TUP - PROGRAMACION II

PROGRAMACIÓN ORIENTADA A OBJETOS

(OOP)

POO

- La programación orientada a objetos, es un paradigma de programación que viene a innovar la forma de obtener resultados.
- Los objetos manipulan los datos de entrada para la obtención de datos de salida específicos, donde cada objeto ofrece una funcionalidad especial.
- Muchos de los objetos pre-diseñados de los lenguajes de programación actuales permiten la agrupación en bibliotecas o librerías
- Muchos de estos lenguajes permiten a los desarrolladores la creación de sus propias bibliotecas.
- Está basada en varias técnicas, incluyendo herencia, cohesión, abstracción, polimorfismo, acoplamiento y encapsulamiento.

Abstracción

- Denota las características esenciales de un objeto, donde se capturan sus comportamientos.
- Cada objeto en el sistema sirve como modelo de un "agente" abstracto que puede realizar trabajo, informar y cambiar su estado, y "comunicarse" con otros objetos en el sistema sin revelar "cómo" se implementan estas características.
- El proceso de abstracción permite seleccionar las características relevantes dentro de un conjunto e identificar comportamientos comunes para definir nuevos tipos de entidades en el mundo real.
- La abstracción es clave en el proceso de análisis y diseño orientado a objetos, ya que mediante ella podemos llegar a armar un conjunto de clases que permitan modelar la realidad o el problema que se quiere atacar.

Encapsulamiento

- Significa reunir todos los elementos que pueden considerarse pertenecientes a una misma entidad, al mismo nivel de abstracción.
- Permite aumentar la cohesión (diseño estructurado) de los componentes del sistema.

Principio de Ocultamiento

- Cada objeto está aislado del exterior, es un módulo natural, y cada tipo de objeto expone una "interfaz" a otros objetos que especifica cómo pueden interactuar con los objetos de la clase.
- El aislamiento protege a las propiedades de un objeto contra su modificación por quien no tenga derecho a acceder a ellas; solamente los propios métodos internos del objeto pueden acceder a su estado.
- Otros objetos no pueden cambiar el estado interno de un objeto de manera inesperada, eliminando efectos secundarios e interacciones inesperadas.

Polimorfismo

- Comportamientos diferentes, asociados a objetos distintos, comparten el mismo nombre; al llamarlos por ese nombre se utilizará el comportamiento correspondiente al objeto que se esté usando.

Sobrecarga de Operadores

- Algunos lenguajes de programación soportan la “Sobrecarga de Operadores”
- Un mismo objeto permite asociar más argumentos o parámetros a los métodos y funciones que tiene definido a modo de realizar, algunas veces, Comportamientos diferentes.

Herencia

- Las clases no se encuentran aisladas, sino que se relacionan entre sí, formando una jerarquía de clasificación.
- Los objetos heredan las propiedades y el comportamiento de todas las clases a las que pertenecen.
- La herencia organiza y facilita el polimorfismo y el encapsulamiento, permitiendo a los objetos ser definidos y creados como tipos especializados de objetos preexistentes.
- Cuando un objeto hereda de más de una clase se dice que hay herencia múltiple; siendo de alta complejidad técnica por lo cual suele recurrirse a la herencia virtual para evitar la duplicación de datos.

Modularidad

- Se denomina "modularidad" a la propiedad que permite subdividir una aplicación en partes más pequeñas (llamadas módulos), cada una de las cuales debe ser tan independiente como sea posible de la aplicación en sí y de las restantes partes.
- Estos módulos se pueden compilar por separado, pero tienen conexiones con otros módulos. Al igual que la encapsulación, los lenguajes soportan la modularidad de diversas formas.

