Paradigma de Objetos

Cátedra de Paradigmas de Programación
Facultad Regional Buenos Aires
Universidad Tecnológica Nacional

Ing. Lucas Spigariol

Buenos Aires – 2008

Presentación

¿Por qué objetos?

En el estado actual de las ciencias de la computación y más precisamente en el ámbito del desarrollo de sistemas es innegable el papel preponderante que asume el Paradigma de Objetos.

El mundo cambia a un ritmo acelerado, las sociedades se transforman y en medio de la vorágine, quienes se adaptan mejor a los escenarios cambiantes pueden mantenerse con firmeza e incluso crecer. No se trata de plantear una apología de la "supervivencia del más apto" desde una concepción evolucionista lineal porque el panorama es mucho más complejo y con múltiples variables a analizar, pero el paradigma de objetos, al plantearse como unos de sus postulados fundamentales la reusabilidad y extensibilidad de sus soluciones, encuentra en este aparentemente difícil contexto el ámbito apropiado para su crecimiento y aceptación creciente. Otros enfoques de la programación originalmente valorados por su solidez y confiabilidad se encuentran muchas veces limitados a la hora de responder ágilmente a los nuevos desafíos que se plantean en la actualidad.

El objetivo de este trabajo es presentar de una manera simple y clara las principales características del Paradigma de Objetos, como modesta introducción para quienes quieran incursionar por el mundo de los objetos. Lejos de ser una tarea ciclópea e ingrata, como afirman los profetas de la inercia escéptica, aprender a desarrollar un sistema y, más aún, poder "pensarlo" desde esta perspectiva se convierte en una muy interesante y productiva tarea, que aunque no todos coincidan con que es "apasionante", como afirmamos quienes desde hace años nos movemos en este ámbito, no podrán negar que redunda en amplios beneficios de productividad.

Dentro de los numerosos lenguajes de programación orientada a objetos, para ilustrar y ejemplificar las explicaciones hemos elegido al iniciador y más emblemático de ellos, el Smalltalk. La claridad con que permite mostrar los conceptos, la coherencia de su esquema interno y la "pureza" y "fidelidad" a los elementos principales del paradigma motivan la opción, junto con la existencia y facilidad de acceso a distintos ambientes de desarrollo y una actualización permanente de versiones.

En los primeros capítulos se desarrollan los principales conceptos de objetos, con algunos ejemplos.

En los siguientes, se presenta en detalle las características del lenguaje de programación con una recorrida por las principales clases y sus métodos.

Por último, un anexo con una serie de ejercicios resueltos a modo de ejemplos integradores donde se aplican los conceptos del paradigma, codificados en Smalltalk.

El presente material es una actualización de otros textos que anteriormente había elaborado, con nuevos ejemplos, ampliaciones conceptuales y ejemplos renovados, elaborado en base a mi experiencia de 15 años como docente en la carrera de Ingeniería en Sistemas de Información, en la Facultad Regional Buenos Aires de la Universidad Tecnológica Nacional, y en particular, en la cátedra de Paradigmas de Programación, desde su creación en el marco del plan de estudios de 1995.

El objetivo principal de este trabajo es que en sus manos se convierta en una herramienta de utilidad para su formación permanente y su ejercicio profesional.

Ing. Lucas Spigariol

Capítulo 1

Conceptos generales

- **El paradigma de Objetos**
- **Principales características**
- **Historia**
- **Lenguajes**
- **Smalltalk**

El paradigma de Objetos

El paradigma de objetos, o como se lo conoce generalmente, la Programación Orientada a Objetos, se fundamenta en concebir a un sistema como un conjunto de entidades que representan al mundo real, los **"objetos"**, que tienen **distribuida la funcionalidad e información** necesaria y que **cooperan entre sí** para el logro de un objetivo común.

"La Programación Orientada a Objetos es una 'filosofía' de desarrollo de software que permite crear unidades funcionales extensibles y genéricas, de forma que el usuario las pueda aplicar según sus necesidades y de acuerdo con las especificaciones del sistema a desarrollar. Permite una representación más directa del modelo de mundo real, reduciendo fuertemente la transformación radical normal desde los requerimientos del sistema, definidos en términos del usuario, a las especificaciones del sistema, definidas en términos del computador "[1].

En otras palabras, el paradigma de objetos pretende:

- Desarrollar los sistemas con modelos **más cercanos a la realidad** que a las especificaciones computacionales.
- Construir componentes de software que sean **reutilizables**.
- Diseñar soluciones de manera que puedan ser **extendidas** y **modificadas** con el mínimo impacto en el resto de su estructura.

Si bien los desarrollos hechos en otros paradigmas también pueden tener estas características, el Paradigma de Objetos no sólo provee herramientas para que sea más sencillo de lograr, sino que las considera como intrínsecas de la

[1] ALONSO AMO, F y SEGOVIA PEREZ, F. *Entornos y Metodologías de Programación*. Capítulo 1

configuración de sus lenguajes. A su vez, no se descarta las propiedades procedimentales que tienen que ver, por ejemplo, con la asignación en memoria o la utilización de estructuras de control para determinar el flujo de secuencia, sino que manteniendo sus propiedades procedimentales, utiliza estas herramientas dentro de otras más amplias y potentes que lo caracterizan como paradigma.

Principales características

Estructura de desarrollo modular basada en **objetos**, que son definidos a partir de **clases**, como implementación de tipos abstractos de datos.

Encapsulamiento como forma de abstracción que separa las interfaces de las implementaciones de la funcionalidad del sistema (**métodos**) y oculta la información (**variables**).

Mecanismo de envío de **mensajes**, que posibilita la interacción entre los objetos y permite la **delegación** de responsabilidades de unos objetos a otros.

Polimorfismo, basado en el **enlace dinámico**, de forma que los objetos del programa puedan interactuar indistintamente con otros, generando soluciones genéricas y extensibles.

Herencia, que permite que los objetos se definan sencillamente como una extensión o modificación de otros objetos.

Historia

El paradigma de objetos surge en base al paradigma imperativo, provocando un giro importante en sus principios y consideraciones básicas, en el que más que dejarlos de lado, los reorganiza y capitaliza dentro de un modelo más amplio, con otros conceptos característicos.

Su progreso no fue sencillo, sino que llevó muchos años. Smalltalk es considerado el primero de los lenguajes orientados a objetos y el emblemático de este paradigma. En Smalltalk, todo es un objeto, incluso los números enteros. Se basó en ideas de Simula (un lenguaje de simulaciones), pero no es sólo un lenguaje, sino un entorno completo, prácticamente un sistema operativo que se ejecuta encima de una "máquina virtual", lo que asegura su máxima portabilidad entre plataformas.

A pesar de ser un lenguaje simple, poderoso y que promueve buenas prácticas de programación, Smalltalk no llegó a ser un lenguaje muy popular. Esto se debe en parte a la poca aceptación de lenguajes interpretados en los años 1980 y 1990 y fundamentalmente al desconocimiento y cierto escepticismo por la idea de objetos que implicada un cambio grande en la forma de pensar y encarar soluciones del momento. A pesar de eso, hubo grandes empresas que llegaron a tener relativo éxito con él y el paradigma de objetos empezó a consolidarse y expandirse.

Cuando a mediados de los 90 surgió Java, un lenguaje que retoma las principales características de Smalltalk, se masificó la utilización del paradigma de objetos. Surgieron otros lenguajes de objetos y hubo algunos que siendo originarios de otros paradigmas incorporaron características de objetos, como el conocido C++.

La programación de objetos fue ganando campos de aplicación velozmente desde, y en la actualidad, aunque siguen siendo numerosos los sistemas desarrollados en lenguajes de otros paradigmas la tendencia marca una preferencia por los desarrollos que basan en lenguajes orientados a objetos, especialmente los nuevos desarrollos. En muchos casos, antes de seguir manteniendo y actualizando sistemas en otros lenguajes, se opta por migrarlos a soluciones en lenguajes orientados a objetos.

Lenguajes

El lenguaje originario y paradigmático de la programación en objetos es Smalltalk. Actualmente existen otros lenguajes de uso más extendido, como Java. Hay también algunos lenguajes como el C++ o el Eiffel, Visual Basic que son extensiones de otros lenguajes que fueron diseñados básicamente como imperativos, pero que tienen extensiones que incorporan, en mayor o menor medida, los principios del paradigma de objetos.

Smalltalk

Smalltalk plantea un modelo puro orientado a objetos lo que significa que todo, en el entorno, es tratado como un objeto. Entre todos los lenguajes orientados a objetos, Smalltalk es el muy consistente en cuanto al manejo de las definiciones y propiedades del paradigma.

Smalltalk es más que un lenguaje, es un completo entorno de desarrollo con numerosas herramientas de desarrollo.

Existen distintas versiones de Smalltalk que presentan variantes en sus entornos de desarrollo para la implementación de aplicaciones orientadas a objetos. Los más comunes y simples de utilizar contienen los siguientes componentes:

1. **Un modelo de objetos a partir del cual los objetos están definidos:** El modelo de objetos define cómo se comportan los objetos. Este modelo soporta la herencia simple, el comportamiento de clases e instancias, el enlace dinámico, liberación automática de memoria y el manejo de mensajes.
2. **Un conjunto de clases reutilizables:** Smalltalk tiene una gran cantidad de clases que pueden ser reutilizadas. Estas clases proveen las funcionalidades básicas de procesamiento de información y el soporte para la portabilidad a diferentes plataformas, incluyendo las interfaces gráficas de usuario y de la misma definición de las clases. Existe un

núcleo de clases básicas para el tratamiento de la información a la que se suman otras clases que permiten realizar funcionalidades específicas, como por ejemplo estructuras de datos dinámicas como las colecciones, acceso a base de datos, conectividad, tratamiento de excepciones, etc.

3. **Un conjunto de herramientas de desarrollo:** Estas herramientas habilitan a los usuarios a interiorizarse de la implementación y la posibilidad de modificar las clases existentes como también crear nuevas clases, ya que el código es abierto. Sirven para la detección de errores a nivel fuente, poder realizar seguimientos, observar datos, modificar datos, crear ejecutables, etc.
4. **Un entorno en tiempo de ejecución:** Esto permite a los usuarios a ejecutar un programa en Smalltalk mientras se cambia el código fuente. Los cambios realizados al código fuente son reflejados instantáneamente en la aplicación que se está ejecutando.
5. **Un sistema de codificación débilmente tipado**: El desarrollo de aplicaciones se realiza mediante una codificación en la que las variables y expresiones en general que referencia a los objetos del sistema no deben ser declaradas de un tipo de datos o clase en particular. Esto permite una gran simplicidad, promueve aplicaciones genéricas a la hora de programar y facilita las futuras modificaciones o reutilizaciones del código.

Capítulo 2

Objetos y mensajes

- **Objetos**
- **Mensajes**
- **Estructura de un objeto**
  - Atributos
  - Métodos
- **Encapsulamiento**
- **Delegación**
  - Mensajes de un objeto a sí mismo
  - Métodos básicos de acceso
- **Múltiples referencias**

Objetos

Los objetos son **abstracciones** que representan a las "**cosas" del mundo real** que forman parte del **dominio del problema**, y a toda entidad que realiza alguna tarea en función de la funcionalidad del sistema. En objetos, **"todo" es pensado como un objeto.**

Hay objetos que representan elementos de existencia física, que se pueden ver y tocar, y que en general son fácilmente reconocibles.

Ejemplo:

En un sistema comercial, probablemente sean objetos los clientes, los empleados, los productos que se venden, las sucursales, los depósitos, las góndolas, etc.

Otros objetos representan elementos reales que son propios de una construcción lógica, conceptual, aunque no tengan una presencia que se pueda percibir por los sentidos. Suelen ser situaciones que en la vida real son representadas o acompañadas por documentos escritos que describen sus características.

Ejemplo:

En el mismo sistema anterior, podría haber objetos que representen a las compras que hacen los clientes, las entregas de mercaderías, los movimientos contables, los pagos de sueldos y en general a toda transacción que forma parte de la dinámica del sistema.

Objetos que existen en cualquier sistema y que usan permanentemente son los que representan los elementos de los códigos, lenguajes y signos propios de la realidad y que en otros paradigmas constituyen los tipos de datos básicos.

Ejemplo:

Son objetos los caracteres, los números en sus diferentes precisiones y sistemas de representación, las palabras y cadenas de caracteres, los valores booleanos, las fechas, los horarios y toda clase de secuencias y convenciones que se desee representar.

Para ser consistente con su propia lógica, todos los elementos de software y hardware del sistema son representados por objetos.

Ejemplo:

Son también objetos las líneas de código, las estructuras de datos, las ventanas, con sus botones, cuadros de texto, menús y demás elementos gráficos, los procesos, los dispositivos, las bases de datos, etc.

Asumiendo que un sistema está formado por un conjunto de objetos que coopera para lograr un fin, toda tarea, más grande o más pequeña, que el sistema realice será responsabilidad de alguno de los objetos que lo componen. Por lo tanto, la razón de existir de un objeto en un sistema es realizar alguna tarea. En otras palabras, "todo lo que se hace, alguien (algún objeto) lo hace" y no tiene sentido objetos que no hagan nada.

Mensajes

Los objetos interactúan solicitándose servicios e intercambiando información mediante el **envío de mensajes**.

Para pedirle a un objeto que realice una acción, se le envía un mensaje. El objeto, en respuesta, realiza la tarea solicitada.

Cada objeto podrá responder a cierto conjunto de mensajes y a otro no, por lo que es fundamental enviar los mensajes a los objetos que sean capaces de responderlos.

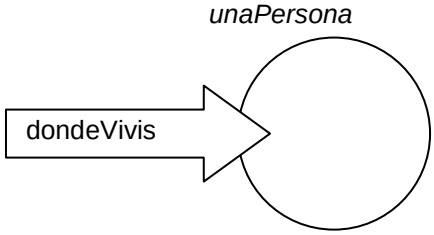
Si la acción pedida consiste en obtener un resultado, el objeto dará por concluida su respuesta al mensaje al retornar el valor correspondiente.

Ejemplo:

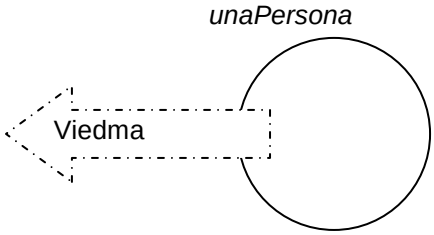
Hay un objeto que representa a una persona, a la que se le envían mensajes para preguntarle dónde vive y cómo se llama. La persona se la representa con el identificador **unaPersona**, y a los mensajes se los denomina **dondeVivis** y **comoTeLlamas.**

Cuando se le quiere preguntar a una persona donde vive, al objeto que la representa se le envía el mensaje.

unaPersona dondeVivis



La respuesta podría ser "Viedma", en caso que la persona viva allí.



En forma análoga, ante le envío del mensaje

unaPersona comoTeLlamas

El objeto devuelve el valor **Juan**, suponiendo que la persona se llama así.

En otras ocasiones, la tarea solicitada no tiene por objetivo obtener un resultado, sino modificar la información del sistema.

Para indicarle con precisión cuál es la tarea solicitada, muchas veces los mensajes deben incluir **argumentos**.

Ejemplo:

Para pedirle a la persona se mude a otra ciudad, se le debe enviar un mensaje donde se le pasa como argumento el nuevo lugar de residencia.

unaPersona mudarse: 'Rosario'

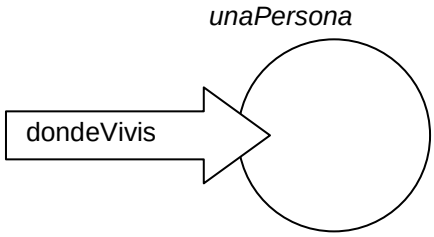
El objeto **unaPersona** realizará la tarea encomendada. En este caso el objetivo del mensaje no es obtener un resultado a modo de respuesta, sino provocar un efecto determinado en el objeto.

Una propiedad fundamental de los objetos es que existen antes y permanecen después del envío del mensaje, manteniendo su entidad y su esencia. El objeto sigue siendo el mismo aunque se haya modificado alguna de sus características.

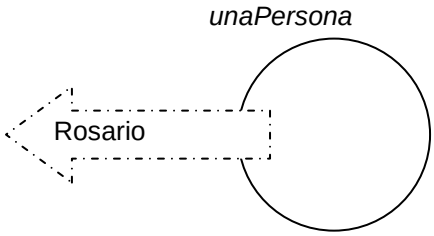
Ejemplo:

Si a continuación se le vuelve a preguntar a la persona dónde vive, su respuesta será diferente a la anterior.

unaPersona dondeVivis



La respuesta será ahora "Rosario", ya que el objeto realizó correctamente la tarea anteriormente solicitada de mudarse de ciudad.



La **funcionalidad de un sistema en objetos**, en definitiva, se implementa mediante **el envío de mensajes a objetos**. Asumiendo que hablar de "programa" en el paradigma de objetos es muy relativo, en comparación a otros paradigmas, el código del programa, en vez de estar constituido por "instrucciones", consiste en mensajes que se envían a objetos.

Estructura de un objeto

Todo objeto tiene, dispone o conoce:

- Un **estado interno**, conformado por **atributos** o **variables de instancia** que describen cómo es la entidad y que contienen los valores que representan su información.

- El **comportamiento**, que consiste en el conjunto de **mensajes** que puede recibir, lo que se corresponde con las acciones o funcionalidades que puede realizar la entidad. Para ello dispone de **métodos**.

