

Objetos y Clases

El desarrollo de un sistema de software comienza con la elaboración de un **modelo del problema** a resolver. A medida que se avanza en el proceso de desarrollo se obtiene un **modelo de la solución del problema**.

Un **modelo** es una representación que permite describir, explicar, analizar o simular un sistema o proceso a partir de sus entidades relevantes y el modo en que se relacionan. Existen distintos **tipos de modelos**. Una maqueta de un estadio de fútbol es un modelo físico. Un sistema de ecuaciones que describe el equilibrio en la resistencia de los materiales con los cuales se construyen las plateas del estadio, es un modelo matemático. Un plano catastral que muestra un terreno y sus límites, es un modelo gráfico. También son representaciones gráficas los planos del sistema de iluminación y el plano de alzada de los vestuarios.

El mecanismo fundamental que se utiliza para elaborar un modelo es la **abstracción**. Una abstracción se genera a partir de una serie de operaciones mentales que permiten **identificar** entidades y caracterizarlas de acuerdo a sus propiedades y al modo en que interactúan con las demás. Una misma entidad puede ser caracterizada de diferentes maneras según qué propiedades se consideren esenciales y qué nivel de detalle se pretenda en la representación.

En un modelo del sistema solar, por ejemplo, algunas de las entidades relevantes son el Sol, la Luna, Mercurio, Venus, Tierra, Marte, Júpiter, Saturno, Urano, Neptuno, Plutón, Ceres. Algunas propiedades de la Tierra son el radio, la masa, el período orbital, el período de rotación y la atmósfera. La Tierra puede rotar sobre sí misma y trasladarse siguiendo una órbita alrededor del Sol.

Otro mecanismo fundamental para la construcción de modelos es la **clasificación**. Una clasificación se genera sobre la base de una serie de operaciones mentales que permiten **agrupar** entidades a partir de sus **semejanzas y diferencias**. Las semejanzas o diferencias se establecen en función de diferentes **criterios**. El criterio de clasificación va a estar ligado a las propiedades que se identificaron como relevantes.

En el modelo del sistema solar, Mercurio, Venus, Tierra, Marte, Júpiter, Saturno, Urano y Neptuno pueden agruparse en la clase Planeta. Todas las entidades de la clase Planeta se caracterizan por las mismas propiedades, aunque los valores para cada propiedad van a variar de una entidad a otra. La clase Planeta es un patrón para las entidades Mercurio, Venus, Tierra, Marte, Júpiter, Saturno, Urano y Neptuno.

La capacidad para clasificar entidades se desarrolla de manera natural en la infancia a través del contacto con objetos concretos. Se refuerza progresivamente y se especializa a medida que aumenta el conocimiento en un determinado dominio.

Cada miembro de un equipo de desarrollo de software debe desarrollar capacidad para clasificar e interpretar una clasificación, en particular cuando se adopta el paradigma de programación orientada a objetos. El **principio fundamental del paradigma de programación orientada a objetos** es **construir un sistema de software** con base en a las entidades de un **modelo** elaborado a partir de un proceso de abstracción y clasificación. Los conceptos de objeto y clase, esenciales en la programación orientada a objetos, están fuertemente ligados a los procesos de abstracción y clasificación.

El concepto de objeto

El término **objeto** es central en la programación orientada a objetos y se refiere a dos conceptos relacionados pero diferentes.

Durante el **desarrollo de requerimientos** de un sistema de software el **analista** es responsable de identificar los **objetos del problema** y caracterizarlos a través de sus **atributos**. En la etapa de diseño se completa la representación modelando también el **comportamiento** de los objetos.

Un **objeto del problema** es una entidad, física o conceptual, caracterizada a través de **atributos** y **comportamiento**. El comportamiento queda determinado por un conjunto de **servicios** que el objeto puede brindar y un conjunto de **responsabilidades** que debe asumir.

Cuando el sistema de software está en ejecución, se crean objetos de software.

Un **objeto de software** es un **modelo**, una representación de un objeto del problema. Un objeto de software tiene una **identidad** y un **estado interno** y recibe **mensajes** a los que responde ejecutando un **servicio**. El estado interno mantiene los valores de los atributos.

Los objetos de software se comunican a través de mensajes para solicitar y brindar servicios. La ejecución de un servicio puede modificar el estado interno del objeto que recibió el mensaje y/o computar un valor.

El **modelo computacional** propuesto por la programación orientada a objetos es un mundo poblado de objetos comunicándose a través de mensajes.

La ejecución se inicia cuando un objeto recibe un mensaje y en respuesta a él envía mensajes a otros objetos.

Caso de Estudio: Identificación de algunos objetos en un Sistema de Gestión Hospitalaria

En un hospital se mantiene la historia clínica de cada paciente, incluyendo nombre, sexo, fecha de nacimiento y obra social. Las historias clínicas registran además información referida a cada práctica de diagnóstico, por ejemplo las radiografías. Cada radiografía incluye una imagen, que puede representarse como un conjunto de píxeles. En una imagen digital, un pixel es la menor unidad homogénea en color.

Durante el desarrollo del Sistema de Gestión Hospitalaria, se identifican como objetos del problema a cada una de las historias clínicas. Cada historia clínica tiene varios atributos, entre ellos el paciente. Cada paciente es a su vez un objeto, cuyos atributos son, entre otros, nombre, sexo, fecha de nacimiento y obra social. La fecha de nacimiento es un objeto, cuyos atributos son día, mes y año.

Una radiografía realizada a un paciente es un objeto, con un atributo imagen. Una imagen es en sí misma un objeto, cuyo principal atributo es una matriz de píxeles. Cada pixel es un objeto, cuyos atributos son tres números enteros. El aparato con el que se realiza el procedimiento es un objeto, como también lo son el médico que ordena la realización del procedimiento, la enfermera que la registra en la historia clínica y el técnico que la hace.

La medición de la presión arterial es un objeto, con atributos para registrar el valor máximo y mínimo. El conjunto de mediciones de presión arterial de un paciente es un objeto formado por otros objetos.

En un momento determinado de la ejecución del Sistema de Gestión del Hospital, se mantendrá en memoria un objeto de software que modelará a una historia clínica en particular. Algunos de los atributos serán a su vez objetos.

La palabra **objeto** se utiliza entonces para referirse a:

- Los **objetos del problema**, es decir, las entidades identificadas durante el desarrollo de requerimientos o el diseño: cada historia clínica, paciente, radiografía, imagen, pixel, medición de presión arterial, etc.
- Los **objetos de software**, esto es, cada representación que modela en ejecución a una entidad del problema: cada historia clínica, paciente, radiografía, imagen, pixel, medición de presión arterial, etc.

Una vez que se han identificado los objetos del problema es posible concentrarse en la especificación de algunos de ellos en particular.

Caso de Estudio: Especificación de Requerimientos de Medición de la Presión Arterial

La presión arterial es la fuerza de presión ejercida por la sangre circulante sobre las arterias y constituye uno de los principales signos vitales de un paciente. Se mide por medio de un esfigmomanómetro, que usa la altura de una columna de mercurio para reflejar la presión de circulación. Los valores de la presión sanguínea se expresan en kilopascales (kPa) o en milímetros del mercurio (mmHg). Para convertir de milímetro de mercurio a kilopascales el valor se multiplica por 0,13.

La presión sistólica se define como el máximo de la curva de presión en las arterias y ocurre cerca del principio del ciclo cardíaco durante la sístole o contracción ventricular; la presión diastólica es el valor mínimo de la curva de presión en la fase de diástole o relajación ventricular del ciclo cardíaco. La presión de pulso refleja la diferencia entre las presiones máxima y mínima medidas. Estas medidas de presión no son estáticas, experimentan variaciones naturales entre un latido del corazón a otro a través del día y tienen grandes variaciones de un individuo a otro.

La hipertensión se refiere a la presión sanguínea que es anormalmente alta, y se puede establecer un umbral para la máxima y otro para la mínima que permitan considerar una situación de alarma.

Desarrollar el sistema completo para la Gestión del Hospital de manera integrada es una tarea compleja, que probablemente demandará la participación de un equipo de profesionales con diferentes roles. La complejidad se reduce si durante la etapa de diseño se divide el problema en subproblemas más simples que el problema original. La especificación de la presión arterial de un paciente es un problema de pequeña escala, con relación al problema global de desarrollar un Sistema de Gestión para el Hospital. Sin embargo, establecer adecuadamente los requerimientos es una tarea fundamental, aun para problemas simples.

El concepto de clase

Durante el desarrollo de requerimientos de un sistema de software se identifican los objetos relevantes del **problema**, se los caracteriza a través de sus atributos y se los agrupa en **clases**. Todos los objetos que son instancias de una clase van a estar caracterizados por los mismos atributos. En la etapa de diseño se especifica el comportamiento de los objetos y probablemente se agregan nuevas clases relevantes para la **solución** del problema.

Desde el punto de vista del diseño de un sistema de software, una **clase** es un *patrón* que establece los atributos y el comportamiento de un conjunto de objetos.

Durante la implementación, el programador escribe el código de cada clase en un lenguaje de programación.

En la implementación de un sistema de software una **clase** es un **módulo de software** que puede construirse, verificarse y depurarse con cierta independencia a los demás.

Así, un **sistema de software orientado a objetos** es una **colección de módulos de código**, cada uno de los cuales implementa a una de las clases modeladas en la etapa de diseño. En la ejecución del sistema se crean **objetos de software**, cada uno de ellos es instancia de una clase.

Desde el punto de vista **estático** un **sistema de software orientado a objetos** es una **colección de clases** relacionadas. Desde el punto de vista **dinámico** un **sistema de software orientado a objetos** es un **conjunto de objetos** comunicándose.

Cada objeto brinda el conjunto de servicios que define su clase.

El diseño de una clase

Durante el desarrollo de requerimientos el **analista** construye un **diagrama de clases** que modela el **problema**. El **diseñador** del sistema completa el diagrama de clases para modelar la **solución**.

Un **diagrama de clases** es una representación visual que permite modelar la estructura de un sistema de software a partir de las clases que lo componen.

El diagrama de clases se construye usando un **lenguaje de modelado**. El diagrama de cada clase en el lenguaje de modelado tiene la siguiente forma:

<Nombre>
<<Atributos>>
<<Constructores>>
<<Comandos>>
<<Consultas>>
<<Responsabilidades>>

El **nombre** de una clase representa la abstracción del conjunto de instancias. Por lo general es un sustantivo y debe elegirse cuidadosamente para representar al conjunto completo.

Un **atributo** es una propiedad o cualidad relevante que caracteriza a todos los objetos de una clase.

Es posible distinguir los **atributos de clase** de los **atributos de instancia**. En el primer caso, el valor es compartido por todos los objetos que son instancias de la clase. Los valores de los atributos de instancia varían en cada objeto.

Un **servicio** es una operación que todas las instancias de una clase pueden realizar.

Los **servicios** pueden ser **métodos** o **constructores**.

Un **método** es un servicio que un objeto ejecuta en respuesta a un mensaje.

Un **constructor** es un servicio que se invoca para crear un objeto.

Los métodos pueden ser **comandos** o **consultas**.

Un **comando** es un método que al ejecutarse modifica el valor de uno o más atributos del objeto que recibió el mensaje.

Una **consulta** es un método que al ejecutarse no modifica el valor de ningún atributo del objeto que recibió el mensaje.

Cada comando o consulta tiene una **funcionalidad** que establece su propósito.