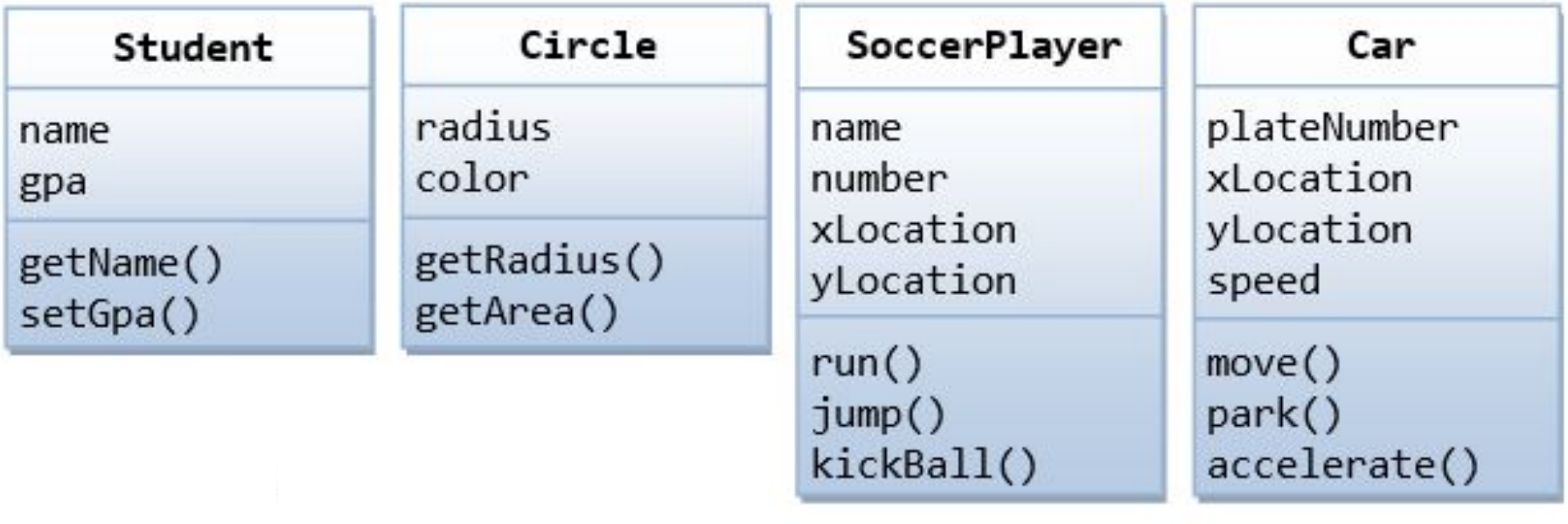
Recolección de basura (Java o C#)

- La recolección de basura (garbage collection) es la técnica por la cual el entorno de objetos se encarga de destruir automáticamente, y por tanto desvincular la memoria asociada, los objetos que hayan quedado sin ninguna referencia a ellos.
- El desarrollador no debe preocuparse por la asignación o liberación de memoria, ya que el entorno la asignará al crear un nuevo objeto y la liberará cuando nadie lo esté usando.