El **envío de un mensaje** se representa mediante el objeto receptor (una variable que lo referencia), seguido del identificador del mensaje (que coincidirá con el nombre de un método que el objeto receptor conoce) y por último un argumento, en caso que fuera necesario (o más de uno).

objeto mensaje: argumento

Atributos

Para que el objeto pueda responder a los mensajes, es necesario que conozca determinada información, que posea ciertos datos. **Todo aquello que el objeto conoce** constituye una característica propia de cada objeto y diferente a la de otros, y se denomina en general **atributos o estado interno del objeto.**

Por otra parte, a quién solicitó la tarea le interesa que ésta se realice, sin importar el detalle de los datos utilizados para hacerlo. Si se le solicitó la tarea a un objeto en particular y no otro es porque se sabe, se supone o se confía que el objeto cuenta con la información necesaria. (De todas maneras, si hicieran falta datos adicionales, se pueden enviar como argumento)

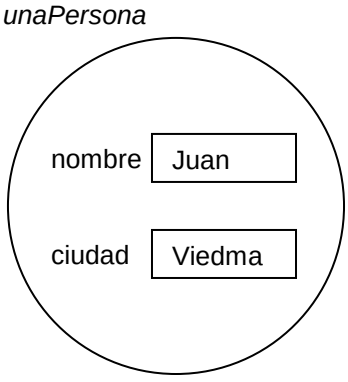
En Smalltalk se llama **variables,** y más precisamente variables de instancia, a cada uno de los valores que el objeto conoce y que contienen la **información propia del estado interno de un objeto**. Cada una es identificada con un nombre y contiene un valor. Mediante asignaciones su valor puede ir cambiando.

Las variables son **privadas** al objeto. Sólo pueden ser accedidas, utilizadas, modificadas por el mismo objeto.

Ejemplo:

Una forma en que el objeto **unaPersona** podría estar implementado internamente para responder a los mensajes anteriores, es mediante dos variables de instancia, una que representa su nombre y otra la ciudad donde vive.

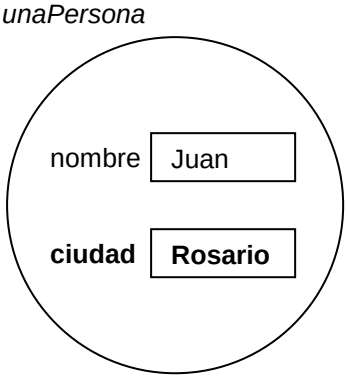
En la situación inicial anterior, el objeto **unaPersona**, en la variable **nombre** contenía el valor **Juan** y en la variable **ciudad**, el valor **Viedma**



Luego del envío del mensaje

unaPersona mudarse: 'Rosario'

El objeto resulta con la variable **ciudad** modificada con el nuevo valor.



Métodos

Para que el objeto pueda responder a los mensajes, además del conocimiento de los atributos, debe contar con **la inteligencia o habilidad necesaria para poder procesarlos de manera adecuada para la tarea que se solicita**, lo que se implementa mediante **métodos**.

Un método consta de una porción de código donde se detalla lo que el objeto debe hacer para realizar una tarea solicitada. **En respuesta a un mensaje el objeto ejecuta un método**. Visto al revés, los métodos que un objeto dispone, pueden ser invocados mediante el envío de mensajes por parte de quien lo requiera.

Lo importante es que quien envía el mensaje se limita a enunciar qué es lo que quiere y es el objeto receptor el que se ocupa de la forma en que se realiza la tarea, de seguir la secuencia de pasos que sea necesario dar y de utilizar, consultar o modificar los atributos que el mismo objeto tiene.

Cada método tiene un nombre que lo identifica y puede recibir argumentos, lo que constituye su interfaz pública. A su vez, tiene una implementación que es privada. Para que la comunicación se produzca, el mensaje que se envía debe coincidir literalmente con el nombre de alguno de los métodos. Es responsabilidad del objeto receptor ubicar el método con su correspondiente implementación para responder al mensaje.

Cuando un objeto recibe un mensaje determina qué método es el solicitado y pasa a ejecutar sus sentencias.

Ejemplo:

Retomando el ejemplo anterior, **unaPersona**, para responder al mensaje que se le envía preguntándolo dónde vive, tiene un método cuya implementación consiste en retornar el valor de la variable **ciudad** de esa persona.

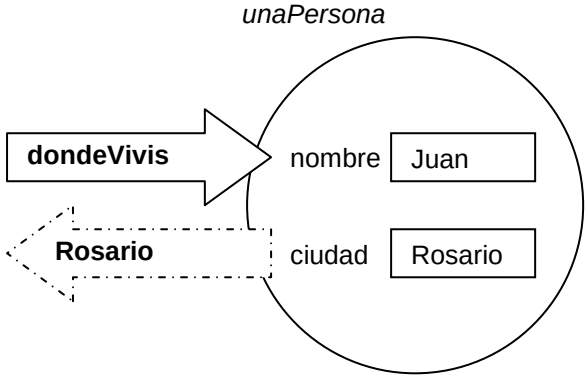
```
dondeVivis
    ^ciudad
```

(El carácter **^** indica la devolución del valor de la expresión que se encuentra a su derecha)

Ante la invocación

unaPersona dondeVivis

Se ejecuta el método indicado y se devuelve el valor **Rosario.**



Ejemplo:

Otro método que conoce unaPersona para poder realizar la tarea de cambiar de ciudad, es el denominado **mudarse:** que está preparado para recibir un argumento que se lo identifica como **nuevaCiudad**. En su implementación, se modifica la variable **ciudad** asignándole el valor del argumento recibido.

```
mudarse: nuevaCiudad
    ciudad := nuevaCiudad
```

(En la interfaz, los dos puntos (:) forman parte del nombre del método e indican la presencia de un argumento. En el cuerpo del método, los caracteres "**:=**" indican la asignación a la variable de la izquierda, del valor de la expresión que se encuentra a la derecha.)

Encapsulamiento

Un objeto no conoce el funcionamiento interno de los demás objetos y no lo necesita para poder interactuar con ellos, sino que le es suficiente con conocer su interfaz, es decir, saber la forma en que debe enviarles sus mensajes y cómo va a recibir la respuesta. Ante la modificación de una funcionalidad en particular del sistema, en la medida que su implementación esté encapsulada en un objeto, el impacto que produce su cambio no afectará a los otros objetos que interactúan con él.

Es necesario diferenciar entre el comportamiento de un objeto, o sea las acciones que es capaz de realizar, y cómo lleva a cabo este comportamiento. Esta abstracción de datos se realiza a través de la interfaz del objeto. Mediante esta interfaz, un objeto emisor se comunica con otro objeto receptor pero el objeto emisor desconoce la forma en que se lleva a cabo la acción solicitada, ya que su implementación es interna al objeto. El énfasis se produce en **qué se puede obtener** más que en **cómo se lo obtiene**. O sea, la interfaz encapsula los datos y código de un objeto.

El encapsulamiento permite proteger al observador de los complejos detalles de implementación del objeto. El objeto encapsula su funcionamiento interno colaborando en la abstracción del observador, que se concentra en lo que necesita resolver dejando de lado los detalles que no son esenciales. Esto permite **separar la implementación de un objeto de su comportamiento**. Esta separación crea una "caja negra" en donde el observador está alejado de los cambios de la implementación. Mientras la interfaz permanezca igual, cualquier cambio interno a la implementación es transparente al observador. Por otra parte, se garantiza que ninguna tarea se hace dos veces y por lo tanto, toda modificación de ella se hace también una sola vez.

De esta manera, sólo el objeto mismo, en sus métodos, puede referirse y modificar los valores guardados en sus variables. Los métodos de un objeto no pueden acceder a las variables de otros objetos. Un objeto sólo puede enviar mensajes a otros objetos, y éstos, internamente, al ejecutar sus métodos, acceder a sus variables. El encapsulamiento asegura que el proceso para obtener los datos de un objeto sea seguro y transparente para el observador.

Delegación

Un objeto **delega a otro una responsabilidad** al **enviarle un mensaje** solicitándole algún servicio. Un diseño que presenta una adecuada distribución de responsabilidades entre los objetos lleva a que "**cada objeto hace sólo lo que tiene que hacer y que sólo él lo hace"**. De esta manera se organizan las responsabilidades entre los objetos sin duplicar ni redundar en el código del programa y sin realizar tareas ni guardar información innecesaria.

La resolución de una tarea, por más compleja que sea, es solicitada a un objeto en particular, quien es el responsable de hacerla. Para ello, en general, pide ayuda a su vez a otros objetos para que realicen parte de ella. En definitiva, la tarea se resuelve mediante una múltiple delegación de responsabilidades de un objeto hacia otros objetos, y así sucesivamente. Cada objeto se comunica con los otros "confiando" en la tarea que delega.

Cuando un objeto recibe un mensaje, al ejecutar el método adecuado desencadena el envío de otros mensajes a los objetos que conoce. Desde el punto de vista de la implementación, las líneas de código que constituyen el cuerpo del método, son básicamente nuevos envíos de mensajes.

Los atributos de un objeto son siempre otros objetos. Los argumentos que recibe cuando se le envía un mensaje también son objetos. Por lo tanto pueden actuar como receptores de mensajes que el objeto envía. Son estos, en principio, los objetos que un objeto conoce.

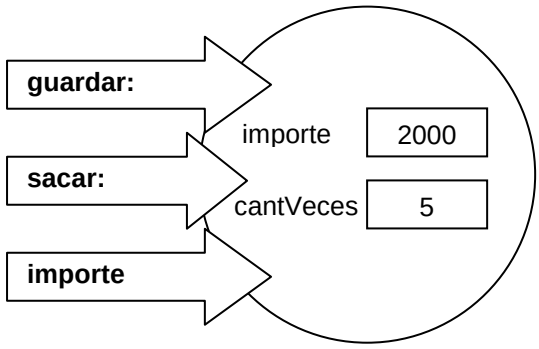
Ejemplo:

Hay un nuevo objeto que representa una caja fuerte. Los mensajes que entiende son **guardar**: que agrega una cierta cantidad de dinero a la existente en la caja fuerte, **vaciar** que retira el total que haya en el momento y **cuantoHay**, que informa la cantidad de dinero que hay en la caja fuerte. Otra responsabilidad de la caja fuerte es registrar la cantidad de veces que se abre para guardar dinero.

Junto con los métodos que permiten realizar esas tareas, el objeto tiene una variable de instancia llamada **importe** que representa la cantidad de dinero que hay en el momento, por ejemplo $2000, y otra, **cantVeces**, con la cantidad de veces que se guardó dinero, por ejemplo, 5.

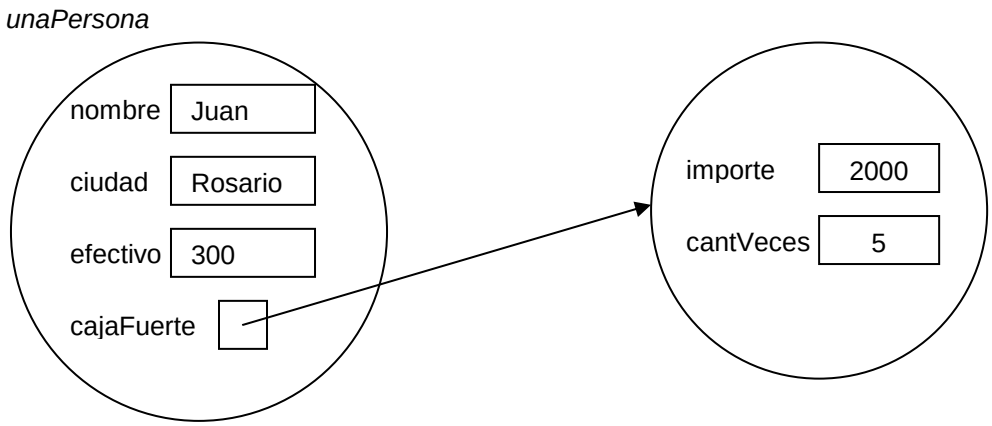
La implementación de los métodos es:

```
guardar: unValor
    importe := importe + unValor
    cantVeces := cantVeces + 1

vaciar
    importe := 0

cuantoHay
    ^importe
```



La caja fuerte, así definida, puede ser utilizada por otros objetos. En particular, la puede aprovechar el objeto **unaPersona**, que es ahora definido con nuevas características.

UnaPersona va a tener la funcionalidad de cobrar su sueldo, que consiste en ahorrar el 40% y dejar el resto disponible para gastar. Otra de sus tareas es informar el dinero total que tiene. Para ello, dispone de una caja fuerte y conoce cuanto dinero tiene disponible en efectivo.



La variable **cajaFuerte** tiene por valor al objeto definido anteriormente. Se puede decir indistintamente que la persona tiene, conoce o referencia a la caja fuerte.

La implementación de los nuevos métodos de la persona puede ser:

```
cobrar: unSueldo
    cajaFuerte guardar: ( unSueldo * 0.4 )
    efectivo := efectivo + ( unSueldo * 0.6 )

dineroTotal
    ^ cajaFuerte cuantoHay + efectivo
```

Cuando a la persona se le envía el mensaje

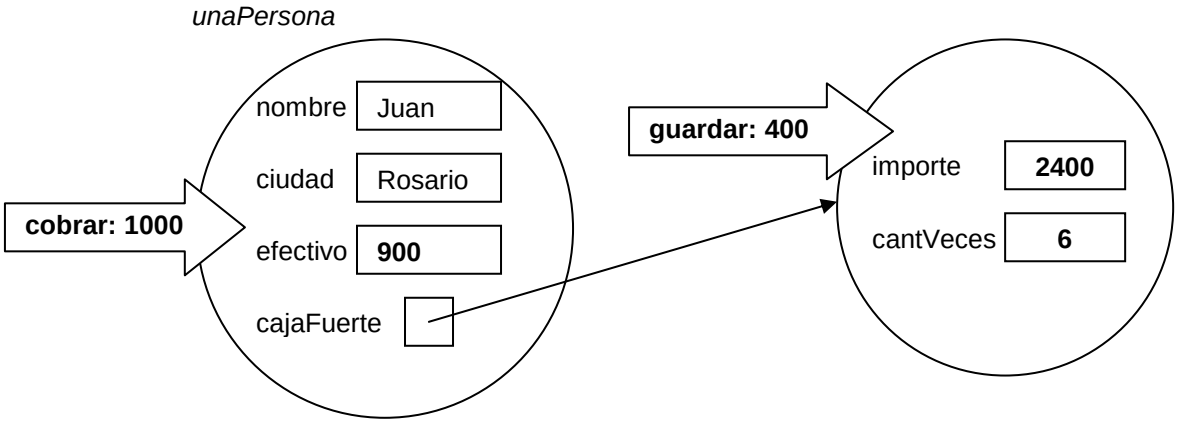
```
unaPersona cobrar: 1000
```

En respuesta, **unaPersona** identifica y ejecuta el método **cobrar:** En la primer línea de código, envía el mensaje **guardar:** al objeto **cajaFuerte**, con un argumento que en este caso es el resultado de la multiplicación, 400**.**

Luego, continúa con la ejecución de la segunda línea del código, en la que guarda en su variable de instancia **efectivo** lo que había anteriormente, 300, más el 60% del sueldo, que es 600, resultando 900.

Para **unaPersona**, la interacción con la **cajaFuerte** consistió en el envío de un mensaje, despreocupándose de su implementación.

Para la **cajaFuerte**, recibir esa solicitud derivó en ejecutar el código del método **guardar:** modificando los valores de sus variables de instancia **importe**, que de 2000 pasó a 2400, y **cantVeces**, que se incrementó de 5 a 6.



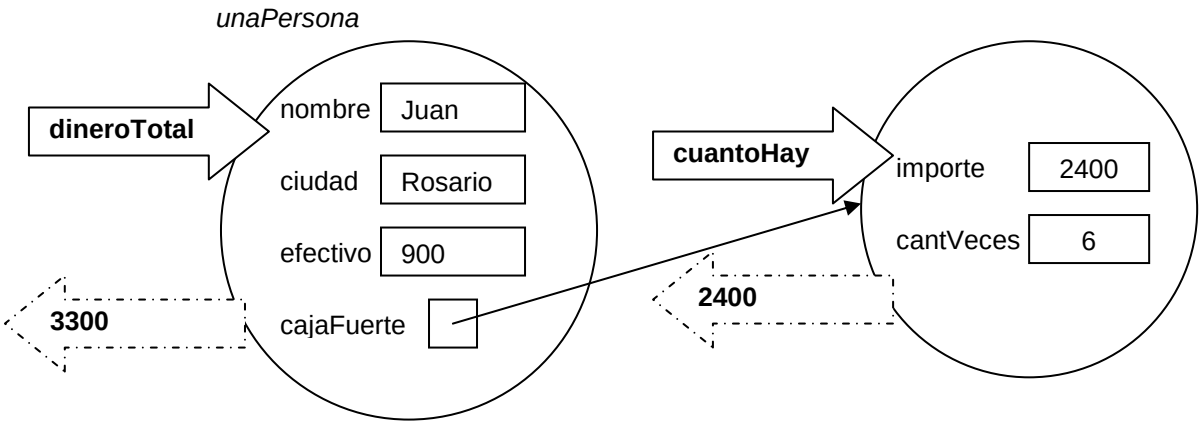
Si a continuación, a la persona se le envía el mensaje

unaPersona dineroTotal

UnaPersona ejecuta su método **dineroTotal**, en el que delega a la **cajaFuerte** el cálculo de cuánto dinero hay allí mediante el envío del mensaje **cuantoHay**, y su resultado, 2400, lo suma al valor de su variable **efectivo**, 900. Por último, retorna el resultado de la suma, 3300 a quien haya enviado el mensaje.

El objeto **unaPersona** conoce su efectivo y por ello accede directamente a el mediante el nombre de la variable, pero no conoce cuál es el importe que hay en la caja fuerte. La tarea de informar cuanto dinero hay dentro, es una responsabilidad propia y exclusiva de la **cajaFuerte**, por lo tanto, **unaPersona** se lo solicita a **cajaFuerte**, a quien sí conoce.

La **cajaFuerte**, en respuesta al pedido ejecuta el método **cuantoHay** y devuelve el valor de su variable **importe**.



El esquema de delegación es permanente en un sistema de objetos y se lo usa muchas veces sin percibirlo. Para la tarea inicial que da arranque a un sistema complejísimo o para hacer una simple operación aritmética como la suma, es necesario utilizar la delegación. Desde cualquier posible objeto creado por los desarrolladores hasta el más básico de los objetos predeterminados, como puede ser un número entero, todos los objetos están preparados para que se les

delegue alguna tarea a ellos. Todas las operaciones y todas las tareas que se deban hacer en el sistema se invocan mediante mensajes.

