1: Introducción a los objetos

El progreso de la abstracción

La **abstracción** consiste en aislar un elemento de su contexto o del resto de los elementos que lo acompañan. En programación, el término se refiere al énfasis en el **"¿qué hace?"** más que en el "¿cómo lo hace?". El común denominador en la evolución de los lenguajes de programación, desde los clásicos o imperativos hasta los orientados a objetos, ha sido el **nivel de abstracción** del que cada uno de ellos hace uso.

Los diferentes paradigmas de programación han aumentado su nivel de abstracción, comenzando desde los lenguajes de máquina, lo *más próximo al ordenador y más lejano a la comprensión humana*; pasando por los lenguajes de comandos, los imperativos, la orientación a objetos (POO), la Programación Orientada a Aspectos(POA); u otros paradigmas como la programación declarativa, etc.

El lenguaje ensamblador es una pequeña abstracción de la máquina subyacente. Muchos de los lenguajes denominados "imperativos" desarrollados a continuación del antes mencionado ensamblador (como Fortran, BASIC y C) eran abstracciones a su vez del lenguaje citado. Estos lenguajes supusieron una gran mejora sobre el lenguaje ensamblador, pero su abstracción principal aún exigía pensar en términos de la **estructura del ordenador** más que en la del **problema en sí** a resolver.

La abstracción encarada desde el punto de vista de la programación orientada a objetos expresa las características esenciales de un objeto, las cuales distinguen al objeto de los demás. Además de distinguir entre los objetos provee límites conceptuales.

Ejemplo

Pensar en términos de objetos es muy parecido a cómo lo haríamos en la vida real. Una analogía sería modelizar un coche en un esquema de POO. Diríamos que el coche es el elemento principal que tiene una serie de características, como podrían ser el color, el modelo o la marca. Además tiene una serie de funcionalidades asociadas, como pueden ser ponerse en marcha, parar o estacionar.

El enfoque orientado a objetos

El **enfoque orientado a objetos** trata de ir más allá, proporcionando herramientas que permitan al programador *representar los elementos del mundo real* tan cerca como sea posible al **dominio de la solución**. Esta representación suele ser lo suficientemente general como para *evitar al programador limitarse a ningún tipo de problema específico*.

La idea es que el programa pueda **autoadaptarse** al problema simplemente añadiendo nuevos tipos de objetos, de manera que la mera lectura del código que describa la solución constituya la lectura de palabras que expresan el problema. Se trata, en definitiva, de una **abstracción** del lenguaje mucho más flexible y potente que cualquiera que haya existido previamente. Por consiguiente, la POO permite al lector describir el problema en términos del propio problema, en vez de en términos del sistema en el que se ejecutará el programa final.

El concepto de objeto

En el paradigma de programación orientada a objetos (POO), un **objeto** es una unidad dentro de un programa de computadora que consta de un **estado** y de un **comportamiento**, que a su vez constan respectivamente de datos almacenados y de tareas realizables durante el tiempo de ejecución.

Estos objetos **interactúan** unos con otros, en contraposición a la visión tradicional en la cual un programa es una colección de subrutinas (funciones o procedimientos), o simplemente una lista de instrucciones para el computador. Cada objeto es capaz de recibir **mensajes**, procesar datos y enviar **mensajes** a otros objetos.

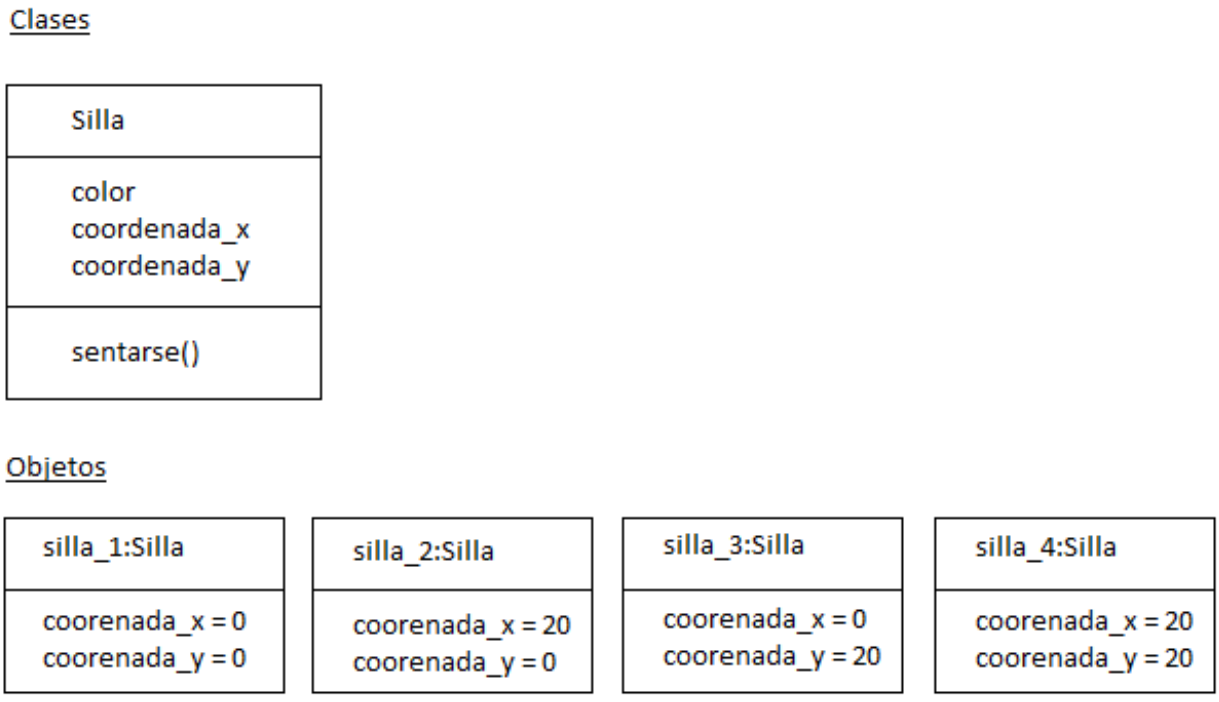
En el mundo de la programación orientada a objetos (POO), un **objeto** es el resultado de la *instanciación de una clase*. Una **clase** es el anteproyecto que ofrece la funcionalidad en ella definida, pero ésta queda implementada sólo al *crear* una **instancia** de la **clase**, en la forma de un **objeto**.

El concepto de clase

En la programación orientada a objetos, una **clase** es una construcción que se utiliza como un modelo (o plantilla) para crear **objetos** de ese tipo. El modelo describe el estado y contiene el comportamiento que todos los objetos creados a partir de esa clase. Un **objeto** creado a partir de una determinada clase se denomina una **instancia** de esa clase.

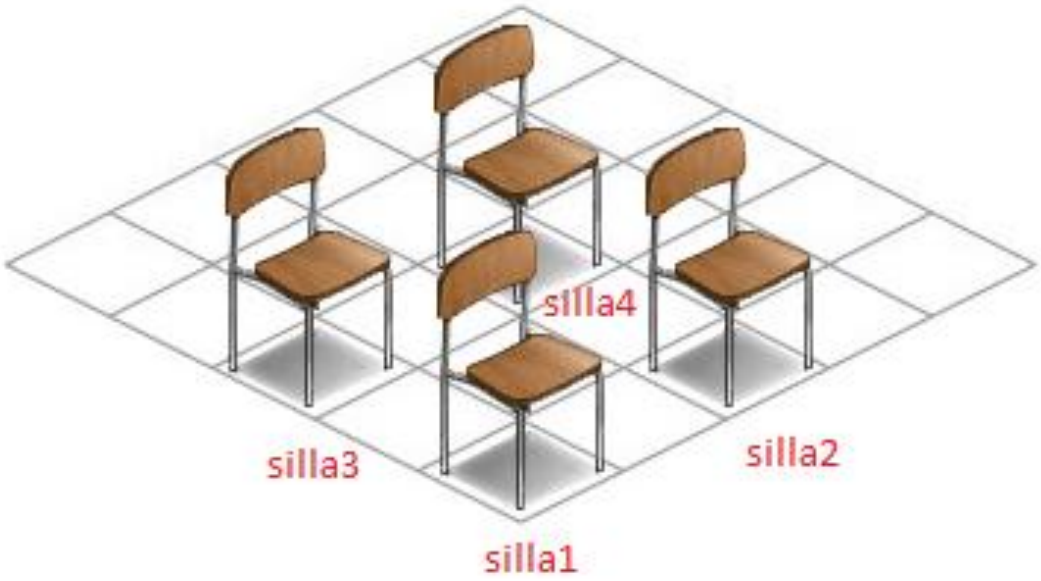
Ejemplo

Si se desea diseñar un juego como el conocido "Sims" en el que se puede diseñar y construir una casa, añadiéndole objetos a las habitaciones, se puede por ejemplo, crear una clase llamada "Silla", en donde contenga propiedades o características de la misma, así como color, coordenada_x, coordenada_y, y también contenga funcionalidades como sentarse. Al crear esta clase se puede instanciar creando tantos objetos de la misma como se desee.



El diagrama anteriormente mostrado sigue el formato del Lenguaje de Modelado Unificado o *Unified Modeling Language* (UML).

De esta manera se puede crear fácilmente un entorno como el representado a continuación.



Y si se crean más clases con más objetos se puede llegar a hacer algo bastante más complejo, de manera sencilla, como se muestra en la siguiente imagen.

La clase como "tipo"

La idea de que todos los **objetos**, aun siendo únicos, son también parte de una **clase de objetos**, todos ellos con características y comportamientos en común, permite la introducción de un nuevo **tipo** dentro de un programa.

Todos los objetos que, con excepción de su estado, son idénticos durante la ejecución de un programa se agrupan en "**clases de objetos**", que es precisamente de donde proviene la palabra clave **clase**. La creación de **clases** es un concepto fundamental en la programación orientada a objetos. Las ***clases*** *funcionan casi como los tipos de datos propios del lenguaje*: es posible la creación de variables de un **tipo** (que se denominan objetos) y manipular estas variables.

Ejemplo

Para que se entienda el concepto anterior se puede hacer un paralelismo entre el enfoque procedimental y el orientado a objetos de la siguiente manera.

```
int     contador;                       // Enfoque procedimental

silla   silla_1  =  new  silla();       // Enfoque orientado a objetos
```

Como se puede observar el tipo en el enfoque procedimental es "**int**" mientras que el tipo en el enfoque orientado a objetos es "**silla**". Así mismo el nombre de la variable en el enfoque procedimental es "**contador**" mientras que el nombre de la variable en el enfoque orientado a objetos, *que en este tipo de enfoque no lo llamaremos nombre de la variable sino que lo llamaremos objeto*, es "**silla_1**".

Dado que una **clase** *describe a un conjunto de objetos* con características (datos) y comportamientos (funcionalidad) idénticos, una clase **es realmente un tipo de datos**, porque un número en coma flotante, por ejemplo, también tiene un conjunto de características y comportamientos. La diferencia radica en que un programador define una **clase** para que encaje dentro de un problema en vez de verse forzado a utilizar un tipo de datos existente que fue diseñado para representar una unidad de almacenamiento en una máquina.

Envío de mensajes o peticiones a un objeto

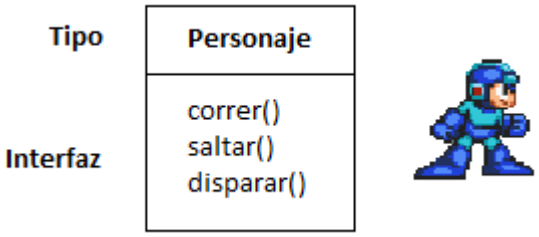
Una vez que se establece una clase, pueden construirse tantos objetos de esa clase como se desee, y manipularlos como si fueran elementos que existen en el problema que se trata de resolver. Pero, ¿cómo se consigue que un objeto haga un trabajo útil para el programador? Debe de haber una forma de hacer **peticiones** al objeto, de manera que éste *desempeñe alguna tarea*, como completar una transacción, dibujar algo en la pantalla o encender un interruptor.

Un **mensaje** es simplemente una **petición** para que *un objeto se comporte de una determinada manera*, ejecutando una de sus funciones miembro. La acción puede ser el envío de otros **mensajes**, el cambio de su estado, o la ejecución de cualquier otra tarea que se requiera que haga el objeto. Esta técnica se conoce como **envío de mensajes**.

Todo objeto tiene una interfaz

Las peticiones que se pueden hacer a un objeto se encuentran definidas en su **interfaz**, y es el tipo de objeto el que determina la **interfaz**.

Un ejemplo podría ser el de un personaje de videojuegos como por ejemplo el conocido "**Mega Man**".




```
Personaje megaMan = new Personaje();
megaMan.disparar();
```

La interfaz establece qué peticiones pueden hacerse a un objeto particular. Sin embargo, debe haber **código** en algún lugar **que permita satisfacer esas peticiones**. Este, junto con los datos ocultos, constituye la **implementación.** Desde el punto de vista de un lenguaje de programación procedural, esto no es tan complicado. Un tipo tiene una función asociada a cada posible petición, de manera que cuando se hace una petición particular a un objeto, se invoca a esa función. Este proceso se suele simplificar diciendo que se ha "**enviado un mensaje**" (*hecho una petición*) a un objeto, y el objeto averigua **qué debe hacer** con el mensaje (*ejecuta el código*).

Aquí, el nombre del **tipo** o clase es **Personaje**, el nombre del **objeto** Personaje particular es **megaMan**, y las **peticiones** que pueden hacerse a un Personaje son **correr**, **saltar** o **disparar**. Es posible crear un Personaje definiendo una "**referencia**" (megaMan) a ese objeto e invocando a **new** para pedir un nuevo objeto de ese tipo. Para enviar un mensaje al objeto, se menciona el nombre del objeto y se conecta al mensaje de petición mediante un punto.

La implementación oculta

Existen dos tipos de programadores: Los **creadores de clases** y los **programadores clientes**. La meta del **programador cliente** es *hacer uso* de un gran repertorio de clases que le permitan acometer el desarrollo de aplicaciones de manera rápida. La meta del **creador de clases** es *construir una clase* que únicamente exponga lo que es necesario al programador cliente, manteniendo **oculto** todo lo demás. ¿Por qué? Porque aquello que esté **oculto** no puede ser utilizado por el **programador cliente**, lo que significa que el **creador de la clase** puede modificar la parte **oculta** a su voluntad, sin tener que preocuparse de cualquier impacto que esta modificación pueda implicar. La parte **oculta** suele representar las interioridades de un objeto que podrían ser corrompidas por un **programador cliente** poco cuidadoso o ignorante, de manera que mientras se mantenga **oculta** su implementación *se reducen los errores en los programas*.

La primera razón que justifica el control de accesos es mantener las manos del programador cliente apartadas de las porciones que no deba manipular.

La segunda razón para un control de accesos es permitir al diseñador de bibliotecas cambiar el funcionamiento interno de la clase sin tener que preocuparse sobre cómo afectará al programador cliente. Por ejemplo, uno puede implementar una clase particular de manera sencilla para simplificar el desarrollo y posteriormente descubrir que tiene la necesidad de reescribirla para que se ejecute más rápidamente. Si tanto la interfaz como la implementación están claramente separadas y protegidas, esto puede ser acometido de manera sencilla.