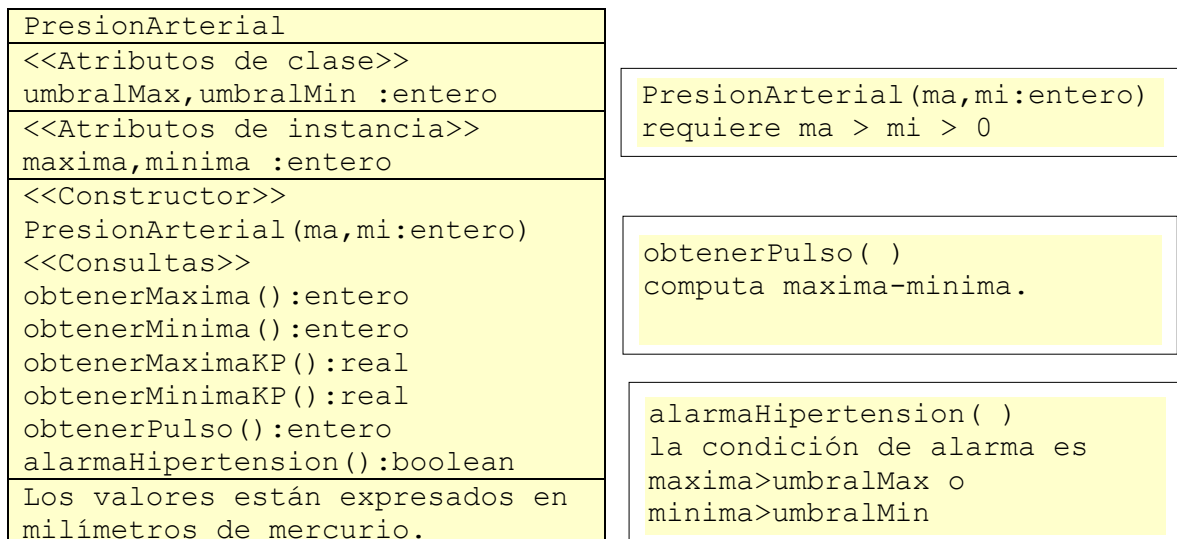
Una **responsabilidad** representa un compromiso o un requisito para una clase.

El conjunto de servicios y las responsabilidades determinan el **comportamiento** de las instancias de una clase.

Al diagrama de una clase se le pueden agregar **notas** o **comentarios** que describan **requisitos** o la **funcionalidad** de los servicios. Pueden dibujarse mediante un rectángulo y pueden tener la esquina superior derecha plegada, como si fuera una hoja. Se puede unir con una línea al elemento de la clase al cual corresponde la nota. Alternativamente los requisitos y la funcionalidad de cada servicio puede especificarse como parte de la descripción del problema.

Caso de Estudio: Diagrama de la clase PresionArterial

Retomando el caso de estudio propuesto, el siguiente diagrama modela la clase **PresionArterial**, con comentarios para especificar requisitos y funcionalidad.



La especificación de requerimientos brinda información útil para la implementación. El siguiente es un modelo alternativo para la clase **PresionArterial**. El diagrama es el mismo pero la notación para describir funcionalidades y responsabilidades es diferente y además se explicita cómo convertir milímetros de mercurio en kilo pascales.

PresionArterial
<<Atributos de clase>> umbralMax, umbralMin :entero
<<Atributos de instancia>> maxima, minima :entero
<<Constructor>> PresionArterial (ma, mi :entero)
<<Consultas>> obtenerMaxima() :entero obtenerMinima() :entero obtenerMaximaKP() :real obtenerMinimaKP() :real obtenerPulso() :entero alarmaHipertension() :boolean
Los valores están expresados en milímetros de mercurio.

- **PresionArterial (ma, mi :entero)** Inicializa los valores de los atributos con los parámetros. Requiere $ma > mi > 0$.
- **obtenerMaximaKP() :entero** Convierte a kilopascales $maxima \cdot 0,13$
- **obtenerMinimaKP() :entero** Convierte a kilopascales $minima \cdot 0,13$
- **obtenerPulso() :entero** computa $maxima - minima$.
- **alarmaHipertension() :true** retorna true si $maxima > umbralMax$ o $minima > umbralMin$.

Ninguno de los modelos incluye atributos para mantener los valores expresados en kilopascales, sino que se computan cuando se ejecutan los métodos `obtenerMaximaKP()` y `obtenerMinimaKP()`.

La clase `PresionArterial` es un **patrón** para cada medición que se va a representar. El diseñador decidió que es relevante representar la máxima y la mínima de cada medición. En la caracterización interesa también conocer el pulso, pero el diseñador resolvió que este valor no se mantenga como un atributo, sino que se calcule como la diferencia entre la máxima y la mínima.

De acuerdo a esta propuesta, en el momento que se crea una medición se establecen los valores de los dos atributos de instancia. El diseñador decidió que la clase `PresionArterial` requiera que la clase que la usa se haga responsable de garantizar que los valores cumplen ciertas restricciones, en este caso que la máxima sea mayor que la mínima.

La clase `PresionArterial` probablemente sea una componente simple que forma parte de una aplicación de gran escala. La implementación de esta clase es una solución para un subproblema, que luego va a integrarse con otras para resolver el problema global. Así, los beneficios de la programación orientada a objetos, que pueden pasar desapercibidos cuando se abordan problemas de pequeña escala, quedarán mejor ilustrados si se considera cada solución en el contexto de un proyecto de mayor envergadura.

La implementación de una clase

En la etapa de implementación de un sistema de software el programador parte del documento elaborado por el diseñador y escribe el código de cada clase del diagrama. La implementación requiere elegir un **lenguaje de programación**. Un lenguaje orientado a objetos permite retener durante la implementación la organización de las clases modeladas durante el desarrollo de requerimientos y la etapa de diseño.

En este libro se ha elegido **Java** como lenguaje de programación y se adoptaron algunas convenciones para el código. En particular el código de cada clase mantiene la estructura de cada diagrama:

<Nombre>
<<Atributos>>
...
<<Constructores>>
...
<<Comandos>>
...
<<Consultas>>
...
<<Responsabilidades>>
...

```
class <Nombre>{
//Atributos de clase
...
//Atributos de instancia
...
//Constructores
...
//Comandos
...
//Consultas
...
}
```

La palabra reservada `class` está seguida por un identificador que es el nombre de la clase. Java es sensible a las minúsculas y mayúsculas. Las `{ }` son los delimitadores de cada unidad de código. Existen otros delimitadores como los corchetes y los paréntesis. El símbolo `//` precede a un comentario de una línea. Los símbolos `/* */` delimitan a un comentario de varias líneas. El símbolo `;` termina cada instrucción.

Los **miembros** de una clase son **atributos** y **servicios**. Manteniendo la misma clasificación que la propuesta por el lenguaje de modelado, los atributos pueden ser de clase o de instancia. Los servicios pueden ser constructores, comandos o consultas. En todos los casos están formados por un encabezamiento y un bloque de instrucciones delimitado por llaves.

Los atributos se definen a través de la declaración de **variables**. Un atributo de clase se declara como una variable estática.

En la declaración de una variable, el tipo precede al nombre de la variable. Un **tipo elemental** determina un conjunto de valores y un conjunto de operaciones que se aplican sobre estos valores. En ejecución, una **variable de un tipo elemental** mantiene un **valor** que corresponde al tipo y participa en las operaciones establecidas por su tipo.

Java brinda siete tipos de datos elementales. Una variable de un tipo elemental se utiliza como operando en cualquier **expresión** en la que se apliquen **operadores** con los cuales su tipo sea compatible. El mecanismo de **conversión implícito** entre tipos elementales permite escribir **expresiones mixtas**, esto es, con operandos de distinto tipo.

Un **constructor** es un **servicio** provisto por la clase y se caracteriza porque recibe el mismo nombre que la clase. El constructor se invoca cuando se crea un objeto y habitualmente se usa para inicializar los valores de los atributos de instancia.

En Java, si una clase no define explícitamente un constructor, el compilador crea automáticamente uno, en ese caso los atributos son inicializados **por omisión**. Una clase puede brindar varios constructores, siempre que tengan diferente número o tipo de parámetros. Si la clase incluye uno o más constructores, el compilador no agrega ningún otro.

Como se indicó antes, los **comandos** son servicios que modifican los valores de uno o más de los atributos del objeto que recibe el mensaje. Las **consultas** son servicios que no modifican los valores de los atributos del objeto que recibe el mensaje y por lo general retornan un valor. Un comando puede retornar también un valor. Los comandos y consultas conforman los **métodos** de una clase.

Cada método está precedido por el **tipo del resultado**. Si el nombre de un método está precedido por un tipo elemental o una clase, el bloque debe incluir una instrucción de retorno. La instrucción de retorno comienza con la palabra `return` y sigue con una expresión que debe ser **compatible** con el tipo del resultado. Si un método no retorna un resultado su nombre va precedido por la palabra `void`.

Una implementación para la clase `PresionArterial`, cuyo diseño se presentó en la sección anterior, puede ser:

Caso de Estudio: El código de la clase `PresionArterial`

```
class PresionArterial {
//Valores representados el milímetros de mercurio
//Atributos de clase
private static final int umbralMax=120;
private static final int umbralMin=80;
//Atributos de instancia
private int maxima;
private int minima;
//Constructor
public PresionArterial(int ma,int mi){
//Requiere ma > mi > 0
    maxima = ma;
    minima = mi;}
//Consultas
public int obtenerMaxima(){
    return maxima;}
public int obtenerMinima(){
    return minima;}
public double obtenerMaximaKP(){
//Convierte a kilopascales
    return maxima*0.13;}
public double obtenerMinimaKP(){
//Convierte a kilopascales
    return minima*0.13;}
public int obtenerPulso(){
    return maxima-minima;}
public boolean armaHipertension(){
    return maxima > umbralMax || minima > umbralMin;}}
```


Algunas convenciones adoptadas para favorecer la legibilidad:

- La primera letra del nombre de la clase se escribe en mayúscula y por lo tanto también el nombre del constructor.
- La primera letra del nombre de atributos y métodos se escribe con minúscula. Se separan las palabras de un identificador escribiendo la inicial de todas las palabras, excepto la primera, con mayúscula. Por ejemplo, `umbralMax`.
- Se escriben como comentarios las notas y responsabilidades del diagrama de clases, excepto cuando resultan evidentes del código como por ejemplo el cálculo del pulso.
- Se usan comentarios para identificar las secciones en las que se definen los atributos, constructores, comandos y consultas.
- Los nombres de las consultas que retornan el valor de un atributo comienzan con la palabra `obtener`. En el caso de que la clase brinde comandos para modificar el valor de un atributo, sus nombres comenzarán con la palabra `establecer`. Otra posibilidad es mantener la convención en inglés usando los prefijos `get` y `set`.

En este caso de estudio, cada atributo de instancia se declara como una variable del tipo elemental `int`. La palabra reservada `private` es un **modificador**, indica que las variables `umbralMax`, `umbralMin`, `maxima` y `minima` sólo son **visibles** y pueden ser usadas dentro de la clase. Fuera de la clase los atributos privados no son visibles.

El modificador `static` establece que las variables `umbralMax` y `umbralMin` presentan atributos de clases, esto es, todos los objetos de clase `PresionArterial` comparten el mismo valor para cada umbral. El modificador `final` indica que se trata de valores constantes, establecidos en la declaración.

La clase brinda un único constructor que establece los valores de los atributos de instancia de acuerdo a los parámetros. La clase `PresionArterial` no ofrece comandos para modificar los valores de los atributos de instancia.

Aunque la clase no mantiene un atributo de instancia para la presión del pulso, las clases que usan los servicios provistos por `PresionArterial` acceden de manera uniforme a la presión máxima, mínima y a la presión del pulso.