La respuesta a un mensaje, en general se resuelve con un método desde el que se envían nuevos mensajes. A su vez, las invocaciones a los mensajes, están definidas en métodos que dan respuesta a otros mensajes. Esta aparente cadena infinita, en un sistema completo, comienza con la ejecución de la aplicación, que se puede interpretar como el envío de un mensaje de inicio, y en el otro extremo, cuando la tarea solicitada es tan sencilla que se resuelve con la asignación o la devolución de un objeto.

Ejemplo:

Teniendo en cuenta que todos los atributos de un objeto son también objetos, **unaPersona** no sólo puede enviar mensajes a **cajaFuerte** como en el ejemplo anterior, sino que puede pedirle tareas al objeto **Rosario**, al **Juan** o al **300** que tiene guardado en sus otras variables. En realidad, lo está haciendo en el método **cobrar:** en la línea donde dice:

```
efectivo := efectivo + ( unSueldo * 0.6 )
```

Al objeto **300** que está guardado en la variable **efectivo** se le envía el mensaje **"+"** teniendo por argumento lo encerrado entre paréntesis. El **300**, entre los métodos que conoce tiene al **"+",** por lo tanto ante la invocación, realiza una suma de la manera que esté implementada en el código del método, y devuelve el resultado, que en el ejemplo es el objeto **900**. Este objeto es guardado como nuevo valor de la variable **efectivo.**

A su vez, el argumento **unSueldo** en este ejemplo es el objeto **1000**, quien entre los mensajes que es capaz de responder tiene al **"*"**, cuyo correspondiente método realiza una multiplicación. Ante la invocación mostrada, devuelve como resultado al objeto **600**.

En definitiva, se trata de dos mensajes anidados. A un objeto se le manda el mensaje **"+"** teniendo por argumento el objeto que devuelve el envío del mensaje **"*"** a otro objeto, con otro como argumento. El objeto resultante es asignado a la variable **efectivo**.

En el método **dineroTotal**, el número resultante de enviar el mensaje **cuantoHay** actúa como receptor del mensaje **"+"** con el objeto referenciado por la variable **efectivo** como argumento. El objeto resultante es devuelto a quien envió el método.

Algo similar ocurre con los métodos de la **cajaFuerte**. Las sumas y restas se realizan mediante el envío del mensaje **"+"** y **"-"** a los respectivos objetos receptores, con otros objetos como argumentos.

Mensajes de un objeto a sí mismo

Puede darse el caso en que el objeto receptor y emisor del mensaje coincida, es decir que el objeto se envíe un mensaje a sí mismo. Tiene el mismo sentido de delegación en el que para realizar una tarea compleja se la descompone en tareas más sencillas que son realizadas por objetos específicos, con la particularidad que en vez de ser otros objetos quienes las realizan, puede tratarse del mismo objeto.

En Smalltalk, cuando se implementa el código, la forma en que un objeto, en alguno de sus métodos puede hacer referencia a sí mismo es mediante una palabra reservada, denominada **self**.

Ejemplo:

Continuando con el mismo ejemplo anterior, si se desea preguntarle a una persona si el dinero total que dispone le alcanza para comprar un producto de un valor determinado, el método que permitiría responderlo, sin usar **self** podría ser.

```
puedeComprarAlgoQueCuesta: unValor
    ^ (cajaFuerte cuantoHay + efectivo) > unValor
```

Este método devuelve el valor booleano (true o false) resultante de comparar el dinero que tiene la persona en efectivo y en la caja fuerte con el valor que se envía como argumento.

Teniendo definido el método **dineroDisponible** que realiza la mayor parte de la tarea, no tiene sentido volver a hacerlo en el presente método. **UnaPersona** se puede pedir ayuda a sí misma para resolver la tarea que le solicitan, puede utilizar la delegación consigo misma, utilizando **self** de esta manera

```
puedeComprarAlgoQueCuesta: unValor
    ^ self dineroTotal > unValor
```

Dada la siguiente invocación:

```
unaPersona puedeComprarAlgoQueCuesta: 3000
```

UnaPersona, el objeto receptor del mensaje, durante la ejecución del método actúa como receptor a su vez del mensaje **dineroTotal**; **self** "es" el objeto mismo.

Métodos básicos de acceso

A los métodos más sencillos, que en vez de permitir enviar nuevos mensajes su implementación se limita a acceder a las variables de un objeto, ya sea para asignar o devolver su valor, se los denomina métodos de acceso ("accesors").

- Los que asignan un valor a las variables se llaman "**setters"**.
- Los que devuelven el valor de las variables son los "**getters**".

Ejemplo:

Para el objeto **unaPersona**, podría definirse un "setter" para la variable nombre

```
bautizar: unNombre
    nombre := unNombre
```

El método "getter" correspondiente a la misma variable sería el ya definido **comoTeLlamas**

```
comoTeLlamas
    ^nombre
```

Los identificadores de las variables de instancia pueden coincidir con los de los métodos del mismo objeto, ya que la forma en que se envían los mensajes y se construyen las expresiones no permite ningún tipo de ambigüedad. Con ver la posición relativa del identificador en el contexto alcanza para saber de qué se trata: Si se le asigna algo, se devuelve o actúa como objeto receptor de un

mensaje, es sin dudas una variable. Si está en el lugar del mensaje, es decir que tiene otro identificador a la izquierda que actúa como su receptor, se trata de un método.

En los métodos de acceso se suele utilizar el mismo identificador que la variable que utilizan.

Ejemplo:

Los mismos métodos anteriores quedarían expresados de la siguiente manera:

```
nombre: unNombre
    nombre := unNombre


nombre
    ^nombre
```

Múltiples referencias

Un objeto puede (y suele) ser conocido no sólo por uno, sino por varios otros objetos, todos los cuales están habilitados a enviarle mensajes. Cada uno de estos objetos tendrá una referencia al objeto en cuestión, que es único.

Por lo tanto, si el objeto es modificado en consecuencia del envío de mensajes de alguno de los objetos que lo conoce, por ejemplo asignando nuevos valores a sus variables de instancia, ese cambio podrá impactar en los demás objetos cuando también interactúen con él, ya que se trata del mismo objeto.

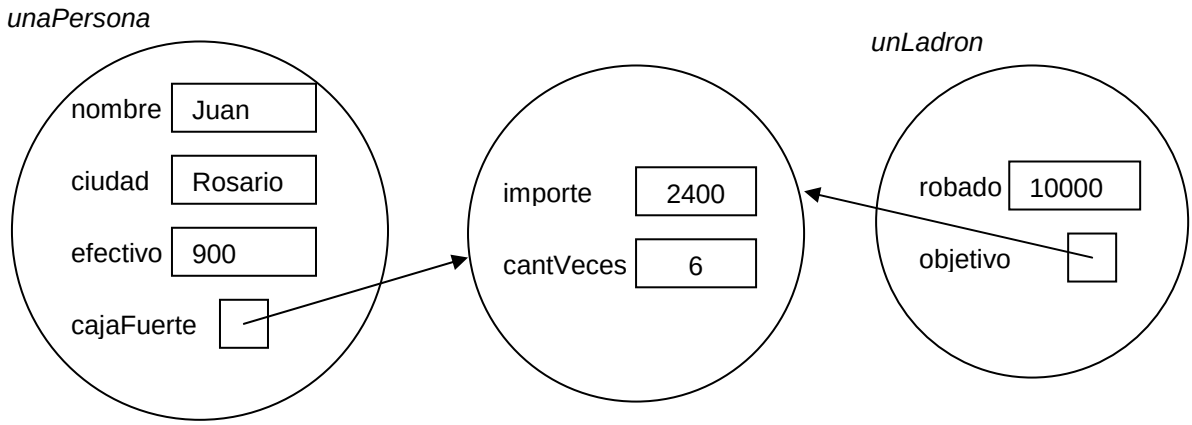
El mecanismo es transparente y se mantiene la lógica normal de envío de mensajes. Cualquiera de los otros objetos que lo conocen le puede mandar un mensaje sin necesitar enterarse que otro objeto también lo hizo.

Ejemplo:

Al ejemplo anterior se le agrega un nuevo objeto **unLadron** que representa a alguien que roba la caja fuerte de la persona.

Uno de los atributos de **unLadron**, denominado **objetivo**, que representa el objetivo del robo que va a realizar, es (referencia) el mismo objeto del atributo **cajaFuerte** de **unaPersona.** Además, del ladrón se conoce cuanto dinero lleva robado, con la variable **robado**.

Los objetos quedan relacionados de esta manera.



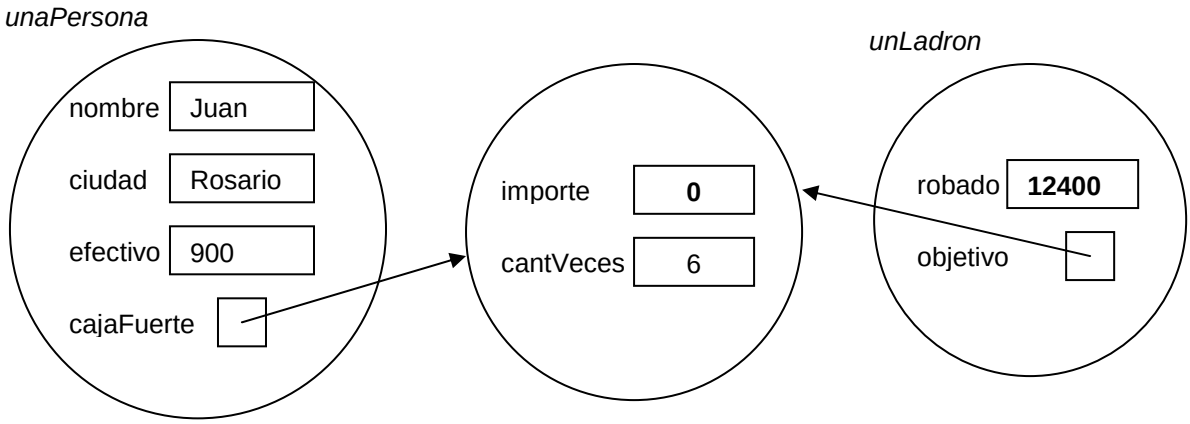
Una de las responsabilidades de **unLadron** es efectuar el robo, que se implementa mediante el método **robar.** La variable objetivo, al ser el objeto desarrollado anteriormente, entiende y sabe responder a los mensajes **cuantoHay** y **vaciar.**

```
robar
    robado := robado + objetivo cuantoHay.
    objetivo vaciar.
```

Luego de enviarse el mensaje

unLadron robar

El estado del sistema queda de la siguiente manera:



Por lo que si se cuando se le vuelve a pregunta a la persona por el dinero disponible:

unaPersona dineroTotal

La respuesta será de sólo **900**, que corresponde al efectivo.

Capítulo 3

Clases

- **Clase**
- **Creación de objetos**
  - Ciclo de vida de un objeto
- **Variables de Clase**
- **Métodos de Clase**

Clase

Entre los muchos objetos que pueden formar parte de un sistema, hay sin duda algunos muy diferentes a otros, con responsabilidades distintas, pero también muchos objetos son similares entre sí en los atributos que posee y en los mensajes que son capaces de responder. Esto motiva la necesidad de "clasificar" los objetos, agrupándolos en función de sus semejanzas.

En concreto, los objetos que responden a los mismos mensajes de igual manera y que tienen una estructura interna igual (las mismas variables, con diferente valor para cada objeto), se clasifican juntos en una determinada "**clase"**.

La clase **describe** completa y detalladamente la **estructura de información** y el **comportamiento** que tendrá **todo objeto de esa clase**, o sea, define el conjunto de variables de instancia y de métodos que determinan cómo van a ser y cómo se van a comportar sus objetos. En otras palabras, determina cómo es y cómo actúa cada objeto.

La clase es lo genérico: es el patrón o modelo ("plano", "molde") para crear objetos. Cada objeto tiene su propia identidad, con una posición de memoria independiente de los otros objetos de la misma clase donde se almacenan sus valores para cada una de sus variables.

Se dice que todo objeto es una "**instancia**" de una clase, porque es creado a partir de ella, es "instanciado". No existen objetos que no sean instancia de alguna clase y todo objeto conoce de qué clase es instancia.

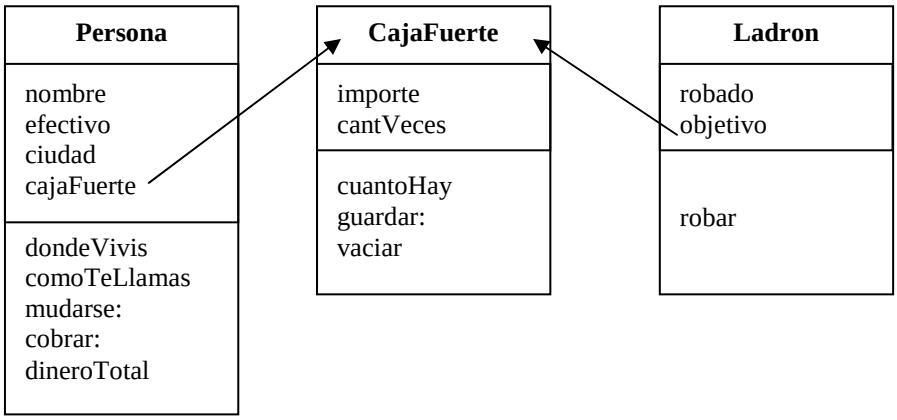
Cuando a un objeto se le envía un mensaje, como sabe de qué clase es instancia, recurre a ella para buscar la implementación del método que le permita responder al pedido.

La clasificación de un objeto se determina en función de los mensajes que se le van a enviar a él los restantes objetos. Para clasificar es necesario tener en cuenta que un objeto pertenece siempre a la misma clase: no puede dejar de ser de una clase y pasar a ser de otra.

Buscando alguna analogía con otros paradigmas de programación, una clase es la forma más elaborada de implementar un tipo abstracto de datos, de manera que los **objetos** vendían a ser a las **clases** lo que los **valores** a los **tipos de datos**.

Ejemplo:

Las clases del ejercicio del capítulo anterior se pueden representar mediante el siguiente diagrama de clases:



Cada cuadro representa a una clase. En la parte superior se indica el nombre de la clase, en el medio se enumeran los nombres de las variables y por último, los métodos.

Las flechas muestran las relaciones entre los objetos de las clases. En este caso, la variable **cajaFuerte** de los objetos **Persona** se espera que sean instancias de la clase **CajaFuerte**. Por su lado, los objetos **Ladrón** también, en su variable **objetivo**, tienen objetos de la clase **CajaFuerte**.

Creación de objetos

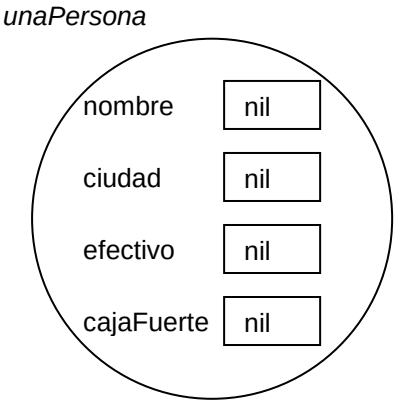
Para crear un objeto se envía un mensaje llamado **new** a la clase de la cual se lo quiere crear. En respuesta, se ejecuta el método **new** que retorna un nuevo objeto de esa clase. Inicialmente, todas las variables del objeto tienen un valor nulo.

Ejemplo:

unaPersona := Persona new.

Se crea un nuevo objeto de la clase **Persona** y se lo asigna a la variable **unaPersona**. Por lo tanto, **unaPersona** ahora es una instancia de **Persona.**

Todas las variables de instancia del objeto **unaPersona** recientemente creado, valen **nil** (valor nulo).



A partir de ese momento, el objeto ya sabe qué clase de objeto es y, por lo tanto, está habilitado para responder al conjunto de mensajes que corresponda a esa clase de objetos.

Ejemplo:

En general, los primeros mensajes que se envían a un objeto recién creado son los que permiten darle valores a las variables de instancia, ya se mediante los setters u otros métodos de inicialización.

unaPersona nombre: 'Juan'.

unaPersona mudarse: 'Viedma'.

unaPersona efectivoInicial

Se le envían a **unaPersona** el mensaje **nombre:** que es respondido con uno de los setters que asigna el argumento a la variable **nombre**, **mudarse:** que de manera similar asigna a la variable **ciudad**, y **efectivoInicial**, que sin necesidad de argumentos dispara un método definido en la clase **Persona**, que inicializa la variable con un valor prefijado:

efectivoInicial

efectivo := 0

De esta manera, quedan con valor todos los atributos del objeto, excepto **cajaFuerte**. Para este último, no existen aún valores u otros objetos que se le puedan asignar, por lo tanto, es necesario antes crear un objeto de la **CajaFuerte**.

unaCaja := CajaFuerte new.

unaCaja inicializar

Se le envía a la clase el mensaje **new** para crear el objeto y se lo guarda en una variable **unaCaja**. Luego se invoca al mensaje **inicializar** para darle valores por defecto a sus variables. Para ello, en la clase **CajaFuerte** está implementado el siguiente método:

inicializar

importe := 0.

cantVeces := 0.