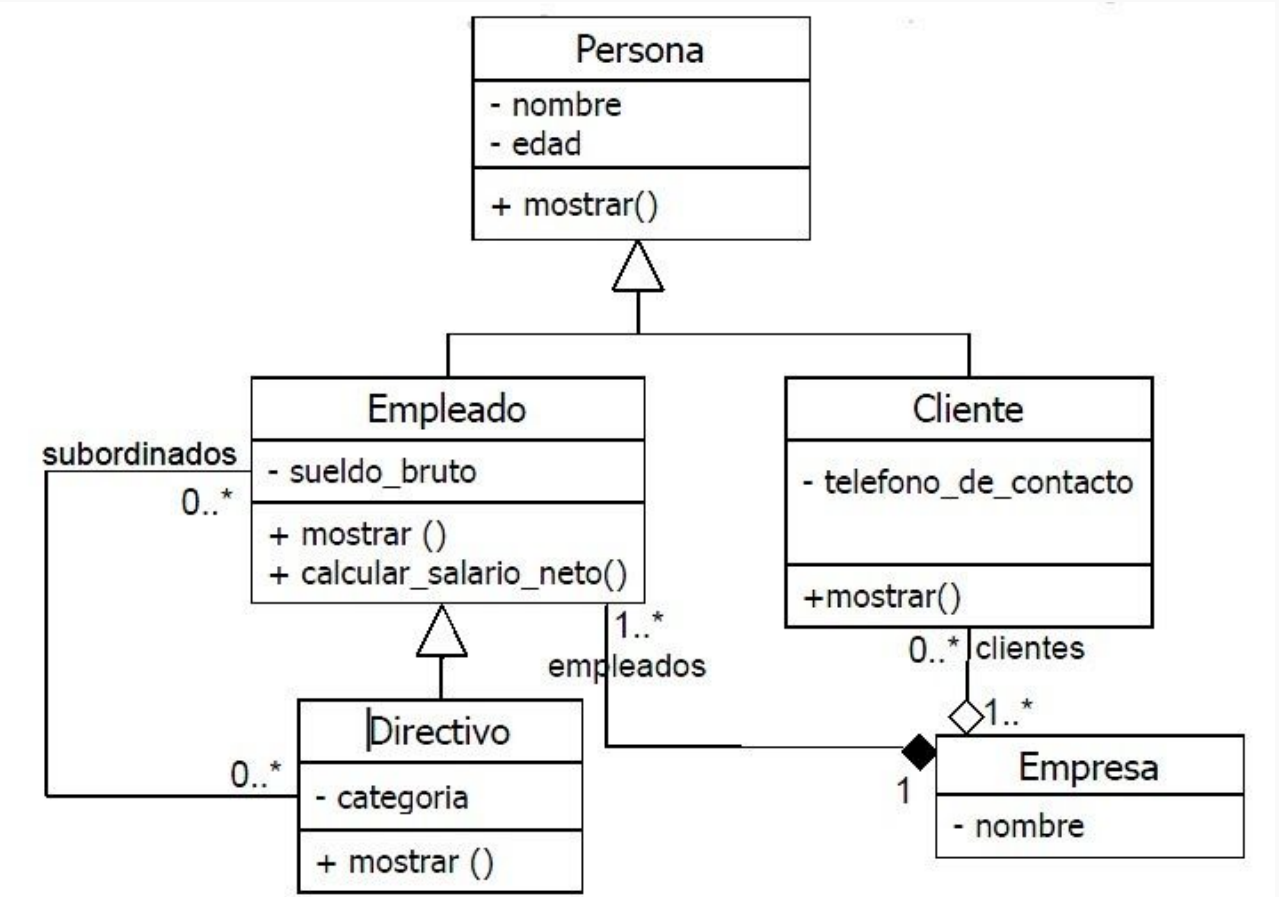
Clase

- Es una plantilla para la creación de objetos de datos según un modelo predefinido.
- Las clases se utilizan para representar entidades o conceptos, como los sustantivos en el lenguaje.
- Cada clase es un modelo que define un conjunto de variables -el estado, y métodos apropiados para operar con dichos datos -el comportamiento.
- Cada objeto creado a partir de la clase se denomina instancia de la clase.

