Sobre la Programación

El concepto de **abstracción** es esencial en ciencias de la computación. La programación puede pensarse como un **proceso de abstracción** basado en reconocer los aspectos relevantes de un problema y descartar aquellos que no lo son. El resultado de este proceso es un **programa** que especifica **cómo** resolver un problema a partir del repertorio de las facilidades que brinda un lenguaje de programación.

Un **programa** es un modelo de la resolución de un problema y una abstracción en dos niveles:

- si consideramos una computadora como un dispositivo concreto, el programa es una abstracción porque se aleja de las instrucciones que brinda la máquina y se acerca al problema.
- si consideramos el problema, el programa es una abstracción porque sólo modela sus aspectos relevantes.

El desarrollo de software requiere de un proceso de abstracción que se aborda siguiendo un **paradigma de programación**. La elección del paradigma de programación está ligada al dominio de aplicación del problema. La elección del lenguaje de programación en el cual se implementará un programa depende fuertemente del paradigma de programación elegido.

Un paradigma de programación es una colección de patrones conceptuales que orientan el proceso de desarrollo de software y determinan la estructura de un programa válido. Usualmente, un paradigma puede ser caracterizado mediante un **principio básico**, fácil de formular. Para llevar a la práctica este principio, es necesario establecer un conjunto de **métodos y técnicas**, coherentes. Un paradigma guía y controla la forma en que un problema va a ser planteado y resuelto, provee una "**filosofía de desarrollo**".

Un lenguaje de programación es una notación formal para especificar un modelo para la resolución de un problema. Un programa escrito en un lenguaje de programación de alto nivel va a tener que ser **traducido**, en una o varias etapas, en una secuencia de instrucciones que la computadora pueda interpretar y ejecutar. El **compilador** es el encargado de detectar errores sintácticos y si no existen generar una primera traducción.

Correctitud: actúa de acuerdo a los requerimientos especificados.
Eficiencia: tiene una baja demanda de recursos de hardware, en particular tiempo de CPU, espacio de memoria y ancho de banda.
Portabilidad: puede ejecutarse sobre diferentes plataformas de hardware y de software.
Simplicidad: es fácil de usar, su interfaz es amigable y no requiere demasiado entrenamiento ni capacitación.
Confiabilidad: reacciona adecuadamente aun en circunstancias no especificadas en los requerimientos.
Extensibilidad: es fácil adaptarlo a cambios en la especificación.
Reusabilidad: puede utilizarse para la construcción de diferentes aplicaciones.

El software es el conjunto de programas que permiten que el conjunto de dispositivos físicos de una computadora puedan ser utilizados. Todo sistema de software tiene un ciclo de vida durante el cual atraviesa diferentes etapas.

1. Estudio de Factibilidad.
2. Especificación de Requerimientos: Los usuarios especifican sus necesidades a través de un requerimiento. El resultado de esta fase es un documento que establece qué hará el sistema y sirve de punto de partida para la documentación.
3. Análisis y Diseño del Sistema: A partir del análisis de los requerimientos se diseña una solución. El resultado de esta etapa es un documento que describe los módulos que integrarán el sistema, sus interfaces y el modo en que se relacionan entre sí y los casos de prueba. En la POO, se identifican **objetos** y se los caracteriza de acuerdo a sus **atributos** y **servicios** y se los agrupan en clases. El diseño especifica una **colección de clases** relacionados entre sí. Una clase **proveedora** *brinda servicios* que otras clases van a usar. Una clase **cliente** *usa los servicios* provistos por otra proveedora. En la etapa de diseño se establece un **contrato** entre cada clase proveedora y sus clientes. El contrato establece un

compromiso entre las clases. Los cambios en el contrato afectan a ambas. Una clase proveedora brinda una **interface** que permite que los clientes accedan a los servicios provistos. Los cambios en la implementación de una clase proveedora no afectan a las clases clientes, los cambios en la interface sí.

4. Implementación: A partir del diseño se genera el programa escrito en un lenguaje de programación y toda la documentación referida al código.
5. Verificación: Se valida la implementación respecto a la especificación de requerimientos. Aquí se ejecutan los casos de prueba para verificar su funcionalidad.
6. Mantenimiento: El mantenimiento involucra todos los cambios en el software que resultan de modificaciones en la especificación.
7. Ejecución: En **ejecución** cada objeto relevante del problema tiene una representación explícita, queda asociado a un **objeto de software** que lo modela.

El producto final de este proceso es un sistema de software que resuelve el problema planteao por la especificación de requerimientos.

Metodologías: Descomponer el problema en función de las acciones (PI), o de los datos (POO).
Paradigma Imperativo: propone que un programa está constituido por un programa principal y un conjunto de subprogramas. Cada subprograma es una secuencia de instrucciones que indican cómo se lleva a cabo la computación.
Metodología de Diseño Top-Down: propone dividir el problema en subproblemas.

Sobre la P.O.O.

La programación orientada a objetos brina un principio simple y una metodología que apoya el proceso de desarrolllo de software en todas sus etapas. La POO propone un modelo computacional en el cual cada objeto relevante de la vida real tiene una representación explícita en ejecución, esto es, queda asociado a un objeto concreto que modela al objeto real.

En el paradigma orientado a objetos un **programa** es una **colección de clases vinculadas a través de relaciones de asociación y herencia**. Las clases son patrones que modelan las entidades del problema. En ejecución, un programa es un mundo poblado de objetos cuyo comportamiento depende de la clase a la que pertenece cada uno y que interactúan entre sí a través de mensajes. El objetivo es aumentar la **calidad** y la **productividad** del software.

Un lenguaje orientado a objetos debe además permitir **encapsulamiento**, de manera tal que los atributos de una clase y la implementación de los servicios queden ocultos de modo que las modificaciones no afecten a los clientes de la clase. Los clientes de la clase conocen únicamente la **interface** que va a estar formada por el conjunto de servicios provistos. Los atributos de instancia solo son visibles dentro de la clase. Esto permite que la representación pueda cambiar y que este cambio afecte fundamentalmente a la clase que se modifica, no a las clases que la usan. El encapsulamiento en Java se da gracias a los modificadores de acceso.