Los tipos `int`, `float`, `double` y `boolean` son tipos elementales. La expresión:

```
maxima*0.13
```

recibe un operando de tipo `int` y otro de tipo `double`, el valor del atributo `maxima` se convierte implícitamente antes de computarse el resultado. Como el resultado de `obtenerMaximaKP` es de tipo `double`, el valor computado se convierte antes de retornar.

La instrucción:

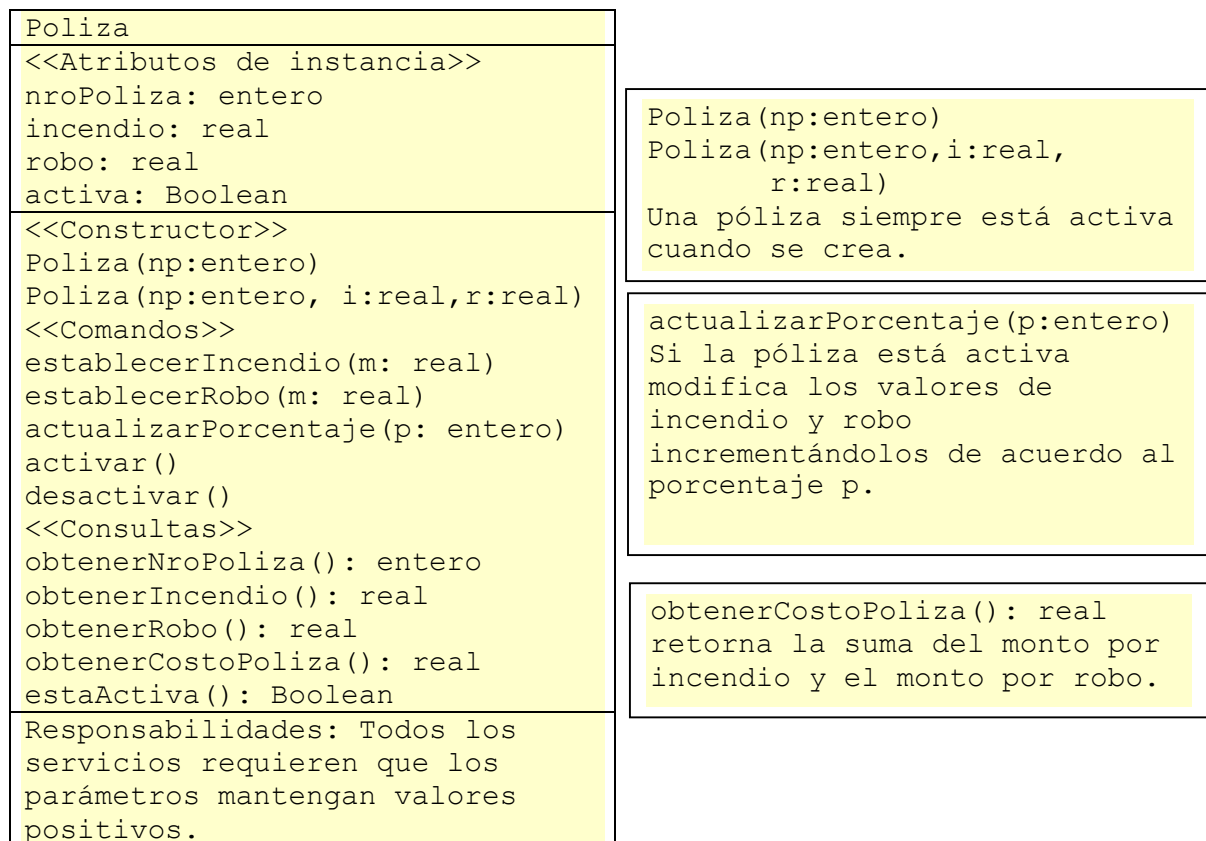
```
return maxima > umbralMax || minima > umbralMin;
```

retorna un valor de verdad que resulta de computar la expresión booleana. El operador `||` se evalúa en **cortocircuito**, es decir, si la primera expresión computa `true`, no computa la segunda. En la siguiente instrucción la evaluación es **completa**, esto es, se evalúan las dos subexpresiones, antes de computar el valor que retorna:

```
return maxima > umbralMax | minima > umbralMin;
```

El diseñador es responsable de especificar correctamente las restricciones sobre los datos, los programadores implementan los controles para validar los datos de acuerdo a lo especificado. Si el diseñador no interpretó o no especificó correctamente estas restricciones puede producirse un **error lógico** o **error de aplicación**. En este ejemplo, el código no tiene errores de compilación, la ejecución terminará normalmente, pero los resultados pueden no ser correctos si las restricciones sobre los valores de los atributos de instancias no especifican cuándo se produce efectivamente una alarma de hipertensión.

Ejercicio: Implemente en Java la clase `Poliza` modelada en el siguiente diagrama:



Ejercicio: Implemente la clase `Surtidor` de acuerdo al diagrama y la especificación de funcionalidades que le sigue:

Una estación de servicio cuenta con surtidores de combustible capaces de proveer Gasoil, Nafta Super y Nafta Premium 2000. Todos los surtidores tienen capacidad para almacenar un máximo de 20000 litros de cada combustible. En cada surtidor se mantiene registro de la cantidad de litros disponibles en depósito de cada tipo de combustible, esta cantidad se inicializa en el momento de crearse un surtidor con la cantidad máxima. En cada surtidor es posible llenar o extraer combustible.

Surtidor
<<atributos de clase>> maximaCarga: 500 <<atributos de instancia>> cantGasoil: entero cantSuper: entero cantPremium: entero
<<Constructor>> Surtidor() <<comandos>> llenarDepositoGasoil() llenarDepositoSuper() llenarDepositoPremium() extraerGasoil(litros: entero) extraerSuper(litros: entero) extraerPremium(litros: entero) <<consultas>> obtenerMaximaCarga(): entero obtenerCantGasoil(): entero obtenerCantSuper(): entero obtenerCantPremium(): entero depositosLlenos(): boolean

- `llenarDepositoGasoil()`, `llenarDepositoSuper()`, `llenarDepositoPremium()` se completa el depósito de acuerdo a la máxima carga.
- `extraerGasoil(litros:entero)`, `extraerSuper(litros: entero)`, `extraerPremium(litros:entero)` si la cantidad de combustibles es mayor a litros, decrementa la cantidad en litros, sino le asigna 0. Requiere litros > 0.
- `depositosLlenos(): boolean` retorna true si los tres depósitos tienen la máxima carga.

Edición, compilación y ejecución

La edición del código de cada clase se realiza dentro de un **entorno de desarrollo** que integra también recursos para compilar, depurar, ejecutar código y crear aplicaciones autónomas. La sigla IDE se utiliza justamente para referirse a un **entorno integrado de desarrollo**. Algunas IDEs brindan otros recursos que favorecen el proceso de desarrollo.

El desarrollador edita el **código fuente** de cada clase y luego las **compila**. Una de las clases debe incluir un método con la siguiente signatura:

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

En Java la ejecución comienza cuando se invoca el método `main` de una clase específica. Aunque Java es un lenguaje que *admite* la programación orientada a objetos, incluye varias características que lo alejan de este modelo de programación. En particular la clase que define al método `main` no modela a objetos del problema. En Java una clase puede ser utilizada para modelar objetos de un problema, tal como propone la programación orientada a objetos, pero también puede ser utilizada con otro propósito. Java no se considera un lenguaje orientado a objetos puro porque admite otros modelos de programación.

Todas las clases que conforman el sistema deben haberse compilado antes de que se invoque a `main`. La ejecución provocará la creación de algunos objetos de otras clases que recibirán mensajes y probablemente crearán objetos de otras clases.

La verificación y depuración de una clase

Un sistema de software orientado a objetos está conformado por una colección de clases. Cada clase implementada debe ser verificada individualmente antes de integrarse con el resto de las clases que conforman la colección. La verificación de una clase aspira a detectar y depurar errores de **ejecución** o de **aplicación**. Los **errores de compilación** ya han sido corregidos antes de comenzar la verificación.

Los **errores de ejecución** provocan la terminación anormal del sistema. Para detectarlos es necesario anticipar distintos **flujos de ejecución**, definiendo **casos de prueba** que conducen a una terminación anormal. Java previene muchos errores de ejecución imponiendo chequeos durante la compilación, sin embargo no controla situaciones como por ejemplo división por 0. En los capítulos que siguen se describen otros tipos de errores de ejecución que es importante detectar y depurar.

Los **errores de aplicación**, también llamados errores lógicos, son probablemente los más difíciles de detectar, el sistema no termina en forma anormal, pero los resultados no son correctos. La causa puede ser que no se hayan especificado correctamente los requerimientos o la funcionalidad, o bien que el diseñador haya elaborado correctamente la especificación, pero la implementación no sea consistente con ella.

En un sistema de mediana o alta complejidad probablemente los roles de analista, diseñador, desarrollador y responsable de verificación y depuración sean ocupados por diferentes miembros del equipo de desarrollo. Sin embargo, con frecuencia el diseñador define un conjunto de **casos de prueba** que el responsable de la verificación debe utilizar.

En todos los casos el objetivo es detectar y depurar los errores, considerando las responsabilidades asignadas a la clase y la funcionalidad establecida para cada servicio. Es importante considerar que la verificación muestra la presencia de errores, no garantiza la ausencia.

Una clase **tester** es aquella que verifica los servicios de una o más clases para un conjunto de casos de prueba. Los casos de prueba pueden ser valores:

- Fijos establecidos en el código de la clase tester
- Leídos de un archivo, por consola o a través una interfaz gráfica
- Generados al azar

Los casos de prueba propuestos aspiran detectar errores internos en el código, no fallas externas al sistema o situaciones que no fueron previstas en el diseño.

La siguiente clase tester utiliza valores fijos e incluye el método `main` que inicia la ejecución:

Caso de Estudio: La clase tester para la clase PresionArterial

```
class TestPresion {
public static void main (String a[]){
    PresionArterial mDia;
    PresionArterial mNoche;
    mDia = new PresionArterial(115,60);
    mNoche = new PresionArterial(110,62);
    int p1 = mDia.obtenerPulso();
    int p2 = mNoche.obtenerPulso();
    System.out.println("pulso en la medición del día "+p1);
    System.out.println("pulso en la medición de la noche "+p2);
}
}
```

La compilación de la clase tester exige que la clase `PresionArterial` compile también. Una vez que el compilador no detecta errores, es posible invocar el método `main` a partir del cual se inicia la ejecución. Las instrucciones:

```
PresionArterial mDia;
PresionArterial mNoche;
```

declaran dos variables de clase `PresionArterial` y son equivalentes a:

```
PresionArterial mDia,mNoche;
```

Las instrucciones:

```
mDia = new PresionArterial(115,60);
mNoche = new PresionArterial(110,62);
```

crean dos objetos, cada uno de los cuales queda **ligado** a una variable. La instrucción:

```
int p1 = mDia.obtenerPulso();
```

envía el mensaje `obtenerPulso()` al objeto ligado a la variable `mDia`. El mensaje provoca la ejecución del método provisto por la clase y retorna un valor de tipo `int` que se asigna a la variable `p1`.

Las instrucciones:

```
System.out.println("pulso en la medición del día "+p1);
System.out.println("pulso en la medición de la noche "+p2);
```

muestran una cadena de caracteres en consola. `System` recibe el **mensaje** `out.println`. El parámetro de este mensaje es una **cadena de caracteres**, esto es un objeto de la clase `String`, provista por Java. La cadena se genera **concatenando cadenas de caracteres**. El valor de `p1` es de tipo `int`, y se convierte automáticamente a una cadena de caracteres, antes de concatenarse para formar otra cadena.