Límites de una clase

Java usa tres palabras clave explícitas para establecer los límites en una clase: **public**, **private** y **protected**. Estos modificadores de acceso determinan **quién** puede usar las definiciones a las que preceden. La palabra **public** significa que las definiciones siguientes están disponibles para **todo el mundo**. El término **private**, por otro lado, significa que **nadie excepto el creador del tipo** puede acceder a esas definiciones. Así, **private** es un muro de ladrillos entre el creador y el programador cliente. Si alguien trata de acceder a un miembro **private**, obtendrá un error en tiempo de compilación. La palabra clave **protected** actúa como private, con la excepción de que una clase heredada tiene acceso a miembros protected pero no a los private. La herencia será definida algo más adelante.

Java también tiene un "acceso por defecto", que se utiliza cuando no se especifica ninguna de las palabras clave descritas en el párrafo anterior. Este modo de acceso se suele denominar "**amistoso**" o **friendly** porque implica que las clases pueden acceder a los miembros amigos de otras clases que estén en el mismo **package** o **paquete**, sin embargo, fuera del paquete, estos miembros amigos se convierten en private.

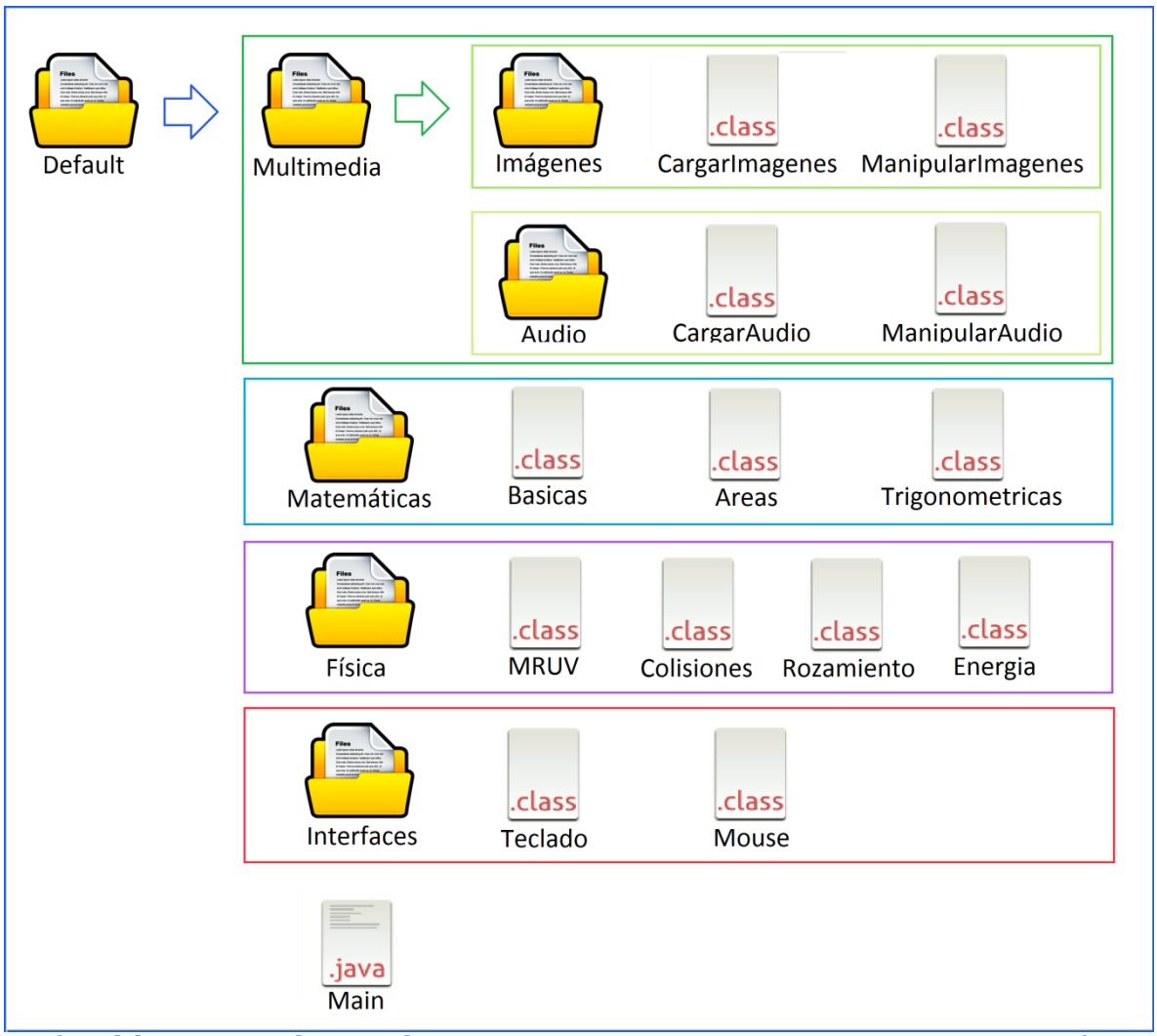
Los paquetes o packages en Java

Un paquete es un espacio que sirve para organizar un conjunto de clases e interfaces relacionadas, al cual cuando es creado se le asigna un nombre para poder identificarlo.

Estos paquetes pueden anidarse obedeciendo una relación jerárquica entre los mismos. Se puede pensar en los paquetes como algo similar a las diferentes carpetas de un ordenador. Puede tener guardadas páginas HTML en una carpeta, imágenes en otra y scripts y aplicaciones en otra distinta. Puesto que un software escrito en el lenguaje de programación Java puede llegar a consistir de cientos o *miles* de clases individuales, tiene sentido que se organicen las clases e interfaces relacionadas en paquetes.

Ejemplo

Para poder ver gráficamente como se organizan los paquetes se supondrá una estructura básica de un video juego. Éste supone un paquete principal que es el "Default" siempre existente y asignado por Java, el cual contiene todos los packages que se crearán y además contiene el archivo principal del programa a construir. Luego dependiendo de las funcionalidades de las clases creadas se pueden separar en paquetes como por ejemplo: Multimedia, Matemáticas, Física, Interfaces, etc.



Reutilizar la implementación

Esta reutilización no es siempre tan fácil de lograr; producir un buen diseño suele exigir experiencia y una visión profunda de la problemática. La reutilización de código es una de las mayores ventajas que proporcionan los lenguajes de programación orientados a objetos.

A este tipo de reutilización se la denomina **composición**. La **composición** es un tipo de relación **dependiente** en dónde una clase está conformada por **objetos de clases** ya existentes: simplemente se crean objetos de la clase existente dentro de la nueva clase.

Esos objetos de la clase existente que se ubican o se crean dentro de la nueva clase se denominan "**objetos miembros**".

La composición se suele representar mediante una relación "**es-parte-de**", como por ejemplo "el motor es una parte de un coche", "el teclado es parte de una computadora", "etc.

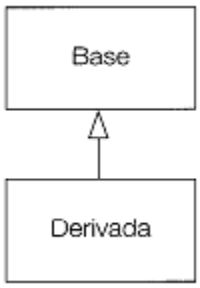
Los **objetos miembros** de la nueva clase suelen ser privados, haciéndolos inaccesibles a los programadores cliente que hagan uso de la clase. Esto permite cambiar los miembros sin que ello afecte al código cliente ya existente. También es posible cambiar los **objetos miembros** *en tiempo de ejecución*, para así cambiar de manera dinámica el comportamiento de un programa. La herencia, que se describe a continuación, no tiene esta flexibilidad, pues el compilador debe imponer restricciones *de tiempo de compilación* en las clases que se creen mediante la herencia.

Dado que la herencia es tan importante en la programación orientada a objetos, casi siempre se enfatiza mucho su uso, de manera que un programador novato puede llegar a pensar que hay que hacer uso de la misma en todas partes. Este pensamiento puede provocar que se elaboren diseños poco elegantes y desmesuradamente complicados. Por el contrario, **primero sería recomendable intentar hacer uso de la composición**, mucho más simple y sencilla. Siguiendo esta filosofía se lograrán diseños mucho más

limpios. Una vez que se tiene cierto nivel de experiencia, la detección de los casos que precisan de la herencia se convierte en obvia.

Herencia: reutilizar interfaz

Parece una pena, acometer todo el problema para crear una clase y posteriormente verse forzado a crear una nueva que podría tener una funcionalidad similar. Sería mejor si pudiéramos hacer uso de una clase ya existente, clonarla, y después hacer al "clon" las adiciones y modificaciones que sean necesarias. Efectivamente, esto se logra mediante la herencia, con la excepción de que si se cambia la clase original (denominada la **clase base**, **clase súper** o **clase padre**), el "clon" modificado (denominado **clase derivada**, **clase heredada**, **subclase** o **clase hijo**) también reflejaría esos cambios.

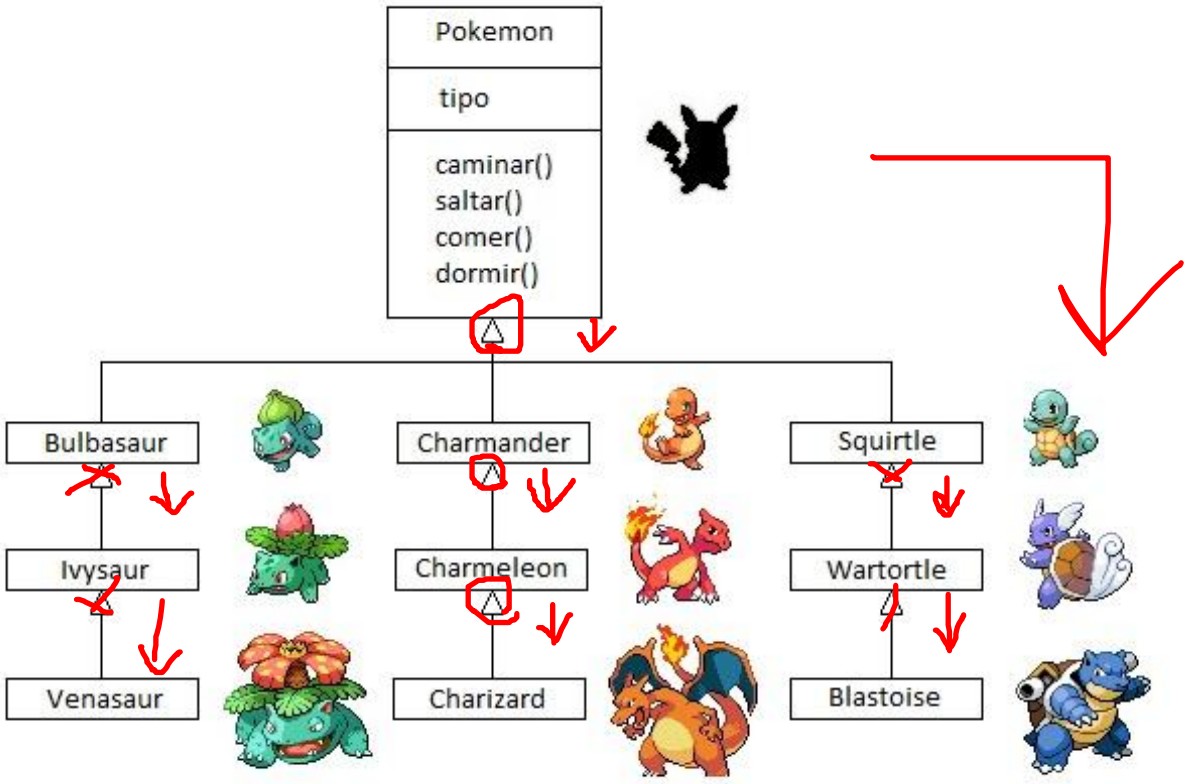


(La flecha de este diagrama UML apunta de la clase derivada a la clase base. Como se verá, puede haber más de una clase derivada.)

Una **clase** hace más que definir los límites de un conjunto de objetos; también tiene relaciones con otras **clases**. Dos **clases** pueden tener características y comportamientos en común, pero una **clase** puede contener más características que otro y también puede manipular más mensajes (o gestionarlos de manera distinta). La herencia expresa esta semejanza entre **clases** haciendo uso del concepto de **clase base** y **clases derivadas**. Una **clase base** contiene todas las características y comportamientos que **comparten** las clases que de él se derivan. A partir de la **clase base**, es posible *derivar* otras **clases** para expresar las distintas maneras de llevar a cabo esta idea.

Ejemplo

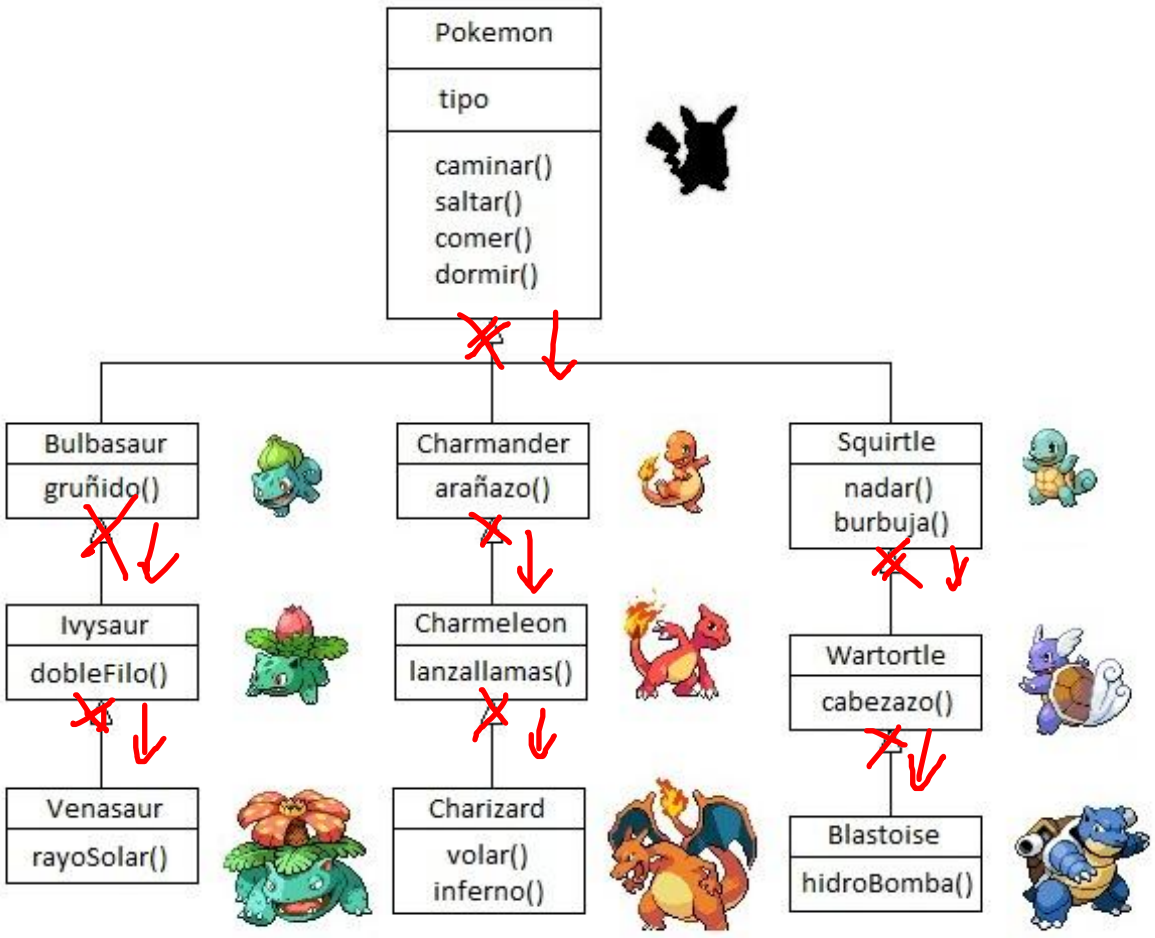
Un ejemplo para ver la herencia pueden ser los conocidos "Pokemons". La clase base es "Pokemon" y cada Pokemon va a tener un tipo (hierba, agua, fuego, etc.). Así mismo cada Pokemon puede caminar, saltar, comer y dormir. A partir de ésta clase, se pueden derivar (heredar) Pokemons específicos: Bulbasaur, Ivysaur, Venasaur, Charmander, Charmeleon, Charizard, Squirtle, Wartortle y Blastoise, pudiendo tener cada uno de los cuales características y comportamientos adicionales. Así mismo Ivysaur heredara todas las propiedades y métodos que Bulbasaur heredo, más los adicionales que pueda tener, lo mismo pasa con Venasaur, ya que son evoluciones de los Pokemons anteriores. De igual manera pueden verse el resto de los Pokemons.