En segundo lugar, un lenguaje orientado a objetos debe permitir la **abstracción de datos** (metodología de diseño que consiste en identificar las entidades de un problema y agruparlas de acuerdo a sus atributos y comportamiento, clases). Los lenguajes que soportan esta metodología brindan mecanismos para definir **Tipos de Datos Abstractos** (TDA). Un tipo de dato abstracto define un patrón de comportamiento para todas las instancias del tipo, de manera tal que la apariencia estructural y operacional de todas las instancias es la misma. Los lenguajes ofrecen además clases predefinidas, generales y reusables.

En muchas aplicaciones esta visión constituye una fuerte limitación. Aunque muchas instancias puedan estar caracterizadas por algunos atributos y algunas operaciones en común, no necesariamente compartirán todos.

Encapsulamiento y TDA

Las Ciencias de la Computación evolucionan para brindar lenguajes y metodologías que permitan cubrir la distancia entre los problemas reales y las facilidades provistas por una computadora. En esta evolución un concepto fundamental es el de **encapsulamiento**, mecanismo que permite la definición de módulos de software que pueden ser utilizados como "cajas negras", esto es, sabiendo **qué** hacen sin saber **cómo**.

El encapsulamiento permite esconder los detalles de la implementación de un módulo, de modo que sus clientes sólo conozcan su funcionalidad. Si cambian la implementación de un módulo, en tanto no cambie su funcionalidad, los módulos que lo usan no se verán afectados. Se **reducen** así las **dependencias entre diferentes unidades de software**, de modo que estos son más fáciles de leer, verificar y modificar.

Todo lenguaje de programación de alto nivel soporta cierto nivel de encapsulamiento en el concepto de **tipo de dato**. Cuando un lenguaje brinda, por ejemplo, el tipo de dato float, el programador puede usar un conjunto de facilidades cuya implementación concreta desconoce.

Conoce el rango de valores que una variable de tipo float puede tomar, conoce las operaciones provistas por el tipo, pero no conoce la **representación interna** en memoria de una variable ni el **código** específico de cada operación. La clase **String** provista por Java es también un tipo de dato a partir del cual es posible declarar variables. **Float** (tipo elemental) y **String** (tipo clase) son **tipos de datos abstractos**, el programador los usa sin conocer la representación interna de los valores del tipo ni la implementación (el código) de las operaciones.

Un lenguaje que soporta encapsulamiento debe brindar algún mecanismo para que el programador pueda no sólo **utilizar** sino también **definir** módulos de software independientes. Cuando un lenguaje permite definir clases y soporta encapsulamiento el programador puede definir **tipos de datos abstractos**.

Un **tipo de dato abstracto** es un patrón a partir del cual es posible crear instancias sin conocer la representación interna de los valores ni la implementación de las operaciones. Un tipo de dato es **abstracto** si la representación de su estado interno y la implementación de sus operaciones queda encapsulada.

La descomposición de un sistema en clases favorece la construcción de software **extensible** y **reusable**, pero es **insuficiente**. Por otra parte la organización de las entidades de un problema en clases también presenta algunas limitaciones. Por lo general cada grupo de objetos relevante en un sistema **comparte** algunos **atributos** y **comportamiento**, pero difiere en otros. Si diseñamos un modelo para todo el grupo de objetos, que incluya a todos los atributos posibles, la caracterización será excesiva y puede no ser significativa. Si en cambio, descomponemos el grupo en subgrupos de entidades con exactamente los mismos atributos, resultará redundante.

La POO propone construir un modelo a partir de la definición de clases pero además brinda el concepto de **herencia** para soportar clasificación. Las clases son las unidades fundamentales a partir de las cuales se construye el programa. Si hablamos de **abstracción** cuando **agrupamos los objetos del problema en clases**, podemos llamar **superabstracción** al proceso de **agrupar clases en superclases**.

La **herencia simple** exige que el proceso de clasificación se realice de manera tal que cada subclase corresponda a una única clase base. Gráficamente la estructura de clases forma un árbol que describe la **jerarquía de herencia**. En cada rama del árbol las clases superiores son los **ancestros** de las clases inferiores, que son sus **descendientes**. Las instancias de una clase derivada son también instancias de las clases ancestros, de modo que heredan sus atributos y métodos. El acceso a los atributos y servicios de una clase desde sus clases derivadas depende del nivel de **encapsulamiento**.

La **herencia múltiple** permite que una clase derivada pueda heredar de dos o más clases más generales. Es una alternativa poderosa pero más compleja.

El concepto de **polimorfismo** es central en la programación orientada a objetos. Polimorfismo significa muchas formas y en ciencias de la computación en particular se refiere a "**la capacidad de asociar diferentes definiciones a un mismo nombre, de modo que el contexto determine cuál**

corresponde usar". En el contexto de la programación orientada a objetos el polimorfismo está relacionado con variables, asignaciones y métodos.

Una **asignación polimórfica** liga un objeto de una clase a una variable declarada de otra clase. La posibilidad de que una variable pueda referenciar a objetos de diferentes clases y de que existan varias definiciones para una misma signatura, brinda **flexibilidad** al lenguaje siempre que además exista ligadura dinámica de código.

La **ligadura dinámica de código** es la vinculación en ejecución de un mensaje con un método.

Generalización, Herencia y Polimorfismo

Los lenguajes orientados a objetos se caracterizan por brindar mecanismos para soportar diferentes formas de relación, en particular **asociación** y **generalización-especialización**. La **generalización** implica reunir los atributos y comportamientos generales de varias clases específicas en una superclase. La clase más general incluye atributos y comportamiento compartido mientras que las clases especializadas sólo los atributos y comportamiento más específicos.

La herencia aumenta el nivel de abstracción porque permite **factorizar atributos** y **comportamientos** compartidos pero además porque con frecuencia permite que los cambios en la especificación del problema provoquen la **incorporación de nuevas clases, si necesidad de modificar las ya existentes**.

La implementación de una jerarquía de herencia implica escribir el código de las clases de manera tal que se respete el encapsulamiento y los compromisos y relaciones entre las clases establecidos durante el diseño. Existen diferentes criterios referidos al nivel de encapsulamiento que debería ligar a clases vinculadas por una relación de herencia.