POO - Conceptos fundamentales - Clase



POO - Conceptos fundamentales - Diagrama de Clases

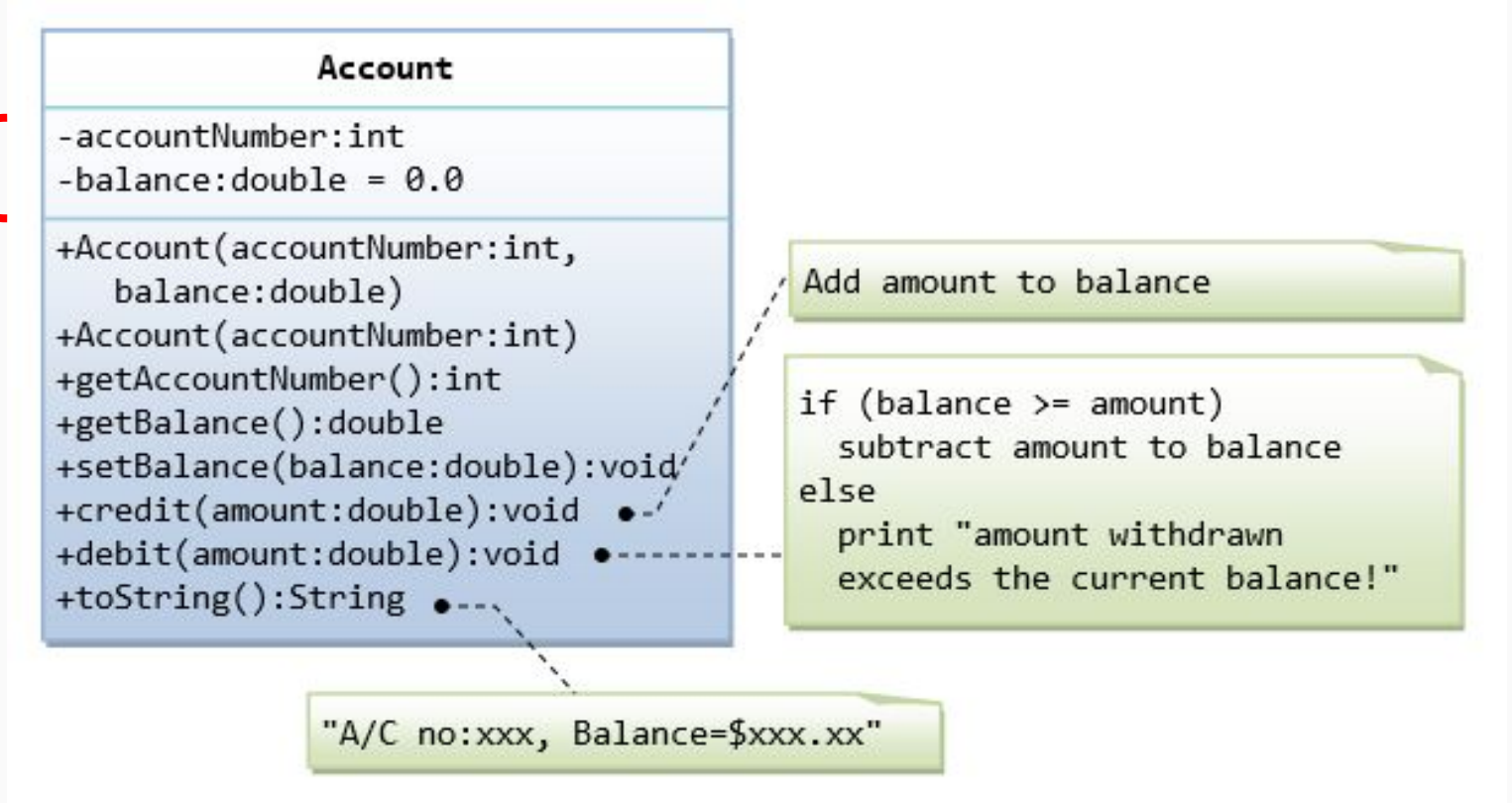


Atributos

- Es una variable que se declara *privada*, que puede ser únicamente accedida y alterada por un método del objeto, y que se utiliza para indicar distintas situaciones posibles para el objeto (o clase de objetos).
- No es visible al programador que maneja una instancia de la clase.