Ejercicio: Dada la clase `Poliza` implementada antes:

- Implemente una clase `TesterPoliza` con un método `main()` que solicite al usuario los datos de una póliza, controle que los tres valores sean positivos, cree un objeto de la clase `Poliza` usando el constructor con tres parámetros y a continuación, actualice con un porcentaje de 20%, desactive, actualice en 10%, active y muestre por pantalla el número de póliza, el costo y el estado (activa o no).
- Modifique el método `main` de la clase `TesterPoliza` agregando instrucciones para crear un objeto de la clase `Poliza` usando el constructor con un parámetro con valor fijo 111, luego establezca valores 1000 y 1200 para robo e incendio, actualícelos en un 15% y a continuación muestre en consola el número de póliza y el costo.

Objetos y referencias

En Java los objetos son **referenciados** a través de variables. La ejecución de la instrucción:

```
PresionArterial mDia = new PresionArterial(115,60);
```

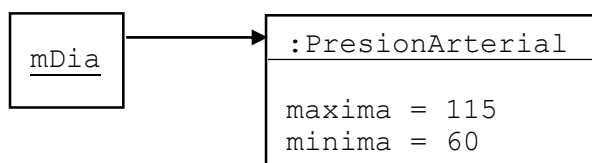
tiene el siguiente significado:

- **Declara** la variable `mDia`
- **Crea** un objeto de clase `PresionArterial`
- **Invoca** al constructor de la clase `PresionArterial`
- **Liga** el objeto a la variable `mDia`

Una **variable ligada** mantiene una **referencia** al **estado interno** de un objeto.

Un **diagrama de objetos** es una representación visual que modela el estado interno de uno o más objetos en ejecución.

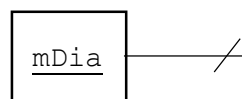
El diagrama de objetos, luego de ejecutada la asignación anterior es:



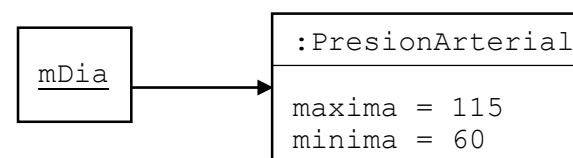
La declaración de la variable puede separarse de la creación del objeto:

```
PresionArterial mDia;  
mDia = new PresionArterial(115,60);
```

La primera instrucción crea una **referencia no ligada** y se grafica como sigue:



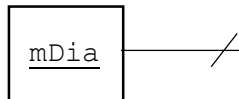
La segunda instrucción invoca al constructor y liga el objeto a la variable. El constructor crea al objeto e inicializa los atributos de instancia. El diagrama es nuevamente:



La variable `mDia` está ligada a un objeto de clase `PresionArterial`, esto es, mantiene una referencia al estado interno del objeto.

El estado interno del objeto ligado a la variable `mDia` está formado por los atributos `maxima` y `minima`. La estructura del estado interno depende de los atributos de instancia de la clase, no de los servicios provistos.

Se utiliza la misma representación gráfica:

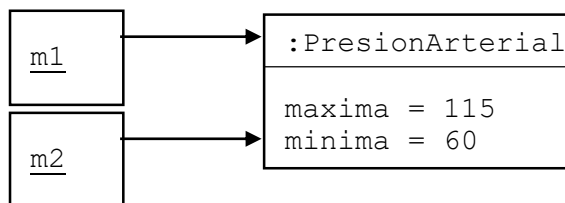


Para la instrucción

```
PresionArterial mDia = null;
```

La variable `mDia` mantiene una referencia nula. Dos variables pueden mantener referencias a un mismo objeto:

```
PresionArterial m1,m2
m1 = new PresionArterial(115,60);
m2 = m1;
```



Las variables `m1` y `m2` están ligadas a un mismo objeto.

Ejercicio: Dada la clase `Surtidor` implementada antes, muestre la salida y elabore el diagrama de objetos al terminar la ejecución del método `main`:

```
class TestDosSurtidores {
public static void main (String a[]){
    Surtidor s1,s2,s3;
    s1= new Surtidor();
    s2 = s1;
    s3 = new Surtidor();
    s1.extraerGasOil(10);
    s2.extraerGasOil(15);
    s3.extraerGasOil(30);
    System.out.println("Después de extraer 10,15,30 en s1,s2 y s3");
    System.out.println("Gasoil en s1 = "+s1.obtenerLitrosGasOil());
    System.out.println("Gasoil en s2 = "+s2.obtenerLitrosGasOil());
    System.out.println("Gasoil en s3 = "+s3.obtenerLitrosGasOil());
    s1.llenarDepositos();
    System.out.println("Después de llenar depósito de gasoil en s1");
    System.out.println("Gasoil en s1 = "+s1.obtenerLitrosGasOil());
    System.out.println("Gasoil en s2 = "+s2.obtenerLitrosGasOil());
    System.out.println("Gasoil en s3 = "+s3.obtenerLitrosGasOil()); }
}
```


Clases y tipos de datos

Como se mencionó antes, cuando el analista o el diseñador de un sistema orientado a objetos especifica una clase, establece sus atributos y servicios. Los servicios incluyen constructores y métodos, estos últimos conformados por comandos y consultas. En la implementación, una clase con esta estructura define un **tipo de dato** a partir del cual se pueden **declarar variables**.

Un **tipo de datos** establece un conjunto de valores y un conjunto de operaciones que se aplican sobre estos valores.

Cuando una clase define un tipo de datos, el conjunto de valores queda determinado por los valores de los atributos, el conjunto de operaciones lo definen los servicios provistos por la clase. Una variable declarada de un tipo definido por una clase, no mantiene un valor dentro del tipo, sino una referencia a un objeto cuyo estado interno mantiene un valor del tipo.

Caso de Estudio: Cuenta Corriente Bancaria

Un banco ofrece cajeros automáticos a través de los cuales los clientes pueden realizar depósitos, extracciones y consultar el saldo de su cuenta corriente. En el momento que se crea una cuenta corriente se establece su código y el saldo se inicializa en 0. También es posible crear una cuenta corriente estableciendo su código y saldo inicial. Una cuenta bancaria puede tener un saldo negativo hasta un máximo establecido por el banco, en ese caso la cuenta está “en descubierto”. En el momento de la creación el saldo debe ser mayor o igual a cero. El código no se modifica, el saldo cambia con cada depósito o extracción.

Cada cuenta corriente bancaria se modela mediante el siguiente diagrama:

CuentaBancaria
<<Atributos de clase>> maxDescubierto:real <<Atributos de instancia>> codigo:entero saldo:real
<<Constructores>> CuentaBancaria(cod:entero) CuentaBancaria(cod:entero, sal:real) <<Comandos>> depositar(mto:real) extraer(mto:real):boolean <<Consultas>> obtenerCodigo():entero obtenerSaldo():entero obtenerMaxDescubierto():entero enDescubierto():boolean toString():String

- CuentaBancaria(cod:entero) Requiere cod mayor a 0
- CuentaBancaria(cod:entero, sal:real) Requiere cod y sal mayores a 0
- depositar(mto:real) Requiere mto mayor a 0
- extraer(mto:real):boolean Requiere mto mayor a 0 Si mto mayor a saldo más maxDescubierto retorna false y la extracción no se realiza, sino modifica saldo y retorna true.

La especificación incluye un texto, el diagrama y la descripción de las funcionalidades y responsabilidades. En este caso es el texto que precede al diagrama el que le indica al programador como implementar el código de la consulta `enDescubierto()`.

Por convención, el método `toString()` retorna la concatenación de los valores de los atributos del objeto que recibe el mensaje. Al igual que los métodos con prefijo `obtener` y `establecer`, solo se especifica la funcionalidad si no es la convencional o tienen requisitos específicos.

La implementación de la clase que modela a una cuenta corriente bancaria es entonces:

```
class CuentaBancaria{
//Atributos de clase
/*Define el monto máximo que se puede extraer sin fondos en la
cuenta*/
private static final int maxDescubierto=1000;
//Atributos de Instancia
/*El codigo se establece al crear la cuenta corriente bancaria y no
cambia. El saldo va a ser siempre mayor a -maxDescubierto*/
private int codigo;
private float saldo;
//Constructores
public CuentaBancaria(int cod) {
//Requiere cod mayor a 0
    codigo = cod; saldo = 0;}
public CuentaBancaria(int cod, float sal) {
//Requiere cod y sal mayores a 0
    codigo = cod; saldo = sal;}
//Comandos
public void depositar(float mto){
//Requiere mto mayor a 0
    saldo+=mto;}
public boolean extraer(float mto){
/*Requiere mto mayor a 0. Si el mto es mayor a saldo+maxDescubierto
retorna false y la extracción no se realiza, sino modifica el saldo
y retorna true */
    boolean puede = true;
    if (saldo+maxDescubierto >= mto)
        saldo=saldo-mto;
    else
        puede = false;
    return puede;}
//Consultas
public int obtenerCodigo(){
    return codigo;}
public float obtenerSaldo(){
    return saldo;}
public float obtenerMaxDescubierto(){
    return maxDescubierto;}
public boolean enDescubierto(){
    return saldo < 0;}
public String toString(){
    return codigo+" "+saldo;} }
```

El identificador `CuentaBancaria` está **sobrecargado**, el tipo o número de parámetros debe ser diferente en cada definición del constructor. El primero recibe como parámetro una variable cuyo valor se utiliza para inicializar el código de la cuenta corriente en el momento de la creación. El segundo constructor recibe dos parámetros, cada uno de los cuales inicializa a un atributo de instancia.

La clase `CuentaBancaria` define un tipo de dato a partir del cual es posible declarar variables. Los servicios provistos por la clase conforman el conjunto de operaciones del tipo.

El siguiente segmento está incluido en el método `main` de una clase tester, responsable de verificar los servicios de `CuentaBancaria`:

```
CuentaBancaria cb, sm;  
cb = new CuentaBancaria(111,1000);  
sm = new CuentaBancaria(112);
```

La instrucción:

```
CuentaBancaria cb, sm;
```

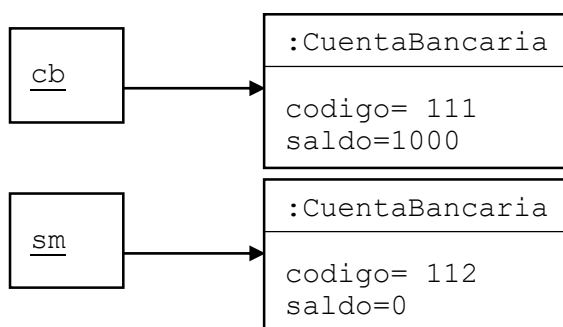
declara dos variables de tipo clase `CuentaBancaria`. Luego de la declaración ninguna de las variables está ligada.

Las instrucciones:

```
cb = new CuentaBancaria(111,1000);  
sm = new CuentaBancaria(112);
```

crean dos objetos de clase `CuentaBancaria`. Las variables `cb` y `sm` están ahora ligadas, es decir, el valor de cada variable es una **referencia ligada**.

En la primera instrucción el constructor está seguido de dos parámetros, se invoca entonces el constructor con dos parámetros definido en la clase `CuentaBancaria`. La segunda instrucción provoca la ejecución del primer constructor provisto por la clase.



Cualquiera sea el constructor que se ejecute, el **estado interno** de cada objeto de clase `CuentaBancaria` mantiene dos atributos, `codigo` y `saldo`. Como establece la especificación de requerimientos, el valor del atributo `codigo` se inicializa cuando se crea la cuenta corriente bancaria y no cambia.