Un argumento a favor de que las clases derivadas accedan a todos sus atributos, aun los que corresponden a las clases superiores en la jerarquía, es que una instancia de una clase específica es también una instancia de las clases más generales de modo que debería poder acceder y modificar su estado interno.

El argumento en contra es que si se modifica la clase base el cambio afectará a todas las clases derivadas que accedan directamente a la representación. **En esta materia consideramos que este es el argumento más fuerte**.

Una clase derivada hereda de la clase base todos sus atributos y métodos, pero no los constructores. Cada clase derivada tendrá sus propios constructores que pueden invocar a los constructores de las clases más generales. Para invocar al constructor de la clase base se usa la **palabra clave super**.

Dado que los atributos son siempre privados una subclase sólo podrá acceder a ellos a través de métodos definidos por la superclase. Un mismo nombre puede utilizarse para definir un método en la clase base y otro en la clase derivada.

Si en la clase derivada se define un método con el mismo nombre, número y tipo de parámetros que un método definido en la clase base, el método de la clase base queda **derogado**. Decimos que la definición de la clase derivada **redefine** al método de la clase base. Un método redefinido conserva la **signatura** de la definición propuesta en la clase base, esto es tiene exactamente el mismo nombre y el mismo número y tipo de parámetros, así como también el mismo tipo de resultado. Cuando en una clase derivada se define un método con el mismo nombre que en alguna clase ancestro, pero con diferente número o tipo de parámetros, el método queda **sobrecargado**.

Java brinda la clase predefinida Object como raíz de la jerarquía de herencia. Es la única clase que no está relacionada con una superclase. Todas las demás clases, explícita o implícitamente, **heredan de la clase Object**. La clase Object brinda el comportamiento común a todos los objetos. Los servicios provistos por Object van a ser redefinidos o sobrecargados por las clases derivadas para implementar comportamiento específico.

En ciencias de la computación **polimorfismo** se refiere a "**la capacidad de asociar diferentes definiciones a un mismo nombre, de modo que el contexto determine cuál corresponde usar**". Dado que una variable puede estar asociada a objetos de diferentes tipos

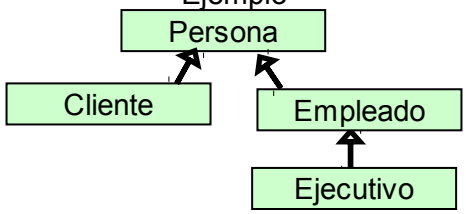
distinguiremos entre:

- El **tipo estático** de una variable, es el tipo que aparece en la declaración.
- El **tipo dinámico** de una variable, que se determina en ejecución y corresponde a la clase a la que corresponde el objeto referenciado.

El tipo estático de una entidad determina el conjunto de tipos dinámicos a los que puede quedar asociada. Una **asignación polimórfica** asocia a una variable declarada de una clase dada, un objeto de una clase derivada. La **variable** determina los mensajes que un objeto puede recibir, pero el **objeto** determina la implementación específica del comportamiento que se ejecuta en respuesta a los mensajes.

El polimorfismo es un mecanismo que favorece la **reusabilidad** pero debe restringirse para brindar **confiabilidad.** Los chequeos de tipos en compilación garantizan que no van a producirse errores de tipo en ejecución. El chequeo de tipos establece restricciones sobre las asignaciones polimórficas y los mensajes que un objeto puede recibir.

Ejemplo



El objeto referenciado por la variable p puede ser de clase Persona, Cliente, Empleado o Ejecutivo. Una asignación polimórfica liga un objeto de una clase a una variable declarada de otra clase.

Persona p;

Empleado e = new Empleado (…) ;Ejecutivo j = new Ejecutivo (…);

Son válidas las siguientes asignaciones polimórficas p = e; p = j; e = j;

En el caso p = new Empleado(…); el tipo estático de p es Persona mientras que el tipo dinámico de p es Empleado. El tipo estático Persona determina el conjunto de tipos dinámicos.

Un método útil para el polimorfismo es <ident>instanceof<Clase> que retorna un valor booleano si el objeto es de esa clase o una subclase de esa clase.

p instanceof Persona true
p instanceof Empleado true
p instanceof Ejecutivo false

Otro es <ident>.getClass() que retorna un String con el nombre de la clase a la que pertenece.

En ocasiones es conveniente que un método no sea redefinido en una clase derivada o incluso que una clase completa no pueda ser extendida. Se utiliza entonces el modificador **final**, que tiene significados levemente distintos según se aplique a una variable, a un método o a una clase. Para una **clase**, final significa que la clase **no puede extenderse** (final de la cadena de clases derivadas). Para un **método**, significa **que no puede redefinirse en una clase derivada**. Para una **variable**, final también significa que no puede ser redefinido en una clase derivada, pero además su valor no puede ser modificado.

Cuando un objeto recibe un mensaje es necesario determinar qué método va a ejecutarse en respuesta a dicho mensaje, esto es establecer la ligadura entre el mensaje y el método que va ejecutarse. Si el método se selecciona considerando la clase de acuerdo a la cual fue **declarada** la variable que mantiene la referencia al objeto, la **ligadura se dice estática** (o vinculación estática de código). En cambio, si el método se selecciona considerando la clase a la cual **pertenece** el objeto que recibe el mensaje, la **ligadura se dice dinámica** (o vinculación dinámica de código).

La ligadura estática es diferente a la dinámica si el método está redefinido y el objeto que recibe el mensaje no es de la misma clase de la variable que lo referencia.

Una **variable polimórfica** determina los mensajes que un objeto *puede* recibir, aunque el **objeto** determina la implementación específica del *comportamiento* que se ejecuta en respuesta a los mensajes. La **clase de la variable** determina el **conjunto de mensajes que un objeto puede recibir**. La **clase del objeto** determina en ejecución **la ligadura**.

Random

Un número aleatorio es un valor obtenido como resultado de computar una función de distribución. La clase Random de Java es un generador de número pseudo-aleatorios. Los números no son realmente aleatorios porque se obtienen a traves de un algoritmo qu genera una secuencia distribuida uniformemente, a partir de una semilla inicial.

Pasos a seguir para crear objetos al azar.