POO - Conceptos fundamentales - Atributos

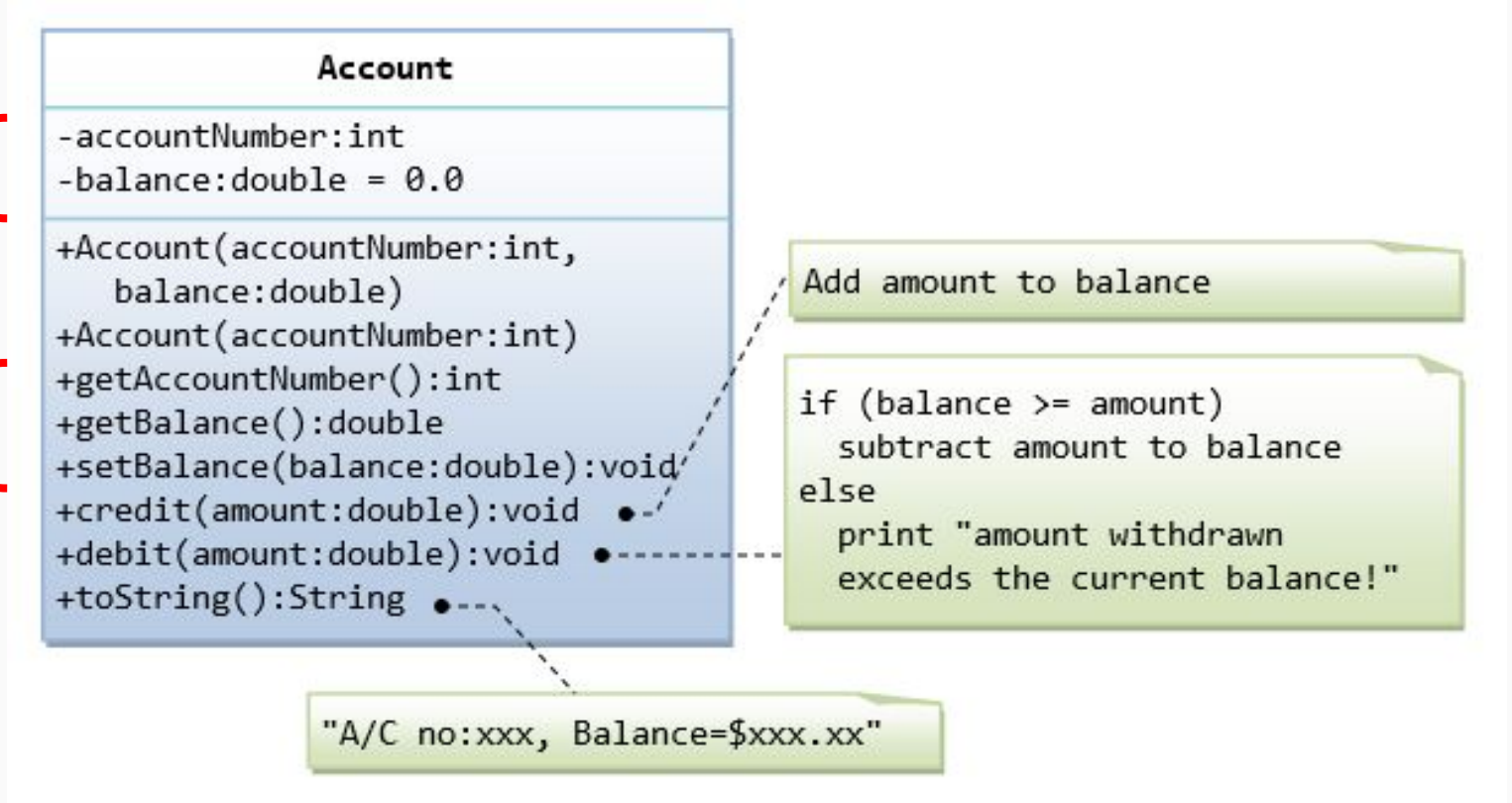
Atributos



Propiedades

- Contenedor de un tipo de datos asociados a un objeto (o a una clase de objetos), que hace los datos visibles desde fuera de la Clase y esto se define como sus características predeterminadas.
- Su valor puede ser alterado por la ejecución de algún método.