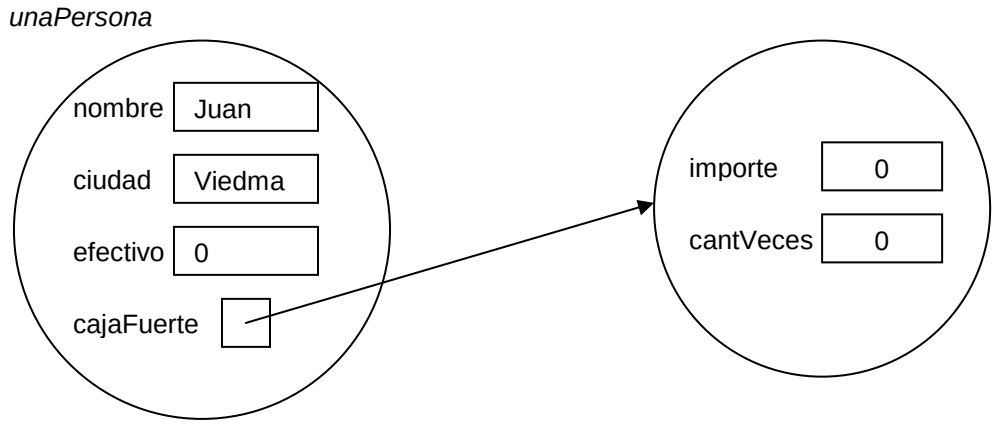
Ahora sí ya existe una instancia de la clase **CajaFuerte** que puede ser utilizada por el objeto **unaPersona**. Se le puede enviar un mensaje a **unaPersona** para indicarle que su caja fuerte va a ser la recién creada:

unaPersona tuCajaFuerteEs: unaCaja

Para que esto funcione, en la clase **Persona** debe haber un método **tuCajaFuertEs**: que asigne el objeto que referencia a la caja fuerte pasado como argumento a su propia variable **cajaFuerte**.

```
tuCajaFuerteEs: algo
    cajaFuerte := algo
```

Luego de esta secuencia de envío de mensajes, los objetos creados son:



Para llegar a la situación inicial planteada en los primeros ejemplos del capítulo anterior, se debe aún enviar varios mensajes más para que se reflejen las cantidades e importes correctos, ya sea haciendo referencia al método **cobrar:** o a otros que pudieran existir.

```
unaPersona cobrar: 1000.
unaPersona cobrar: 200.
...
```

También, para reflejar el caso de la presencia del ladrón, se debe crear el objeto **unLadron** y enviarle algunos mensajes para darle los valores correctos. En particular, para representar adecuadamente la situación descripta por la cual luego el ladrón va a robar la caja fuerte de la persona, es fundamental que se le envíe como argumento del mensaje **objetivo:** el mismo objeto **unaCaja** que se le envió como argumento a **unaPersona.**

```
unLadron := Ladron new.
unLadron objetivo: unaCaja.
unLadron robado: 10000.
```

Ciclo de vida de un objeto

La vida de un objeto comienza cuando un objeto cualquiera invoca la creación de un objeto y se lo referencia desde alguna variable.

A medida que avanza la ejecución del programa, el mismo objeto creado puede ser referenciado por otros objetos desde diferentes variables, ya sean variables de instancia, variables temporales, argumentos o estructuras de datos que conforman lo que se denomina el **ambiente**, contenedor o más gráficamente "**el lugar donde los objetos viven**".

Muchos objetos diferentes pueden tener referencias a un mismo objeto, sin que el objeto esté duplicado. Precisamente, una de las claves de la programación en objetos es que cada objeto sólo existe una sola vez dentro del sistema. Todos los demás objetos que necesiten utilizar sus servicios deberán contar con una referencia a él. En otras palabras, es como si un objeto existente en un único lugar fuera "visto" desde diferentes lugares.

Ejemplo:

Es lo que sucede con el objeto **unaCaja**, que es referenciado desde la variable de instancia **cajaFuerte** de **unaPersona** y desde la variable de instancia **objetivo** de **unLadron**.

También se podrían crear nuevos objetos **Persona** que tengan la misma caja fuerte, lo que tiene sentido de realidad pensando en personas que viven en la misma casa, y otros objetos **Ladron** que tengan como objetivo a la misma caja fuerte, lo cual puede formar parte del dominio del problema que se quiere resolver con el presente sistema, de manera que el objeto que representa la caja fuerte tendría muchas referencias.

Generalizando, la existencia de la clase **CajaFuerte** apunta a que también existan muchas cajas fuertes, por los que las posibles combinaciones de interacción entre los objetos, es decir qué persona y qué ladrón conoce a cuál caja fuerte, son innumerables.

Cuando los demás objetos dejan de necesitar el objeto en cuestión pueden cambian sus referencias hacia otros objetos. Así, cuando el objeto ya no es referenciado por ningún otro, quiere decir que su utilidad ha terminado y concluye el ciclo de vida del objeto. Entonces, internamente se ejecuta un mecanismo de liberación de memoria llamado **Garbage collector**, para que la memoria pueda ser reutilizada por otros objetos.

Ejemplo:

En un caso muy simple, si se crea un objeto **Persona**

```
p := Persona new.
```

y luego se asigna

```
p := 1
```

el objeto **Persona** creado deja de existir, porque la única forma de referenciarlo era mediante la variable **p**, que ahora tiene otro valor.

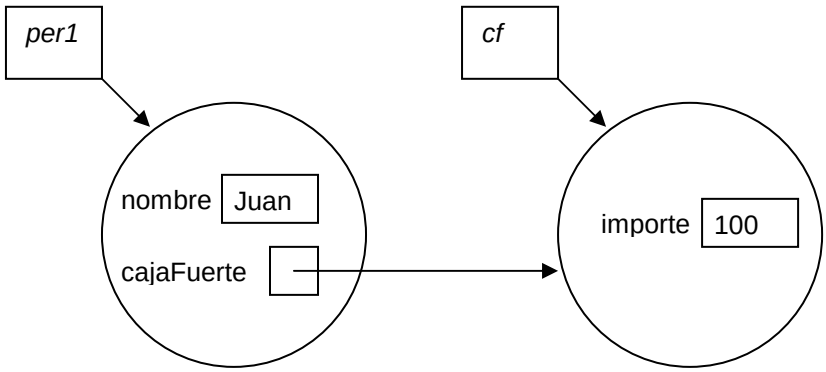
Ejemplo:

Se crea un objeto **Persona** y un objeto **CajaFuerte** y se los relaciona

```
per1 := Persona new.
per1 nombre: 'Juan'
cf := CajaFuerte new.
cf importe: 100.
per1 tuCajafuerteEs: cf.
```

Para este ejemplo no interesa la inicialización de todas las variables de instancia, sino sólo la de la variable **cajaFuerte**, que indica la relación entre los dos objetos y de algún

valor adición para que se identifique mejor a los objetos en ejemplo. Tanto la variable **cf** como la variable de instancia **cajaFuerte** de la persona **per1** referencian a la instancia de **CajaFuerte.**



Se crea otra **Persona** y se la guarda en otra variable auxiliar.

per2 := Persona new.

per2 nombre: 'Ana'

Se crea otra **cajaFuerte** y se la asigna a la misma variable anterior.

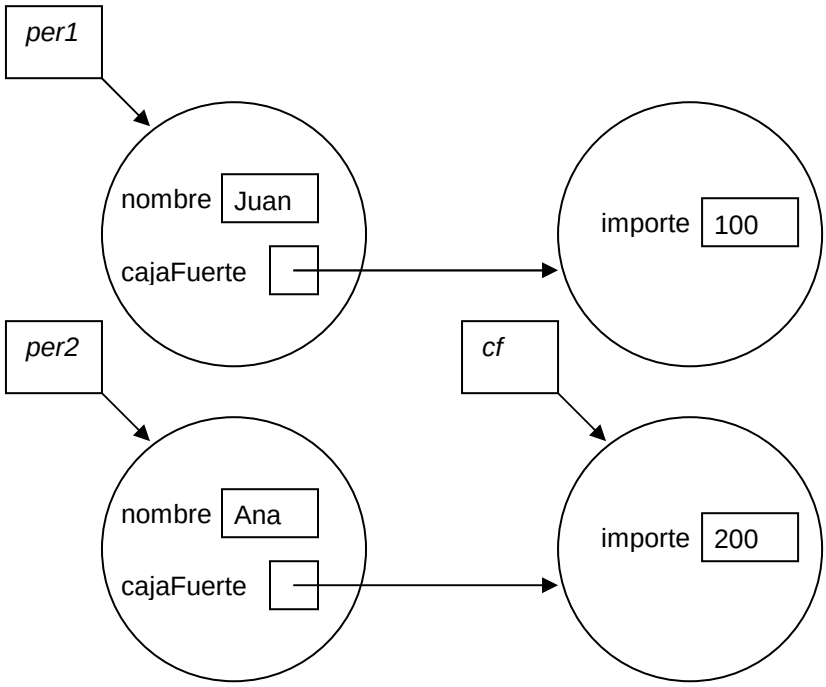
cf := CajaFuerte new.

cf importe: 200

Esta asignación hace que la variable **cf** deje de referenciar a la caja fuerte anterior y lo haga al nuevo objeto creado. La anterior instancia de **CajaFuete** sigue existiendo al ser referenciada desde la variable **cajaFuerte** de la persona **per1**.

Se le asocia a la segunda persona la nueva caja fuerte.

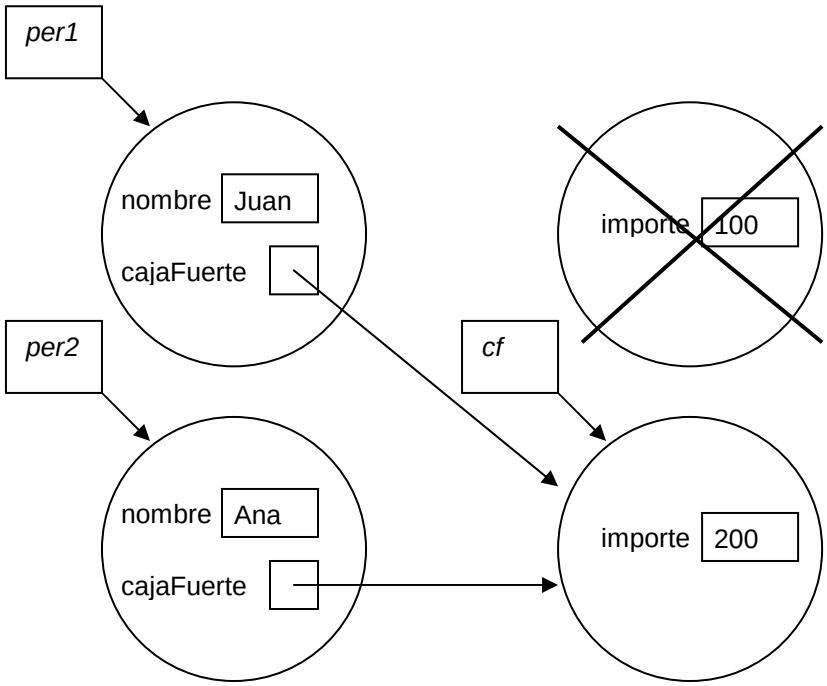
per2 tuCajafuerteEs: cf.



Por último, se le asocia también a la primera persona la segunda caja fuerte.

per1 tuCajafuerteEs: cf.

De esta manera, la instancia de **CajaFuerte** creada en primer lugar ya no es conocida por ningún otro objeto ni hay variable que la referencia, por lo que no hay forma de acceder a ella para enviarle mensajes. Es considero como un objeto que ya no es de utilidad, que está "muerto" y queda sujeto al mecanismo de liberación de memoria.



Por su parte, la otra instancia de **CajaFuerte** tiene tres referencias, dos desde los objetos **Persona** y otra desde la variable **cf**. Mientras siga siendo referenciada por al menos uno de ellas, seguirá siendo un objeto "vivo".

Variables de Clase

En una clase se pueden definir dos tipos de variables. Además de la información propia de cada objeto, que es representada por las **variables de instancia**, se pueden definir otras **variables propias de la clase en sí misma**, que son llamadas **variables de clase**.

Una variable de clase contiene un valor que es **común a todos los objetos de la clase** y es de acceso compartido, por lo que todos ellos pueden consultarla o modificarla mediante sus métodos. Una variable de clase **existe sólo una vez en la clase**.

Ejemplo:

Una nueva responsabilidad de las personas es informar si tienen mucho dinero o no. El criterio para determinarlo consiste es comparar el dinero total de una persona con un valor máximo. Como dicho valor no es fijo sino que puede ir cambiando, es conveniente guardarlo en una variable y como es único y común para todas las personas por igual, lo mejor es que la variable sea de clase.

El método que permite realizar esta tarea puede ser definido como:

```
muchoDinero
    ^self dineroTotal > Maximo
```

Maximo es el nombre de la variable de clase. Se lo utiliza de igual manera que las otras variables, ya sea como objeto receptor o como argumento.

Métodos de Clase

El comportamiento de todos los objetos de una clase es definido mediante métodos, que más precisamente se denominan **métodos de instancia**, que son los que se ejecutan cuando a cualquiera de los objetos de la clase se le envía el mensaje correspondiente. Además, **una clase también puede actuar como receptora de mensajes**, y para responder a ellos, puede tener definidos métodos que son propios de la clase como tal, que son llamados **métodos de clase**.

Los métodos de clase implementan la **funcionalidad que está asociado a la clase** para realizar tareas específicas. El caso más frecuente es para crear nuevos objetos, pero también se puede realizar todo tipo de tareas que no se relacionen específicamente con un objeto en particular de la clase, sino con ella en general.

En el envío de un mensaje no hay ambigüedad posible que confunda si se trata de un método de clase o uno de instancia, ya que el receptor debe ser una clase cuando se invoca a un método de clase y de igual manera, debe ser una instancia cuando se utiliza un método de instancia.

Ejemplo:

Como ya ha sido usado en ejemplos anteriores, el método **new** es el método de clase, definido para todas las clases, que permite crear objetos.

```
unaPersona := Persona new.
```

Una utilidad frecuente para los métodos de clase es la de crear e inicializar objetos. Para la persona, se puede definir un método que cree a un objeto y le asigne algunos valores a las variables.

```
unaPersona := Persona new: 'Juan'.
```

Este mensaje difiere del anterior por el argumento, ya que el carácter ":" forma parte del nombre del método. El objetivo es que en **unaPersona** se guarde el nuevo objeto, con el nombre que se indica y una cantidad de efectivo por defecto. La implementación puede ser:

```
new: unNombre
    | p |
    p := self new.
    p nombre: unNombre.
    p efectivoInicial.
    ^p
```

Entre **"|"** se indica que **p** es una variable local del método. En la línea siguiente, **self** hace referencia al receptor del mensaje, en este caso, tratándose de un método de clase, a

la clase **Persona.** Por lo tanto es la clase la que recibe el mensaje **new** y devuelve una nueva instancia de **Persona**, que se guarda en la variable **p.** Al objeto **p**, la persona recién creada, se le envían mensajes de inicialización y por último se lo devuelve.

Más allá que a partir de una clase se hayan instanciado más o menos objetos, o inclusive ninguno, un método de clase, cuando se lo invoca, su receptor es la clase misma y por lo tanto no puede acceder desde su código a las variables de instancia de ninguno de los objetos de esa clase. No tendría sentido que lo hiciera, ya que no tiene forma de determinar a cuál de todos los objetos hacer referencia. Sí pueden acceder a las variables de clase.

Ejemplo:

La actualización del valor de la variable de clase **Maximo**, si bien se le podría pedir a un objeto de la clase para que lo haga desde un método de instancia, es más frecuente que sea responsabilidad de la clase misma y lo realice desde un método de clase.

Retomando el ejemplo con variables de clase:

```
Persona maximo: 10000
```

Se le pide a la clase persona que asuma 10000 como valor máximo para decidir si una persona tiene mucho dinero. El método de clase que lo realiza, tiene la forma de un setter:

```
maximo: unValor
    Maximo := unValor
```

Capítulo 4

Colecciones

- **Bloques de código**
- **Colecciones**
  - Con índice
  - De tamaño variable
  - De tamaño variable con índice
  - Ordenadas
  - Diccionarios
- **Comportamiento común**
  - size
  - do: unBloque
  - detect: unBloque ifNone: otroBloque
  - select: unBloque
  - collect: unBloque
  - inject: valor into: unBloque
  - includes: unObjeto
  - occurrencesOf: unObjeto
- **Ejemplo integrador**

Bloques de código

Los bloques de código son objetos que permiten agrupar un **conjunto de código Smalltalk sin evaluar aún** para ser tratado como una unidad.

El uso más frecuente de los bloques de código es ser enviados como argumentos en los mensajes. El receptor del mensaje decide cuándo, cuántas veces y con qué valores evaluarse, o eventualmente no evaluarlo. En un bloque se pueden expresar tareas simples o tan complejas como se desee, siempre manteniendo la lógica del envío de mensajes.

Los bloques entienden el mensaje **value** que permite evaluar su código y devolver como resultado el último objeto referenciado en su interior.

Se representan encerrando las sentencias entre corchetes **[ ]**.

Ejemplo:

[x := 1. x + 2] value

Se evalúan las dos expresiones y se retorna 3. x es una variable que está declarada fuera del bloque de código.

El método **and**: recibe como segundo argumento un bloque de código.

(a < b) and: [c < d]

La funcionalidad del mensaje consiste en que si el objeto receptor es **true**, es decir si el valor de **a** es menor que el de **b**, evalúa el bloque de código para retornar el valor de verdad de toda la expresión. En cambio, si resultara que el valor que retorna la comparación es **false**, no se requiere la evaluación del bloque para que también el **and:** retorne **false**.

Los bloques también pueden aceptar argumentos que son enviados al momento de evaluar el bloque de código y utilizados en su interior.

Ejemplo:

[:unNumero | unNumero + 2] value: 1

La variables temporal del bloque de código **unNumero** recibe el valor **1** que es enviado como argumento del bloque. Se evalúa el mensaje "**+**" y retorna **3**.

El método **to: do:** recibe un bloque de código como argumento

1 to: 5 do [:unNumero | aux := aux + unNumero]

La tarea del mensaje consiste en evaluar reiteradas veces el bloque de código, enviando como parámetro cada uno de los números entre el receptor y el primer argumento, en este caso, 10 veces, con **unNumero** valiendo entre **1** y **5**. La variable **aux**, suponiendo que inicialmente valía **0**, terminará con un valor **15**

Un bloque también puede tener más de un argumento, manteniendo la misma lógica de funcionamiento.

Ejemplo:

[:variable1 :variable2 | variable1 + variable2] value: 2 value:3

El mensaje recibe tantos argumentos como variables temporales haya, en el mismo orden. En este caso, el resultado es **5**.