Al heredar a partir de una tipo existente, se crea un **nuevo tipo**. Este **nuevo tipo** contiene no sólo los miembros del tipo existente sino lo que es más importante, *duplica la interfaz*

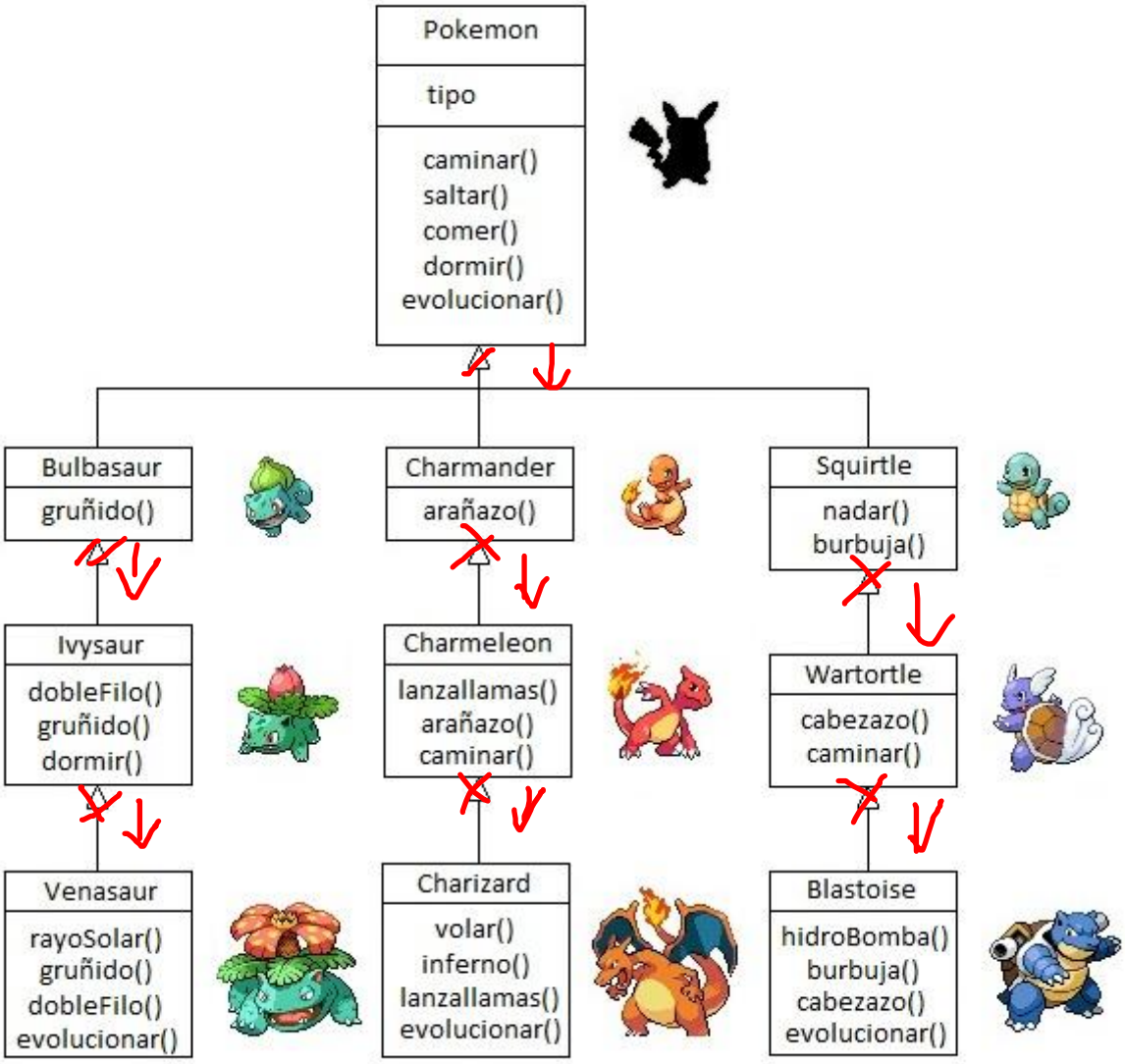
de la clase base. Es decir, todos los mensajes que pueden ser enviados a objetos de la clase base también pueden enviarse a los objetos de la clase derivada. Dado que sabemos el **tipo** de una clase en base a los mensajes que se le pueden enviar, la clase derivada es del mismo **tipo** que la clase base. Así, en el ejemplo anterior, "**Bulbasaur es un Pokemon**".

Generalmente cuando se crea una clase derivada *se añaden nuevas funciones a la misma*. Estas **funciones nuevas** no forman parte de la interfaz de la clase base, lo que significa que la clase base simplemente no hacía todo lo que ahora se deseaba, por lo que fue necesario añadir nuevas **funciones**. Sin embargo, debería considerarse detenidamente la posibilidad de que la clase base llegue también a necesitar estas **funciones adicionales**.



Aunque la herencia puede implicar en ocasiones que se van a añadir funcionalidades a una interfaz, esto no tiene por qué ser siempre así. Una posibilidad importante dentro de la POO es la capacidad de variar el comportamiento de una función ya existente en la clase base. A esto se le llama **redefinición de la función**.

Para **redefinir una función** simplemente se crea una nueva definición de la función dentro de la clase derivada. De esta manera puede decirse que "*se está utilizando la misma función de la interfaz pero se desea que se comporte de manera distinta dentro del nuevo tipo*".



Para que se termine de entender la idea se detallará que puede hacer ahora cada Pokemon.

Bulbasaur: Como propiedad tiene tipo, que en este caso será Hierba. Como métodos tendrá caminar, saltar, comer, dormir, evolucionar y gruñido, entendiéndose este último método como un ataque del Pokemon.

Ivysaur: Como propiedad tiene tipo, que en este caso será Hierba. Como métodos tendrá caminar, saltar, comer, dormir (que fue redefinido para que por ejemplo no duerma este Pokemon), evolucionar, doble filo, entendiéndose como un ataque del Pokemon, y gruñido, que fue redefinido por ejemplo para que el ataque cause más daño.

Venasaur: Como propiedad tiene tipo, que en este caso será Hierba. Como métodos tendrá caminar, saltar, comer, dormir (que fue redefinido en la clase Ivysaur para que no duerma, gruñido, que es nuevamente redefinido para que por ejemplo no pueda utilizar más este ataque, doble filo, que es redefinido para que por ejemplo cause más daño, rayoSolar, entendiéndose éste como un ataque del Pokemon, y evolucionar, el cual es redefinido ya que este no puede evolucionar mas.

Charmander: Como propiedad tiene tipo, que en este caso será Fuego. Como métodos tendrá caminar, saltar, comer, dormir, evolucionar y arañazo, entendiéndose este último método como un ataque del Pokemon.

Charmeleon: Como propiedad tiene tipo, que en este caso será Fuego. Como métodos tendrá caminar, que fue redefinido para que por ejemplo camine a mayor velocidad, saltar, comer, dormir, evolucionar, lanzallamas, entendiéndose como un ataque del Pokemon, y arañazo, que fue redefinido por ejemplo para que el ataque cause más daño.

Charizard: Como propiedad tiene tipo, que en este caso será Fuego. Como métodos tendrá caminar, que fue redefinido en la clase Charmeleon para que camine a mayor velocidad, saltar, comer, dormir, volar, lanzallamas, que fue redefinido para que cause más daño, inferno, entendiéndose éste como un ataque del Pokemon, y evolucionar, el cual es redefinido ya que este no puede evolucionar mas.

Squirtle: Como propiedad tiene tipo, que en este caso será Agua. Como métodos tendrá caminar, saltar, comer, dormir, evolucionar, nadar y burbuja, entendiéndose este último método como un ataque del Pokemon.

Wartortle: Como propiedad tiene tipo, que en este caso será Agua. Como métodos tendrá caminar, que fue redefinido para que por ejemplo camine a menor velocidad, saltar, comer, dormir, evolucionar, nadar y cabezazo, entendiéndose como un ataque del Pokemon.

Blastoise: Como propiedad tiene tipo, que en este caso será Agua. Como métodos tendrá caminar, que fue redefinido en la clase Wartortle para que camine a menor velocidad, saltar, comer, dormir, nadar, burbuja, que fue redefinido para que no se pueda usar más, cabezazo, que fue redefinido para que cause más daño, hidroBomba, entendiéndose éste como un ataque del Pokemon, y evolucionar, el cual es redefinido ya que este no puede evolucionar mas.

Polimorfismo

A menudo se desea tratar un objeto no como el **tipo específico** del que es, sino como su **tipo base**. Esto permite escribir código que no depende de **tipos específicos**. El **polimorfismo** indica que una variable pasada o esperada puede adoptar múltiples formas. Cuando se habla de **polimorfismo** en programación orientada a objetos se suelen entender dos cosas:

1. La primera se refiere a que se puede trabajar con un objeto de una clase sin importar de qué clase se trata. Es decir, se trabajará igual sea cual sea la clase a la que pertenece el objeto. Esto se consigue mediante **jerarquías de clases** y clases abstractas.

2. La segunda suele referirse a la posibilidad de declarar métodos con el mismo nombre que pueden tener diferentes argumentos dentro de una misma clase.

La capacidad de un programa de **trabajar con más de un tipo** de objeto se conoce con el nombre de **polimorfismo**

Ejemplo

Considérese el ejemplo de los **Pokemons**. Para demostrar el **polimorfismo**, se escribirá un único fragmento de código que ignora los detalles específicos de tipo y solamente hace referencia a la clase base. El código está *desvinculado* de información específica del tipo, y por consiguiente es más fácil de escribir y entender. Y si se añade un nuevo tipo -por ejemplo un Pokemon llamado **Pikachu**- mediante herencia, el código que se escriba trabajará tan perfectamente con el nuevo **Pokemon** como lo hacía con los tipos ya existentes, y por consiguiente, el programa es ampliable.

Si se escribe un método en Java

```
void evolucionarPokemon (Pokemon p) {
p.evolucionar() ;
}
```

Esta función se entiende con cualquier **Pokemon**, y por tanto, es independiente del tipo específico de objeto (**Pokemon**) que esté evolucionando. En cualquier otro fragmento del programa se puede usar la función **evolucionarPokemon()** :

```
Bulbasaur b = new Bulbasaur() ;
Charmander c = new Charmander() ;
Squirtle s = new Squirtle() ;
evolucionarPokemon(b) ;
evolucionarPokemon(c) ;
evolucionarPokemon(s) ;
```

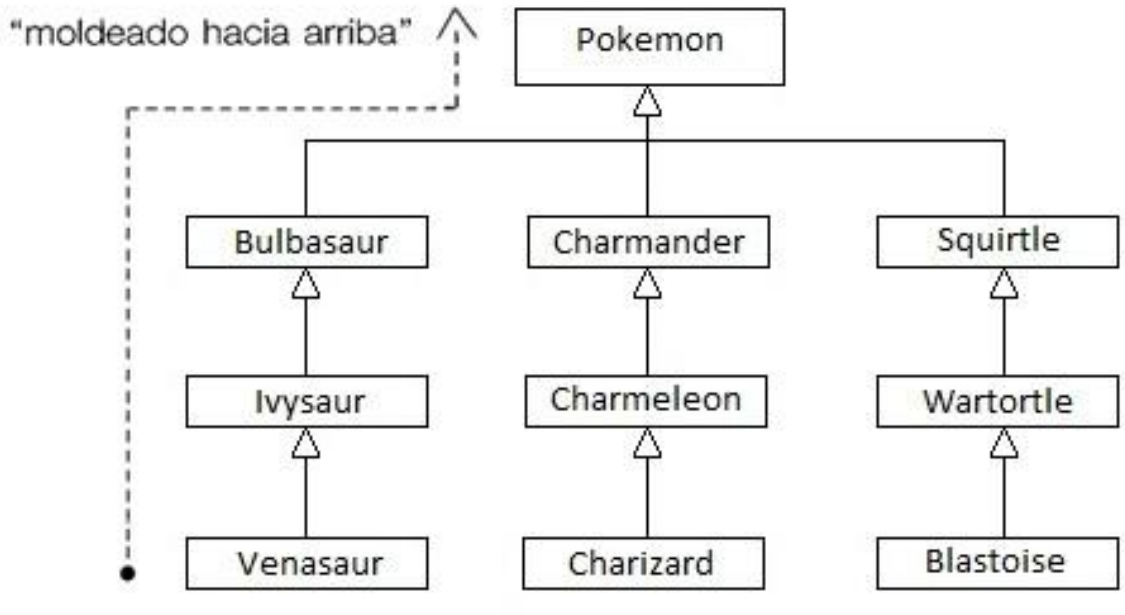
```
Pokemon p;

//-> Menu ingrese pokemon
int opc = s.nextInt();

if(opc==1){
   p = new Pikachu();
} else if(opc==2){
   p = new Bulbasaur();
} ... etc
```

Las llamadas a **evolucionarPokemon()** trabajan correctamente, independientemente del tipo de objeto. Lo que está ocurriendo es que se está pasando un objeto del tipo **bulbasaur** a una función que espera un **Pokemon**. Como **bulbasaur** *es un* **Pokemon**, puede ser tratado como tal por **evolucionarPokemon().** Es decir, cualquier mensaje que **evolucionarPokemon()** pueda enviar a un **Pokemon**, también podrá aceptarlo un objeto del tipo **bulbasaur**. Por tanto, obrar así es algo totalmente seguro y lógico.