1. Importar el paquete que incluye a la clase Random (import java.util.Random;)
2. Crear un objeto de la clase Random
   Random rnd = new Random(); utiliza la hora como semilla.
   Random rnd = new Random(n); n=valor de la semilla.
3. Invocar a uno de los métodos que generan un número aleatorio
   rnd.nextInt(); rnd.nextInt(3); (0, 1 o 2) rnd.nextFloat();

String

La **clase String** provista por Java brinda facilidades para almacenar y procesar cadenas de caracteres. Un estado interno de una instancia de tipo String es una secuencia de caracteres encerrados entre comillas. Un objeto de clase String puede recibir mensajes que se ligan a métodos provistos por la clase. Ninguno de estos métodos modifica el estado interno.

Métodos:

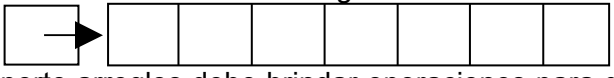
- **length** retorna la longitud.
- **ToLowerCase()** / **toUpperCase()** retorna el string en minúscula / mayúscula.
- **trim()** retorna el string sin espacios.
- **charAt(Pos)** retorna el carácter en la posición dada.
- **substring(Ini)** retorna la subcadena a partir de la posición Ini.
- **substring(Ini, Fin)** retorna la subcadena a partir de la posición Ini hasta la anterior Fin.
- **indexOf(A)** retorna la posición de la primera aparición de la subcadena A en la cadena (retorna -1 si no aparece).
- **indexOf(A,Pos)** retorna la posición de la primera aparición de la subcadena A en la cadena a partir de la posición Pos (retorna -1 si no aparece).
- **lastIndexOf(A)** retorna la posición de la última aparición de la subcadena A en la cadena (retorna -1 si no aparece).
- **A.compareTo(B)** retorna 1 si B es anterior a A, 0 si son iguales y -1 si A es anterior a B.

Arreglos

Un arreglo es una **estructura de datos** que agrupa componentes de un mismo tipo elemental o de una misma clase. La estructura de datos queda ligada a una variable mediante una **declaración** como float [] saldos. Una variable ligada a un arreglo referencia a un objeto que se **crea** mediante una instrucción. La cantidad de elementos queda definida en el momento de la creación. Si no podemos anticipar cuál va a ser la cantidad máxima, se establece un valor suficientemente grande y luego se usa la estructura parcialmente. En memoria se reservará espacio para la cantidad máxima establecida.

Cada **componente** puede accederse a través de una **posición** o **subíndice**. Un objeto arreglo tiene una **longitud** que queda fija en el momento de la creación y se almacena en una variable llamada **length (<Identificador>.length).**

En Java una variable declarada como un arreglo mantiene una referencia a un objeto arreglo.



Un lenguaje que soporte arreglos debe brindar operaciones para declarar una variable, crear la estructura y acceder a sus componentes.

Arreglos de Dos Dimensiones

Un arreglo de dos dimensiones es en realidad un arreglo cuyas componentes son arreglos. No todas las componentes tienen necesariamente la misma cantidad de elementos. Si todas las filas están ligadas a arreglos de una misma dimensión se puede pensar en el arreglo de dos dimensiones como una tabla o matriz.

La operación básica sigue siendo la subindicación, sólo que ahora cuando especificamos sólo un subíndice hacemos referencia a una fila completa en vez de un elemento. La variable length sigue siendo accesible, tabla.length nos proporciona el número de filas, y tabla[i].length, nos indica el número de elementos en la fila i (número de columnas).

La Clase Vector

La clase Vector provista por el paquete java.util permite crear estructuras de datos lineales, genéricas y dinámicas. En una estructura lineal cada elemento tiene uno anterior, excepto el primero, y cada elemento tiene uno siguiente, excepto el último. Una estructura genérica admite elementos de diferentes tipos (deben ser Tipo Clase). En una estructura dinámica la cantidad de elementos se establece en ejecución y puede modificarse.

La clase Vector es una **clase genérica**, se instancia con un tipo **específico cada vez que declaramos una variable**.

Un objeto de clase Vector es similar a un objeto de la clase Array excepto porque:

- los elementos son siempre de un tipo clase.
- la cantidad de elementos no queda determinada en la creación, sino que puede aumentar o disminuir en ejecución.
- el repertorio de servicios que brinda una instancia de Vector no se restringe a la subindicación.
- la clase Vector es más flexible que la clase Array pero como contrapartida es menos eficiente.

La notación para declarar una variable de clase Vector es /Vector < tipo clase > v. Para crear una instancia de la clase Vector v = new Vector <tipo clase>(<Num>);

Métodos:

- void addElement (Object element): agrega un elemento al final.
- void setElementAt (Object element, int index) : modifica el elemento iésimo
- Object elementAt (int index): retorna el elemento iésimo.
- int size (): retorna la cantidad de elementos.
- void insertElement (int index, Object element): agrega un elemento en la posición i, y desplaza al resto uno hacia abajo.
- boolean contains (Object elem)
- void removeElementAt (int index): elimina el elemento que ocupa la iésima posición en v y todos los elementos que siguen al iésimo disminuyen una posición.
- void removeElement (Object elem): elimina la primera aparición de la cadena especificada por el parámetro y todos los elementos que le siguen disminuyen una posición. Si se elimina un elemento la operación retorna true.
- boolean equals (Object o)
- void clear ()
- Object clone()

Recursividad

La recursividad puede ser utilizada para resolver problemas que exhiban las siguientes propiedades:

• Existe al menos una instancia del problema que puede resolverse de manera más o menos directa, llamada caso trivial.

• Resolver cualquier instancia del problema - excepto el o los casos triviales - requiere resolver una o varias instancias más simples del mismo problema. Más simples en el sentido de que están más cerca del caso trivial.

En Java cada nueva declaración de variable de un tipo elemental provoca que, en ejecución, se reserve una nueva celda de memoria. Esta celda se libera en el momento que termina el bloque que contiene la declaración. Si un método p invoca a otro q, cuando comienza la ejecución de q se reservarán locaciones de memoria para mantener los pará
metros de q. Al terminar la ejecución de q este espacio es liberado.