Colecciones

Una colección es un objeto que representa a una **estructura de datos** que referencia a un **conjunto de objetos**. Su responsabilidad es contener y manejar un grupo de objetos.

Las colecciones, en su amplia mayoría, pueden **contener objetos de distintas clases**.

Las colecciones varían en las tareas que pueden realizar y en los mensajes que saben responder, por lo tanto, hay distintas clases de colecciones, aunque hay algunas tareas que puede realizar cualquier colección.

Algunas colecciones pueden crecer y decrecer en tamaño. Otras colecciones no pueden cambiarlo, lo que puede ser apropiado cuando el número de elementos en un grupo es conocido y estable.

Hay colecciones que tienen organizados sus elementos. Esta organización puede consistir en un índice numérico que permite acceder a una posición determinada de la colección, una clave de acceso más compleja para búsqueda directa o tener algún criterio de orden interno.

También tienen definidos un abanico amplio de métodos que permiten manipular su contenido evitando el uso de estructuras de control adicionales.

Con índice

Son colecciones de tamaño fijo que contienen a sus objetos indexados por un número entero que va desde 1 hasta la longitud de la colección. Los elementos se guardan según la posición. Estas colecciones no pueden crecer o disminuir en tamaño y los objetos duplicados están permitidos.

Entre ellas, la clase **Array** representa una estructura de datos de **vector** o **arreglo**, con elementos que pueden ser objetos de cualquier clase.

Ejemplo:

```
unArray := Array new: 3
unArray at: 1 put: 5.
unArray at: 2 put: 'hola'.
unArray at: 3 put: true.
```

Se crea un **Array** de **3** elementos, en el que la primera posición referencia al número **5**, la segunda a la palabra **'hola'** y la tercera al valor booleano **true**, todos objetos de diferente clases.

```
col  := #( 1 3 5 7 9)
```

col es una instancia de **Array** de cinco elementos, definido por extensión

```
x := col at: 4.
```

Asigna a la variable **x** el **7**, el objeto ubicado en la 4º posición de la colección.

```
col at: 4  put: 17
```

Modifica la colección, colocando el **17** en la 4º posición, en vez del **7**

col2 := col , #(2 4)

Concatena la colección que recibe el mensaje con la colección que se recibe como argumento. Ninguna de las dos colecciones se modifica, sino que se crea y se devuelve una nueva colección con todos los elementos de ambas, que se guarda en **col2**.

col first

Devuelve **1**, el primer objeto de la colección

col last

Devuelve **9**, el último objeto de la colección

col copyFrom: 2 to: 3

Devuelve una nueva colección con los elementos de la colección comprendidos entre las posiciones enviadas como argumentos, **#(3 5)**.

Muy similar es la clase **String**, con la restricción de contener solamente objetos que son **caracteres**, lo que a su vez le permite realizar un conjunto de tareas que sólo tienen sentido para las cadenas de caracteres.

Ejemplo:

' hola' trimBlanks

Devuelve **'hola'**, la cadena sin los espacios iniciales

'DOSxCUATRO' upTo: $x

Devuelve **'DOS'**, la cadena hasta el primer carácter ingresado

'el sol' subStrings

Devuelve **#('el' 'sol')**, un **Array** con cada palabra como elemento, de acuerdo a los espacios y otros separadores.

De tamaño variable

Son colecciones que actúan como un contenedor dentro del cual se pueden colocar y quitar objetos. No tienen subíndice para acceder a sus elementos por posición. Guarda sus elementos en un orden arbitrario. Puede incrementar o decrementar su tamaño. Sus elementos pueden ser objetos de cualquier clase.

La clase **Bag** es la más básica.

Ejemplo:

unBag := Bag new .

unBag add: 5.

unBag add: 'hola'.

unBag add: true.

unBag add: 5.

unBag remove: true.

En la colección queda contendido dos veces el objeto **5** y una vez la palabra **'hola'**, en alguna lugar dentro de su estructura interna. El objeto **true** fue agregado y luego quitado.

unBag addAll: #(3 4)

Agrega a la colección receptora todos los objetos de la colección que se envía como argumento.

unBag removeAll

Elimina todos los elementos de la colección, quedando vacía. A su vez, devuelve la colección original, con todos sus elementos.

Un **Set** es una colección con las mismas características que el **Bag**, pero con la particularidad de no permitir elementos duplicados. Cuando se quiere agregar un objeto que ya se encuentra en la colección, el objeto no es agregado.

Ejemplo:

```
unSet := Set new .
unSet add: 5.
unSet add: 'hola'.
unSet add: 5.
```

En la colección queda contendido una sola vez el objeto **5** y la palabra **'hola'**. La segunda invocación del **add:** no agregó el objeto **5** por estar ya presente en la colección.

Se puede obtener un **Set** a partir de otra colección cualquiera, con sus mismos elementos, pero por tratarse de un **Set**, con una sola vez cada uno si hay repeticiones.

Ejemplo:

```
unSet := #(2 3 4 3) asSet.
```

unSet es una colección de la clase **Set** con tres elementos: el **2**, el **3** y el **4**.

De tamaño variable con índice

La **OrderedCollection** es una colección que permite recibir un conjunto de mensajes muy completo. Combina las características de las colecciones con subíndice y las de tamaño variable, por lo que puede ser indexada numéricamente para colocar y acceder a los objetos por su posición y también puede crecer y decrecer dinámicamente. Sus elementos pueden ser objetos de cualquier clase y también permite objetos duplicados. Son, en general, las colecciones más utilizadas.

Ejemplo:

```
unaCol := OrderedCollection new.
unaCol add: 5.
unaCol add: 'hola'.
unaCol add: $a.
unaCol at: 1 put: 7.
unaCol remove: $a.
```

La colección queda con dos elementos, en la primera posición se encuentra el **7**, que reemplaza al **5** agregado inicialmente, y en la segunda posición la palabra **'hola'**. El carácter **$a** fue agregado y luego quitado.

Ordenadas

La **SortedCollection** es una colección con las mismas características de una **OrderedCollection**, que permite mantener ordenados los objetos que contiene mediante un criterio de orden. Si no se indica lo contrario, el criterio de orden es de menor a mayor.

Ejemplo:

```
unaCol := SortedCollection new.
unaCol add: 5.
unaCol add: 3.
unaCol add: 4.
```

La colección se mantiene ordenada de menor a mayor a medida que se agregan los objetos. Queda primero el **3**, luego el **4** y por último el **5**.

Para ordenar por otro criterio, se requiere de un bloque de código con dos argumentos y sentencias donde se especifica un criterio de orden entre dos objetos por el cual un objeto quede ubicado delante de otro. La colección guarda el objeto referenciado por el primer argumento delante del objeto referenciado por el segundo argumento en la colección cuando la evaluación del bloque de código devuelve un valor **true**. Una misma colección puede ser ordenada por un criterio en un momento y luego por otro.

Ejemplo:

```
unaCol := SortedCollection new.
unaCol sortBlock: [:a :b | a size > b size]
unaCol add: 'hola'.
unaCol add: 'que'.
unaCol add: 'decis'.
```

La colección se mantiene ordenada de mayor a menor según la longitud de sus objetos a medida que son agregados. Queda primero el 'decis', luego el 'hola' y por último el 'que'.

La forma de ordenar una colección cualquiera, no ordenada, es creando una nueva colección ordenada, instancia de **SortedCollection**, con los mismos elementos que la colección original, indicando un criterio de orden mediante un bloque de código.

Ejemplo:

```
unArray := #(2 4 3)
```

unaCol := unArray asSortedCollection: [:a :b | a > b].

En **unaCol** queda una nueva **SortedCollection** ordenada de mayor a menor, tal como le indica el bloque de código, que contiene primero al **4**, luego al **3** y por último al **2**. **unArray** se mantiene como estaba.

Diccionarios

El **Dictionary** Es una colección que contiene objetos que son incorporados y accedidos mediante una clave asociada a cada uno.

Las claves pueden ser cualquier objeto y no puede haber dos claves iguales en un diccionario.

Ejemplo:

Se asume que existen tres objetos **Persona** creados anteriormente, cada uno con su nombre.

unDiccionarioConPersonas := Dictionary new.

unDiccionarioConPersonas at: (unaPersona nombre) put: unaPersona.

unDiccionarioConPersonas at: (otraPersona nombre) put: otraPersona.

unDiccionarioConPersonas at: (otraPersonaMas nombre) put: otraPersonaMas.

La colección contiene a los tres objetos **Persona**, cada uno en la posición correspondiente según el orden alfabético de sus nombres, utilizados como claves.

Para acceder a los objetos del diccionario se usan las mismas claves utilizadas para guardar los objetos en él.

p := unDiccionarioConPersonas at: 'Juan'

En **p** se referencia al objeto persona cuyo nombre es **'Juan'**, suponiendo que es alguna de las personas agregadas anteriormente a la colección.

Comportamiento común

Existe en el Smalltalk una serie de métodos de uso frecuente que son comunes a toda clase de colecciones.

size

Devuelve la longitud de una colección, es decir la cantidad de objetos en la colección.

Ejemplo:

#(1 2 3 2 1) size

Retorna **5**

do: unBloque

Evalúa el bloque de código que recibe como argumento para cada uno de los objetos de la colección. Por lo tanto se ejecuta el conjunto de líneas del bloque tantas veces como objetos haya en la colección, siendo cada objeto de la colección el argumento con que se evalúa el bloque.

Ejemplo:

suma := 0.

#(1 2 3 4 5) do: [:unNumero | suma := suma + unNumero].

Se ejecuta el bloque de código cinco veces, con la variable temporal **unNumero** referenciando a cada objeto de la colección. En la variable suma queda el valor **15**, resultado de la suma de todos los objetos de la colección.

detect: unBloque ifNone: otroBloque

Devuelve el primer objeto de la colección para el cual la evaluación del argumento, que es un bloque de código, es verdadera. Si para ninguno de los elementos resulta verdadera, retorna el valor del segundo argumento, que también es un bloque de código.

Ejemplo:

#(1 2 3 4 5) detect: [:nro | nro > 3] ifNone: [3].

Retorna el número **4** que es el primero mayor que **3**.

#(1 2) detect: [:nro | nro > 3] ifNone: [3].

Al no encontrar ningún elemento mayor que **3**, retorna el número **3**.

select: unBloque

Devuelve una colección conteniendo los objetos de la colección original para los cuales la evaluación del bloque de código, es verdadera. Si para ninguno de los elementos resulta verdadera, retorna una colección vacía. La colección no se modifica, sino que se devuelve una nueva colección de la misma clase.

Ejemplo:

'hola ' select: [:letra | letra isVowel]

Devuelve **'oa'**.

#(1 2 3 4 5') select: [:nro | nro > 10]

Devuelve una colección vacía.

collect: unBloque

Devuelve una nueva colección conteniendo los objetos resultantes de evaluar el bloque de código que se envía como argumento, para cada uno de los objetos

de la colección. La colección no se modifica, sino que se devuelve una nueva colección de la misma clase.

Ejemplo:

#('las' 'longitudes' 'de' 'las' 'palabras') collect: [:obj | obj size]

Devuelve **#(3 10 2 3 8)**

inject: valor into: unBloque

El bloque que se recibe, que debe tener dos argumentos, se evalúa para todos los objetos de la colección. Uno de ellos, el segundo, es cada uno de los elementos de la colección, el otro es un valor que va cambiando en cada evaluación. Toma como valor inicial el primer argumento del mensaje, y en cada evaluación toma el resultado de la evaluación anterior del bloque. Su valor luego de la última evaluación es devuelto por el mensaje como resultado

Ejemplo:

#(1 2 3 4 5) inject: 0 into: [:resul :elem | resul + elem]

Devulve **15**, que es la sumatoria de todos los elementos. El valor inicial para **resul** fue el **0** pasado como primer argumento. Luego de la primer evaluación su valor es **1** (**resul** que era **0** más **elem** que era **1**), en la siguiente evaluación cambio su valor a **3**, ya que **resul** vale **1** y **elem 2**. Así sucesivamente hasta que se suman todos los elementos de colección.

includes: unObjeto

Devuelve si el objeto enviado como argumento se encuentra en la colección. Para buscar, utiliza como criterio la igualdad.

Ejemplo:

#(2 3 4) includes: 4.

Devuelve true.

occurrencesOf: unObjeto

Devuelve la cantidad de veces que el objeto enviado como argumento se encuentra en la colección. Para buscar, utiliza como criterio la igualdad.

Ejemplo:

#(2 3 4 3) occurrencesOf: 3.

Devuelve **2**.

Ejemplo integrador

En un taller mecánico, donde trabajan varios mecánicos, se hacen reparaciones a automóviles en las que se utilizan diferentes repuestos. Se quiere calcular:

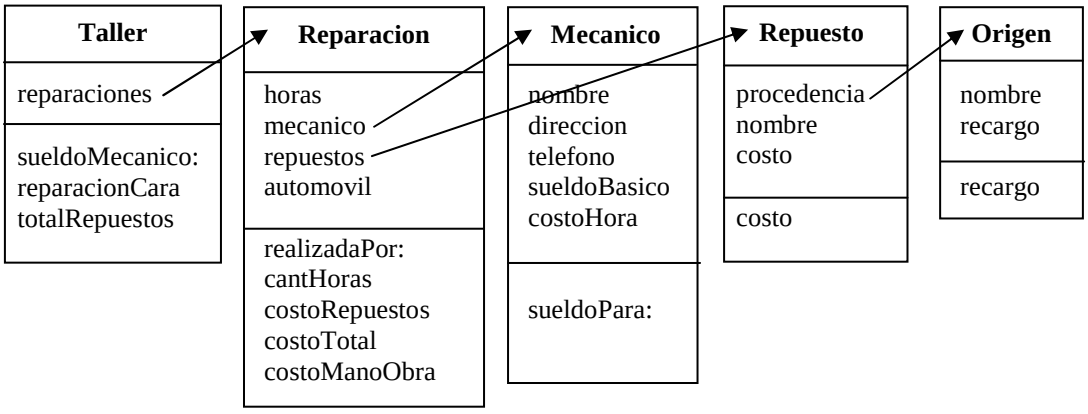
El sueldo a pagar a un mecánico dado. El sueldo del mecánico se calcula como el costo de las horas que trabajó en las diferentes reparaciones, más un importe correspondiente a un sueldo básico acordado entre el dueño y cada mecánico.

La suma del costo de los repuestos de todas las reparaciones hechas en el taller. Los repuestos tienen costo básico más un recargo según el país de origen: si son países limítrofes es del 18%, si es del resto de América, el 25% y si son de Europa el 28%. Los nacionales tienen un 10%. En estos momentos no se están recibiendo repuestos de otros lugares, pero se estima puede suceder en un futuro cercano.

Cuál es la reparación más cara que se hizo en el taller. En cada reparación a un automóvil trabaja un mecánico una cierta cantidad de horas y se utilizan un conjunto de repuestos. El costo total de la reparación se conforma por el costo de los repuestos más el costo de las horas de trabajo del mecánico (cada mecánico cobran un valor distinto por hora), más una comisión del 30% sobre el total.

Solución:

Una solución posible, puede implementarse con las siguientes clases



La clase **Taller** representa al taller mecánico. Mediante la colección **reparaciones** conoce a todas las reparaciones que se hicieron en el taller. Habrá un solo objeto de esta clase y será su responsabilidad realizar las tareas solicitadas. Probablemente haya más información que el taller debiera conocer, como por ejemplo los mecánicos, pero para las tareas solicitadas es suficiente.

De la clase **Reparación** hay muchas instancias, una para cada reparación que se hizo en el taller. Todas ellas están en la colección de reparaciones del taller. De cada reparación se sabe cuantas horas duró, el mecánico que la realizó, una colección con todos los repuestos utilizados y el automóvil que fue reparado.

En la clase **Mecanico** está la información propia de cada mecánico: los datos básicos de nombre dirección y teléfono, y los valores necesarios para los cálculos, el sueldo básico mensual y lo que cobra por hora de trabajo. Hay tantos objetos como mecánicos haya en el taller. Es de esperar que sean muchos menos que las reparaciones. Todas las reparaciones de un mismo mecánico, en el atributo **mecanico** referencian al mismo objeto.

La clase **Repuesto** indica los atributos de todos los repuestos: un nombre, sólo a fines descriptivos, el costo y el lugar de procedencia, que se trata de un objeto de la clase **Origen**, donde se indica el nombre del lugar de origen y el porcentaje de recargo correspondiente. El recargo no es en sí un atributo del repuesto, sino de su lugar de procedencia. Repuestos puede haber muchos, orígenes, habrá los 4 indicados como ejemplo en el planteo inicial y si es necesario agregar otros se crearán nuevos objetos, sin modificar la estructura de clases.