A este proceso de tratar un tipo derivado como si fuera el tipo base se le llama conversión de tipos *hacia arriba*. El nombre **cast** se utiliza en el sentido de convertir un molde, y es hacia arriba. Por consiguiente, hacer un **casting** a la clase base es moverse hacia arriba por el diagrama de herencias.



Un programa orientado a objetos siempre tiene algún moldeado hacia arriba pues ésta es la manera de desvincularse de tener que conocer el tipo exacto con que se trabaja en cada instante. Si se echa un vistazo al código de **evolucionarPokemon()** :

```
p.evolucionar();
```

Obsérvese que no se dice "caso de ser **bulbasaur**, hacer esto; caso de ser **charmander**, hacer esto otro, etc.". Si se escribe código que compruebe todos los tipos posibles que puede ser un **Pokemon**, el tipo de código se complica, además de hacerse necesario modificarlo cada vez que se añade un nuevo tipo de **Pokemon**. Aquí, simplemente se dice que "en el caso de los Pokemons, se sabe que es posible aplicarles la acción de **evolucionar()**.
Lo que más llama la atención del código de **evolucionarPokemon()** es que, de alguna manera, se hace lo correcto. Invocar a **evolucionar()** para **bulbasaur** hace algo distinto que invocar a **evolucionar()** para **charmander** o **squirtle**, pero cuando se envía el mensaje **evolucionar()** a un **Pokemon** anónimo, se da el comportamiento correcto basándose en el tipo actual de **Pokemon**.

Clases base abstractas e interfaces

A menudo es deseable que la clase base *únicamente* presente una interfaz para sus clases derivadas. Es decir, no se desea que nadie cree objetos de la clase base, sino que sólo se hagan moldeados hacia arriba de la misma de manera que se pueda usar su interfaz. Esto se logra convirtiendo esa clase en *abstracta* usando la palabra clave **abstract**. Si alguien trata de construir un objeto de una clase **abstracta** el compilador lo evita. Esto es una herramienta para fortalecer determinados diseños.

También es posible utilizar la palabra clave **abstract** para describir un método que no ha sido aún implementado -indicando "he aquí una función interfaz para todos los tipos que se hereden de esta clase, pero hasta la fecha no existe una implementación de la misma". Se puede crear un método **abstracto** sólo dentro de una clase **abstracta**. Cuando se hereda la clase, debe implementarse el método o de lo contrario también *la clase heredada se convierte en abstracta*. La creación de métodos **abstractos** permite poner un método en una interfaz sin verse forzado a proporcionar un fragmento de código, posiblemente sin significado, para ese método.

Localización de objetos y longevidad

La POO consiste en tipos de datos abstractos, herencia y polimorfismo, aunque también hay otros aspectos no menos importantes.

Uno de los factores más importantes es la manera de **crear** y **destruir** objetos. ¿Dónde están los datos de un objeto y cómo se controla su longevidad (tiempo de vida)? En este punto hay varias filosofías de trabajo. En C++ el enfoque principal es el **control de la eficiencia**. Para lograr un **tiempo de ejecución óptimo**, es posible determinar el espacio de almacenamiento y la longevidad en **tiempo de programación**, ubicando los objetos en la pila o en el área de almacenamiento estático. De esta manera se prioriza la **velocidad de la asignación** y **liberación de espacio** de almacenamiento, cuyo control puede ser de gran valor en determinadas situaciones. Sin embargo, se sacrifica en **flexibilidad** puesto que es necesario conocer la cantidad exacta de objetos, además de su longevidad y su tipo, mientras se escribe el programa. Si se está tratando de resolver un problema más general, este enfoque resulta demasiado restrictivo.

El segundo enfoque es crear objetos **dinámicamente** en un espacio de memoria denominado el montículo o montón (*heap*). En este enfoque, no es necesario conocer hasta **tiempo de ejecución** el número de objetos necesario, cuál es su longevidad o a qué tipo exacto pertenecen. Estos aspectos se determinarán justo en el preciso momento en que se ejecute el programa. Si se necesita un nuevo objeto, simplemente se construye en el *montículo* en el instante en que sea necesario. Dado que el almacenamiento se gestiona dinámicamente, en tiempo de ejecución, la **cantidad de tiempo** necesaria para asignar espacio de almacenamiento en el montículo es bastante **mayor** que el tiempo necesario para asignar espacio a la pila.

En Java cada vez que se desea crear un objeto se usa la palabra clave **new** para construir una instancia dinámica de ese objeto.

Hay otro aspecto, sin embargo, a considerar: la **longevidad** de un objeto (el tiempo desde que el objeto es creado hasta que se destruye). Con los lenguajes que permiten la creación de objetos en el montículo (**Java**), el compilador determina cuánto dura cada objeto y puede destruirlo cuando no es necesario. Sin embargo, si se crea en la pila, el compilador no tiene conocimiento alguno sobre su longevidad (**C++**). En un lenguaje

como C++ hay que determinar en **tiempo de programación** cuándo destruir el objeto, lo cual puede conducir a fallos de memoria si no se hace de manera correcta. **Java** proporciona un **recolector de basura** que descubre automáticamente cuándo se ha dejado de utilizar un objeto, que puede, por consiguiente, ser destruido. Un **recolector de basura** es muy conveniente, al reducir el número de aspectos a tener en cuenta, así como la cantidad de código a escribir. Y lo que es más importante, el recolector de basura proporciona un **nivel de seguridad** mucho **mayor** contra el problema de los fallos de memoria.

El recolector de basura de Java

Cada objeto requiere recursos simplemente para poder existir, fundamentalmente **memoria**. Cuando un objeto deja de ser necesario debe ser **eliminado** de manera que estos **recursos** queden disponibles para poder **reutilizarse**. En situaciones de programación sencillas la cuestión de cuándo eliminar un objeto no se antoja complicada: se crea el objeto, se utiliza mientras es necesario y posteriormente debe ser destruido. Sin embargo, no es difícil encontrar situaciones en las que esto se complica.

Ejemplo

Supóngase, por ejemplo, que se está diseñando un juego donde el jugador debe diseñar y construir un aeropuerto, podría haber una parte en donde se necesite gestionar el tráfico aéreo. A primera vista, parece simple: construir un **contenedor** para albergar **aviones**, crear a continuación un nuevo **avión** y ubicarlo en el **contenedor** (para cada avión que aparezca en la zona a controlar). En el momento de **eliminación**, se **borra** (suprime) simplemente el objeto **avión** correcto cuando un avión abandona el aeropuerto.

Pero quizás, se tiene otro sistema para guardar los **datos** de los aviones*, datos que no requieren atención inmediata*. Quizás, se trata de un registro del plan de viaje de todos los pequeños aviones que abandonan el aeropuerto. Es decir, se dispone de un **segundo contenedor** de

aviones pequeños, y siempre que se crea un objeto **avión** también se introduce en este segundo contenedor si se trata de un avión pequeño. Posteriormente, algún proceso en segundo plano lleva a cabo operaciones con los objetos de este segundo contenedor cada vez que el sistema está ocioso.

Ahora el problema se complica: ¿cómo se sabe cuándo destruir los objetos? Cuando el programa principal ha acabado de usar el **objeto**, puede que otra parte del sistema *lo esté usando* (*o lo vaya a usar en un futuro*), ya que un mismo objeto se encuentra en **dos contenedores** (en dos lugares). Este problema surge en numerosísimas ocasiones, y los sistemas de programación en los que los objetos deben de borrarse explícitamente cuando acaba de usarlos, pueden volverse bastante **complejos**.

Con Java, el problema de vigilar que se libere la memoria se ha implementado en el recolector de basura. El recolector "sabe" cuándo se ha dejado de utilizar un objeto y libera la memoria que ocupaba automáticamente. Esto hace que el proceso de programar en Java sea mucho más sencillo que el hacerlo en C++. Hay muchas menos decisiones que tomar y menos obstáculos que sortear.

Los recolectores de basura frente a la eficiencia y flexibilidad

Si todo esto es tan buena idea, ¿por qué no se hizo lo mismo en C++? Bien, por supuesto, hay un precio que pagar por todas estas comodidades de programación, y este precio consiste en sobrecarga en tiempo de ejecución. Como se mencionó anteriormente, en C++ es posible crear objetos en la pila (pero no se dispone de la flexibilidad de crear tantos como se desee en tiempo de ejecución). La creación de objetos en la pila es la manera más eficiente de asignar espacio a los objetos y de liberarlo.

Java es más simple que C++, pero a cambio es menos eficiente y en ocasiones ni siquiera aplicable. Sin embargo, para un porcentaje elevado de problemas de programación, Java es la mejor elección.

Colecciones e iteradores

Si se desconoce el número de objetos necesarios para resolver un problema en concreto o cuánto deben durar, también se desconocerá cómo almacenar esos objetos. ¿Cómo se puede saber el espacio a reservar para los mismos? De hecho, no se puede, pues esa información se desconocerá hasta tiempo de ejecución.

La solución a la mayoría de problemas de diseño en la orientación a objetos parece sorprendente: se crea **otro tipo de objeto**. El nuevo **tipo de objeto** que resuelve este problema particular tiene referencias a otros **objetos**. Por supuesto, es posible hacer lo mismo con un **array**, disponible en la mayoría de lenguajes. Pero hay más. Este nuevo objeto, generalmente llamado ***contenedor***, se expandirá a sí mismo cuando sea necesario para albergar cuanto se coloque dentro del **contenedor**. Simplemente se crea el objeto **contenedor**, y él se encarga de los detalles.

Un tema que aparece frecuentemente en la programación es el de organizar una **colección de objetos** en memoria. Este tema también es conocido con el título “estructuras de datos”, y se relaciona también a distintos **algoritmos** para agilizar el acceso.

Pero *¿para qué usar colecciones si puedo almacenar objetos dentro de un Array?*

Todo programador aprende, apenas empieza, un par de **estructuras de datos** simples. Se podría decir que la primer **estructura de datos** es la **variable**. Almacena un solo objeto, y el tiempo de acceso es **ínfimo**. La segunda **estructura de datos** que siempre se aprende es el **arreglo** (en inglés "*array*"). Esta colección permite obtener un valor dado una posición en una tabla. Es muy veloz ya que esta estructura tiene un fuerte paralelo con cómo se organiza internamente la memoria de una computadora. Los valores en un arreglo están en orden, y en ese mismo orden están dispuestos en la memoria, por lo tanto el acceso lo maneja directamente el hardware (porque la memoria funciona físicamente como un arreglo).

Un arreglo en Java se hace así

```
String[] nombre = new String[10];
```

Se puede usar arreglos para todo. Se puede usar para guardar una lista de helados, para guardar las notas que se sacaron los alumnos de una clase, para guardar las coordenadas que forman un mapa, etc. Supóngase que se quiere guardar las notas de los alumnos de una clase en un arreglo. Primero se crea una clase con el nombre del alumno y su nota:

```
class Nota
{
	String alumno;
	int nota;

	Nota(String alumno, int nota)
	{
	this.alumno = alumno;
	this.nota = nota;
	}

public int get Nota() { return nota; }
public string get Alumno() { return alumno; }
}
```

Ahora se almacena las notas de 500 alumnos:

```
Nota[] notas = new Nota[500];

notas[0] = new Nota("Juan", 7);
notas[1] = new Nota("Ezequiel", 7);
notas[2] = new Nota("Pedro", 8);
notas[3] = new Nota("Manuel", 4);
notas[4] = new Nota("David", 6);

// ...

notas[499] = new Nota("Elías", 9);
```

Ahora... ¿Qué pasa se desea averiguar la nota de Elías? Se desconoce "a priori" en qué posición del arreglo está guardada la nota. La única manera de proceder es simplemente recorrer el arreglo y preguntar si ya llegamos al alumno deseado:

```
for(Nota n : notas)
{
	if(n.getNombre().equals("Elías"))
	{
		return n.getNota();
	}
}
```

¡Esto es lentísimo! Está bien, “Elías” es el peor caso, ya que se encuentra último. A Elías se lo encuentra después de haber efectuado 499 comprobaciones. Pero si se supone que todos los alumnos son accedidos más o menos con una misma probabilidad se tiene que el número de comprobaciones promedio que se necesitarán por búsqueda de alumno es de 250!

¿Se puede hacer mejor? ¿Qué tal si se pudiera contar con que el orden de las notas coincide con el orden alfabético del nombre de los alumnos? En ese caso se podría ir al alumno 250, y preguntar: ¿Elías es mayor o menor alfabéticamente que vos? Si el alumno 250 es Luis, Elías claramente está antes qué él. Luego, Elías está entre el 0 y el 249. Entonces vamos a comprobar al alumno 125, que es, por ejemplo, “Carlos”. Elías está después que Carlos, entonces sabemos que su posición es mayor a 125 (y era menor a 249). Así vamos acotando hasta llegar a Elías en, a lo sumo, 9 comprobaciones. A este algoritmo se le suele llamar “**búsqueda dicotómica** o **binaria**”.

Mediante algoritmos, se redujo la necesidad de comprobar 250 veces en promedio, a comprobar a lo sumo nueve veces. Y si en vez de 500 nombres hubiera 100.000 sería necesario comprobar, a lo sumo, 17 veces. El uso de este algoritmo depende del hecho de que los datos están ordenados. Éste es el enfoque que usan los **contenedores**, sirven para guardar objetos de algún tipo en particular, y estos **contenedores** están preparados para manipular esos tipos de datos a los cuales correspondan los **objetos** que se les pase, de la manera más eficiente.

Afortunadamente, un buen lenguaje POO viene con un conjunto de **contenedores** como parte del propio lenguaje. Java también tiene **contenedores** en su biblioteca estándar. En algunas bibliotecas, se considera que un **contenedor genérico** es lo suficientemente bueno para todas las necesidades, mientras que en otras (como en **Java**) la biblioteca tiene distintos tipos de **contenedores** para distintas necesidades.

Todos los **contenedores** tienen alguna manera de *introducir y extraer cosas*; suele haber funciones para añadir elementos a un contenedor, y otras para extraer de nuevo esos elementos. Pero sacar los elementos puede ser más problemático porque una función de selección única suele ser restrictiva. ¿Qué ocurre si se desea manipular o comparar un conjunto de elementos del contenedor y no uno sólo?

La solución es un **iterador**, que es un objeto cuyo trabajo es seleccionar los elementos de dentro de un contenedor y presentárselos al **usuario del iterador**. Como clase, también proporciona cierto nivel de abstracción. Esta abstracción puede ser usada para separar los detalles del contenedor del código al que éste está accediendo. El contenedor, a través del iterador, se llega a abstraer hasta convertirse en una simple secuencia, que puede ser recorrida gracias al **iterador** sin tener que preocuparse de la estructura subyacente. Esto proporciona la flexibilidad de cambiar fácilmente la estructura de datos subyacente sin que el código de un programa se vea afectado.

Hay dos razones por las que es necesaria una selección de contenedores. En primer lugar, los contenedores proporcionan distintos tipos de interfaces y comportamientos externos. Cualquiera de éstos podría proporcionar una *solución mucho más flexible a un problema*. En segundo lugar, distintos contenedores tienen distintas eficiencias en función de las operaciones.