Si durante la ejecución de q se produce una llamada a sí mismo, tendremos una segunda "instancia" de q en ejecución, la primera instancia se suspende hasta que la instancia recursiva termine. Las locaciones reservadas durante la ejecución inicial de q son inaccesibles para la segunda instancia de q. Cuando la segunda instancia de q termine, el control vuelve a la primera instancia de q, exactamente al lugar siguiente a la llamada recursiva.

Modificadores de acceso

Los modificadores de acceso permiten que el programador determine el nivel de encapsulamiento.

Los **atributos de instancia** se declaran como **privados** para que no sean accesibles desde otras clases. Los **constructores** y los **métodos** que brindan servicios para las clases clientes se declaran como **públicos** para que sean visibles y puedan ser usados desde el exterior.

Para clases

- public: es visible para cualquier otra clase.
- final: no puede ser extendida.
- abstract: debe ser extendida.

Para métodos y atributos:

- public: es visible en todas las clases.
- private: sólo es visible dentro de la clase en la que se declara.
- protected es visible en el paquete y en las clases derivadas.

Para métodos:

- static: no hacen referencia a variables de instancia de la clase, sólo a sus parámetros.
- final: no pueden ser redefinidos.
- abstract: no está definido, se implementa en una clase derivada. Si un método se declara abstracto, la clase debe declararse como abstracta.

Para atributos:

- static: están compartidos por todas las instancias de la clase.
- final: el valor es constante.

Wrapper Class

Cuando usamos un cuadro de texto para tipear números que van a ser asignados a variables de tipos elementales es necesario convertir el tipo y aplicar el concepto de wrapper class. Una wrapper class permite crear objetos que se corresponden con valores de tipos elementales. Por cada tipo elemental existe una wrapper class. Por ejemplo el tipo elemental **int** está asociado al tipo clase **Integer**.

Pasar de String a int: int j=Integer.parseInteger(unaLinea.getText());

Clase Embebida

Una clase embebida es una clase que se define dentro de otra. Esta característica permite agrupar clases relacionadas y controlar la visibilidad. El acceso a una clase interna se limita el acceso a la clase externa. Un objeto de la clase interna estará siempre asociado a una instancia de la clase externa. Las instancias de la clase interna son creadas por instancias de la clase externa y tienen acceso a los atributos y servicios de la clase externa.

Adoptamos la convención de que el constructor invoca al constructor de la clase base y luego invoca a un método provisto por la misma clase que realiza el resto de la inicialización. Super() para crear el JFrame extendido, y armarGUI() (método interno) para crear los elementos, agregarle los oyentes y agregarlos al contenedor.

Objeto

La palabra objeto puede utilizarse para referirse a dos conceptos relacionados pero diferentes. En un problema a resolver podemos identificar los objetos o **entidades relevantes**. Un programa es un modelo de la resolución de un problema. En ejecución, cada objeto del problema esta asociado a una representación abstracta, un **objeto de software**.

En ejecución una clase puede pensarse como un generador a partir del cual se crean objetos de software. Cada objeto del problema está caracterizado a través de atributos y brinda un repertorio de servicios. Cada objeto de software tiene una **identidad** y un **estado interno** y ejecuta **operaciones** en respuesta a los **mensajes** que recibe. Las variables de instancia mantienen el estado interno de un objeto de software. Cuando hablamos del estado interno de un objeto estamos refiriéndonos al **estado de los atributos de instancia en un momento determinado**.

Cada objeto de software modela a un objeto del problema identificado en la etapa de diseño. El estado interno de un objeto puede contener referencias a otros objetos, de modo que un sistema complejo puede modelarse a partir de objetos simples.

Hay dos **diferencias concretas** entre una variable asociada a un objeto y una variable que contiene valores de un tipo elemental:

- la **declaración**
- la **modificación**

Variables

La memoria puede visualizarse como una secuencia de **celdas** cada una de las cuales tiene asociado una **dirección** y un **contenido**. Tanto la dirección como el contenido son secuencias de bits (0 y 1). Dos celdas pueden coincidir en su contenido, pero no en la dirección. Si el contenido de una celda o de un bloque de celdas mantiene almacenado un dato, la secuencia de bits mantiene el tipo y el valor del dato.

Una variable puede ser de tipo elemental o tipo clase. El tipo indica cómo va a interpretarse la secuencia de 0s y 1s, determina el conjunto de valores que puede tomar y las operaciones en las que puede participar. El concepto de tipo permite factorizar propiedades y establecer chequeos para prevenir errores.

Tipo elemental

byte →short→int→long→float→double / char / boolean

Una **variable de tipo elemental** es una **abstracción** de una celda de memoria o un bloque de celdas de memoria que contiene un **valor simple**. En el caso del tipo elemental el lenguaje nos permite desentendernos un poco de esa visión de celdas, direcciones y contenidos y pensar en función de sus atributos **valor**, **nombre** y **tipo**.

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 atómico indivisible** que corresponde al tipo y participa en las operaciones establecidas por su tipo. El programador no conoce la representación interna de cada valor ni el código de las operaciones, es un **cliente** de un tipo cuya implementación esta **encapsulada**.

Una variable de un tipo elemental se declara mediante una **instrucción simple**. Cuando se ejecuten estas instrucciones se reservan locaciones de memoria y se asignan directamente lo valores especificados. Una variable de un tipo elemental **recibe valores a través de asignaciones**.

Las expresiones son fundamentales para computar, consta de operandos y operadores, donde los operadores determinan el tipo esperado de los operandos y el tipo del valor computado. Si el tipo esperado difiere del operando puede suceder una conversión implícita, puede saltar un error de tipo o se puede hacer un casting. Una variable de un tipo elemental puede aparecer en cualquier expresión en la que se apliquen operadores con los cuales este tipo sea compatible.

Tipo clase

Una **variable de un tipo clase**, a diferencia de una de tipo elemental, no contiene al objeto propiamente dicho, sino una **referencia** al objeto. El **valor** de una variable de tipo clase es entonces una referencia que puede ser **nulo** o **asociado** a un objeto. El objeto será una **abstracción de una entidad** identificada durante la etapa de análisis y tendrá asociado un **estado interno** determinado por el estado de los atributos en un momento determinado.