Para calcular el sueldo de un mecánico, en la clase **Taller** se define el siguiente método

```
sueldoMecanico: unMecanico
    | reps horas |
    total := 0.
    reps := reparaciones select: [:repa | repa realizadaPor: unMecanico ].
    reps do: [:repa | horas := horas + repa cantHoras ].
    ^ unMecanico sueldoPara: horas
```

El método recibe como argumento **unMecanico** al que se quiere calcular su sueldo. Primero, de todas las reparaciones del taller se seleccionan las que han sido realizadas por **unMecanico**, mediante el mensaje **select**:, y se las guarda en la colección **reps**. Luego se recorre la colección **reps** y se suma el total de horas trabajadas, que se guarda en la variable **horas**. Con ese valor, el objeto **unMecanico** puede calcular su sueldo, lo que se le solicita mediante el mensaje **sueldoPara:.**

Para saber si la reparación era del mecánico en cuestión, a cada reparación se le envió un mensaje que fue respondido por el siguiente método de la clase **Reparacion**.

```
realizadaPor: unMec
    ^ mecanico = unMec
```

También en la clase **Reparacion** debe estar definido el método "getter" que devuelve la cantidad de horas

```
cantHoras
    ^ horas
```

Por último, el método **sueldoPara**: ubicado en la clase **Mecanico**, que realiza el cálculo utilizando sus propias variables de instancia y el argumento con la cantidad de horas.

```
sueldoPara: horas
    ^ sueldoBasico + horas * costoHora
```

Para resolver el pedido del total de repuestos del taller, en la clase **Taller** se define el siguiente método, cuya única tarea es acumular el costo de los repuestos de todas las reparaciones a partir de pedirle a cada reparación el costo de sus repuestos.

```
totalRepuestos
    ^reparaciones inject: 0 into: [ :total :repa | total + repa costoRepuestos ].
```

En forma similar, cada reparación acumula el costo de sus repuestos, preguntándole a cada repuesto por su costo.

```
costoRepuestos
    ^repuestos inject: 0 into: [:total :repu | total + repu costo ].
```

En la clase **Repuesto** se resuelve el cálculo de su costo, delegando a su vez en el lugar de procedencia.

```
costo
    ^ costo * (1+ procedencia recargo)
```

En la clase **Origen**, sólo resta el método "getter" que devuelve el recargo.

```
recargo
    ^recargo
```

Para el último pedido, de la reparación más cara, el método **reparacionCara** ubicado en la clase Taller tiene por estrategia ordenar todas las reparaciones de acuerdo a su costo y tomar la primera.

```
reparacionCara
    ^ (reparaciones asSortedCollection: [:a :b | a costoTotal > b costoTotal ] )
        first
```

En cada reparación, el costo se calcula sumando el costo de los repuestos (que se obtiene delegando en el método **costoRepuestos**, ya realizado), más el costo de mando de obra (que se delega en otro método de la clase **Reparación**) y aplicando el 30% de incremento.

```
costoTotal
    ^ (self costoRepuestos + self costoManoOtra ) * 1.30
```

Para terminar, el método que se ocupa de realizar el producto de las horas trabajadas por el costo horario del mecánico.

```
costoManoOtra
    ^mecanico costoHora * horas
```

Capítulo 5

Polimorfismo

- **Polimorfismo**
- **Enlace dinámico**
- **Ejemplo integrador**

Polimorfismo

El objetivo general del **polimorfismo** es desarrollar objetos con métodos genéricos que puedan trabajar indistintamente con diferentes objetos, que esos objetos sean intercambiables.

Hay polimorfismo cuando un objeto puede enviar un mismo mensaje a diferentes objetos y todos ellos responden adecuadamente. Para ello, cada objeto tiene la posibilidad de tener definidos métodos con la misma interfaz, pero con una propia implementación. Un objeto puede interactuar con cualquiera de los otros objetos de acuerdo a las características de la interfaz común, y el objeto receptor realizará la tarea solicitada de acuerdo a la propia implementación que tenga definida, independientemente de las otras implementaciones que tengan los otros objetos.

Para responder al mensaje, los diferentes objetos, siendo instancias de diferentes clases, se remiten a la suya para ejecutar el método correspondiente. Por lo tanto, diferentes clases pueden tener métodos con el mismo nombre e implementaciones diferentes.

Desde el punto de vista del objeto que invoca, el proceso es transparente, y es en definitiva éste, el que se ve beneficiado por el polimorfismo. Tampoco le interesa saber si ante la misma invocación hecha a diferentes objetos, la forma en que cada uno de ellos respondió, de acuerdo a su implementación, fue la misma o no.

El polimorfismo permite a un objeto comunicarse con otros objetos sin tener que saber de qué clase son, sólo necesita que dichos objetos entiendan el mensaje indicado, o sea, que tengan acceso a la implementación de un método del mismo nombre. En otras palabras, dos objetos son polimórficos si puedo intercambiarlos sin que el emisor del mensaje perciba cambios en la manipulación de esos objetos para un fin específico. Es decir, que para desde el punto de vista del objeto emisor, puede tener como receptores del mensaje a

diferentes objetos en diferentes momentos de la ejecución del programa, siempre y cuando todos ellos tengan implementados sus correspondientes métodos.

El **polimorfismo** permite a los programadores separar las cosas que cambian de las que no cambian, lo genérico de lo particular, y de esta manera hacer más fácil la ampliación, el mantenimiento y la reutilización de los programas.

Ejemplo:

Un objeto **unGato,** instancia de la clase **Gato** tiene definido un método llamado **edad**, con su correspondiente codificación, y simultáneamente otro objeto, como puede ser **unPerro**, instancia de la clase **Perro** también tiene definido otro método llamado **edad**. La interfaz de ambos métodos es la misma, pero la implementación no tiene porqué coincidir.

```
unGato edad
```

Ante esta invocación se ejecuta el método correspondiente al a persona, ya que el receptor sabe su identidad. En este caso, el método puede ser tan simple como retornar una variable de instancia que contenga esa información.

```
edad
    ^edad
```

Otra invocación posible puede ser:

```
unPerro edad
```

En este caso el objeto receptor también conoce su identidad y va a ejecutar su propio método **edad**, que puede tener un cálculo a partir de su fecha de nacimiento y la fecha actual.

```
edad
    ^Date today yearsSince: fechaNacimiento
```

En una situación cualquiera se encuentra la siguiente invocación

```
x edad
```

En respuesta al mensaje, el objeto **x** responderá con el método de la clase de la que sea instancia, ya sea un perro o un gato. Para esta porción de código, no es necesario distinguir si se trata de un objeto de la clase **Perro** o **Gato**.

Ejemplo:

Complejizando el ejemplo anterior, se agregar un objeto Persona al que se le desea preguntar además si considera vieja a la mascota que tiene.

El diagrama de clases puede ser

Persona
mascota maximo
suMascotaEsVieja

Perro
fechaNacimiento
edad

Gato
edad
edad

El atributo **mascota** referencia a un objeto perro o gato, según lo que cada persona tenga. El atributo **maximo** indica el valor a partir del cual considera vieja a su mascota. Los métodos y variables de perros y gatos son los del ejemplo anterior.

A la persona se le pregunta

unaPersona suMascotaEsVieja

que para responder ejecuta el método

suMascotaEsVieja

^mascota edad > maximo

En este método se ve la importancia del polimorfismo. La persona interactúa con la mascota que sea que tenga, sin importarle que sea un gato o un perro.

En un futuro podría haber muchas otras clases de animales que pudieran ser mascotas de una persona. Siempre y cuanto tengan implementado el comportamiento para responder al mensaje **edad**, todo seguirá funcionando sin un mínimo cambio en la implementación de la clase **Persona**

Enlace dinámico

Cuando se envía un mensaje a un objeto el lenguaje debe decidir durante la ejecución del programa cuál método se ejecuta, lo que se denomina **enlace dinámico**. El polimorfismo se basa en que el enlace que se establece entre la llamada a un método y el código que efectivamente se asocia con dicha llamada se produce en tiempo de ejecución.

Ejemplo integrador

Una consultora quiere calcular lo que tiene que cobrarle a los clientes y lo que debe pagar a sus empleados.

La consultora cuenta con un plantel de programadores y analistas, que desarrollan tareas a distintos clientes que contratan los servicios de la consultora.

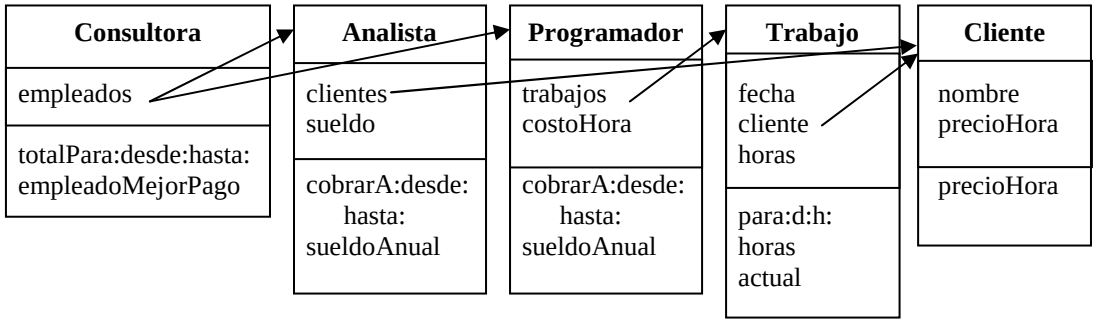
Los programadores van cambiando de cliente para el que trabajan, por lo que se registra cuidadosamente cuantas horas trabajó cada uno para cada cliente de la consultora. Los analistas tienen asignado un conjunto de clientes para los que habitualmente trabajan, pero no hay un registro detallado. Puede haber más de un analista para el mismo cliente.

Los analistas cobran un sueldo mensual, mientras que los programadores cobran un monto por hora trabajada, que es fijo para cada uno, independientemente de para qué cliente trabajaron.

Para cobrarle a sus clientes, el monto se calcula para un plazo indicado. Se le cobra el total de horas que los programadores trabajaron para ese cliente, valuadas al monto acordado con cada cliente, más el 1% del sueldo mensual por cada día del plazo, de los analistas que tienen asignados al cliente.

En concreto, se desea saber cuánto se le tiene que cobrar a un cliente dado y cuál es el empleado mejor pago en lo que va del presente año.

Solución:



La consultora conoce a todos sus empleados y los tiene juntos en una colección. Allí están mezclados objetos de la clase **Analista** y **Programador**.

De cada analista esta registrado el sueldo mensual y una colección con los clientes que tiene asignado. De cada programador se tiene el costo de la hora de trabajo y una colección con objetos de la clase **Trabajo**, que representan el tiempo trabajado por un programador para un cliente en un día. Por lo tanto, en cada objeto trabajo se referencia a un cliente.

Para saber cuanto cobrarle a un cliente, está en la clase **Consultora** el siguiente método, que recibe como argumento a un objeto cliente y dos fechas.

```
totalPara: unCliente desde: unDia hasta: otroDia
    ^empleados inject: 0 into:
        [ :tot :emp | tot + (emp cobrarA: unCliente desde: unDia hasta: otroDia ) ].
```

Lo que hace es tener en cuenta a todos sus empleados y sumar lo que le corresponda según el trabajo de cada uno en relación a dicho cliente. Dicha tarea la delega en cada empleado. Lo hace polimórficamente, preguntando lo mismo a todos los objetos de la colección, sin distinguir si se trata de analistas o programadores.

Para el caso de cada analista, el calculo depende en primer instancia si se trata de uno de los clientes que tiene asignado, fijándose si el cliente que se recibe como argumento está incluido en su propia colección de clientes. Si es así, calcula el porcentaje de acuerdo a la cantidad de días y a su sueldo. Si no, retorna 0.

```
cobrarA: unCliente desde: unDia hasta: otroDia
    (clientes includes: unCliente ) ifTrue:
        [ ^(unDia substractDate: otroDia) * 0,01 * sueldo ]
    ^0
```

Para el caso de cada programador, se seleccionan todos los trabajos realizados por el para el cliente entre las fechas indicadas, se suman las horas y se las multiplica por el precio de la hora correspondiente al cliente.

```
cobrarA: unCliente desde: unDia hasta: otroDia
    | trabs |
    trabs:= trabajos select: [ :tr |  tr para: unCliente d: unDia h: otroDia ].
    ^(trabs inject: 0 into: [ :total :tr | total + tr horas] ) * unCliente precioHora
```

Para completar los métodos, en la clase **Trabajo** se encuentra el siguiente método, que devuelve si es un trabajo a tener en cuenta o no.

```
para: unCliente d: unDia h: otroDia
    ^ (fecha between: unDia and: otroDia) and: [ cliente = unCliente]
```

También un método getter que devuelva la cantidad de horas trabajadas

```
horas
    ^horas
```

Y por último otro getter para el precio por hora del cliente

```
precioHora
    ^precioHora
```

Para obtener el empleado mejor pago, el siguiente método implementado en la clase consultora ordena la colección de todos los empleados, sin importar que haya dentro analistas y programadores, ya que lo hace enviándoles un mensaje polimórfico que calcula el sueldo de cada uno en el año.

```
empleadoMejorPago
    (empleados asSortedCollection:
        [:e1 :e2 | e1 sueldoAnual > e2 sueldoAnual ] ) first.
```

Para el analista, el sueldo anual corresponde al sueldo mensual por la cantidad de meses que van del año.

```
sueldoAnual
    ^ sueldo * Date today monthIndex
```

Para el programador, el sueldo anual se calcula seleccionando todos los trabajos del año, sumando las horas de cada uno de ellos y multiplicándolo por su propio valor de la hora.

```
sueldoAnual
    ( (trabajos select: [ :trab |  trab actual ] )
        inject: 0 into:  [ :tot :trab | tot + trab horas ] ) * costoHora
```

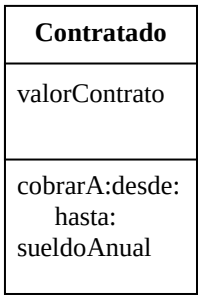
Se debe agregar en la clase **Trabajo** el método que permite saber si el trabajo corresponde al año actual.

```
actual
    ^ fecha year = Date today year
```

En un futuro, la consultora amplia su planta de empleados e incorpora a personal contratado. Se firma con cada uno un contrato de enero a diciembre por un monto global que se les va pagando mensualmente de a partes. A los clientes no se les cobra por el trabajo que ellos puedan hacer.

Nueva solución:

De lo realizado, no se modifica nada. Sólo se agrega una nueva clase para el personal contratado y a sus objetos se los guarda en la misma colección de empleados de la consultora junto con todos los demás.



Lo que es importante, para mantener el polimorfismo y que sean tratados indistintamente por la consultora para hacer sus cálculos, es que los nuevos objetos puedan responder a todo lo que respondían los objetos analistas y programadores.

Por un lado, el método **sueldoAnual**, que se calcula de la siguiente manera

```
sueldoAnual

    ^ valorContrato / 12 * Date today monthIndex
```

Por último, para que no influya en el total a cobrar al cliente, este método devuelve 0 sin importar ninguna condición.

```
cobrarA: unCliente desde: unDia hasta: otroDia

    ^0
```

Si cualquiera de estos métodos no estuviera, cuando desde la consultora se evalúa a todos los objetos de la colección de empleados enviándoles los mensajes anteriores, funcionaría para analistas y programadores, pero cuando apareciese el primer contratado se produciría un error.

Capítulo 6

Herencia

- **Herencia**
  - Variables
  - Métodos
  - Mecanismo de búsqueda
- **Redefinición**
  - Uso de super
- **Clases abstractas**
- **Herencia simple y múltiple**
- **Ejemplo integrador**

Herencia

La herencia permite facilitar la extensibilidad de un programa existente, de manera que realice nuevas funcionalidades aprovechando las actuales. La herencia permite que se puedan **definir nuevas clases basadas en clases existentes**, lo que facilita reutilizar código previamente desarrollado.

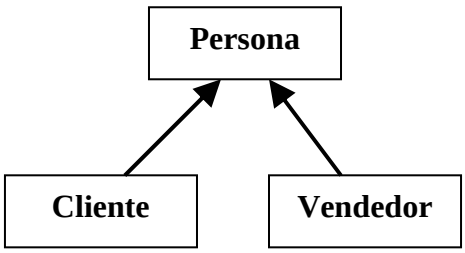
Si una clase hereda de otra tiene como propios todas sus variables y métodos. La clase que hereda puede agregar nuevas variables y métodos y modificar los métodos heredados, redefiniéndolos.

Se dice que la nueva clase es "subclase" de la ya existente y, en forma inversa, ésta es "superclase" de aquélla. De esta manera, los objetos de una subclase asumen el comportamiento y la estructura de información de su superclase y poseen además todas las características definidas en la propia clase. Las clases se organizan jerárquicamente siendo unas subclases de otras, de una manera que optimiza la organización de la información y la implementación de los métodos. La herencia dentro de la jerarquía de clases es ilimitada, por lo que las subclases pueden tener subclases que también tengan subclases y así sucesivamente. La subclase toma el comportamiento y la estructura interna de todas sus superclases.

La herencia permite realizar un doble movimiento de generalización y especialización. Por un lado, una superclase de varias clases permite generalizar agrupando sus variables y métodos comunes. Por otro, teniendo definida una clase, la creación de una nueva subclase permite obtener objetos que se especialicen en algún comportamiento o propiedad adicional, que los diferencia de los objetos de la otra clase, que carecen de ella. De esta manera, para construir la jerarquía de clases hay que tener en cuenta que las clases van de lo más general a lo más particular.

Ejemplo:

Existiendo una clase **Persona**, que representa objetos con sus propios métodos y variables, como pueden ser el nombre y la edad, se puede definir a partir de ella una nueva clase llamada **Cliente**, en la que sólo se define la información y comportamiento específico de los clientes, como un dato sobre su situación impositiva o un método que permita realizar una compra, y asume por herencia todos los métodos y variables de la clase **Persona**. De similar manera, se puede definir una clase **Vendedor**, cuyos objetos también tenga todas las características de una persona, pero se le agreguen las características propias que debe tener todo vendedor.



Variables

Las subclases hereden las variables de instancia de sus superclases y pueden poseer variables propias. Para los objetos de la subclase, todas las variables son consideradas de igual manera independientemente de si fueron heredadas o no.

Mensajes y métodos

La herencia de métodos es útil para permitir a una clase aprovechar el comportamiento de su superclase. Los objetos de la subclase pueden ejecutar todos los métodos definidos en la superclase, es decir, que todos los mensajes que se le envían a los objetos de las superclases se pueden enviar también a los objetos de la subclase y ser respondidos de igual manera. Además, los objetos de la subclase pueden tener nuevos métodos, por lo que el comportamiento puede ser mayor, o redefinir los métodos de la superclase, por lo que puede ser diferente.

Mecanismo de búsqueda

Todo objeto de la subclase está habilitado a ejecutar los métodos definidos en ella además todos los métodos de sus superclases. Cuando un objeto recibe un mensaje busca en primer lugar en su propia clase si existe el correspondiente

método. Si no lo encuentra, busca en la superclase inmediata y así sucesivamente va recorriendo la jerarquía, pasando de subclase a superclase, hasta que encuentra en alguna el método con ese nombre. Se produce un error si en la clase más general, Object, no se encuentra un método para responder al mensaje.