Clases y paquetes

Un **paquete** en Java es un contenedor que agrupa un conjunto de clases con características comunes. El uso de paquetes favorece en primer lugar la reusabilidad porque permite utilizar clases que ya han sido diseñadas, implementadas y verificadas, fuera del contexto del sistema

que se está desarrollando. También favorece la eficiencia porque el lenguaje brinda un repertorio reducido de recursos, pero permite agregar otros **importando paquetes**.

En el caso de estudio de la Cuenta Corriente Bancaria la clase tester puede implementarse generando valores al azar, dentro de cierto rango:

```
import java.util.Random;
class Simulacion CuentaBancaria{
public static void main (String[] args) {
    Random gen;
    gen = new Random();
    CuentaBancaria cta = new CuentaBancaria(111,1000);
    int cantMovimientos = 0; int monto;
    int tipoMov; boolean desc=false;
    while (cantMovimientos<50 && !desc) {
        cantMovimientos ++;
        monto = gen.nextInt(500)+10;
        tipoMov = gen.nextInt(2)+1;
        if (tipoMov==1){
            cta.depositar(monto);
            System.out.println (cantMovimientos + " deposito " +
            monto+" Cuenta "+cta.obtenerCodigo()+
            " "+cta.obtenerSaldo());}
        else
            if (cta.extraer(monto))
                System.out.println (cantMovimientos + " extrajo "+ monto+
                " Saldo "+cta.obtenerSaldo());
            else
                System.out.println (i+" NO pudo extraer "+ monto+ " Saldo "
                +cta.obtenerSaldo());}} }
```

La definición de la clase `Simulacion_CuentaBancaria` está precedida por una instrucción que **importa** al paquete `java.util.Random` que ofrece a la clase `Random`. La clase `String`, provista por Java, está incluida dentro de los recursos básicos del lenguaje. Así, el lenguaje brinda algunos recursos básicos y permite utilizar otros importando los paquetes que los ofrecen.

La clase `Random` define un tipo de dato a partir del cual es posible crear instancias, desconociendo la representación de los datos y la implementación de las operaciones, por ejemplo `nextInt`. Sólo se conoce su **funcionalidad y signatura**, esto es, el tipo del resultado y el número y tipo de los parámetros.

En el simulador propuesto, los valores de las variables `monto` y `tipoMov` se generan al azar. El primero toma valores entre 10 y 510 y el segundo toma los valores 1 o 2. Si `tipoMov` es 1 se realiza un depósito, sino una extracción. El comando `depositar` requiere que su parámetro sea mayor a 0, la clase cliente es responsable de garantizar este requisito, sin embargo el tester no valida que la variable `monto` sea mayor a 0 ya que `nextInt` va a generar un valor positivo.

La cantidad de movimientos va a ser como máximo 50, puede ocurrir que la variable `desc` tome el valor `true` antes de que `cantMovimientos` sea 50 y en ese caso el bloque iterativo termina.

Ejercicio: Dada la clase `Surtidor` implementada antes, implemente en Java la clase `SimulacionSurtidor` que permita verificar los servicios provistos por la clase `Surtidor`, de acuerdo al algoritmo:

```
Algoritmo SimuladorSurtidor
crea un surtidor
n = leer cantidad de iteraciones
repetir n veces
    mostrar la cantidad actual en el depósito de cada combustible
    opcion = leer una opción entre 1 y 6
    según opción sea:
        1: leer litros a cargar y cargar Gasoil
        2: leer litros a cargar y cargar Super
        3: leer litros a cargar y cargar Premium
        4: llenar Deposito Gasoil
        5: llenar Deposito Super
        6: llenar Deposito Premium
```

Variables y alcance

La **declaración** de una variable establece su **nombre**, su **tipo** y su **alcance**. Java impone algunos requisitos para los nombres. El tipo puede ser elemental o una clase. El alcance de una variable determina su **visibilidad**, es decir, el segmento del código en el cual puede ser **usada**.

En Java una **variable declarada en una clase**, ya sea como atributo de clase o de instancia, es visible en toda la clase. Si se declara como **privada**, sólo puede ser usada dentro de la clase, esto es, el alcance es la clase completa.

En los casos de estudio propuestos los atributos, de instancia y de clase se declaran como privados y sólo se usan **atributos de clase** para representar valores **constantes**. En particular, en la clase `CuentaBancaria` las variables `codigo` y `saldo` son los atributos de instancia de la clase y pueden ser usadas en cualquiera de los servicios provistos por la clase. Como se declaran privados, desde el exterior sus valores sólo pueden ser accedidos por los servicios públicos que brinda la clase. La variable `maxDescubierto` es un atributo de clase, que sólo es visible en la clase.

En Java una **variable declarada local a un bloque**, se crea en el momento que se ejecuta la instrucción de declaración y se destruye cuando termina el bloque que corresponde a la declaración. El alcance de una variable local es entonces el bloque en el que se declara, de modo que sólo es **visible** en ese bloque.

En la clase `CuentaBancaria` la variable `puede`, declarada e inicializada en el método `extraer`, sólo es visible y puede ser accedida en el bloque de código de ese método.

```
public boolean extraer(float mto){
/*si mto es mayor a saldo+maxdescubierto retorna false y la
extracción no se realiza*/
    boolean puede = true;
    if (saldo+maxDescubierto >= mto)
        saldo=saldo-mto;
    else    puede = false;
    return puede;}

```

Una **variable declarada como parámetro formal de un servicio** se trata como una variable local que se crea en el momento que comienza la ejecución del servicio y se destruye cuando termina. Se inicializa con el valor del argumento o parámetro real. El pasaje de parámetros en Java es entonces **por valor**.

En el método `extraer` la variable `mto` es un parámetro, su alcance es el bloque del comando. Fuera del comando, la variable no es visible.

Cada bloque crea un nuevo **ambiente de referenciamiento**, formado por todos los identificadores que pueden ser usados. Los operandos de una **expresión** pueden ser constantes, atributos, variables locales y parámetros visibles en el ambiente de referencia en el que aparece la expresión.

Mensajes y métodos

Cuando un objeto recibe un mensaje, su clase determina el método que se va a ejecutar en respuesta a ese mensaje. En todos los casos de estudio propuestos, cuando un objeto recibe un mensaje, el **flujo de control** se interrumpe y el control pasa al método que se liga al mensaje. Al terminar la ejecución del método, el control vuelve a la instrucción que contiene al mensaje. Java permite modelar también procesamiento paralelo, esto es, ejecutar varios métodos simultáneamente. En este libro no se presentan conceptos de paralelismo.

Si un método retorna un valor, el tipo de la expresión que sigue a la palabra `return` debe ser compatible con el tipo que precede al nombre del método en el encabezamiento.

La clase `CuentaBancaria` brinda dos comandos para modificar el saldo: `depositar` y `extraer`. Ambos reciben como parámetro una variable `mto` que se requiere no negativa.

El siguiente segmento forma parte del método `main` en una clase tester de `CuentaBancaria`:

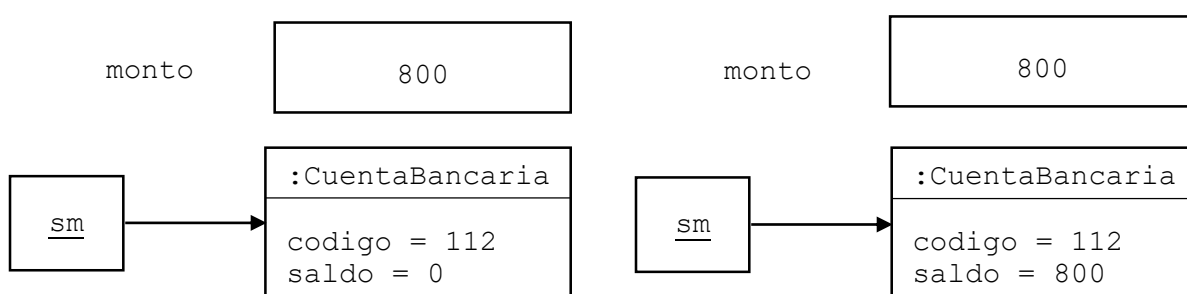
```
int monto = 800;
CuentaBancaria sm = new CuentaBancaria(112);
sm.depositar(monto);
if (!sm.extraer(100))
    System.out.println("No se pudo realizar la extracción");
System.out.println("El saldo es "+sm.toString());
```

En la instrucción `sm.depositar(monto)` la cuenta bancaria ligada a la variable `sm` recibe el mensaje `depositar` que provoca la ejecución del método:

```
public void depositar(float mto){
//Requiere mto > 0
    saldo+=mto;}

```

El diagrama de objetos antes y después de la ejecución de `sm.depositar(monto)` es:



La clase `tester` no controla que la variable `monto` contenga un valor positivo porque la verificación se realiza con **valores fijos**. Si los valores los ingresa el usuario, es necesario **validar la entrada**.

El comando `depositar` se declara de tipo `void`, para indicar que no retorna un resultado. El valor del parámetro real `monto` se utiliza para inicializar el valor del parámetro formal `mto`. La asignación modifica el valor del atributo de instancia `saldo` incrementando su valor de acuerdo al valor del parámetro `mto`, en este ejemplo el valor del saldo de la cuenta bancaria era `0` y pasó a ser `800`.

Si el valor de la variable `mto` se modificara en el bloque que corresponde al método, el cambio no sería visible al terminar la ejecución de `depositar`, porque el pasaje de parámetros es por valor. Es decir, cuando el control retorna al método `main`, el parámetro real conserva el valor, aun cuando el parámetro formal se modifique.

En el método `main` la instrucción condicional `if (!sm.extraer(100))` envía el mensaje `extraer` al objeto ligado a la variable `sm`. Cuando el objeto recibe el mensaje se ejecuta el comando:

```
public boolean extraer(float mto){  
    /*Si el mto es mayor a saldo+maxDescubierto retorna false y la  
    extracción no se realiza, sino modifica el saldo y retorna true */  
    boolean puede = true;  
    if (saldo+maxDescubierto >= mto)  
        saldo=saldo-mto;  
    else  
        puede = false;  
    return puede;}  

```

El parámetro formal `mto` se inicializa con el valor del parámetro real que en este caso es la constante `100`. La variable local `puede` se inicializa en `true`. En este ejemplo el condicional `(saldo+maxDescubierto >= mto)` computa `true`, de modo que la extracción puede realizarse, `saldo` se decrementa de acuerdo al valor del parámetro y la consulta retorna el valor `true`.

Cuando el comando `extraer` termina de ejecutarse, el control retorna a la instrucción condicional del método `main`. El resultado es justamente el valor de la expresión lógica del condicional.

Si en la clase `tester` la instrucción condicional fuera `if(!sm.extraer(10000))` nuevamente el objeto ligado a la variable `sm` recibe el mensaje `extraer`. Sin embargo, la expresión `(saldo+maxDescubierto >= mto)` en este caso computa `false`, de modo que la variable `puede` toma el valor `false` y `saldo` no se modifica.

La variable local `puede` se declara del tipo que corresponde al resultado, ya que su valor es el que retorna al terminar la ejecución del comando. Un método que modifica el valor de uno o más atributos del objeto que recibe el mensaje, es un comando, retorne o no un valor.

La última instrucción del segmento del método `main`:

```
System.out.println ("El saldo es "+sm.toString());
```

Envía el mensaje `toString()` al objeto ligado a la variable `sm`, que provoca la ejecución de la consulta:


```
public String toString(){
    return codigo+" "+saldo;}
```

La consulta retorna un resultado de tipo `String`, la expresión que sigue a la palabra `return` es la cadena de caracteres `112 400.0` que se genera concatenando los valores de los atributos `codigo` y `saldo`, separados por un espacio. Cuando la ejecución del método `toString()` termina, el control vuelve a la instrucción que contiene el mensaje `toString()`. La cadena que retorna se concatena con un cartel y se muestra en consola:

```
El saldo es 112 400.0
```

Como se mencionó antes, método `toString()` retorna la concatenación de los valores de los atributos del objeto que recibe el mensaje.