Finalmente, debe recordarse que un **contenedor** es sólo un **espacio de almacenamiento** en el que colocar objetos. Si este espacio resuelve todas las necesidades, no importa realmente cómo está implementado.

La jerarquía de raíz Única

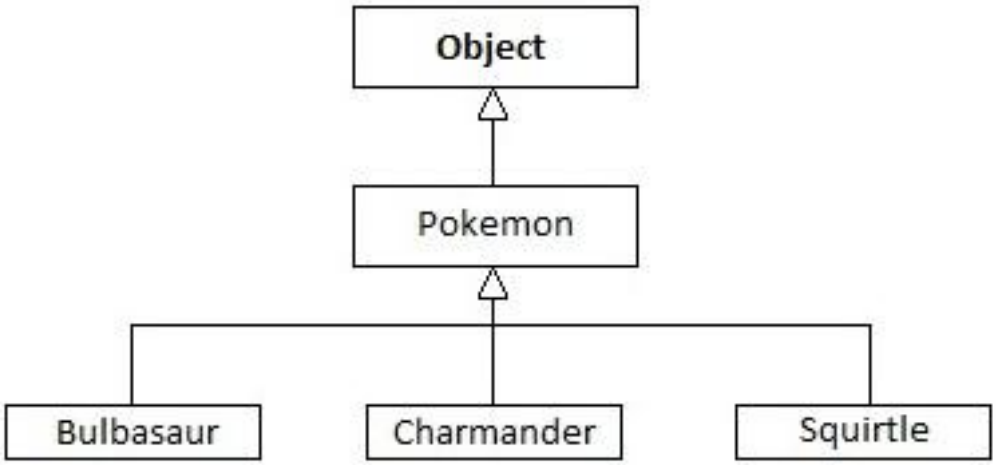
Uno de los aspectos de la POO que se ha convertido especialmente prominente desde la irrupción de C++ es si todas las clases en última instancia *deberían ser heredadas de una única clase base*. En Java la respuesta es "sí" y el nombre de esta última clase base es simplemente **Object**. Resulta que los beneficios de una jerarquía de raíz única son enormes.

Todos los objetos en una jerarquía de raíz única tienen una **interfaz en común**, por lo que en última instancia son del **mismo tipo**. La alternativa (proporcionada por C++) es el desconocimiento de que todo pertenece al mismo tipo fundamental. Esto encaja en el modelo de C mejor, y puede pensarse que es menos restrictivo, pero cuando se desea hacer programación orientada a objetos pura, es necesario proporcionar una **jerarquía completa** para lograr el mismo nivel de conveniencia intrínseco a otros lenguajes POO. Y en cualquier nueva biblioteca de clases que se adquiera, se utilizará alguna interfaz incompatible. Hacer funcionar esta nueva interfaz en un diseño lleva un gran esfuerzo. Si se empieza de cero, otras alternativas, como Java, resultarán mucho más productivas.

Puede garantizarse que todos los objetos de una jerarquía de raíz única (como la proporcionada por Java) tienen cierta funcionalidad. Una jerarquía de raíz única, junto con la creación de todos los objetos en el montículo, simplifica enormemente el paso de argumentos.

Una jerarquía de raíz única simplifica muchísimo la implementación de un recolector de basura. Si no existiera este tipo de jerarquía ni la posibilidad de manipular un objeto a través de referencias, sería muy difícil implementar un recolector de basura.

Dado que está garantizado que en tiempo de ejecución la información de tipos está en todos los objetos, jamás será posible encontrar un objeto cuyo tipo no pueda ser determinado.



De esta manera por ejemplo, Bulbasaur aparte de ser del tipo Bulbasaur, sería del tipo Pokemon y también del tipo Object. Y Pokemon aparte de ser del tipo Pokemon sería también del tipo Object.

Manejo de excepciones: tratar con errores

El **manejo de errores** ha sido, desde el principio de la existencia de los lenguajes de programación, uno de los aspectos más difíciles de abordar.

El **manejo de excepciones** está íntimamente relacionado con el lenguaje de programación y a veces incluso con el sistema operativo. Una excepción es un **objeto** que es "**lanzado**", "**arrojado**" desde *el lugar en que se produce el error*, y que puede ser "**capturado**" por el gestor de excepción apropiado diseñado para manejar ese tipo de error en concreto. Es como si la gestión de excepciones constituyera un *cauce de ejecución diferente*, **paralelo**, que puede tomarse cuando algo va mal. Y dado que usa un cauce de ejecución distinto, no tiene por qué interferir con el código de ejecución normal. De esta manera el código es más simple de escribir puesto que no hay que estar comprobando los errores continuamente. Además, una excepción lanzada no es como un valor de error devuelto por una función, o un indicador (bandera) que una función pone a uno para indicar que se ha dado cierta condición de error (éstos podrían ser ignorados). Una excepción no se puede ignorar, por lo que se garantiza que será tratada antes o después. Finalmente, las excepciones proporcionan una manera de recuperarse de manera segura de una situación anormal. En vez de simplemente terminar el programa,

muchas veces es posible volver a poner las cosas en su sitio y restablecer la ejecución del programa, logrando así que éstos sean mucho más robustos.

El manejo de excepciones de Java destaca entre los lenguajes de programación pues en Java, éste se encuentra imbuido desde el principio, y es **obligatorio** utilizarlo. Si no se escribe un código de manera que maneje excepciones correctamente, se obtendrá un **mensaje de error** en tiempo de compilación. Esta garantía de consistencia hace que la gestión de errores sea mucho más sencilla.

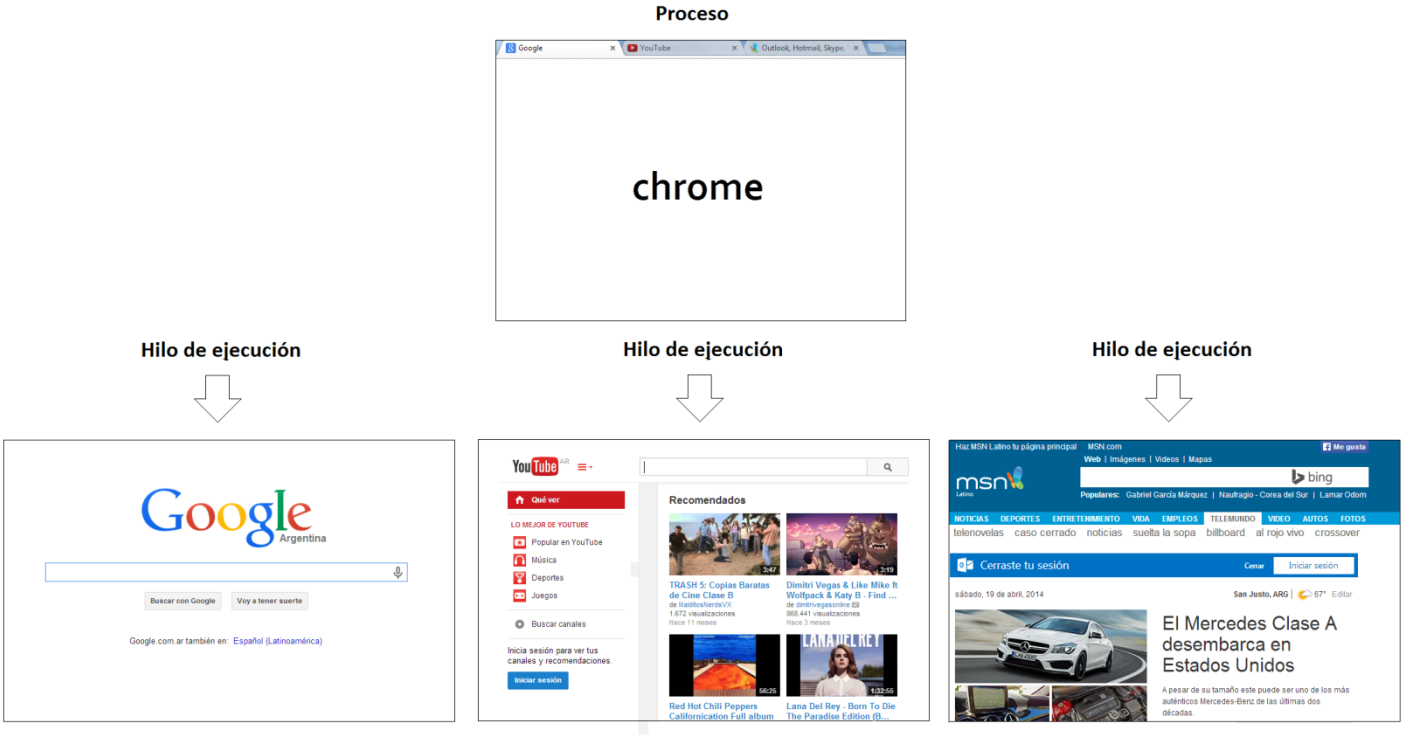
Multihilo

Un concepto fundamental en la programación es la idea de *manejar más de una tarea en cada instante*. Muchos problemas de programación requieren que el programa sea capaz de detener lo que esté haciendo, llevar a cabo algún otro problema, y volver a continuación al proceso principal. Se han buscado múltiples soluciones a este problema. Inicialmente, los programadores que tenían un conocimiento de **bajo nivel** de la máquina, escribían rutinas de servicio de interrupciones, logrando la **suspensión** del **proceso principal** mediante interrupciones hardware. Aunque este enfoque funcionaba bastante bien, era dificultoso y no portable, por lo que causaba que transportar un programa a una plataforma distinta de la original fuera **lento y caro**.

A veces, es necesario hacer uso de las interrupciones para el manejo de tareas críticas en el tiempo, pero hay una gran cantidad de problemas en los que simplemente se intenta dividir un problema en fragmentos de código que pueden ser ejecutados por separado, de manera que se logra un menor tiempo de respuesta para todo el programa en general. Dentro de un programa, estos fragmentos de código que pueden ser ejecutados por separado, se denominan **hilos**, y el concepto general se denomina **multihilos**.

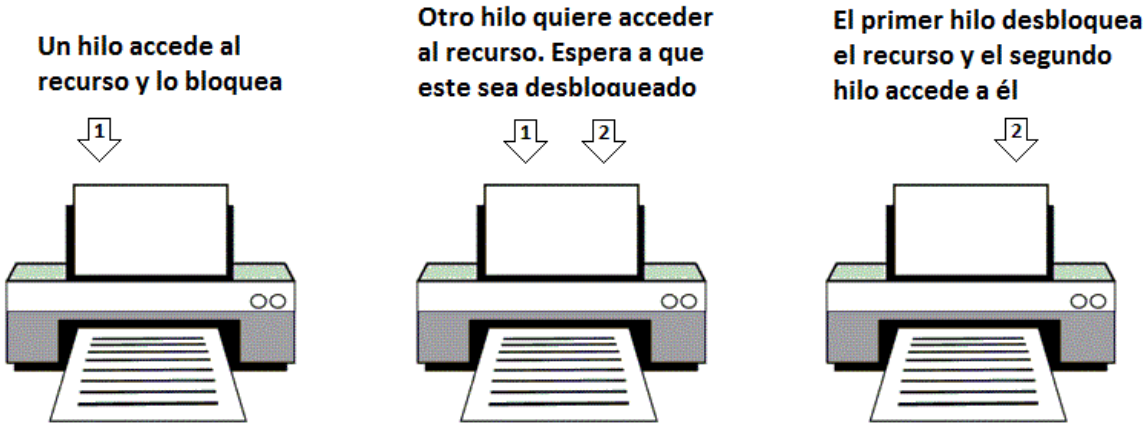
Ejemplo

Un ejemplo común de aplicación multihilo es la interfaz de usuario. Por ejemplo el navegador Google Chrome sería un proceso, mientras que cada una de las ventanas que se pueden tener abiertas simultáneamente trayendo páginas HTML estaría formada por al menos un hilo.



Normalmente, los hilos no son más que una **herramienta** para facilitar la planificación en un **monoprocesador**. Pero si el sistema operativo soporta múltiples procesadores, es posible asignar cada hilo a un procesador distinto de manera que los hilos se ejecuten verdaderamente en paralelo. Uno de los aspectos más destacables de la programación *multihilo* es que el programador no tiene que preocuparse de si hay uno o varios procesadores. El programa se divide de forma lógica en hilos y si hay más de un procesador, se ejecuta más rápidamente, sin que sea necesario llevar a cabo ningún ajuste adicional sobre el código.

Todo esto hace que el manejo de hilos parezca muy sencillo. Hay un inconveniente: los *recursos compartidos*. Si se tiene más de un hilo en ejecución tratando de acceder al mismo **recurso**, se plantea un problema. Por ejemplo, dos procesos no pueden enviar simultáneamente información a una impresora. Para resolver el problema, los recursos que pueden ser compartidos como la impresora, deben bloquearse mientras se están usando. Por tanto, un hilo bloquea un recurso, completa su tarea y después libera el bloqueo de manera que alguien más pueda usar ese recurso.



El hilo de Java está incluido en el propio lenguaje, lo que hace que un tema de por sí complicado se presente de forma muy sencilla. El manejo de hilos se soporta a **nivel de objeto**, de manera que un hilo de ejecución se representa por un objeto. Java también proporciona bloqueo de recursos limitados; puede bloquear la memoria de cualquier objeto (que en el fondo, no deja de ser un tipo de recurso compartido) de manera que sólo un objeto pueda usarlo en un instante dado. Esto se logra mediante la palabra clave **synchronized**. Otros tipos de recursos deben ser explícitamente bloqueados por el programador, generalmente, creando un objeto que represente el bloqueo que todos los hilos deben comprobar antes de acceder al recurso.

Persistencia

Al crear un objeto, existe tanto tiempo como sea necesario, pero bajo ninguna circunstancia sigue existiendo una vez que el programa acaba. Hay situaciones en las que sería increíblemente útil el que un objeto pudiera *existir y mantener su información* incluso cuando el programa ya no esté en ejecución. De esta forma, la siguiente vez que se lance el programa, el objeto estará ahí y seguirá teniendo la misma información que tenía la última vez que se ejecutó el programa. Por supuesto, es posible lograr un **efecto similar** escribiendo la información en un **archivo** o en una **base de datos**, pero con la idea de hacer que todo sea un objeto, sería deseable poder declarar un objeto como persistente y hacer que alguien o algo se encargue de todos los detalles, sin tener que hacerlo uno mismo.

Java proporciona soporte para "**persistencia ligera**", lo que significa que es posible **almacenar objetos** de manera sencilla en un disco para más tarde recuperarlos. La razón de que sea "**ligera**" es que es necesario hacer llamadas explícitas a este almacenamiento y recuperación. Además, los *JavaSpaces* proporcionan cierto tipo de almacenamiento persistente de los objetos. En alguna versión futura, podría aparecer un soporte completo para la persistencia.

Java e Internet

¿Qué es la Web?

Es necesario entender los sistemas cliente/servidor, otro elemento de la computación lleno de aspectos que causan también confusión.