Herencia y polimorfismo

La herencia potencia el polimorfismo, lo amplia. Objetos de diferentes clases pueden entender un mismo mensaje, es decir, comportarse polimórficamente, sin necesidad de que en cada una de sus clases haya una implementación para el método en cuestión. Si se da el caso en que todas estas clases son subclases de otra, y en esta otra se encuentra implementado el método, es suficiente para que funcione el polimorfismo, ya que todos los objetos de las subclases, gracias a la herencia, pueden ejecutar al método.

Redefinición

La redefinición se produce cuando una clase **vuelve a definir**, o sea redefine, **alguno de los métodos heredados** de su superclase. El nuevo método sustituye al heredado para todos los objetos de la clase que lo ha redefinido, de manera que sus objetos tienen un comportamiento modificado respecto de los objetos de la superclase.

Así, la redefinición permite que al definir una nueva clase sus objetos no sólo extiendan o amplíen el funcionamiento de los objetos de la superclase, sino también los modifiquen, ajustándolo a los requerimientos y necesidades específicas para los cuales se creó la subclase.

El mecanismo de búsqueda que permite que la herencia funcione es el que garantiza que la redefinición de métodos no genere ambigüedades o conflictos a la hora de establecer qué método ejecutar. Cuando un objeto recibe un mensaje, para responder, busca la implementación del método de acuerdo al mecanismo ya explicado. Si se trata de un método redefinido en varias clases, como lo búsqueda la realiza recorriendo todas las clases desde las subclases hacia las superclases, apenas encuentra en una de ellas el método requerido lo ejecuta sin seguir buscando, o sea, sin enterarse siquiera que el método estaba también definido en otra superclase.

Uso de super

Es frecuente que cuando se redefine un método para modificar su comportamiento en una subclase sea de utilidad el método de la superclase, para lo cual, dentro del nuevo método, se puede **invocar al método de igual nombre de la superclase**.

La palabra reservada **super** representa al **objeto receptor del mensaje**, pero causa que **la búsqueda del método a ejecutar comience por la superclase** del objeto receptor.

Dentro de la codificación de un método, **super** actúa como receptor de un mensaje, al igual que **self**[2], referencia que el objeto receptor del nuevo mensaje es el mismo del que se está ejecutando, pero con la diferencia que la búsqueda del método invocado no comienza en la misma clase, sino en la superclase del objeto receptor.

Super es la forma en que permite invocar desde un método de una subclase al método de la superclase que se está redefiniendo.

Clase abstracta

Una clase abstracta es **una clase de la que no se crean objetos**, es decir, que nunca se la va a instanciar. Lejos de que esta característica parezca convertirlas en clases sin sentido, las clases abstractas, como todas las clases que tienen subclases, permiten proporcionar una estructura de datos y un comportamiento de utilidad general que puede ser utilizado por todas las subclases que hereden de ella.

No existe ninguna expresión para predeterminar que una clase sea abstracta, Las clases abstractas proveen de un comportamiento a sus subclases, pero no se definen con el propósito de tener sus propias instancias.

Ejemplo:

Object

Es la clase abstracta de cual heredan todas las clases

Collection

Es una clase abstracta, ya que todas las colecciones son instancia de alguna de sus subclases, pero nunca de si misma

Herencia simple y múltiple

Existen dos tipos de herencia denominada simple o múltiple. La herencia simple plantea que una clase puede tener una y sólo una superclase, mientras que la herencia múltiple consiste en permitir que una clase herede de dos o más superclases simultáneamente. La herencia múltiple trae aparejada posibles complicaciones de ambigüedad, como por ejemplo la herencia simultánea de métodos polimórficos de diferentes superclases, que en algunos lenguajes son resueltos con esquemas fuertemente tipados o con estrategias que no son propias del paradigma. Por lo tanto, Smalltalk, al igual que otros lenguajes, soporta sólo herencia simple.

[2] Ver capítulo 2 "Objetos y Mensajes" Mensajes de un objeto a sí mismo.

Herencia en métodos y variables de clase

Los métodos de clase se heredan, es decir, que todo mensaje que se le envía a una clase, también se le puede enviar a sus subclases y la forma de respuesta sería la misma.

Por su parte, también se pueden redefinir, para ello, en el caso que se quiera reutilizar el método que se está redefiniendo, se puede utilizar la palabra reservada **super** para invocar al método de clase original.

Ejemplo:

```
new
    |obj|
    obj := super new.
    obj inicializar.
    ^obj
```

Si este método está definido en la clase **Persona**, cuando se invoque

```
Persona new.
```

En vez de ejecutarse el método **new** estandar, por herencia, se ejecuta el presente método, que lo redefine. Sin embargo, este método necesita invocar al método original, cosa que hace mediante el envío del mensaje

```
super new.
```

super referencia a la clase **Persona**, pero en vez de que el método que se ejecute sea el **new** de la misma **clase** (lo que provocaría una invocación recursiva infinita), se invoca al método **new** de la **superClase.** El receptor sigue siendo la clase **Persona**, que es lo que se quiere crear, pero el método que se ejecuta se busca en la superclase, que en este ejemplo no importa cuál es porque en definitiva se hereda del método **new** estandar.

Ante la presencia de subclases, las variables de clase, además de ser compartidas (consultadas o modificadas) por todos los objetos de su clase, también lo son con todos los objetos de las subclases de su clase. Sigue existiendo un único valor para la variable de clase, la diferencia es que potencialmente, más y distintos objetos la pueden utilizar.

Ejemplo integrador

Se trata de un sistema de control de acceso al subterráneo con tarjetas. Para pasar por los molinetes e ingresar al subte, se utilizan tarjetas de cartón que pueden tener una validez de entre 1 y 10 viajes. También hay tarjetas recargables de material plástico más duradero, donde se carga un monto de dinero, con dos modalidades: recarga en efectivo o débito automático.

A las tarjetas recargables se les descuenta el importe de cada viaje según la tarifa al momento de su uso, mientras que a las tarjetas de cartón, que se abonan previamente, no importa la tarifa al momento del viaje, ya que su capacidad

está medida en viajes. Quienes llevan tarjetas de cartón no pueden ingresar al subte si se les acaba la cantidad de viajes disponibles. Cuando las tarjetas de recarga manual se quedan sin saldo suficiente para pagar el viaje, la persona no puede ingresar.

Las tarjetas recargables con débito automático están asociadas a una cuenta bancaria desde donde se hace el débito. Cuando quieren ingresar y no tienen saldo suficiente, la tarjeta automáticamente se recarga con el importe equivalente a 30 viajes, que se debitan de la cuenta. Si no hubiera saldo en la cuenta se impide el paso de la persona.

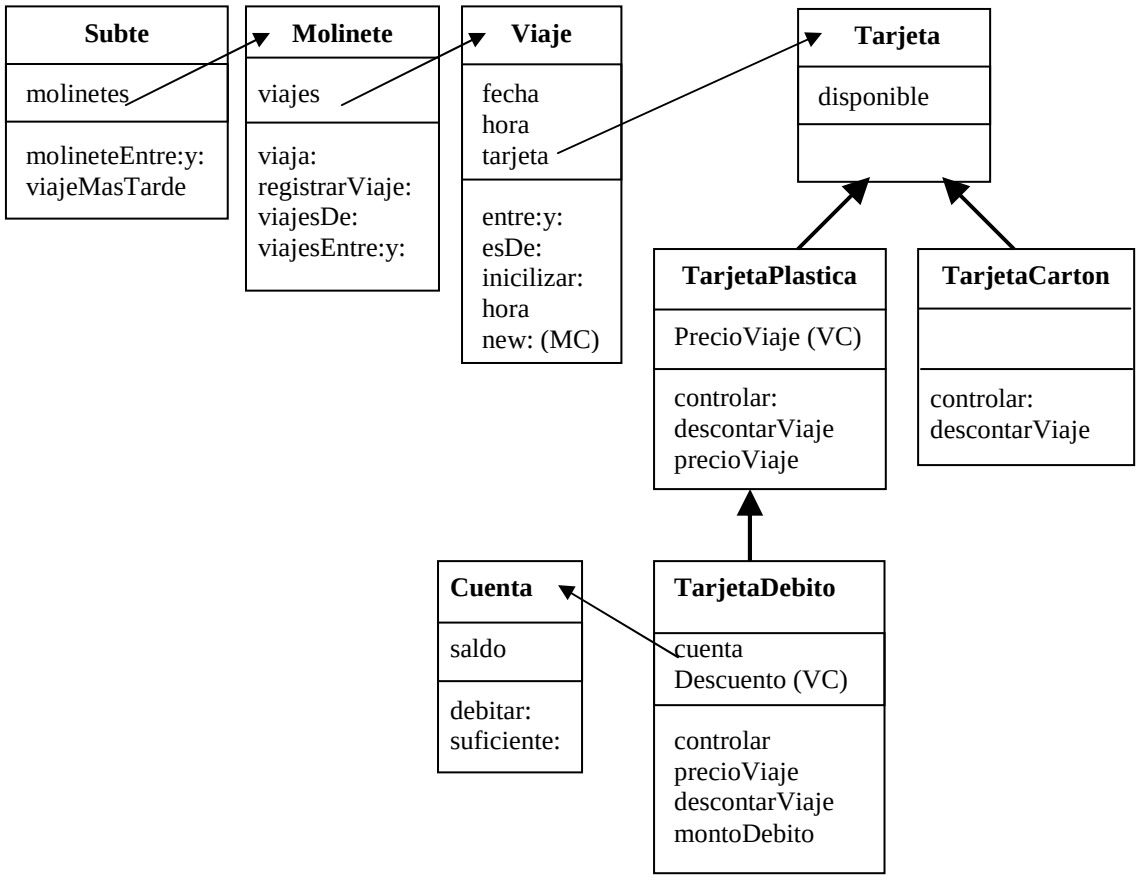
Todos los molinetes están habilitados para cualquiera de los dispositivos. El sistema lleva el registro de todos los viajes realizados.

La tarifa de un viaje en subte es actualmente de $0,90, pero es un valor que va cambiando. Actualmente, hay un descuento del 5% para los viajes hechos con tarjetas recargables con débito automático.

Se debe realizar

- Controlar cuando una persona atraviesa un molinete con su tarjeta, actualizando los saldos y registrando toda la información necesaria.
- Devolver el molinete por el que más personas pasaron entre dos fechas.
- La hora más tarde en que se hizo un viaje con una tarjeta, en cualquiera de los molinetes

Solución:



La clase **Subte**, de la que probablemente habrá una sola instancia, tiene por atributo una colección de todos los molinetes.

Cada objeto de la clase **Molinete** tiene una colección con los viajes que se realizaron en dicho molinete.

Un viaje está representado por objetos con tres atributos: la fecha y hora del viaje y la tarjeta con la que se lo hizo.

Hay tres clases de tarjetas de las cuales va a haber instancias, **TarjetaPlastica**, **TarjetaCarton** y **TarjetaDebito**, mientras que la clase **Tarjeta** es abstracta y se definió con el fin de agrupar el comportamiento y la información común, en este caso el disponible.

Las tarjetas de cartón heredan el atributo **disponible** y lo utilizan para representar la cantidad de viajes disponibles. En cambio, las otras tarjetas, que también lo heredan, lo usan para guardar el saldo disponible, expresado en dinero. El atributo tiene el mismo nombre y representa lógicamente lo mismo, pero en cada clase se lo usa con diferentes métodos.

Las tarjetas plásticas tienen el valor del viaje como variable de clase y un método para devolverlo. Las tarjetas recargables con débito automático tienen lo mismo y además la información de la cuenta bancaria. En particular, van a redefinir parte del comportamiento de las tarjetas plásticas.

Por ultimo, la clase **Cuenta** representa una cuenta bancaria conocida desde la tarjeta, tiene un saldo y comportamiento para operar con ese saldo.

El siguiente método, ubicado en la clase molinete, realiza el control de la tarjeta y la actualización de los valores. Recibe como argumento a **unaTarjeta** y lo primero que hace es delegar en ella el control de si puede pasar, con el mensaje **controlar**. Si no puede, se sale del método devolviendo **false**. En caso que pueda pasar se actualiza la tarjeta enviándole el mensaje **descontarViaje** y se registran los datos del viaje realizado en el mismo molinete con el mensaje **registrarViaje**, pasando la tarjeta como argumento.

```
viaja: unaTarjeta
    (unaTarjeta controlar) ifFalse: [ ^ false ].
    unaTarjeta descontarViaje.
    self registrarViaje: unaTarjeta.
    ^ true
```

A cualquiera de las tarjetas se les preguntaba de la misma manera con el mensaje controlar, sin tener que saber de antemano de qué clase de tarjeta se trata. Los controles son diferentes según la clase de tarjeta. Para las de cartón, se debe controlar si tiene viajes disponibles

```
controlar
    ^ disponible > 0
```

Las tarjetas plásticas manuales se fijan si alcanza el saldo para el precio del viaje.

```
controlar
    ^ disponible >= self precioViaje.
```

Las tarjetas plásticas recargables redefine el método **controlar.** Además de fijarse el saldo como las de recarga manual, en caso que no sea suficiente, se fijan si hay saldo en la cuenta bancaria. Para ello aprovechan la implementación de la superclase, mediante el envío del mensaje **controlar** a **super**.

```
controlar
    ^ super controlar or: [ cuenta suficiente: self montoDebito ]
```

A su vez, en la clase **Cuenta**, hay un método que controla si hay un saldo suficiente.

```
suficiente: unaCantidad
    ^ saldo >= unaCantidad
```

Para el cálculo del precio del viaje, en la clase **TarjetaPlastico** el método se limita a devolver una variable de clase con el valor del viaje.

```
precioViaje
    ^ PrecioViaje
```

En cambio, para la clase **TarjetaDebito**, como hay descuento, se hace el cálculo correspondiente mediante una redefinición del método, en el que usando **super** se aprovecha la implementación de la superclase.

```
precioViaje
    ^ super precioViaje * (1 -  Descuento)
```

El monto a debitar automáticamente de la cuenta bancaria, se realiza con este método, ubicado en la clase **TarjetaDebito**.

```
montoDebito
    ^ 30 * self precioViaje
```

Por otro lado, la actualización de los viajes realizados en las tarjetas, también se resuelve polimórficamente, con distintas implementaciones en las clases. Para las tarjetas de cartón es sólo restar un viaje.

```
descontarViaje
    disponible := disponible - 1
```

Para las tarjetas de plástico, el método consiste en decrementar del saldo el valor del viaje. El envío del mensaje **precioViaje** a **self**, delega el cálculo en el método ya realizado de las tarjetas de plástico.

```
descontarViaje
    disponible := disponible - self precioViaje.
```

Este método también se redefine en la clase **TarjetaDebito**, ya que el comportamiento si bien en parte es el mismo, también debe actualizar la cuenta bancaria cuando es necesario y hacer la recarga automática.

```
descontarViaje
    super descontarViaje.
    (disponible < 0) ifTrue: [ cuenta debitar: self montoDebito.
                               disponible:= disponible + self montoDebito ]
```

En la **Cuenta**, se agrega este método:

```
debitar: unaCantidad
    saldo := saldo - unaCantidad
```

Una vez hechos todos los controles y actualizaciones en las tarjetas, con el siguiente método de la clase **Molinete** se guarda la información del viaje realizado en el molinete. Para ello se crea una instancia de la clase viaje a partir de la tarjeta y se la agrega a la colección de viajes del molinete.

```
registrarViaje: unaTarjeta
    viajes add: (Viaje new: unaTarjeta )
```

El método de clase **new:** recibe a la tarjeta, invoca al mensaje **new** para crear el objeto y luego lo inicializa mediante el mensaje inicializar.

```
new: unaTarjeta
    | v |
    v := self new.
    v inicializar: unaTarjeta.
    ^v
```

El método de instancia inicializar, ubicado en la clase **Viaje**, setea los valores de las variables con la fecha y hora del sistema, además de registrar la tarjeta.

```
inicializar: unaTarjeta
    fecha := Date today.
    hora := Time now.
    tarjeta := unaTarjeta
```

La responsabilidad de saber cuál de todos los molinetes tuvo más viajes, es del objeto subte, quien conoce a todos los molinetes.

```
molineteEntre: unaF y: otraF
    ^ ( molinetes asSortedCollection:
    [:m1 :m2 | (m1 viajesEntre: unaF y: otraF) > (m2 viajesEntre: unaF y: otraF) ] )
            first
```

Cada molinete cuenta los viajes en las fechas indicadas

```
viajesEntre: unaF y: otraF
    ^ (viajes select: [ :via | via entre: unaF y: otraF ] ) size
```

Cada viaje sabe si está en el plazo dado

```
estaEntre: unaF y: otraF
    ^ fecha between: unaF and: otraF
```

Para saber la hora del viaje más tarde hecho por una tarjeta, una estrategia posible es obtener todos los viajes hechos por la tarjeta en todos los molinetes del subterráneo, ponerlos todos en una colección y luego ordenarla para obtener el de hora mayor.

```
viajeMasTarde: unaTarjeta
    | viajes |
    viajes := OrderedCollection new.
    molinetes do: [:mol | viajes addAll: (mol  viajesDe: unaTarjeta) ].
    ^ (viajes asSortedCollection: [:v1 :v2 | v1 hora < v2 hora] ) last hora
```

Cada molinete devuelve una colección con los viajes de esa tarjeta

```
viajesDe: unaTarjeta
    ^ viajes select: [ :v | v esDe: unaTarjeta ]
```

Cada viaje sabe con qué tarjeta se hizo

```
esDe: unaTarjeta
    ^ tarjeta = unaTarjeta
```

Por último, el método getter de la hora del viaje

```
hora
    ^hora
```

Una variante del método anterior puede ser la siguiente:

```
viajeMasTarde: unaTarjeta
    ^ ( ( molinetes inject: (OrderedCollection new )
        into: [:viajes :mol | viajes , mol  viajesDe: unaTarjeta ] )
            asSortedCollection: [:v1 :v2 | v1 hora < v2 hora] ) last hora
```

Anexo

Smalltalk

- **Codificación**
  - Identificadores
  - Asignación
  - Devolución
- **Envío de mensajes**
  - Mensajes sucesivos
  - Clasificación de los mensajes
  - Orden de evaluación
- **Números**
- **Caracteres**
- **Valores de verdad**
  - Conjunciones y disyunciones
  - Decisiones
- **Valor nulo**
- **Igualdad e identidad**
- **Definición de clases y métodos**

Codificación

La codificación de programa consiste en la definición de un conjunto de clases con sus correspondientes variables y métodos. Ejecutándose, es un conjunto de objetos cooperantes.