Parámetros y resultados de tipo clase

Un método puede recibir como parámetro o retornar como resultado a un objeto. En ambos casos, la variable que se recibe como parámetro o se retorna como resultado es de tipo clase.

Cuando en ejecución un mensaje se vincula a un método, el número de parámetros formales y reales es el mismo y los tipos son consistentes. Como se mencionó antes, en Java el pasaje de parámetros es por valor. Por cada parámetro formal se crea una variable y se le asigna el valor del parámetro real. Si el parámetro es de tipo clase, se asigna una referencia, de modo que el parámetro formal y el parámetro real mantienen una referencia a un mismo objeto.

Si un método modifica el valor de un parámetro formal, cuando la ejecución de ese método termina el cambio no persiste. En cambio, si un método modifica el estado interno de un objeto referenciado por un parámetro formal, cuando el método termina el estado interno se ha modificado.

El siguiente diagrama agrega nuevos servicios a la clase `CuentaBancaria`:

CuentaBancaria
<<Atributos de clase>> maxDescubierto:real <<Atributos de instancia>> codigo:entero saldo:real
<<Constructores>> CuentaBancaria(c:entero) CuentaBancaria(c:entero,s:float) <<Comandos>> depositar(mto:real) extraer(mto:real):boolean <<Consultas>> obtenerCodigo():entero obtenerSaldo():real obtenerMaxDescubierto():real enDescubierto():boolean mayorSaldo(cta:CuentaBancaria):entero ctaMayorSaldo(cta:CuentaBancaria): CuentaBancaria toString():String

La especificación para los nuevos servicios es:

- `mayorSaldo(cta:CuentaBancaria): entero` Requiere `cta` ligada. Retorna el código de la cuenta con mayor saldo.
- `ctaMayorSaldo(cta:CuentaBancaria):CuentaBancaria` Requiere `cta` ligada. Retorna la cuenta con mayor saldo.

La implementación de los dos nuevos servicios es:

```
public int mayorSaldo(CuentaBancaria cta){
/* Requiere cta ligada. Retorna el código de la cuenta con mayor
saldo. */
    if (saldo > cta.obtenerSaldo())
        return codigo;
    else
        return cta.obtenerCodigo();}
public CuentaBancaria ctaMayorSaldo(CuentaBancaria cta){
/* Requiere cta ligada. Retorna la cuenta con mayor saldo.*/
    if (saldo > cta.obtenerSaldo())
        return this;
    else
        return cta;}
```

En el siguiente segmento del método `main` en una clase tester:

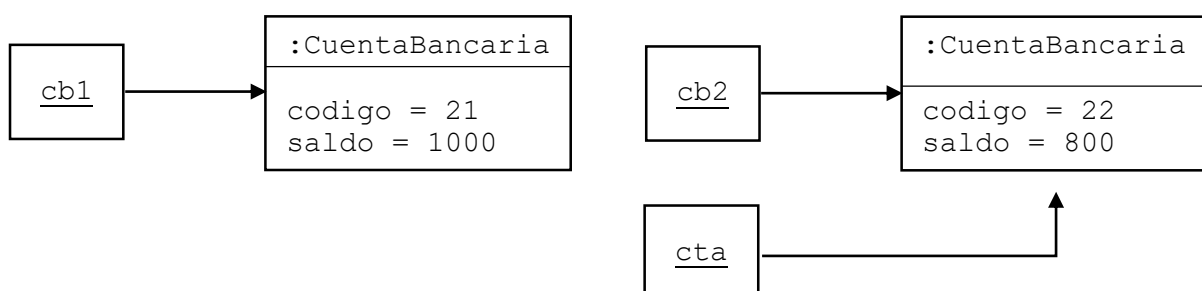
```
CuentaBancaria cb1,cb2,cb3;
cb1 = new CuentaBancaria(21,1000);
cb2 = new CuentaBancaria(22,800);
System.out.println ("Codigo Cuenta Mayor Saldo "+
    cb1.mayorSaldo(cb2));
cb3 = cb1.ctaMayorSaldo(cb2);
System.out.println ("Cuenta Mayor Saldo "+ cb3.toString());
```

La instrucción que envía el mensaje `mayorSaldo` al objeto ligado a la variable `cb1`, provoca la ejecución del servicio `mayorSaldo` provisto por `CuentaBancaria`.

```
public int mayorSaldo(CuentaBancaria cta){
/*Compara el saldo de la cuenta corriente bancaria con el saldo de
cta y retorna el CODIGO de la cuenta corriente bancaria que tiene
mayor saldo. Requiere cta ligada*/
    if (saldo > cta.obtenerSaldo())
        return codigo;
    else
        return cta.obtenerCodigo(); }
```

La consulta `mayorSaldo` recibe una cuenta corriente como parámetro y retorna el código de la cuenta con mayor saldo, comparando el saldo de la cuenta corriente que recibe el mensaje con el saldo de la cuenta corriente recibida como parámetro.

El diagrama de objetos cuando comienza la ejecución de `mayorSaldo` es:



El parámetro formal `cta` está ligada a un objeto de clase `CuentaBancaria`, que en este caso también está referenciado por el parámetro real `cb2`. Las variables `cb1` y `cb2` no son visibles en el bloque de código del método `mayorSaldo`. La variable `cta` sólo es visible y puede ser usada en el bloque de código de `mayorSaldo`. Cuando el método termina, la variable `cta` se destruye, aun cuando el objeto sigue existiendo.

La ejecución de `cta.obtenerSaldo()` retorna el valor `800` que corresponde al saldo de la cuenta `cb2` en la clase `tester`. La expresión `(saldo > cta.obtenerSaldo())` computa `true` al comparar `1000` y `800` de modo que se ejecuta `return codigo` y el resultado es `21`, el código de la cuenta corriente con mayor saldo entre el objeto que recibió el mensaje y el que pasó como parámetro.

La consulta `mayorSaldo` es una operación binaria, los operandos son el objeto que recibe el mensaje y el parámetro. Los atributos del objeto que recibe el mensaje se acceden directamente, los atributos del objeto que se recibe como parámetro se acceden a través de las operaciones `obtenerCodigo()` y `obtenerSaldo()`.

La programación estructura recomienda que cada instrucción tenga un único punto de entrada y uno de salida, en particular un método debe incluir una única instrucción de retorno al final, o dos instrucciones, sólo en el caso de que el código contenga una única instrucción `if-else` con una instrucción simple en cada alternativa.

La instrucción:

```
System.out.println ("Codigo Cuenta Mayor Saldo  
"+cb1.mayorSaldo(cb2));
```

Muestra en consola:

```
Codigo Cuenta Mayor Saldo 21
```

La instrucción:

```
cb3 = cb1.ctaMayorSaldo(cb2);
```

Envía el mensaje `ctaMayorSaldo` al objeto ligado a la variable `cb1`. El mensaje provoca la ejecución del método con el mismo nombre.

```
public CuentaBancaria ctaMayorSaldo(CuentaBancaria cta){  
/*Retorna la cuenta corriente bancaria que tiene mayor saldo  
Requiere cta ligada*/  
    if (saldo > cta.obtenerSaldo())  
        return this;  
    else  
        return cta;    }
```

La consulta `ctaMayorSaldo` recibe una cuenta corriente como parámetro y retorna la cuenta corriente con mayor saldo, comparando el saldo de la cuenta que recibe el mensaje, con el saldo de la cuenta recibida como parámetro. El alcance de la variable de `cta` es el bloque del método `ctaMayorSaldo` que la recibe como parámetro. Sólo se accede a los atributos de `cta` a partir de los servicios provistos por su clase.

La instrucción condicional `if (saldo > cta.obtenerSaldo())` compara nuevamente el saldo del objeto que recibió el mensaje con el saldo del objeto que pasó como parámetro. La variable `this` mantiene referencia al objeto que recibe el mensaje, como la expresión

condicional computa `true`, se ejecuta `return this` y retorna al método `main` el objeto que recibió el mensaje, que se liga a la variable `cb3`.

Si el método `main` de la clase `tester` incluye la instrucción:

```
cb3 = cb2.ctaMayorSaldo(cb1);
```

El objeto ligado a la variable `cb2` recibe el mensaje `ctaMayorSaldo` con `cb1` como parámetro. En este caso la expresión `(saldo > cta.obtenerSaldo())` computa `false` de modo que se ejecuta el bloque del `else`, esto es `return cta`. Al terminar la ejecución del método `ctaMayorSaldo` y retornar el control al método `main`, el resultado se asigna a `cb3`.

El método `ctaMayorSaldo` recibe como parámetro a un objeto y retorna como resultado a un objeto. El tipo de la expresión que sigue a la palabra `return` se corresponde con el tipo que precede al nombre de la consulta en el encabezamiento.

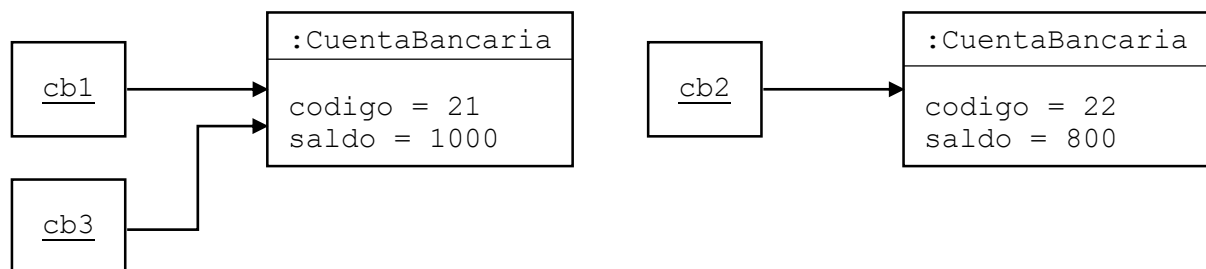
Tanto al terminar la ejecución de la instrucción:

```
cb3 = cb1.ctaMayorSaldo(cb2);
```

como luego de ejecutarse:

```
cb3 = cb2.ctaMayorSaldo(cb1);
```

el diagrama de objetos será:



Es decir, en ambos casos las variables `cb1` y `cb3` quedan ligadas a un mismo objeto.

La ejecución de:

```
System.out.println ("Cuenta Mayor Saldo "+ cb3.toString());
```

Envía el mensaje `toString` al objeto ligado a `cb3`. En este ejemplo, la salida en consola será la misma que la salida que provoca la ejecución de:

```
System.out.println ("Cuenta Mayor Saldo "+ cb1.toString());
```

El método `toString()` retorna un resultado de tipo `String`, esto es, una cadena de caracteres. Expresado con mayor precisión, el método `toString()` retorna una referencia a un objeto de clase `String`. El mensaje `System.out.println` requiere como parámetro un objeto de clase `String`. Cuando un parámetro es de tipo clase, el método recibe una referencia.