Una vez que una clase ha sido definida podemos **declarar** variables de tipo clase mediante una instrucción y a partir de esta declaración es posible **crear** un objeto ligado a la variable.

La creación de un objeto provoca tres efectos:

1. *Ocupa* memoria para mantener el estado interno del objeto de acuerdo a la estructura determinada por la clase.
2. *Invoca* al constructor de la clase para inicializar el estado interno mantenido en los atributos de instancia.
3. *Liga* el objeto creado a la variable.

El objeto es una **instancia de la clase**. La creación reserva espacio de almacenamiento en memoria para mantener el estado interno del objeto. Si el estado está encapsulado, sólo puede ser **consultado o modificado en respuesta a un mensaje**.

Es importante distinguir entre el **nombre** de un objeto, el objeto mismo y el objeto del problema que representa. Un mismo objeto puede tener dos nombres diferentes y también es posible que dos objetos diferentes, con diferentes nombres, representen a un mismo objeto del problema.

Una clase puede pensarse como un **Proveedor de servicios**. Cada servicio va a ser usados desde una clase **Cliente**. Una clase Cliente solo tiene acceso a la **interface** de la clase Proveedora.

Clase

La unidad básica de programación en Java es la **clase**. Una clase es un equema que especifica los atributos y servicios compartidos por todos los objetos que pertenecen a ella. Un programa en Java está constituido por una colección de clases. La implementación de una clase consiste en definir sus **miembros**:

- **Nombre:** representa la abstracción del conjunto de instancias.
- **Atributos:** representan propiedades o cualidades relevantes y que caracterizan a todos los objetos de una clase. Son atributos de instancia y atributos de clase. Mantienen el **estado interno** de un objeto y son accesibles desde todos los servicios definidos en la clase. Cada atributo está asociado a un **tipo**.
- **Responsabilidades:** especifican las obligaciones y compromisos de la clase.
- **Servicios**: representan **operaciones** que todas las instancias de una clase pueden realizar cuando el sistema se ejecuta. El conjunto de servicios modela el **comportamiento** de los objetos de la clase. Son constructores y métodos. Los métodos pueden ser **comandos** o **consultas**.
- **Casos de prueba:** verifican si cada servicio satisface su funcionalidad y si en conjuno la clase cumple con sus responsabilidades.

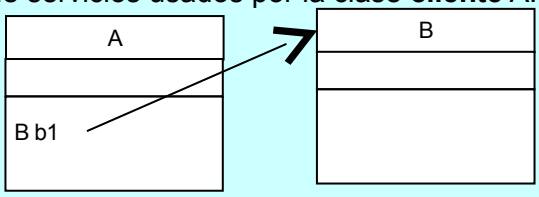
Constructores: forman parte de los servicios provistos por una clase y se caracterizan porque reciben el mismo nombre que la clase. La invocación de un constructor se realiza en la creación de un objeto (new Constructor()) y permite inicializar los atributos de instancia.

Consulta: Una consulta es un método que retorna un valor de un tipo establecido. La **interfaz** de una consulta establece el tipo del resultado y la implementación incluye una o más instrucciones para retornar un valor. Son métodos que computan un valor y devuelven un resultado (getter).

Comando: Un comando brinda un servicio sin retornar necesariamente un valor. Si no retorna un valor el método se define de tipo void. La instrucción return en este caso es opcional. Son métodos que en general modifican el estado interno (setter o mutators).

Relaciones Entre Clases

Las clases que conforman un sistema pueden **relacionarse** de diferentes maneras. Si una clase A crea objetos de la clase B, locales a algún método, decimos que A **usa** a B. Como vimos la clase B se dice **proveedora** de servicios usados por la clase **cliente** A.



Clase Abstracta

En la jerarquía de clases que modelan un sistema, las clases de los niveles superiores son generales y las de los niveles inferiores son más específicas. **Una clase es abstracta si no está asociada a entidades del problema**. En ejecución no va a haber instancias de una clase abstracta. Las clases abstractas pueden identificarse en la etapa de diseño de la aplicación o pueden crearse artificialmente durante la implementación.

Toda clase abstracta tiene que **especializarse** en al menos una clase concreta, esto es una clase que modele a entidades del problema. Es posible declarar variables como clases abstractas pero no crear objetos de una clase abstracta. De modo que la declaración que sigue es válida MaquinaExpendedora me; pero el compilador reportará un error si se intenta crear una instancia me = new MaquinaExpendedora().

Con frecuencia una clase general establece algunos de los servicios que brindan todas sus instancias, sin llegar a implementarlos de manera concreta. En Java, el modificador **abstract** permite declarar una clase abstracta. Una clase abstracta ***puede*** contener uno o más **métodos abstractos**, esto es, la clase **especifica un servicio pero no indica cómo se implementa**. Las clases derivadas deben redefinir los métodos abstractos provistos por la clase base, implementando el comportamiento específico. Si una clase derivada de una clase abstracta no implementa los métodos abstractos de su clase base, debe definirse también como abstracta o el compilador notifica un error.

Como no existen instancias de una clase abstracta, el constructor sólo puede ser invocado desde los constructores de las clases derivadas.

Genericidad

La programación orientada a objetos tiene como principales objetivos favorecer la **confiabilidad**, **reusabilidad** y **extensibilidad** del software. La extensibilidad se refiere a reducir el impacto de los cambios. Las modificaciones con frecuencia pueden resolverse definiendo nuevas clases específicas, sin necesidad de cambiar las que ya han sido verificadas. La reusabilidad evita escribir el mismo código repetidamente, para lograrlo es necesario distinguir el comportamiento general del particular.

La **genericidad** favorece la reusabilidad. El proceso consiste en abstraer lo que es común y esencial en un conjunto de entidades para formar un concepto general que comprenda a todas. Una clase derivada puede pensarse como una **especialización** de una clase más general. Alternativamente podemos pensar a una clase derivada como una **extensión** de la clase base.

Una **clase genérica** encapsula a una estructura cuyo comportamiento es independiente del tipo de las componentes. Los datos de aplicaciones muy diferentes puede modelarse con frecuencia a partir de tipos de datos cuyas operaciones no dependen del tipo de las componentes. La genericidad puede modelarse en Java de dos maneras diferentes: usando **tipos de datos parametrizados** o usando **herencia**.