Computación cliente/servidor

La idea principal de un sistema cliente/servidor es que se dispone de un depósito (**servidor**) central de información -cierto tipo de datos, generalmente en una base de datos- que se desea distribuir bajo demanda a cierto conjunto de máquinas o personas (**cliente**). Una clave para comprender el concepto de cliente/servidor es que el depósito de información está ubicado centralmente, de manera que puede ser modificado y de forma que los cambios se propaguen a los consumidores de la información. A la(s) máquina(s) en las que se ubican conjuntamente el depósito de información y el software que la distribuye se la denomina el servidor. El software que reside en la máquina remota se comunica con el *servidor*, toma la información, la procesa y después la muestra en la máquina remota, denominada el *cliente*.

El concepto básico de la computación cliente/servidor, por tanto, no es tan complicado. Aparecen problemas porque se tiene un único servidor que trata de dar servicio a múltiples clientes simultáneamente. Generalmente, está involucrado algún sistema gestor de base de datos de manera que el diseñador "reparte" la capa de datos entre distintas tablas para lograr un uso óptimo de los mismos. Además, estos sistemas suelen admitir que un cliente inserte nueva información en el servidor. Esto significa que es necesario asegurarse de que el nuevo dato de un cliente no machaque los nuevosdatos de otro cliente, o que no se pierda este dato en el proceso de su adición a la base de datos (procesamiento de la transacción). Al cambiar el software cliente, debe ser construido, depurado e instalado en las máquinas cliente, lo cual se vuelve bastante más complicado y caro de lo que pudiera parecer. Finalmente, hay un aspecto de rendimiento muy importante: es posible tener cientos de clientes haciendo peticiones simultáneas a un mismo servidor, de forma que un mínimo retraso sea crucial. Para minimizar la latencia, los programadores deben empeñarse a fondo para disminuir las cargas de las tareas en proceso, generalmente repartiéndolas con las máquinas cliente, pero en ocasiones, se dirige la carga hacia otras máquinas ubicadas junto con el servidor, denominadas intermediarios "*middleware*"(que también se utiliza para mejorar la *mantenibilidad* del sistema global).

La simple idea de distribuir la información a la gente, tiene muchas capas de complejidad en la fase de implementación, y el problema como un todo puede parecer desesperanzador. E incluso

puede ser crucial: la computación cliente/servidor se lleva prácticamente la mitad de todas las actividades de programación. Es responsable de todo, desde recibir las órdenes y transacciones de tarjetas de crédito hasta la distribución de cualquier tipo de datos -mercado de valores, datos científicos, delgobierno. El problema cliente/servidor completo debe resolverse con un enfoque global.

La Web como un servidor gigante

La Web es, de hecho, un sistema cliente/servidor gigante. Es un poco peor que eso, puesto que todos los servidores y clientes coexisten en una única red a la vez. No es necesario, sin embargo, ser conscientes de este hecho, puesto que simplemente es necesario preocuparse de saber cómo conectarse y cómo interactuar con un servidor en un momento dado.

Inicialmente, este proceso era unidireccional. Se hacía una petición de un servidor y éste te proporcionaba un archivo que el software navegador (por ejemplo, el cliente) de tu máquina podía interpretar dándole el formato adecuado en la máquina local. Pero en poco tiempo, la gente empezó a demandar más servicios que simplemente recibir páginas de un servidor. Se pedían capacidades cliente/servidor completas, de manera que el cliente pudiera retroalimentar de información al servidor; por ejemplo, para hacer búsquedas en base de datos en el servidor, añadir nueva información al mismo, o para ubicar una orden (lo que requería un nivel de seguridad mucho mayor que el que ofrecían los sistemas originales). Éstos son los cambios de los que hemos sido testigos a lo largo del desarrollo de la Web.

El navegador de la Web fue un gran paso hacia delante: el concepto de que un fragmento de información pudiera ser mostrado en cualquier tipo de computador sin necesidad de modificarlo. Sin embargo, los navegadores seguían siendo bastante primitivos y pronto se pasaron de moda debido a las demandas que se les hacían. No eran especialmente interactivos, y tendían a saturar tanto el servidor como Internet puesto que cada vez que requería hacer algo que exigiera programación había que enviar información de vuelta al servidor para que fuera procesada. Encontrar algo que por ejemplo, se había tecleado incorrectamente en una solicitud, podía llevar muchos minutos o segundos. Dado que el navegador era únicamente un visor no podía desempeñar ni siquiera las tareas de computación más simples. (Por otro lado, era seguro, puesto que no podía ejecutar programas en la máquina local que pudiera contener errores (*bugs*) o virus.)

Para resolver el problema, se han intentado distintos enfoques. El primero de ellos consistió en mejorar los estándares gráficos para permitir mejores animaciones y vídeos dentro de los navegadores. El resto del problema se puede resolver incorporando simplemente la capacidad de ejecutar programas en el cliente final, bajo el navegador. Esto se denomina programación en la parte cliente.

Programación en el lado del cliente

El diseño original servidor-navegador de la Web proporcionaba contenidos interactivos, pero la capacidad de interacción la proporcionaba completamente el servidor. Éste producía páginas estáticas para el navegador del cliente, que simplemente las interpretaba y visualizaba. El HTML básico contiene mecanismos simples para la recopilación de datos: cajas de entrada de textos, cajas de prueba, cajas de radio, listas y listas desplegables, además de un botón que sólo podía programarse para borrar los datos del formulario o "enviar" los datos del formulario de vuelta al servidor. Este envío de datos se lleva a cabo a través del *Common Gateway Interface* (CGI), proporcionado por todos los servidores web. El texto del envío transmite a CGI qué debe hacer con él. La acción más común es ejecutar un programa localizado en el servidor en un directorio denominado generalmente "cgi-bin". Estos programas pueden escribirse en la mayoría de los lenguajes

Muchos de los sitios web importantes de hoy en día se siguen construyendo estrictamente con CGI, y es posible, de hecho, hacer casi cualquier cosa con él. Sin embargo, los sitios web cuyo funcionamiento se basa en programas CGI se suelen volver difíciles de mantener, y presentan además problemas de tiempo de respuesta. (Además, poner en marcha programas CGI suele ser bastante lento.) Los diseñadores iniciales de la Web no previeron la rapidez con que se agotaría el ancho de banda para los tipos de aplicaciones que se desarrollaron. Por ejemplo, es imposible llevar a cabo cualquier tipo de generación dinámica de gráficos con consistencia, pues es necesario crear un archivo GIF que pueda ser después trasladado del servidor al cliente para cada versión del gráfico.

La solución es la programación en el lado del cliente. La mayoría de las máquinas que ejecutan navegadores Web son motores potentes capaces de llevar a cabo grandes cantidades de trabajo, y con el enfoque HTML estático original, simplemente estaban allí "sentadas", esperando ociosas a que el servidor se encargara de la página siguiente. La programación en el lado del cliente quiere decir que el servidor web tiene permiso para hacer cualquier trabajo del que sea capaz, y el resultado para el usuario es una experiencia mucho más rápida e interactiva en el sitio web.

La programación en el lado cliente no es muy distinta de la programación en general. Los parámetros son casi los mismos, pero la plataforma es distinta: un navegador web es como un sistema operativo limitado.

Conectables (plug-ins)

Uno de los mayores avances en la programación en la parte cliente es el desarrollo de los conectables (*plug-ins*). Éstos son modos en los que un programador puede añadir nueva funcionalidad al navegador descargando fragmentos de código que se conecta en el punto adecuado del navegador. Le dice al navegador "de ahora en adelante eres capaz de llevar a cabo esta nueva actividad". (Es necesario descargar cada conectable únicamente una vez.) A través de

los conectables, se añade comportamiento rápido y potente al navegador, pero la escritura de un conectable no es trivial y desde luego no es una parte deseable para hacer como parte de un proceso de construcción de un sitio web. Por consiguiente, los navegadores proporcionan una "puerta trasera que permite la creación de nuevos lenguajes de programación en el lado cliente.

Lenguajes de guiones (Scripting)

Los conectables condujeron a la explosión de los lenguajes de guiones (*scripting*). Con uno de estos lenguajes se integra el código fuente del programa de la parte cliente directamente en la página HTML, y el conectable que interpreta ese lenguaje se activa automáticamente a medida que se muestra la página HTML. Estos lenguajes tienden a ser bastante sencillos de entender y, dado que son simplemente texto que forma parte de una página HTML, se cargan muy rápidamente como parte del único acceso al servidor mediante el que se accede a esa página. El sacrificio es que todo el mundo puede ver (y robar) el código así transportado. Sin embargo, generalmente, si no se pretende hacer cosas excesivamente complicadas, los lenguajes de guiones constituyen una buena herramienta, al no ser complicados.

Esto muestra que los lenguajes de guiones utilizados dentro de los navegadores web se desarrollaron verdaderamente para resolver tipos de problemas específicos, en particular la creación de interfaces gráficos de usuario (IGUs) más interactivos y ricos. Sin embargo, uno de estos lenguajes puede resolver el 80% de los problemas que se presentan en la programación en el lado cliente. Este 80% podría además abarcar todos los problemas de muchos programadores, y dado que los lenguajes de programación permiten desarrollos mucho más sencillos y rápidos, es útil pensar en utilizar uno de estos lenguajes antes de buscar soluciones más complicadas, como la programación en Java o ActiveX.

Los lenguajes de guiones de navegador más comunes son JavaScript (que no tiene nada que ver con Java; se denominó así simplemente para aprovechar el buen momento de marketing de Java), VBScript (que se parece bastante a Visual Basic), y Tcl/Tk, que proviene del popular lenguaje de construcción de IGU (Interfaz Gráfica de Usuario) de plataformas cruzadas. Hay otros más, y seguro que se desarrollarán muchos más.

JavaScript es probablemente el que recibe más apoyo. Viene incorporado tanto en el navegador Netscape Navigator como en el Microsoft Internet Explorer (IE). Además, hay probablemente más libros de JavaScritpt que de otros lenguajes de navegador, y algunas herramientas crean páginas automáticamente haciendo uso de JavaScript. Sin embargo, si se tiene habilidad en el manejo de Visual Basic o Tcl/Tk, será más productivo hacer uso de esos lenguajes de guiones en vez de aprender uno nuevo.

Java

Si un lenguaje de programación puede resolver el 80 por ciento de los problemas de programación en el lado cliente, ¿qué ocurre con el 20 por ciento restante? La solución es Java. Este proporciona aspectos de lenguaje y bibliotecas que manejan de manera elegante problemas que son complicados para los lenguajes de programación tradicionales, como la ejecución multihilo, el acceso a basede datos, la programación en red, y la computación distribuida. Java permite programación en el lado cliente a través del *applet*.

Un *applet* es un miniprograma que se ejecutará únicamente bajo un navegador web. El *applet* se descarga automáticamente como parte de una página web. Cuando se activa un *applet*, ejecuta un programa que proporciona una manera de distribuir automáticamente software cliente desde el servidor justo cuando el usuario necesita software cliente, y no antes. El usuario se hace con la última versión del software cliente, sin posibilidad de fallo, y sin tener que llevar a cabo reinstalaciones complicadas. Gracias a cómo se ha diseñado Java, el programado simplemente tieneque crear un único programa, y este programa trabaja automáticamente en todos los computadores que tengan navegadores que incluyan intérpretes de Java.

No sólo se consigue un incremento de velocidad y capacidad de respuesta inmediatas, sino que el tráfico en general de la red y la carga en los servidores se reduce considerablemente, evitando que toda Internet se vaya ralentizando.

Una ventaja que tienen los *applets* de Java sobre los lenguajes de guiones es que se encuentra en formato compilado, de forma que el código fuente no está disponible para el cliente.

ActiveX

El competidor principal de Java es el ActiveX de Microsoft.Este era originalmente una solución válida exclusivamente para Windows, aunque ahora se está desarrollando mediante un consorcio independiente de manera que acabará siendo multiplataforma. Por consiguiente, ActiveX no se limita a un lenguaje particular. Si, por ejemplo, uno es un programador Windows experimentado, haciendo uso de un lenguaje como C++, Visual Basic o en Delphi de Borland, es posible crear componentes ActiveX sin casi tener que hacer ningún cambio a los conocimientos de programación que ya se tengan.

Seguridad

La capacidad para descargar y ejecutar programas a través de Internet puede parecer el sueño de un constructor de virus. Si se hace clic en el sitio web, es posible descargar automáticamente cualquier número de cosas junto con la página HTML. Java también fue diseñado para ejecutar sus applets dentro de un "envoltorio" de seguridad, lo que evita que escriba en el disco o acceda a la memoria externa a ese envoltorio.

Programar con ActiveX es como programar Windows -es posible hacer cualquier cosa. De esta manera, si se hace *clic* en una página web que descarga un componente ActiveX, ese componente podría llegar a dañar los archivos de un disco.

La solución parece aportarla las "firmas digitales", que permiten la verificación de la autoría del código. Este sistema se basa en la idea de que un virus funciona porque su creador puede ser anónimo, de manera que si se evita la ejecución de programas anónimos, se obligará a cada persona a ser responsable de sus actos. Esto parece una buena idea, sin embargo, si un programa tiene un error (bug) inintencionadamente destructivo, seguirá causando problemas.

El enfoque de Java es prevenir que estos problemas ocurran, a través del envoltorio. El intérprete de Java que reside en el navegador web local examina el *applet* buscando instrucciones adversas a medida que se carga el *applet*. Más en concreto, el *applet* no puede escribir ficheros en el disco o borrar ficheros (una de las principales vías de ataque de los virus). Los *applets* se consideran generalmente seguros, y dado que esto es esencial para lograr sistemas cliente/servidor de confianza, cualquier error (bug) que produzca virus en lenguaje Java será rápidamente reparado. (Aparteel navegador refuerza estas restricciones de seguridad).

Uno puede desear construir una base de datos o almacenar datos para utilizarlos posteriormente, finalizada la conexión. La solución es el "*applet* firmado" que utiliza cifrado de clave pública para verificar que un *applet* viene efectivamente de donde dice venir. Un *applet* firmado puede seguir destrozando un disco local, pero la teoría es que dado que ahora es posible localizar al creador del *applet*, éstos no actuarán de manera perniciosa. Java proporciona un marco de trabajo para las firmas digitales, de forma que será posible permitir que un *applet* llegue a salir fuera del envoltorio si es necesario.

Internet frente a Intranet

La Web es la solución más general al problema cliente/servidor, de forma que tiene sentido que se pueda utilizar la misma tecnología para resolver un subconjunto del problema, cliente/servidor *dentro* de una compañía. Cuando se utiliza tecnología web para una red de información restringida a una compañía en particular, se la denomina una Intranet. Las Intranets proporcionan un nivel de seguridad mucho mayor que el de Internet, puesto que se puede controlar físicamente el acceso a los servidores dentro de la propia compañía.

Si se está ejecutando código en una Intranet, es posible tener un conjunto de limitaciones distinto. No es extraño que las máquinas de una red puedan ser todas plataformas Intel/Windows. En una Intranet, uno es responsable de la calidad de su propio código y puede reparar errores en el momento en que se descubren. El tiempo malgastado en instalar actualizaciones (*upgrades*) es la razón más apabullante para comenzar a usar navegadores, en los que estas actualizaciones son invisibles y automáticas.