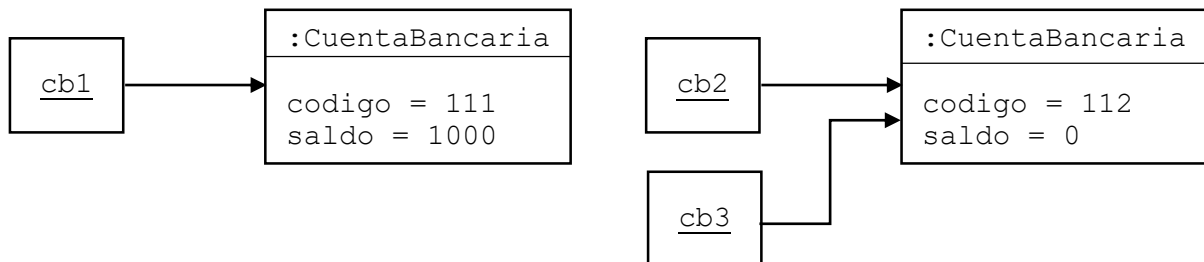
Identidad, igualdad y equivalencia

Cada objeto de software tiene una **identidad**, una propiedad que lo distingue de los demás. La **referencia** a un objeto puede ser usada como propiedad para identificarlo.

Si varias variables están ligadas a un mismo objeto, todas mantienen una misma referencia, esto es, comparten una misma identidad. La siguiente secuencia:

```
CuentaBancaria cb1 = new CuentaBancaria(111,1000);  
CuentaBancaria cb2,cb3;  
cb2 = new CuentaBancaria(112);  
cb3 = cb2;
```

Se representa gráficamente mediante el siguiente diagrama de objetos:



Las variables `cb2` y `cb3` tienen la misma identidad, mantienen referencias a un mismo objeto. Si el objeto recibe un mensaje que modifica su estado interno, como por ejemplo en la primera instrucción de:

```
cb2.depositar (200);  
System.out.println (cb3.toString());
```

Se modifica el estado interno del objeto ligado a las variables `cb2` y `cb3`. La segunda instrucción muestra en consola:

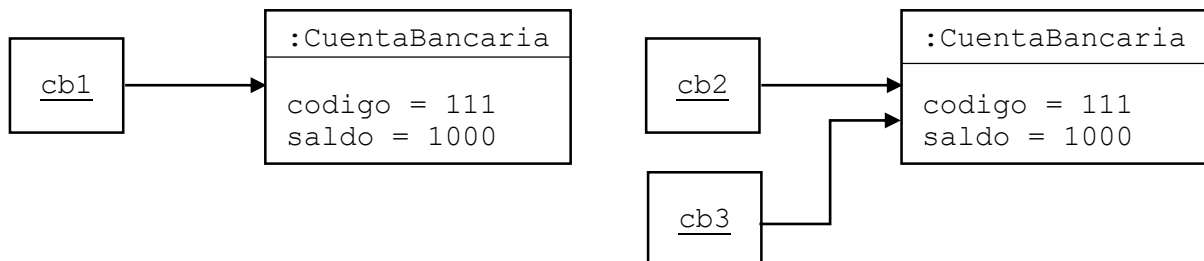
```
112 200.0
```

Como las dos variables están ligadas a un mismo objeto, el mensaje `depositar` modifica el atributo de instancia `saldo` y el cambio es visible al ejecutarse el método `toString()`.

Los **operadores relacionales** comparan los valores de las variables. Cuando se aplica el operador de igualdad sobre variables de tipo clase, se comparan los valores de las variables. El valor de cada variable es una referencia nula o ligada a un objeto. Por ejemplo, en el siguiente segmento de código:

```
CuentaBancaria cb1,cb2,cb3;  
cb1 = new CuentaBancaria(111,1000);  
cb2 = new CuentaBancaria(111,1000);  
cb3 = cb2;
```

Las variables `cb2` y `cb3` referencian a un mismo objeto, tienen entonces una misma identidad. El diagrama de objetos es:



Si a continuación se ejecutan las siguientes instrucciones:

```
boolean b1,b2;  
b1 = cb2 == cb1;  
b2 = cb2 == cb3;
```

La variable lógica `b1` toma el valor `false` porque `cb1` y `cb2` mantienen referencias a distintos objetos. La variable `b2` toma el valor `true` porque `cb2` y `cb3` mantienen un mismo valor, esto es, referencian a un mismo objeto.

Para comparar el **estado interno** de los objetos, se comparan los valores de los atributos de instancia. El método `main` en la clase `tester` puede incluir por ejemplo la siguiente instrucción condicional:

```
if (cb1.obtenerCodigo()==cb2.obtenerCodigo() &&
    cb1.obtenerSaldo() == cb2.obtenerSaldo())
```

En este caso la comparación de los estados internos puede resolverse con una expresión con dos subexpresiones. Si una clase tiene 10 atributos, cada vez que se comparan dos objetos es necesario comparar los 10 atributos. Más aun, si la representación interna de la cuenta bancaria cambia, es necesario modificar todas las clases que usan los servicios provistos por `CuentaBancaria` y comparan el estado interno.

Cuando una clase define un tipo de dato, por lo general va a incluir un método que decide si dos objetos que son instancias de esa clase, tienen el mismo estado interno. Por convención se utiliza el nombre `equals` para este método.

También es habitual que una clase incluya métodos para copiar y clonar. El primero copia el estado interno de un objeto en otro. El segundo crea un clon del objeto que recibe el mensaje; el nuevo objeto puede evolucionar luego de manera autónoma al original. En este libro se adopta la convención de llamar `copy` y `clone` a estos métodos.

El siguiente diagrama agrega nuevos servicios a la clase `CuentaBancaria`:

CuentaBancaria
<<Atributos de clase>> maxDescubierto:real <<Atributos de instancia>> codigo:entero saldo:real
<Cconstructores>> CuentaBancaria(c:entero) CuentaBancaria(c:entero,s:float)
<<Comandos>> depositar(mto:real) extraer(mto:real):boolean copy(cta:CuentaBancaria)
<<Consultas>> obtenerCodigo():entero obtenerSaldo():real obtenerMaxDescubierto():real enDescubierto():boolean mayorSaldo(cta:CuentaBancaria):entero ctaMayorSaldo(cta:CuentaBancaria):CuentaBancaria toString():String equals(cta:CuentaBancaria):boolean clone():CuentaBancaria

Describimos las responsabilidades solo de los nuevos servicios:

- `equals(cta:CuentaBancaria):boolean` Requiere `cta` ligada

- `copy(cta:CuentaBancaria)` Requiere `cta` ligada

El método `equals` por convención compara el estado interno del objeto que recibe el mensaje, con el estado interno del objeto que pasa como parámetro. El resultado debe ser `true` si el `codigo` del objeto que recibe el mensaje es igual al `codigo` del objeto que se liga a la variable `cta` y el `saldo` del objeto que recibe el mensaje es igual al `saldo` del objeto ligado a `cta`. Si la funcionalidad no fuera la convencional debería describirse en la especificación.

```
public boolean equals(CuentaBancaria cta){  
    //Requiere cta ligada  
    return codigo == cta.obtenerCodigo() &&  
        saldo == cta.obtenerSaldo();}
```

El estado del objeto que recibe el mensaje se accede directamente a través de los atributos de instancia. El estado del objeto que pasa como parámetro se accede indirectamente a través de las operaciones provistas por la clase.

La consulta requiere que el parámetro esté ligado. Si no fuera así, esto es, si la variable `cta` mantuviera una referencia nula, la expresión `cta.obtenerCodigo()` provocaría la terminación anormal del programa.

Un diseño alternativo podría indicar expresamente que el método `equals` tiene la responsabilidad de controlar si el parámetro está ligado. También podría ocurrir que la especificación no indique explícitamente de quién es la responsabilidad de realizar este control. En ambos casos la implementación de `equals` sería:

```
public boolean equals(CuentaBancaria cta){  
    boolean e = false;  
    if (cta != null)  
        e = codigo == cta.obtenerCodigo() &&  
            saldo == cta.obtenerSaldo();  
    return e;}
```

Cualquiera sea el diseño y la implementación de `equals`, la clase `tester` puede usarlo para comparar el estado interno de los objetos:

```
if (cb1.equals(cb2))
```

El método `equals` es una **operación binaria**, un operando es el objeto que recibe el mensaje, el otro operando es el objeto que pasa como parámetro.

Los objetos ligados `cb1` y `cb2` tienen distinta **identidad** pero pueden considerarse **equivalentes**, en el sentido de que modelan a objetos con el mismo estado interno.

El operador relacional `==` decide si dos objetos tienen la misma identidad. El servicio `equals` provisto por la clase `CuentaBancaria` decide si dos objetos son equivalentes.

El método `copy` modifica el estado interno del objeto que recibe el mensaje, con el estado interno del objeto que pasa como parámetro. El estado del objeto que recibe el mensaje se accede directamente a través de los atributos de instancia.

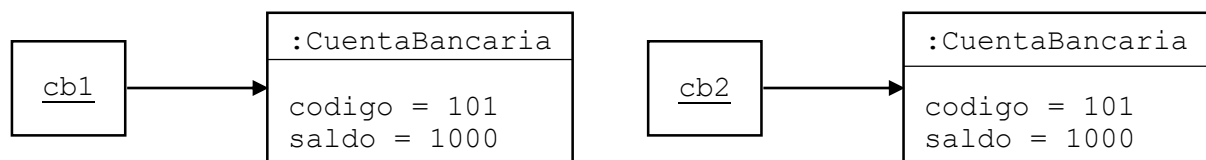
```
public void copy (CuentaBancaria cta) {  
    //Requiere cta ligada  
    codigo = cta.obtenerCodigo();  
    saldo = cta.obtenerSaldo();}
```


El estado del objeto que pasa como parámetro se accede indirectamente a través de las operaciones obtener provistas por la clase. Nuevamente, de acuerdo a la especificación, el método requiere que la variable `cta` esté ligada.

Dado el siguiente segmento de código en la clase tester:

```
CuentaBancaria cb1,cb2;
cb1 = new CuentaBancaria(101,1000);
cb2 = new CuentaBancaria(102,500);
cb2.copy(cb1);
```

Cuando el objeto ligado a la variable `cb2` recibe el mensaje `copy` su estado interno se modifica con los valores del objeto ligado a `cb1`, el diagrama de objetos al terminar la ejecución del segmento, es entonces:



Si a continuación se ejecuta el siguiente segmento en la clase tester:

```
boolean b1 = cb1==cb2;
boolean b2 = cb1.equals(cb2);
```

Los valores de las variables `b1` y `b2` serán `false` y `true` respectivamente. Los objetos no tienen la misma identidad, pero son equivalentes. Si el objeto ligado a `cb1` recibe en mensaje que modifica su estado interno, el objeto ligado a `cb2` no modifica su estado interno.

La consulta `clone` retorna la referencia a un objeto con el mismo estado interno que el objeto que recibe el mensaje.

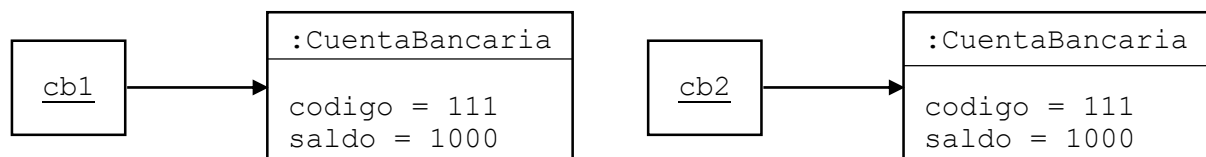
```
public CuentaBancaria clone() {
    CuentaBancaria aux = new CuentaBancaria(codigo,saldo);
    return aux;}

```

Después de la ejecución del siguiente segmento:

```
CuentaBancaria cb1,cb2;
cb1 = new CuentaBancaria(111,1000);
cb2 = cb1.clone();
```

El diagrama de objetos será:



Es importante observar que `cb2` no estaba ligada en el momento en que se envió el mensaje `clone`. Si a continuación se ejecutan las siguientes instrucciones:

```
boolean b1 = cb1==cb2;
boolean b2 = cb1.equals(cb2);
```

las expresiones computan `false` y `true` para `b1` y `b2` respectivamente. Los objetos no tienen la misma identidad, pero son equivalentes.