La codificación de un programa está organizada fundamentalmente en métodos, en los que se especifica el comportamiento de los objetos. A su vez, la codificación de cada método consiste en su mayor parte en nuevas invocaciones a mensajes, por lo que se ejecutarán otros métodos. Complementando los envíos de mensajes, forman también parte de la sintaxis las expresiones que representan las asignaciones y devolución de valores, junto con declaraciones y comentarios. Existen algunas reglas que gobiernan la sintaxis y gramática en que se debe hacer la codificación.

Identificadores

Los identificadores de variables y métodos, comúnmente llamados "nombres", consisten en una secuencia de caracteres que comienza con una letra en minúscula y pueden tener cualquier combinación de letras y números, pero sin espacios en blanco. Cuando el identificador contiene más de una palabra, las palabras extras comienzan con una letra en mayúscula.

Ejemplo:

unNombreDeVariable

unMetodo

algo

Asignación

La asignación permite darle un valor a una variable, y se representa con "**:=**". Provoca que la variable ubicada a la izquierda del signo referencie al objeto resultante de evaluar la instrucción de la derecha.

Ejemplo:

unaVariable := unObjeto	Asigna **unObjeto** a **unaVariable**
x := 5	Asigna el objeto **5** a la variable **x**
x := a + b	Asigna el resultado de la suma de **a** y **b** a la variable **x**

Devolución

La devolución de valores permite retornar un resultado al objeto emisor de un mensaje. Se representa con un "**^**" delante del objeto que se desea retornar.

Ejemplo:

^unObjeto	Retorna **unObjeto**
^ false	Retorna un valor de verdad falso
^ x + y	Retorna el resultado de la suma entre **x** e **y**

Al combinar el uso de la asignación y la devolución con invocaciones de mensajes, las reglas de precedencia indican que primero se ejecutan los mensajes, luego las asignaciones y por último la devolución.

Ejemplo:

^x := x + 7

Primero realiza la suma, luego asigna el nuevo valor a **x** y por último retorna **x**

Envío de mensajes

Los mensajes se expresan con el objeto receptor (representado por una expresión literal o por un nombre de variable) y a continuación, separado por un espacio, el nombre del método. Cuando hay varios mensajes que se envían, se los separa mediante un "."

Ejemplo:

```
unObjeto nombreDelMetodo.
unaPersona dameTuNombre.
3 factorial.
```

Para aquellos mensajes que necesitan pasar argumentos al objeto receptor al nombre del método le siguen sus argumentos.

Ejemplo:

```
unObjeto nombreDelMetodo: argumento.
unaPersona mudarseA: unaCiudad.
2 + 2.
unArray at: 1 put: 'hola'.
```

Mensajes sucesivos

Cuando hay una secuencia de mensajes que se envían a un mismo objeto receptor, se puede escribir el primer mensaje de la forma habitual, colocar un ";" como separador en vez del "." y a continuación los mensajes sucesivos, omitiendo repetir al objeto receptor, también separados por ";". Al final, se coloca un "."

Ejemplo:

En vez de realizar

```
unaPersona nombre: unNombre.
unaPersona edad: unaEdad.
unaPersona domicilio: unDomicilio.
```

Se puede hacer

```
unaPersona nombre: unNombre;
        edad: unaEdad;
        domicilio: unDomicilio.
```

Clasificación de los mensajes

Según los argumentos de los mensajes se clasifican en unarios, binarios y de palabra clave.

Unarios: Son los mensajes que no tienen argumentos.

Ejemplo:

x sin	Devuelve el resultado del seno de **x**
3 factorial	Devuelve el factorial de **5**
'hola' size	Devuelve **4**, la longitud de la cadena **'hola'**

En estos ejemplos **x**, **3** y **'hola'** son objetos y **sin**, **factorial** y **size** son mensajes, o sea también nombres de métodos.

Binarios: Tienen un solo argumento y su nombre está formado por caracteres especiales, por lo que no lleva el ":" para indicar el argumento.

Ejemplo:

a + b	Devuelve la suma de **a** y **b**
a & b	Devuelve la conjunción lógica entre **a** y **b**
a >= b	Devuelve si es cierto que a sea mayor o igual que **b**

De Palabra Clave: Son mensajes con uno o más argumentos. Utilizan el ":" como último carácter del nombre, para indicar la presencia del argumento. Cuando hay varios argumentos, el nombre tiene tantas partes como argumentos, cada una terminada también con el carácter ":".

Ejemplo:

```
unObjeto nombreDelMetodo: argumento
unaPersona mudarseA: unaCiudad
unArray at: 1 put: 'hola'
unaPersona nombre: 'Juan' edad: 21
unBooleano ifTrue: unBloqueDeCodigo ifFalse: otroBloqueDeCodigo
```

Orden de evaluación

Si en una expresión hay varios mensajes que se están enviando, estos se ejecutan de acuerdo a un orden de precedencia. El resultado de un mensaje que se evalúa antes es tomado como objeto receptor o argumento del mensaje que es evaluado a continuación dentro en la expresión.

Las reglas de precedencia que indican el orden de evaluación de los mensajes son muy simples. Para alterarlas, se debe indicar mediante paréntesis.

1. Dentro de una expresión, los mensajes unarios se ejecutan primero, luego los mensajes binarios y finalmente los mensajes de palabra clave.

Ejemplo:

4 + 3 factorial

Se evalúa primero el mensaje factorial, por ser unario, y su resultado es considerado como argumento del mensaje binario de la suma, de manera que luego se evalúa **4 + 6**.

unaPersona nombre: 'Ju' , 'an' edad: 20 + 1

Se evalúa primero la concatenación entre **'Ju'** y **'an'** y la suma entre **20** y **1**, que son binarios y sus resultados son enviados como argumentos del mensaje de palabra clave que luego se evalúa como.

unaPersona nombre: 'Juan' edad: 21

2. Entre mensajes de igual categoría se ejecutan los mensajes de izquierda a derecha.

Ejemplo:

2 * 3 + 1

Evalúa primero la multiplicación y luego la suma. Resultado **7**

1 + 2 * 3

Evalúa primero la suma y luego la multiplicación. Resultado **9**

Números

Los números se representan mediante objetos que son instancias de las subclases de la clase **Number**, que es abstracta. Se representan en el código por la expresión literal de los dígitos comunes y un reducido conjunto de caracteres especiales, según sean números enteros, positivos, fraccionarios, con decimales, etc. No se los instancia explícitamente enviando el mensaje **new** a la clase correspondiente, sino directamente con su representación.

Ejemplo:

123	Un número entero (Clase **SmallInteger**)
-23412345345	Un número entero negativo (Clase **LargeInteger**)
3/4	Una fracción (Clase **Fraction**)
4.56	Un número real (Clase **Float**)
2.34e10	Un número real, en notación exponencial (Clase **Float**)

Las operaciones aritméticas básicas se realizan con mensajes binarios polimórficos para todas las clases numéricas.

Ejemplo:

3 + 4	Devuelve **7**, la suma del **3** y 4
3.3 * 2	Multiplica el número real por el entero y obtiene **6.6**
6.8 / 2.4	Realiza la división y obtiene como resultado **2.8333333333**
9 // 4	Realiza la división entera (cociente) y devuelve **2**,
9 \\ 4	Devuelve el resto de la división entera: **1**
4 >= 3	Devuelve **true**, por ser el número **4** mayor o igual que el **3**
3.11 < 5	Devuelve **true**, por ser el **3.11** menor que el **5**

Caracteres

Cualquier carácter es una instancia de la clase **Character**, se representa por el mismo carácter precedido por el signo $. No se instancian mediante el método **new**.

Ejemplo:

$G	el carácter **G**
$5	el carácter **5**
$$	el carácter **$**

Hay varios mensajes para operar con los caracteres.

Ejemplo:

$G asciiValue	Devuelve **71**, el valor ASCII de la letra **G**
$5 isDigit	Devuelve **true**, porque el **5** es un carácter que es un dígito
$$ asString	Devuelve un **String** con el carácter **$** como único elemento.
$e < $f	Devuelve **true**, porque el carácter **e** es menor al **f** en el ASCII

Valores de verdad

Los valores de verdad se representan con las palabras reservadas **true** y **false,** que son las únicas instancias de las clases **True** y **False**, respectivamente, ambas subclases de **Boolean**.

Para hacer una negación, existe el mensaje not.

Ejemplo:

false not	Retorna **true**

(5 > 1) not Retorna **false**

Conjunciones y disyunciones

Para unir varias expresiones booleanas mediante conjunciones o disyunciones hay dos estrategias básicas, según se quiera o no evaluar todas las expresiones.

Ejemplo:

(a > 0) & (b > 0)

Devuelve **true** si **a** y **b** son positivos.

(a > 0) | (b > 0)

Devuelve **true** si **a** es positivo o **b** es positivo.

(a > 0) | ((b := b+1) > 0)

Devuelve **true** si **a** es negativo o **b**, luego de realizarse la suma, es positivo. Independientemente del resultado final, **b** es incrementada.

De esta manera, el **|** recibe como argumento la expresión ya evaluada, o sea, el valor booleano que resulte finalmente. Lo mismo sucede con el **&.**

(a > 0) and: [b > 0]

Devuelve **true** si **a** y **b** son positivos. Si **a** no fuera positivo, devuelve **false** sin evaluar el bloque.

(a > 0) or: [b > 0]

Devuelve **true** si **a** es positivo o **b** es positivo. Si **a** fuera positivo, devuelve **true** sin evaluar el bloque.

(a > 0) or: [(b := b+1) > 0]

Devuelve **true** si a es negativo o **b**, luego de realizarse la suma, es positivo. Si **a** fuera positivo, devuelve **true** sin evaluar el bloque, por lo que la variable **b** no sería incrementada.

En conclusión, tanto para el mensaje **and:** como para el **or:** el argumento no es el valor booleano, sino un bloque de código que sólo se evalúa si es necesario para resolver el valor final de la expresión.

Decisiones

Para tomar una decisión sobre dos posibles flujos de ejecución dependiendo del valor de verdad de una expresión booleana, se puede enviar el mensaje de palabra clave **ifTrue:ifFalse:** El receptor es un objeto booleano y los argumentos son dos bloques de códigos, uno de los cuales se evaluará y el otro no.

Ejemplo:

(a > b) ifTrue: [x := a] ifFalse: [x := b].

Si **a** es mayor a **b**, se evaluará el primer argumento y en caso contrario, se evaluará el segundo. El valor de la variable **x** dependerá entonces del valor de verdad de la comparación.

También se puede enviar el mensaje **ifFalse:ifTrue:** que realiza lo mismo pero invirtiendo el orden de los argumentos, el mensaje **ifTrue:** y el **ifFalse:** ambos con un solo argumento.

Valor nulo

Toda variable, inicialmente, antes que le sea asignado algún valor, apunta a un valor nulo representado por la palabra reservada **nil**. Este valor es una instancia de la clase **UndefinedObject**.

Cualquier variable puede ser apuntada a **nil** durante la ejecución, mediante una asignación. También puede usarse **nil** como un valor de retorno para indicar que una operación no fue exitosa.

Ejemplo:

```
elevar: unNumero aLaPotencia: unaPotencia
( unaPotencia < 0 )
        ifFalse: [ ^nil ].
^aNumber raisedTo: aPower
```

Este método, eleva **unNumero** a la potencia **unaPotencia**. Devuelve **nil** si **unaPotencia** no es positiva. Caso contrario envía un nuevo mensaje que calcula el resultado y lo devuelve.

Se puede preguntar si el valor de una expresión es nil mediante el mensaje **isNil**.

Ejemplo:

```
empleados
    ( empleados isNil) ifTrue:[ empleados := OrderedCollection new].
    ^empleados
```

Este método, ubicado en alguna clase que tenga como variable de instancia una colección de empleados, permite acceder a la colección, y si esta aún no fue inicializada, es decir que sigue valiendo **nil**, la inicializa con una nueva colección de la clase **OrderedCollection**.

Igualdad e identidad

Un objeto es idéntico a otro sólo si es el mismo objeto. Al comparar dos variables o expresiones cualesquiera, la comparación por idéntico es cierta cuando ambas referencian al mismo objeto. El mensaje que responde si un objeto es idéntico a otro es "**==**". Para preguntar si un objeto no es idéntico a otro, el mensaje es "**~~**".

Ejemplo:

2 == 2

El objeto 2 es idéntico al objeto 2, son el mismo objeto. Devuelve **true**

2 ~~ 2

Devuelve **false**.

2 == 2.0

Son distintos objetos, uno es **SmallInteger** y el otro es **Float**. Devuelve **false**.

x := unObjeto. y := unObjeto. x == y.

x e **y** referencian a **unObjeto**. Devuelve **true**

Un objeto es igual a otro si su valor es el mismo. Por defecto, para el común de las clases la igualdad coincide con la identidad, pero existen varias clases, como por ejemplo las numéricas, para los que la igualdad toma en cuenta el valor que representan los objetos más allá que no sean los mismos. El mensaje que indica si un objeto es igual a otro es el "=", mientras que saber si son diferentes, el mensaje es el "**~=".**

Ejemplo:

2 = 2

El objeto **2** es igual al objeto **2**. Devuelve **true**

2 ~= 2

Devuelve **false**.

2 = 2.0

El valor de ambos es el mismo. Devuelve **true**.

x := unObjeto. y := unObjeto. x = y.

Si son el mismo objeto, seguro valen lo mismo. Devuelve **true**

Definición de clases y métodos

Para definir una clase se especifica un identificador que actúa como nombre de la clase. Los nombres de las clases siempre empiezan en mayúsculas. Además, se especifica su ubicación jerárquica respecto de otras clases.

Luego se enumeran entre comillas simples, los nombres de cada una de las variables de instancia, que comienzan con minúsculas. En la siguiente línea, también entre comillas simples, los nombres de las variables de clase, comenzando con mayúsculas.

Ejemplo:

```
Object subclass: #Auto
    instanceVariableNames: 'nroMotor marca modelo'
    classVariableNames: 'TasaDeImpuesto'
```

...

La clase **Auto** está ubicada jerárquicamente como subclase de la clase **Objetc**.

Un método, en su primera línea, tiene la interfaz, compuesta por el nombre del método junto con la declaración de los identificadores de cada uno de los argumentos que recibe, que se consideran variables locales al método.

Es una convención aceptada el tener comentarios al principio de un método para explicar su comportamiento. Los comentarios son cualquier secuencia de caracteres que no deben ser ejecutados, que se representa encerrándola entre comillas dobles (“ ”).

Un método pueden necesitar utilizar variables locales, que se declaran encerrándolas entre barras verticales (|) al principio de un método y separándolas entre sí por espacios. Su ámbito de validez es la totalidad del método. Su nombre debe ser único dentro del método, por lo que no puede coincidir con una variable de instancia ni con los argumentos.

Por último, viene el cuerpo principal del código del método, con las expresiones que indican el envío de nuevos mensajes, asignaciones, etc. Las sentencias que forman parte del código de un método se separan unas de otras por puntos, excepto en la última, que es opcional. Si en un método no se especifica la devolución de un objeto, se asume por defecto como valor de retorno al mismo objeto receptor del mensaje.

Ejemplo:

```
promedioEntre: unNumero yTambien: otroNumero
    “este es un método de ejemplo que recibe dos números y calcula su promedio”
    | variableAuxiliar otraVariableAuxiliar |
    variableAuxiliar := unNumero + otroNumero.
    otraVariableAuxiliar := variableAuxiliar / 2.
    ^ otraVariableAuxiliar
```

La interfaz incluye el nombre del método **promedioEntre:yTambien:**, que se corresponde con un mensaje de palabra clave con dos argumentos y los identificadores de cada uno de ellos, **unNumero** y **otroNumero**. En la siguiente línea, el comentario explicativo del sentido del método.

La declaración de variables locales abarca a las variables **variableAuxiliar** y **otraVariableAuxiliar**.

En la línea siguiente se le envía al objeto representado por la variables local del argumento, llamada **unNumero**, el mensaje **+** con **otroNumero** como argumento; el resultado es asignado a la variable local declarada con el nombre de **variableAuxiliar**. Luego, al objeto recientemente referenciado desde **variableAuxiliar** se le envía un mensaje de división **(/)** con el objeto **2** como argumento; el resultado es asignado a la variable local **otraVariableAuxiliar**. Por último, se realiza la devolución del objeto referenciado por **otraVariableAuxiliar**, que concluye con la ejecución del método.

El código de un método puede hacerse más compacto y eficiente reduciendo la cantidad de variables auxiliares y combinando el envío de varios mensajes. Es necesario establecer un equilibrio criterioso, sin abusar de variables auxiliares ni anidando demasiado las expresiones, para aumentar la claridad y legibilidad del código.

Ejemplo:

El método del ejemplo anterior puede optimizarse en su codificación y sin perder claridad dada la sencillez del cálculo. De modo que el método pudo haber sido codificado también como:

```
promedioEntre: unNumero yTambien: otroNumero
    ^  (unNumero + otroNumero ) / 2
```

Bibliografía

- **ALONSO AMO, F y SEGOVIA PEREZ, F**. *Entornos y Metodologías de Programación*. Paraninfo.
- **WATT, David**. *Programming Languajes Concepts and Paradigms*. Prentice Hall.
- **GHEZZI, Carlo y JAZAYERI, Mehdi**. *Conceptos de Lenguajes de Programación*. Díaz de los Santos.
- **RAVI y SETHI.** *Lenguajes de programación - Conceptos y constructores.* Addison Wesley.

Índice