Programación en el lado del servidor

¿Qué ocurre cuando se hace una petición a un servidor? La mayoría de las veces la petición es simplemente "envíame este archivo". A continuación, el navegador interpreta el archivo de la manera adecuada: como una página HTML, como una imagen gráfica, un *applet* Java, un programa de guiones, etc. Una petición más complicada hecha a un servidor puede involucrar una transacción de base de datos. Un escenario común involucra una petición para una búsqueda compleja en una base de datos, que el servidor formatea en una página HTML para enviarla a modo de resultado. Un usuario también podría querer registrar su nombre en una base de datos al incorporarse a un grupo o presentar unaorden, lo cual implica cambios a esa base de datos. Estas peticiones deben procesarse vía algún código en el lado servidor, que se denomina generalmente programación en el lado servidor.

Entre éstos se encuentran los servidores web basados en Java que permiten llevar a cabo toda la programación del lado servidor en Java escribiendo lo que se denominan *servlets*. Éstos y sus descendientes, los JSP, son las dos razones principales por las que las compañías que desarrollan sitios web se están pasando a Java, especialmente porque eliminan los problemas de tener que tratar con navegadores de distintas características.

Análisis y diseño

El paradigma de la orientación a objetos es una nueva manera de enfocar la programación. Son muchos los que tienen problemas a primera vista para enfrentarse a un proyecto de POO. A medida que se aprende a pensar de forma orientada a objetos, es posible empezar a crear "buenos" diseños y sacar ventaja de todos los beneficios que la PO0 puede ofrecer.

Una *metodología* es un conjunto de procesos y heurísticas utilizadas para descomponer la complejidad de un problema de programación. Especialmente en la POO, la metodología es un área de intensa experimentación, por lo que es importante entender qué problema está intentando resolver el método antes de considerar la adopción de uno de ellos.

Se haga lo que se haga al diseñar y escribir un programa, se sigue un método. Puede que sea el método propio de uno, e incluso puede que uno no sea consciente de utilizarlo, pero es un proceso que se sigue al crear un programa. Mientras se está en el propio proceso de desarrollo, el aspecto más importante es no perderse. Algunos tipos de procesos, sin importar lo limitados que puedan ser, le permitirán encontrar el camino de manera más sencilla que si simplemente se empieza a codificar.

También es fácil desesperarse, caer en "parálisis de análisis", cuando se siente que no se puede avanzar porque no se han cubierto todos los detalles en la etapa actual. Debe recordarse que, independientemente de cuánto análisis lleve a cabo, hay cosas de un sistema que no aparecerán hasta la fase de diseño, y otras no aflorarán incluso hasta la fase de codificación o en un extremo,

hasta que el programa esté acabado y en ejecución. Debido a esto, es crucial moverse lo suficientemente rápido a través de las etapas de análisis y diseño e implementar un prototipo del sistema propuesto.

La clase de problema de programación examinada en el presente capítulo es un "juego de azar". Intentar analizar completamente un problema al azar antes de pasar al diseño e implementación conduce a una parálisis en el análisis, al no tener suficiente información para resolver este tipo de problemas durante la fase de análisis. A menudo, se propone "construir uno para desecharlo". Con PO0 es posible tirar parte, pero dado que el código está encapsulado en clases, durante la primera pasada siempre se producirá algún diseño de clases útil y se desarrollarán ideas que merezcan la pena para el diseño del sistema de las que no habrá que deshacerse. Por tanto, la primera pasada rápida por un problema no sólo suministra información crítica para las ulteriores pasadas por análisis, diseño e implementación, sino que también crea la base del código.

Si se está buscando una metodología se recomienda preguntarse acerca de lo que se está intentando descubrir.

1. ¿Cuáles son los objetos? (¿Cómo se descompone su proyecto en sus componentes?)

2. ¿Cuáles son las interfaces? (¿Qué mensajes es necesario enviar a cada objeto?)

Si se delimitan los objetos y sus interfaces, ya es posible escribir un programa. El proceso puede descomponerse en cinco fases, y la Fase O no es más que la adopción de un compromiso para utilizar algún tipo de estructura.

Fase 0: Elaborar un plan

En primer lugar, debe decidirse qué pasos debe haber en un determinado proceso. Suena bastante simple y la gente no obstante, suele seguir sin tomar esta decisión antes de empezar a codificar. Si el plan consiste en "empecemos codificando", entonces, perfecto (en ocasiones, esto es apropiado). Al menos, hay que estar de acuerdo en que eso también es tener un plan.

También podría decidirse en esta fase que es necesaria alguna estructura adicional de proceso, pero no toda una metodología completa. A algunos programadores les gustatrabajar en "modo vacación", en el que no se imponga ninguna estructura en el proceso de desarrollar de su trabajo; "se hará cuando se haga". Esto puede resultar atractivo a primera vista, pero a medida que se tiene algo de experiencia uno se da cuenta de que es mejor ordenar y distribuir el esfuerzo en distintas etapas en vez de lanzarse directamente a "finalizar el proyecto". Además, de esta manera se divide el proyecto en fragmentos más asequibles, y se resta miedo a la tarea de enfrentarse al mismo.

Incluso cuando uno piensa que el plan consiste simplemente en empezar a codificar, todavía se atraviesan algunas fases al plantear y contestar ciertas preguntas.

El enunciado de la misión

Cualquier sistema que uno construya, tiene un propósito fundamental: La necesidad básica que cumple. Esto es denominado *alto concepto* y uno puede describir el propósito de un programa en dos o tres fases. Esta descripción, pura, es el punto de partida.

El alto concepto es bastante importante porque establece el tono del proyecto; es el enunciado de su misión. Por ejemplo, en un sistema de control de tráfico aéreo, uno puede comenzar con un alto concepto centrado en el sistema que se está construyendo: "El programa de la torre hace un seguimiento del avión". Pero considérese qué ocurre cuando se introduce el sistema en un pequeño aeródromo; quizás sólo hay un controlador humano, o incluso ninguno. Un modelo más usual no abordará la solución que se está creando como describe el problema: "Los aviones llegan, descargan, son mantenidos y recargan, a continuación, salen".

Fase 1: ¿Qué estamos construyendo?

En la generación previa del diseño del programa (denominada diseño procedural) a esta fase se le de nominaba "creación del *análisis de requisitos* y *especificación del sistema*". Éstas, por supuesto, eran fases en las que uno se perdía; documentos con nombres intimidadores que podían de por sí convertirse en grandes proyectos. Sin embargo, su intención era buena. El análisis de requisitos dice: "Construya una lista de directrices que se utilizarán para saber cuándo se ha acabado el trabajo y cuándo el cliente está satisfecho". La especificación del sistema dice: "He aquí una descripción de lo que el programa hará (pero *no cómo*) para satisfacer los requisitos hallados". El análisis de requisitos es verdaderamente un contrato entre usted y el cliente. La especificación del sistema es una exploración de alto nivel en el problema, y en cierta medida, un descubrimiento de si puede hacerse y cuánto tiempo llevará. Se podría tener otras limitaciones que exijan expandirlos en documentos de mayor tamaño, pero si se mantiene que el documento inicial sea pequeño y conciso, es posible crearlo en unas pocas sesiones de tormenta de ideas (*brainstorming*) en grupo, con un líder que dinámicamente va creando la descripción. Este proceso no sólo exige que todos aporten sus ideas sino que fomenta el que todos los miembros del equipo lleguen a un acuerdo inicial. Quizás lo más importante es que puede incluso ayudar a que se acometa el proyecto con una gran dosis de entusiasmo.

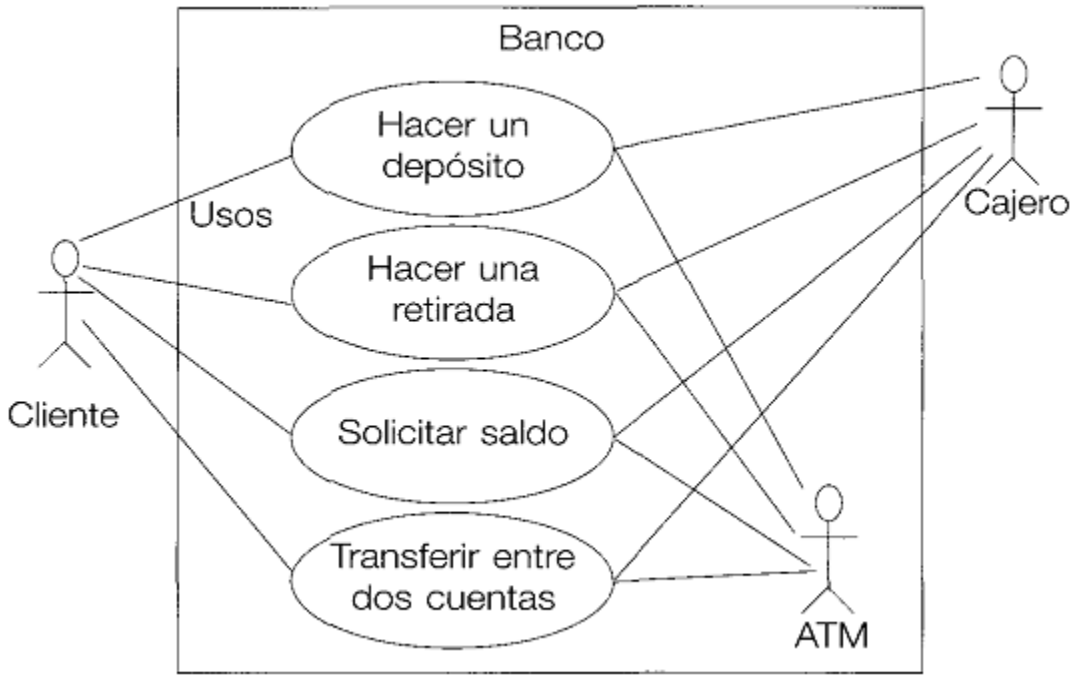
Es necesario mantenerse centrado en el corazón de lo que se está intentando acometer en esta fase: determinar qué es lo que se supone que debe hacer el sistema. La herramienta más valiosa para esto es una colección de lo que se denomina "casos de uso". Los casos de uso identifican los aspectos claves del sistema, que acabarán por revelar las clases fundamentales que se usarán en éste. De hecho, los casos de uso son esencialmente soluciones descriptivas a preguntas como1":

- “¿Quién usará el sistema?”
- “¿Qué pueden hacer esos actores con el sistema?”

- “¿Cómo se las ingenia cada actor para hacer eso con este sistema?"
- "¿De qué otra forma podría funcionar esto si alguien más lo estuviera haciendo, o si el mismo actor tuviera un objetivo distinto?" (Para encontrar posibles variaciones.)
- “¿Qué problemas podrían surgir mientras se hace esto con el sistema?" (Para localizar posibles excepciones.)

Si se está diseñando, por ejemplo, un cajero automático, el caso de uso para un aspecto particular de la funcionalidad del sistema debe ser capaz de describir qué hace el cajero en cada situación posible. Cada una de estas "situaciones" se denomina un *escenario*, y un caso de uso puede considerarse como una colección de escenarios. Uno puede pensar que un escenario es como una pregunta que empieza por: “¿Qué hace el sistema si...?". Por ejemplo: “¿Qué hace el cajero si un cliente acaba de depositar durante las últimas 24 horas un cheque y no hay dinero suficiente en la cuenta, sin haber procesado el cheque, para proporcionarle la retirada el efectivo que ha solicitado?"

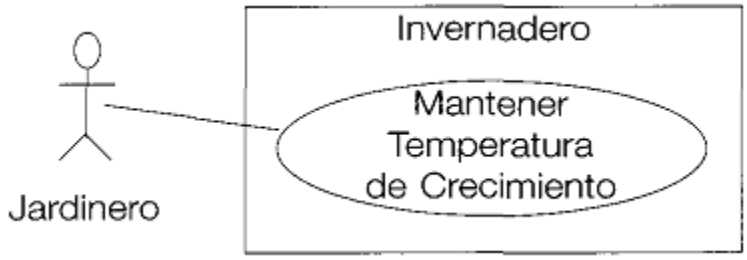
Deben utilizarse diagramas de caso de uso intencionadamente simples para evitar ahogarse prematuramente en detalles de implementación del sistema:



Cada uno de los monigotes representa a un "actor", que suele ser generalmente un humano o cualquier otro tipo de agente (por ejemplo, otro sistema de computación, como "ATM"). La caja representa los límites de nuestro sistema. Las elipses representan los casos de uso, que son

descripciones del trabajo útil que puede hacerse dentro del sistema. Las líneas entre los actores y los casos de uso representan las interacciones.

Un caso de uso no tiene por qué ser terriblemente complejo, aunque el sistema subyacente sea complejo. Solamente se pretende que muestre el sistema tal y como éste se muestra al usuario. Por ejemplo:



Los casos de uso proporcionan las especificaciones de requisitos determinando todas las interacciones que el usuario podría tener con el sistema. Se trata de descubrir un conjunto completo de casos de uso para su sistema, y una vez hecho esto, se tiene el núcleo de lo que el sistema se supone que hará.Sise tiene un conjunto completo de casos de uso, es posible describir el sistema y pasar a la siguiente fase.

Cuando uno se quede bloqueado, es posible comenzar esta fase utilizando una extensa herramienta de aproximación: describir el sistema en unos pocos párrafos y después localizar los sustantivosy los verbos. Los sustantivos pueden sugerir quiénes son los actores, el contexto del caso de uso(por ejemplo, "*corredor*"), o artefactos manipulados en el caso de uso. Los verbos pueden sugerir interacciones entre los actores y los casos de uso, y especificar los pasos dentro del caso de uso. También será posible descubrir que los sustantivos y los verbos producen objetos y mensajes durante la fase de diseño.

Ahora se tiene una visión de lo que se está construyendo, por lo que probablemente se pueda tener una idea de cuánto tiempo le llevará.

Debería empezarse por una primera estimación del tiempo que llevaría, para posteriormente multiplicarla por dos y añadirle el 10 por ciento. Aunque si se desea tener suficiente tiempo como para lograr un producto verdaderamente elegante y disfrutar durante el proceso, el multiplicador correcto puede ser por tres o por cuatro.

Fase 2: ¿Cómo construirlo?

En esta fase debe lograrse un diseño que describe cómo son las clases y cómo interactuarán. Una técnica excelente para determinar las clases e interacciones es la tarjeta *Clase-Responsabilidad-Colaboración* (**CRC**). Parte del valor de esta herramienta se basa en que es de muy baja tecnología: se comienza con un conjunto de tarjetas de 3 x 5, y se escribe en ellas. Cada tarjeta representa una única clase, y en ella se escribe:

1. El nombre de la clase. Es importante que este nombre capture la esencia de lo que hace la clase, de manera que tenga sentido a primera vista.

2. Las "responsabilidades" de la clase: qué debería hacer. Esto puede resumirse típicamente escribiendo simplemente los nombres de las funciones miembros (dado que esas funciones deberían ser descriptivas en un buen diseño), pero no excluye otras anotaciones.