Durante la ejecución de la consulta `clone()` las variables `cb1` y `cb2` **no son visibles**, están fuera del **alcance** del método. La variable `aux` es local a la consulta, cuando termina la ejecución de este método, la variable se destruye.

EJERCICIO. Implemente y verifique la clase `NaveEspacial` de acuerdo a la siguiente especificación.

NaveEspacial
<<Atributos de instancia>> velocidad:entero combustible:entero
<<Constructor>> NaveEspacial(v,c:entero) <<Comandos>> cambiarVelocidad(v:entero) cambiarCombustible(c:entero) copy(n:NaveEspacial) <<Consultas>> obtenerVelocidad():entero obtenerCombustible():entero equals(n:NaveEspacial):boolean clone():NaveEspacial

`copy(n:NaveEspacial)` y `equals(n:NaveEspacial):boolean` Requieren n ligado.

EJERCICIO. Implemente y verifique la clase `Refugio` de acuerdo a la siguiente especificación.

En un videojuego las criaturas habitan en refugios que disponen de cierta cantidad de alimentos, bebidas y camas. El atributo `camas` indica cuántas camas están ocupadas, `alimentos` indica cuántos alimentos hay en la alacena y `bebidas` cuántas bebidas hay en la alacena.

Refugio
<<Atributos de clase>> capacidadAlacena = 20, cantidadCamas = 10 <<Atributos de instancia>> alimentos, bebidas, camas: entero
<<Constructor>> Refugio(a, b, c: entero) <<Comandos>> consumirAlimento() consumirBebida() ocuparCama(): boolean desocuparCama(): boolean reponerAlimentos(n: entero): boolean reponerBebidas(n: entero): boolean <<Consultas>> obtenerAlimentos(): entero

```
obtenerBebidas(): entero
obtenerCamas(): entero
obtenerCapacidadAlacena(): entero
esHabitable(): boolean
disponibilidad(): entero
diasSupervivencia(): entero
mayorAlimentos(r: Refugio): Boolean
equals(r: Refugio): boolean
clone(): Refugio
toString(): String
```

- `Refugio(a, b, c: entero)` si la suma de los valores de los parámetros `a` y `b` es mayor a la capacidad de la alacena, asigna la mitad de esta capacidad a `alimentos` y la mitad a `bebidas`, sino asigna el valor de `a` al atributo `alimentos`, y el valor de `b` al atributo `bebidas`. Si el parámetro `c` es mayor a la cantidad de camas, asigna esta constante al atributo `camas`, sino le asigna `c`.
- `consumirAlimento()` decrementa el valor de los `alimentos` en 1, requiere que la clase cliente haya controlado que hay al menos un alimento.
- `consumirBebida()` decrementa el valor de los `bebidas` en 1, requiere que la clase cliente haya controlado que hay al menos una bebida.
- `reponerAlimentos(n: entero)` incrementa el valor de `alimentos` en `n`, siempre y cuando `n` sea positivo y la capacidad de la alacena lo permita; en ese caso el comando retorna `true`. Si `n` no es positivo o al sumar `n` a la cantidad de alimentos el valor supera a la capacidad de la alacena, el atributo no se modifica y el comando retorna `false`.
- `reponerBebidas(n: entero)` incrementa el valor de `bebidas` en `n`, siempre y cuando `n` sea positivo y la capacidad de la alacena lo permita; en ese caso el comando retorna `true`. Si `n` no es positivo o al sumar `n` a la cantidad de bebidas el valor supera a la capacidad de la alacena, el atributo no se modifica y el comando retorna `false`.
- `desocuparCama(): boolean` si hay camas ocupadas decrementa en 1 camas y retorna `true`, si no retorna `false`.
- `ocuparCama(): boolean` si hay camas disponibles incrementa `camas` en 1 y retorna `true`, si no retorna `false`.
- `esHabitable(): boolean` retorna `true` sí y solo sí `alimentos` y `bebidas` mantienen valores positivos y hay al menos una cama disponible.
- `disponibilidad(): entero` retorna la cantidad de camas disponibles.
- `diasSupervivencia(): entero` retorna el menor valor entre los atributos `alimentos` y `bebidas`.
- `mayorAlimentos(r: Refugio): boolean` retorna `true` si `r` está ligado y el refugio que recibe el mensaje tiene más alimentos que el refugio `r`, en caso contrario retorna `false`.
- `equals(r: Refugio): boolean` retorna `true` si `r` está ligado y tiene el mismo estado interno que el refugio que recibe el mensaje, en caso contrario retorna `false`.

Ejercicio: Dada la siguiente especificación:

Sensor
<pre><<atributos de clase>> max = 0,01 <<atributos de instancia>> p1: real p2: real</pre>
<pre><<constructor>> Sensor(p1, p2: real) <<comandos>> establacerP1(p: real) establacerP2(p: real) copy(s: Sensor) <<consultas>> obtenerP1(): real obtenerP2(): real</pre>
<pre>riesgo(): boolean emergencia(): boolean equals(s: Sensor): boolean clone(): Sensor</pre>

copy(s: Sensor) Requiere s
ligado

riesgo() retorna true si p2 es
mayor que p1

emergencia() retorna true si p1
es menor a max.

- Implemente la clase `Sensor` modelada en el diagrama de la clase. Tenga en cuenta que como no se especifica quién es responsable de controlar que el parámetro está ligado, es el método `equals` quien debe hacerlo y retornar `false` si no lo está.
- Elabore un diagrama de objetos al finalizar el siguiente segmento de código:

```
Sensor s1, s2, s3;
s1 = new Sensor(1.5, 1.6);
s2 = s1.clone();
s3 = new Sensor(1.5, 1.5);
System.out.println(s1 == s2);
System.out.println(s1.equals(s2));
System.out.println(s1.equals(s3));
```

- Dibuje un nuevo diagrama de objetos luego de que la ejecución continúa con el siguiente segmento de código:

```
s3.copy(s1);
s1 = s2;
System.out.println(s1 == s2);
System.out.println(s1.equals(s2));
System.out.println(s1.equals(s3));
```

Representación en memoria

La memoria de una computadora es el dispositivo en el que se almacenan datos e instrucciones. Toda la información se almacena utilizando el sistema de numeración binario. Textos, números, imágenes, sonidos y casi cualquier otra forma de información puede ser transformada en una sucesión de bits, o dígitos binarios, cada uno de los cuales tiene un valor de 1 o 0. La estructura de la memoria es así sumamente simple y puede graficarse como:

Dirección	Contenido
00000000	11011111
00000001	10101010
00000010	11110000
00000011	00000001

Cada celda de memoria tiene una **dirección** y un **contenido**. En este ejemplo, tanto la dirección como el contenido se representan mediante 8 bits. Un conjunto de celdas consecutivas pueden agruparse para definir un **bloque de memoria** que contenga a una **unidad de información**, por ejemplo, una cadena de caracteres. El contenido de una celda puede ser una dirección en memoria, en ese caso la celda contiene una referencia a otro bloque de memoria. En el diagrama de memoria propuesto, el contenido de la cuarta celda es la dirección de memoria de la segunda celda.

En Java el tipo de una variable puede ser elemental o una clase. La **representación interna en memoria** es diferente en cada caso. Una variable de tipo elemental almacena un valor de su tipo. Una variable de tipo clase almacena una **referencia** a un objeto de software de su clase.

Cuando en ejecución se alcanza una declaración de una variable local de tipo elemental se reserva un **bloque de memoria**. Por ejemplo:

```
int i=3;
```

reserva un bloque de memoria ligado a la variable **i** y se almacena el valor 3, o mejor dicho, la representación binaria del valor 3. Así, el bloque de memoria ligado a la variable mantiene un valor dentro del conjunto de valores que determina el tipo de dato. La asignación:

```
i=i+1;
```

Modifica el valor almacenado en el bloque de memoria ligado a **i**, almacenando el valor que resulta de computar la expresión **i+1**. La instrucción:

```
if (i==3)
```

Compara el valor almacenado en la celda de memoria ligada a la variable **i** con la constante 3. Se comparan las representaciones binarias de los valores, aun cuando se mantiene una visión más abstracta.

Cuando en ejecución se alcanza una **declaración de variable** cuyo tipo es una clase, la variable es de **tipo clase**, se reserva también un **bloque de memoria** que mantendrá inicialmente una **referencia no ligada**.

```
CuentaBancaria cb;
```

La **creación de un objeto** reserva un nuevo bloque de memoria para mantener el estado interno del objeto. El valor de la variable no pertenece al conjunto de valores que determina el tipo, sino que es una **dirección al bloque de memoria** en el que se almacena el estado interno del objeto.

```
cb = new CuentaBancaria (111,1000);
```

La variable **cb** mantiene la referencia al bloque de memoria en el cual reside efectivamente el objeto de la clase **CuentaBancaria**.

La variable `cb` está **ligada** al objeto que le fue asignado, esto es, es una referencia al estado interno del objeto en el cual se almacenan los valores de los atributos. Los atributos de instancia determinan la estructura del estado interno del objeto, no de la variable.

La ejecución de la secuencia:

```
CuentaBancaria cb;  
cb = new CuentaBancaria (111,1000);
```

es equivalente a:

```
CuentaBancaria cb = new CuentaBancaria (111,1000);
```

En términos de la memoria, el significado es:

- Reservar un bloque de memoria para almacenar el valor de la variable `cb`.
- Invocar al constructor con dos parámetros de clase `CuentaBancaria`.
- Almacenar en la variable `cb` la dirección del bloque de memoria que almacena el estado interno del objeto creado.

La invocación del constructor a su vez da lugar a la siguiente secuencia de acciones:

- Reservar un bloque de memoria para los parámetros formales.
- Asignar los valores de los parámetros reales a los parámetros formales.
- Reservar un bloque en memoria para almacenar el estado interno de un objeto de software de clase `CuentaBancaria`.
- Ejecutar el código del constructor.
- Liberar el bloque de memoria ocupado por los parámetros formales.

Es decir, cuando el constructor termina de ejecutarse se libera el bloque de memoria que ocuparon los parámetros formales, pero siguen almacenado en memoria el estado interno del objeto que acaba de crearse.

Dada la siguiente secuencia de instrucciones:

```
CuentaBancaria cb1,cb2;  
cb1 = new CuentaBancaria(13,450);  
cb2 = cb1;
```

La segunda asignación almacena en el bloque de memoria ligado a la variable `cb2`, el mismo valor que almacena el bloque de memoria ligado a la variable `cb1`. De modo que las dos variables mantienen la referencia a un mismo objeto.

La instrucción:

```
if (cb1==cb2)
```

Compara el valor almacenado en el bloque de memoria ligado a la variable `cb1`, con el valor almacenado en el bloque de memoria ligado a `cb2`, retorna `true` si ambos valores son iguales.

El diagrama de objetos es un modelo abstracto de la representación en memoria, no grafica la memoria o las direcciones en memoria, sino el estado interno de los objetos y las referencias.

Contenido

El concepto de objeto	2
El concepto de clase.....	3
El diseño de una clase	4
La implementación de una clase.....	7
La verificación y depuración de una clase	12
Objetos y referencias	14
Clases y tipos de datos.....	16
Clases y paquetes.....	18
Variables y alcance.....	20
Mensajes y métodos	21
Parámetros y resultados de tipo clase.....	23
Identidad, igualdad y equivalencia	26
Representación en memoria.....	33