En esta materia definiremos clases genéricas usando herencia.

Un arreglo es una estructura de datos que permite agrupar elementos de un mismo tipo, elemental o de clase. Un arreglo puede usarse para representar un TDA que brinda un conjunto de

operaciones que permiten esconder la manipulación del arreglo y ofrecer servicios más abstractos. Algunos de los servicios pueden **generalizarse**:

- insertar (unElem : TipoElemento)
- eliminar (unElem : TipoElemento)
- hayElementos(): boolean
- estaLlena () boolean
- existeElemento (unElem : TipoElemento):boolean

Es posible separar los servicios **generales** de los **específicos** y definir una **clase genérica.** Es una decisión de diseño, no la impone el lenguaje. El objetivo es aumentar la abstracción y obtener soluciones generales que puedan **rehusarse**.

Ordenamiento Burbuja: compara cada elemento de la estructura con el siguiente y los intercambia si corresponde. El proceso se repite hasta que la estructura está ordenada en su totalidad.

```
public void burbuja()
   {int n=array.length;
   boolean esNec=true;
   while(esNec)
         {int i=0;
         esNec=false;
         while(i+1<n)
               {if (array[i]>array[i+1])        {intercambiar(i,i+1);
                                                 esNec=true;}
               i=i+1;}
         n=n-1;}
   }
```

Merge Sort: Recursivamente parte la estructura en mitades, las ordena y las intercala si corresponde.

```
public void mergesort(){auxMerge(0,array.length-1);}
```

```
   private void auxMerge(int lo, int hi)
   {if (lo<hi) {
         int m=(lo+hi)/2;
         auxMerge(lo, m);
         auxMerge(m+1, hi);
         merge(lo, m, hi);}
    }
```

```
    private void merge(int lo, int m, int hi)
    {int i, j, k;
    int[] aux=new  int[array.length];
   // copy both halves of a to auxiliary array b
   for (i=lo; i<=hi; i++)
      aux[i]=array[i];
   i=lo; j=m+1; k=lo;
   // copy back next-greatest element at each time
   while (i<=m && j<=hi)
         if (aux[i]<=aux[j])              array[k++]=aux[i++];
         else                             array[k++]=aux[j++];
   // copy back remaining elements of first half (if any)
   while (i<=m)
      array[k++]=aux[i++];
}
```

QuickSort: Consiste en acomodar un elemento llamado Pivot en su possición definitiva y luego ordenar la estructura a su izquierda y a su derecha.

```
public void quicksort () {quicksort2(0,array.length-1);}
```

```
private void quicksort2 (int lo, int hi){
        int i=lo;
        int j=hi,
        int h;
        // comparison element x
        int x=array[(lo+hi)/2];
         do{
                while (array[i]<x) i++;
                while (array[j]>x) j--;
                if (i<=j)           {h=array[i];
                                     array[i]=array[j];
                                     array[j]=h;
                                     i++; j--;}
        } while (i<=j);
        //  recursion
        if (lo<j) quicksort2(lo, j);
        if (i<hi) quicksort2(i, hi);
}
```

Busqueda Binaria: Requiere que la estrctura esté ordenada. Consiste en partir la estructura en mitades y escoger la adecuada considerando que el elemento buscado es igual al medio, mayor o menor.

```
public boolean Binaria(int elem) {return AuxBinary(0,array.length-1,elem);}
```

```
public boolean AuxBinary(int ini, int fin, int elem)
{int mid=(ini+fin)/2;
if (array[mid]==elem)  return true;
else                {if (ini>=fin)     return false;
                     else     {if (array[mid]<elem)   ini=mid+1;
                               else                          fin=mid-1;
                               return AuxBinary(ini,fin,elem);}
                     }
 }
```

TODOS LOS METODOS ESTAN IMPLEMENTADOS PARA ARREGLOS DE ENTEROS, HAY QUE MODIFICAR LOS COMPARADORES <,>,= POR LOS CORRESPONDIENTES METODOS DE COMPARACION PARA QUE FUNCIONE.

Interface y Programación Basada en Eventos

Una **interface** es un conjunto de métodos que luego van a ser implementados por una o más clases, es similar a una clase pero **sólo brinda métodos abstractos**. Todos los métodos provistos son públicos.

Una interface define un tipo a partir del cual es posible declarar variables pero **no crear instancias**. Las interfaces pueden organizarse en una estructura jerárquica, donde cada nivel especializa al anterior. Enfaticemos que una interface no es una clase, no tiene variables de instancia ni implementa los servicios provistos. **La definición de interfaces permite *simular* herencia múltiple**. Una clase C puede **extender** una clase B e **implementar** una clase C. Java brinda interfaces y permite definir otras nuevas. Por el momento no vamos a definir nuevas interfaces, pero sí definiremos clases que implementan interfaces provistas por el lenguaje.

Una **interfaz** (superficie de contacto) es un medio que permite que dos entidades se comuniquen, es la conexión física y funcional entre dos aparatos o sistemas independientes. En una **interfaz de usuario** una de las entidades es una **persona** y la otra un **sistema** que el usuario intenta controlar. El sistema puede ser una herramienta, un automóvil o cualquier dispositivo electrónico, en particular una computadora. En una computadora el usuario ejecuta acciones como oprimir teclas o el botón del mouse y estas acciones son percibidas por la interfaz del sistema.

Las **interfaces gráficas de usuario (GUI)** explotan la capacidad de las computadoras para reproducir imágenes y gráficos en pantalla y brindan un ambiente más amigable, simple e intuitivo. La implementación de una GUI en **Java** se realiza usando un **paquete gráfico**. Un paquete gráfico brinda clases generales a partir de las cuales es posible crear objetos gráficos o bien definir nuevas clases más específicas a partir de las cuales crear objetos. Una GUI es una **colección de componentes** con una **representación gráfica** y capacidad para **percibir eventos** generados por las acciones del usuario. Las componentes son las partes individuales a partir de las cuales se conforma una interfaz gráfica. Una componente está asociada a un **objeto gráfico** que puede interactuar con el usuario. Un objeto gráfico es una instancia de una **clase gráfica**.