3. Las "colaboraciones" de la clase: ¿con qué otras clases interactúa? "Interactuar" es un término amplio intencionadamente; vendría a significar agregación, o simplemente que cualquier otro objeto existente ejecutara servicios para un objeto de la clase. Las colaboraciones deberían considerar también la audiencia de esa clase. Por ejemplo, si se crea una clase Petardo, ¿quién la va a observar, un **Químico** o un **Observador**? En el primer caso estamos hablando de punto de vista del químico que va a construirlo, mientras que en el segundo se hace referencia a los colores y las formas que libere al explotar.

Uno puede pensar que las tarjetas deberían ser más grandes para que cupiera en ellas toda la información que se deseara escribir, pero son pequeñas a propósito, no sólo para mantener pequeño el tamaño de las clases, sino también para evitar que se caiga en demasiado nivel de detalle muy pronto. La clase ideal debería ser comprensible a primera vista. La idea de las tarjetas CRC es ayudar a obtener un primer diseño de manera que se tenga un dibujo a grandes rasgos que pueda ser después refinado.

Una de las mayores ventajas de las tarjetas CRC se logra en la comunicación. Cuando mejor se hace es en tiempo real, en grupo y sin computadores. Cada persona se considera responsable de varias clases (que al principio no tienen ni nombres ni otras informaciones). Se ejecuta una simulación en directo resolviendo cada vez un escenario, decidiendo qué mensajes se mandan a los distintos objetos para satisfacer cada escenario. A medida que se averiguan las responsabilidades y colaboraciones de cada una, se van rellenando las tarjetas correspondientes. Cuando se han recorrido todos los casos de uso, se debería tener un diseño bastante completo.

Una vez que se tiene un conjunto de tarjetas CRC se desea crear una descripción más formal del diseño haciendo uso de UML. No es necesario utilizar UML, pero puede ser de gran ayuda, especialmente si se desea poner un diagrama en la pared para que todo el mundo pueda ponderarlo, lo cual es una gran idea. Una alternativa a UML es una descripción textual de los objetos y sus interfaces, o, dependiendo del lenguaje de programación, el propio código.

También puede ser necesario describir las estructuras de datos, en sistemas o subsistemas en los que los datos sean un factor dominante (como una base de datos).

Sabemos que la Fase 2 ha acabado cuando se han descrito los objetos y sus interfaces.

Las cinco etapas del diseño de un objeto

El diseño de un objeto conlleva una serie de etapas. Es útil tener esta perspectiva porque se deja de esperar la perfección; por el contrario, uno comprende lo que hace un objeto y el nombre que debería tener surge con el tiempo. El patrón para un tipo de programa particular emerge al enfrentarse una y otra vez con el problema. Los objetos, también tienen su patrón, que emerge a través de su entendimiento, uso y reutilización.

1. **Descubrimiento de los objetos**. Esta etapa ocurre durante el análisis inicial del programa. Se descubren los objetos al buscar factores externos y limitaciones, elementos duplicados en el sistema, y las unidades conceptuales más pequeñas. Algunos objetos son obvios si ya se tiene un conjunto de bibliotecas de clases. La comunidad entre clases que sugieren clases bases y herencia, puede aparecer también en este momento, o más tarde dentro del proceso de diseño.

2. **Ensamblaje de objetos.** Al construir un objeto se descubre la necesidad de nuevos miembros que no aparecieron durante el descubrimiento. Las necesidades internas del objeto pueden requerir de otras clases que lo soporten.

3. **Construcción del sistema.** De nuevo, pueden aparecer en esta etapa más tardía nuevos requisitos para el objeto. Así se aprende que los objetos van evolucionando. La necesidad de un objeto de comunicarse e interconectarse con otros del sistema puede hacer que las necesidades de las clases existentes cambien, e incluso hacer necesarias nuevas clases.

4. **Aplicación del sistema.** A medida que se añaden nuevos aspectos al sistema, puede que se descubra que el diseño previo no soporta una ampliación sencilla del sistema. Con esta nueva información, puede ser necesario reestructurar partes del sistema, generalmente añadiendo nuevas clases o nuevas jerarquías de clases.

5. **Reutilización de objetos.** Ésta es la verdadera prueba de diseño para una clase. Si alguien trata de reutilizarla en una situación completamente nueva, puede que descubra pequeños inconvenientes. Al cambiar una clase para adaptarla a más programas nuevos, los principios generales de la clase se mostrarán más claros, hasta tener un tipo verdaderamente reutilizable. Sin embargo, no debe esperarse que la mayoría de objetos en un sistema se diseñen para ser reutilizados.

Guías para el desarrollo de objetos

1. Debe permitirse que un problema específico genere una clase, y después dejar que la clase crezca y madure durante la solución de otros problemas.

2. Debe recordarse que descubrir las clases (y SUS interfaces) que uno necesita es la tarea principal del diseño del sistema. Si ya se disponía de esas clases, el proyecto será fácil.

3. No hay que forzar a nadie a saber todo desde el principio; se aprende sobre la marcha.Y esto ocurrirá poco a poco.

4. Hay que empezar programando; es bueno lograr algo que funcione de manera que se pueda probar la validez o no de un diseño. No hay que tener miedo a acabar con un código de estilo procedimental malo -las clases segmentan el problema y ayudan a controlar la anarquía y la entropía. Las clases malas no estropean las clases buenas.

5. Hay que mantener todo lo más simple posible. Los objetos pequeños y limpios con utilidad obvia son mucho mejores que interfaces grandes y complicadas. Hay que empezar con algo pequeño y sencillo.

Fase 3: Construir el núcleo

Ésta es la conversión inicial de diseño pobre en un código compilable y ejecutable que pueda ser probado, y especialmente, que pueda probar la validez o no de la arquitectura diseñada. Este proceso no se puede hacer de una pasada, sino que consistirá más bien en una serie de pasos que permitirán construir el sistema de manera iterativa.

Su objetivo es encontrar el núcleo de la arquitectura del sistema que necesita implementar para generar un sistema ejecutable, sin que importe lo incompleto que pueda estar este sistema en esta fase inicial. Está creando un *armazón* sobre el que construir en posteriores iteraciones. También se está llevando a cabo la primera de las muchas integraciones y pruebas del sistema, a la vez que proporcionando a los usuarios una realimentación sobre la apariencia que tendrá su sistema, y cómo va progresando. Se descubrirán posibles cambios y mejoras que se pueden hacer sobre el diseño original -cosas que no se hubieran descubierto de no haber implementado el sistema.

Una parte de la construcción del sistema es comprobar que realmente se cumple el análisis de requisitos y la especificación del sistema. Debe asegurarse que las pruebas verifican los requerimientos y los casos de uso. Cuando el corazón del sistema sea estable, será posible pasar a la siguiente fase y añadir nuevas funcionalidades.

Fase 4: Iterar los casos de uso

Una vez que el núcleo del sistema está en ejecución, cada característica que se añada es en sí misma un pequeño proyecto. Durante cada *iteración*, entendida como un periodo de desarrollo razonablemente pequeño, se añade un conjunto de características.

Cada iteración dura de una a tres semanas. Al final de ese periodo, se tiene un sistema integrado y probado con una funcionalidad mayor a la que tenía previamente. Pero lo particularmente interesante es la base de la iteración: un único caso de uso. Éste es una unidad de desarrollo fundamental a lo largo de todo el proceso de construcción de software.

Se deja de iterar al lograr la funcionalidad objetivo, o si llega un plazo y el cliente se encuentra satisfecho con la versión actual (debe recordarse que el software es un negocio de suscripción). Dado que el proceso es iterativo, uno puede tener muchas oportunidades de lanzar un producto, más que tener un único punto final; los proyectos abiertos trabajan exclusivamente en entornos iterativos de gran nivel de realimentación, que es precisamente lo que les permite acabar con éxito.

Fase 5: Evolución

Denominado tradicionalmente "mantenimiento", que quiere decir cualquier cosa, desde "hacer que funcione de la manera que se suponía que lo haría en primer lugar", hasta "añadir aspectos varios que el cliente olvidó mencionar", pasando por el tradicional "arreglar los errores que puedan aparecer" o "la adición de nuevas características a medida que aparecen nuevas necesidades".Quizás haya un término mejor para describir lo que está pasando, como por ejemplo el término ***evolución***. "Uno no acierta a la primera, por lo que debe concederse la libertad de aprender y volver a hacer nuevos cambios". A corto y largo plazo, será el propio programa el que se verá beneficiado de este proceso continuo deevolución. De hecho, ésta permitirá que el programa pase de bueno a genial, haciendo que se aclaren aquellos aspectos que no fueron verdaderamente entendidos en la primera pasada. También es en este proceso en el que las clases se convierten en recursos reutilizables, en vez de clases diseñadas para su uso en un solo proyecto.

El soporte a la evolución podría ser el beneficio más importante de la POO. Con la evolución, se crea algo que al menos se aproxima a lo que se piensa que se está construyendo, se compara con los requisitos, y se ve dónde se ha quedado corto. Después, se puede volver y ajustarlo diseñando y volviendo a implementar las porciones del programa que no funcionaron correctamente.

La evolución también se da al construir un sistema, ver que éste se corresponda con los requisitos, y descubrir después que no era, de hecho, lo que se pretendía. Al ver un sistema en funcionamiento, se puede descubrir que verdaderamente se pretendía que solucionase otro problema. Si uno espera que se dé este tipo de evolución, entonces se debe construir la primera versión lo más rápidamente posible con el propósito de averiguar sin lugar a dudas qué es exactamente lo que se desea.

Quizás lo más importante que se ha de recordar es que por defecto, si se modifica una clase, sus súper y subclases seguirán funcionando. Uno no debe tener miedo a la modificación. Los cambios no tienen por qué estropear el programa.

Programación extrema (Extreme Programming, XP)

XP es tanto una filosofía del trabajo de programación como un conjunto de guías para acometer esta tarea. Las dos contribuciones más distintivas e importantes en mi opinión son "escribir las pruebas en primer lugar" y "la programación a pares". Si se adoptan únicamente estas dos prácticas, uno mejorará enormemente su productividad y nivel de confianza.

Escritura de las pruebas en primer lugar

El proceso de prueba casi siempre ha quedado relegado al final de un proyecto. Implícitamente, tenía una prioridad bastante baja, y la gente que se especializa en las pruebas nunca ha gozado de un gran estatus, e incluso suele estar ubicada en el sótano, lejos de los "programadores de verdad". XP revoluciona completamente el concepto de prueba dándole una prioridad igual (o incluso mayor) que a la codificación. De hecho, se escriben los tests*antes* de escribir el código a probar, y los códigos se mantienen para siempre junto con su código destino. Es necesario ejecutar con éxito los tests cada vez que se lleva a cabo un proceso de integración del proyecto.

Al principio la escritura de las pruebas tiene dos efectos extremadamente importantes.

El primero es que fuerza una definición clara de la interfaz de cada clase. La estrategia de pruebas XP especifica exactamente qué apariencia debe tener la clase para el consumidor de la clase, y cómo ésta debe comportarse exactamente. No puede haber nada sin concretar. Nada es igual que un conjunto de pruebas. Lo primero es una lista de deseos,pero las pruebas son un contrato reforzado por el compilador y el programa en ejecución.

Al crear los tests, uno se ve forzado a pensar completamente en la clase, y a menudo, descubre la funcionalidad deseada que podría haber quedado en el tintero durante las experiencias de pensamiento de los diagramas XML, las tarjetas CRC, los casos de uso, etc.

El segundo efecto importante de escribir las pruebas en primer lugar, proviene de la ejecución de las pruebas cada vez que se construye un producto software. Esta actividad proporciona la otra mitad de las pruebas que lleva a cabo el compilador.

Las pruebas integradas permitidas por el diseño del lenguaje no pueden ir mucho más allá. En cierto punto, *cada uno* debe continuar y añadir el resto de pruebas que producen una batería de pruebas completa que verifique todo el programa. Y, exactamente igual que si se dispusiera de un compilador observando por encima del hombro, ¿no desearía uno que estas pruebas le ayudasen a hacer todo biendesde el principio? Por eso es necesario escribir las pruebas en primer lugar y ejecutarlas cada vez que se reconstruya el sistema. Las pruebas se convierten en una extensión de la red de seguridad proporcionada por el lenguaje.

Programación a pares

Los programadores suelen considerarse abanderados de la individualidad y por el contrario, XP, que trata, de por sí, de luchar contra el pensamiento convencional, enuncia lo contrario, afirmando que el código debería siempre escribirse entre dos personas por cada estación de trabajo. Y esto debería hacerse en áreas en las que haya grupos de estaciones de trabajo, sin las barreras de las que la gente de facilidades de diseño suelen estar tan orgullosos. La primera tarea para convertirse a XP es aparecer con destornilladores y llaves Allen y desmontar todo aquello que parezca imponer barreras o separaciones.

El valor de la programación en pareja es que una persona puede estar, de hecho, codificando mientras la otra piensa en lo que se está haciendo. El pensador es el que tiene en la cabeza todo el esbozo. Y si el codificador se queda clavado, pueden cambiar de sitio. Si los dos se quedan parados, puede que alguien más del área de trabajo pueda contribuir al oír sus meditaciones. Trabajar a pares hace que todo fluya mejor y a tiempo. Y lo que probablemente es más importante: convierte la programación en una tarea mucho más divertida y social.

Por qué Java tiene éxito

El objetivo de Java es mejorar la productividad y fue diseñado para ser práctico; las decisiones de diseño del lenguaje Java se basaban en proporcionar al programador la mayor cantidad de beneficios posibles.

Los sistemas son más fáciles de expresar y entender

“Pon el bit en el chip que indica que el relé se va cerrar". Uno maneja conceptos de alto nivel y puede hacer mucho más con una única línea de código.

El otro beneficio del uso de esta expresión es la mantenibilidad. Si un programa es fácil de entender, entonces es fácil de mantener.

Ventajas máximas con las bibliotecas

La manera más rápida de crear un programa es utilizar código que ya esté escrito: una biblioteca. Uno de los principales objetivos de Java es facilitar el uso de bibliotecas. Dado que el compilador de Java se encarga del buen uso de las bibliotecas uno puede centrarse en lo que desea que haga la biblioteca en vez de como tiene que hacerlo.

Manejo de errores

Si se está construyendo un programa grande y complejo, no hay nada peor que tener un error enterrado en algún sitio sin tener ni siquiera una pista de dónde puede estar. El manejo de excepciones de Java es una forma de garantizar que se notifiquen los errores, y que todo ocurre como consecuencia de algo.

Programación a lo grande

Java está diseñado para ayudar a *programar a lo grande* -es decir, para borrar esos límites de complejidad entre un programa pequeño y uno grande. Uno no tiene por qué usar PO0 al escribir un programa de utilidad del estilo de "iHola, mundo!", pero estas características siempre están ahí cuando son necesarias. Y el compilador se muestra agresivo a la hora de descubrir las causas generadora de errores, tanto tanto el caso de programas grandes, como pequeños.