Java ofrece diferentes paquetes gráficos para soportar la implementación de gráficos y GUIs. **AWT** (**Abstract Window Toolkit**) y **Swing** son paquetes gráficos independientes de la plataforma, para desarrollar interfaces gráficas. Ambos brindan una colección de clases que pueden especializarse para crear botones, cajas de texto, menúes, etc. Swing no reemplaza a AWT sino que la usa y agrega nuevas clases. Cada clase de Swing modela un tipo de componente. La implementación de interfaces gráficas se basará en **la redefinición de los métodos provistos por las clases que conforman el paquete Swing**. Así, un mensaje enviado desde una operación provista por una clase dada del paquete Swing, puede quedar asociado a un método redefinido en una clase que extiende a otra clase provista por Swing. Esta es una característica totalmente nueva para nosotros, hasta el momento el código que generábamos podía usar métodos provistos por el lenguaje o definidos por nosotros mismos. En lo que sigue, algunos de los métodos definidos por el lenguaje usarán métodos definidos por el programador.

Los tipos de componentes pueden agruparse en:

- **controles básicos**: son componentes simples que se usan principalmente para tomar la entrada del usuario y mostrar un estado simple. JButton, JCheckBox, JComboBox, JList, JMenu, JRadioButton, JSlider.
- **displays interactivos con información altamente formateada**: JColorChooser, JEditorPane, JTextPane.
- **displays con información no editable**: JLabel, JProgressBar, JSeparator, JToolTip.
- **Contenedores**: componentes que son **contenedores** de otros componentes. Un contenedor tiene atributos especiales como el **layout** o **diagramado** de acuerdo al cual se organizan las componentes contenidas. Una **ventana** es un contenedor de alto nivel. Un **frame** es un tipo especial de ventana sobre el que puede ejecutarse una aplicación. JFrame, JApplet, JDialog, JPanel, JScrollPane, JSplitPane.

La **implementación** de una interfaz gráfica consiste en:

- **importar paquetes gráficos** para crear objetos gráficos de las clases provistas o definir clases más específicas.

• **crear objetos fuentes de evento** capaces de percibir las acciones del usuario sobre las componentes gráficas.
• **registrar objetos oyente** a los objetos fuente de evento
• implementar interfaces para **manejar los eventos internos** disparados en respuesta a las acciones del usuario.
• crear paneles contenedores de las componentes gráficas
• establecer el diagramado de los paneles y la apariencia visual de las componentes especificando los valores de los atributos gráficos de los objetos
• insertar las componentes en los paneles y los paneles en el panel del contenido del frame.

Algunos de las componentes de una GUI son **reactivas**, están asociadas a objetos fuente de evento que ***perciben*** **eventos externos** y ***disparan*** **eventos internos**. Cada evento interno se dispara creando un objeto evento de software que es pasado como **parámetro en un mensaje enviado a un objeto oyente**. En respuesta al mensaje el oyente ejecuta un método que corresponde a la acción del usuario. El objeto oyente está registrado al objeto fuente de evento asociado a la componente reactiva.

El **evento externo** provocado por la **acción** del usuario al hacer una acción, es detectado por el objeto fuente de evento. El objeto fuente de evento detecta el evento externo y dispara un **evento interno**, esto es, se crea un objeto implícitamente de la clase ActionEvent. La definición de la clase Oyente incluye la **redefinición del método actionPerformed** que recibe como argumento el objeto evento creado implícitamente. El programador redefine el método **actionPerformed** que no va a ser invocado explícitamente por él mismo, sino que el mensaje que provoca su ejecución está incluido en algún método definido en una clase provista por Java. Esto es, **el programador define (o redefine) un método que él mismo nunca va a invocar *al* menos** explícitamente.

En la **programación basada en eventos** el orden en el cual se ejecutan las instrucciones va a quedar determinado por **eventos**, en nuestro caso generados por acciones realizadas por el usuario.

Observemos que estamos considerando tres tipos de objetos:

- Los **objeto fuente del evento** están asociados a una componente de la interfaz, tienen una **representación gráfica** y son capaces de percibir y **reaccionar ante un evento externo** provocado por una acción del usuario y **disparar eventos de software**.
- Los **objetos evento** son disparados implícitamente por un objeto fuente del evento.
- Los **objetos oyentes** (listeners) que se ejecutan para manejar un evento. La clase a la que pertenece un objeto oyente brinda métodos para manejar eventos, es decir especifica el curso de acción a seguir en respuesta a diferentes tipos de eventos.

Cuando un usuario interactúa con una interface gráfica realiza acciones que generan eventos de **alto nivel** y de **bajo nivel**. Estos eventos pueden ser percibidos por la aplicación o no.

Objeto Evento	Interface de Oyente	Manejador
ActionEvent	ActionListener	actionPerformed(ActionEvent)
ItemEvent	ItemListener	itemStateChanged(ItemEvent)
MouseEvent	MouseListener MouseMotionListener	mousePressed(MouseEvent) mouseReleased(MouseEvent) mouseEntered(MouseEvent) mouseExited(MouseEvent) mouseClicked(MouseEvent)
KeyEvent	KeyListener	keyPressed(KeyEvent) keyRealesed(KeyEvent) keyTyped(KeyEvent)

Ejemplo:

```
botonRojo = new JButton("Rojo");
   OyenteBotonR ponerRojo = new OyenteBotonR();
   botonRojo.addActionListener(ponerRojo);
```

```
class OyenteBotonR implements ActionListener {
  public void actionPerformed( ActionEvent event) {
    getContentPane().setBackground(Color.red);}}
```

El desarrollo de una interfaz gráfica está fuertemente ligada a los conceptos de evento, herencia, polimorfismo y ligadura dinámica. En una interfaz gráfica el usuario interactúa con el programa realizando acciones que producen eventos. Un programa puede responder ante cada evento o ignorarlo. Si un programa va a responder debe incluir objetos con la capacidad de reaccionar ante el evento y manejarlo. La herencia, el polimorfismo y la ligadura dinámica van a ser los mecanismos que permiten que un método redefinido por el programador sea invocado desde un método que no está definido por el programador.