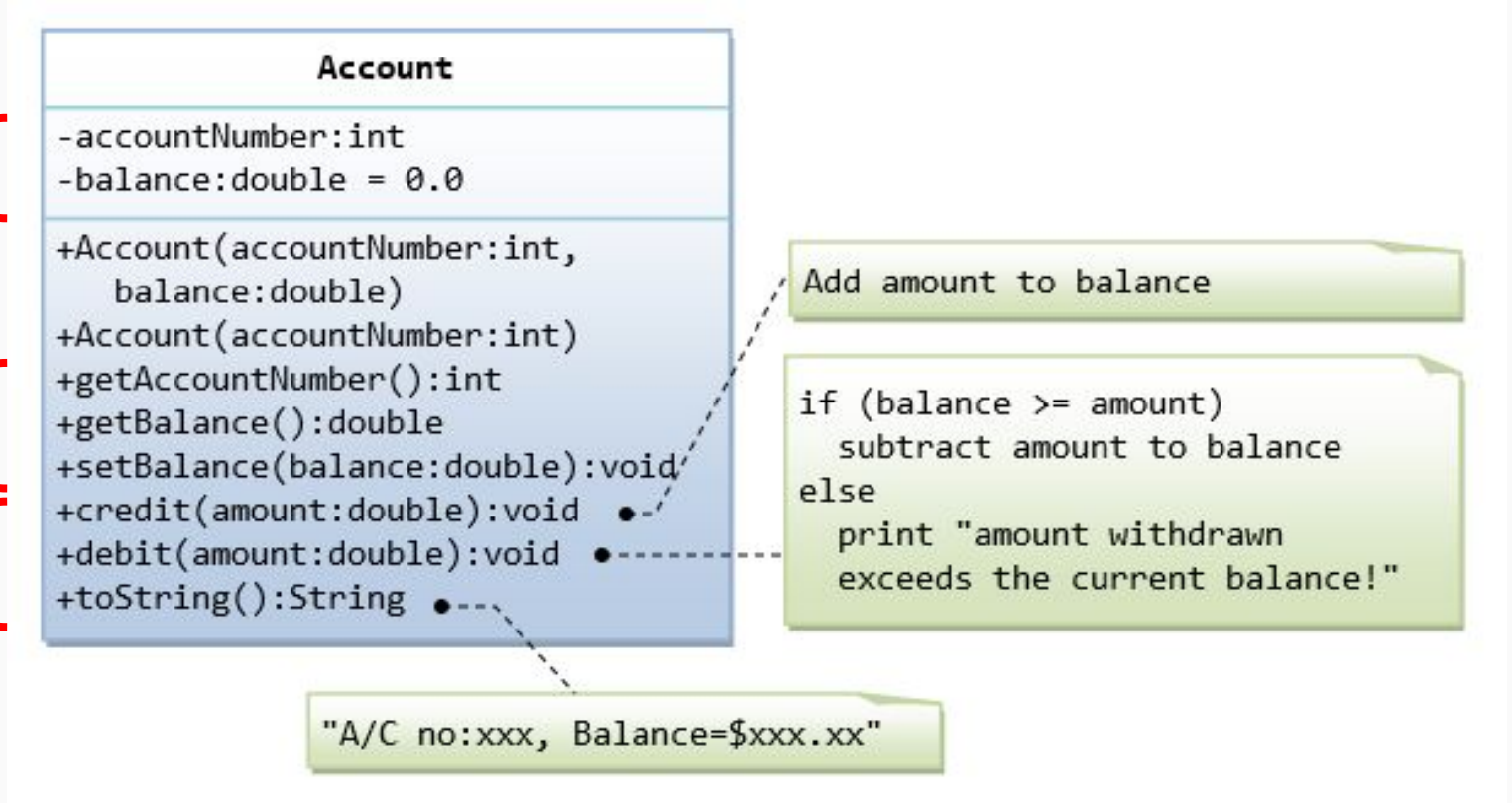
POO - Conceptos fundamentales - Atributos



Método

- Desde el punto de vista del comportamiento, es lo que el objeto puede hacer.
- Su código es definido en una clase y puede pertenecer tanto a una clase, como es el caso de los métodos de clase o estáticos, como a un objeto, como es el caso de los métodos de instancia.
- Algoritmo asociado a un objeto (o a una clase de objetos), cuya ejecución se desencadena tras la recepción de un "mensaje".
- Un método puede producir un cambio en las propiedades del objeto, o la generación de un "evento" con un nuevo mensaje para otro objeto del sistema.
- El método podría ser también una función.

POO - Conceptos fundamentales - Atributos



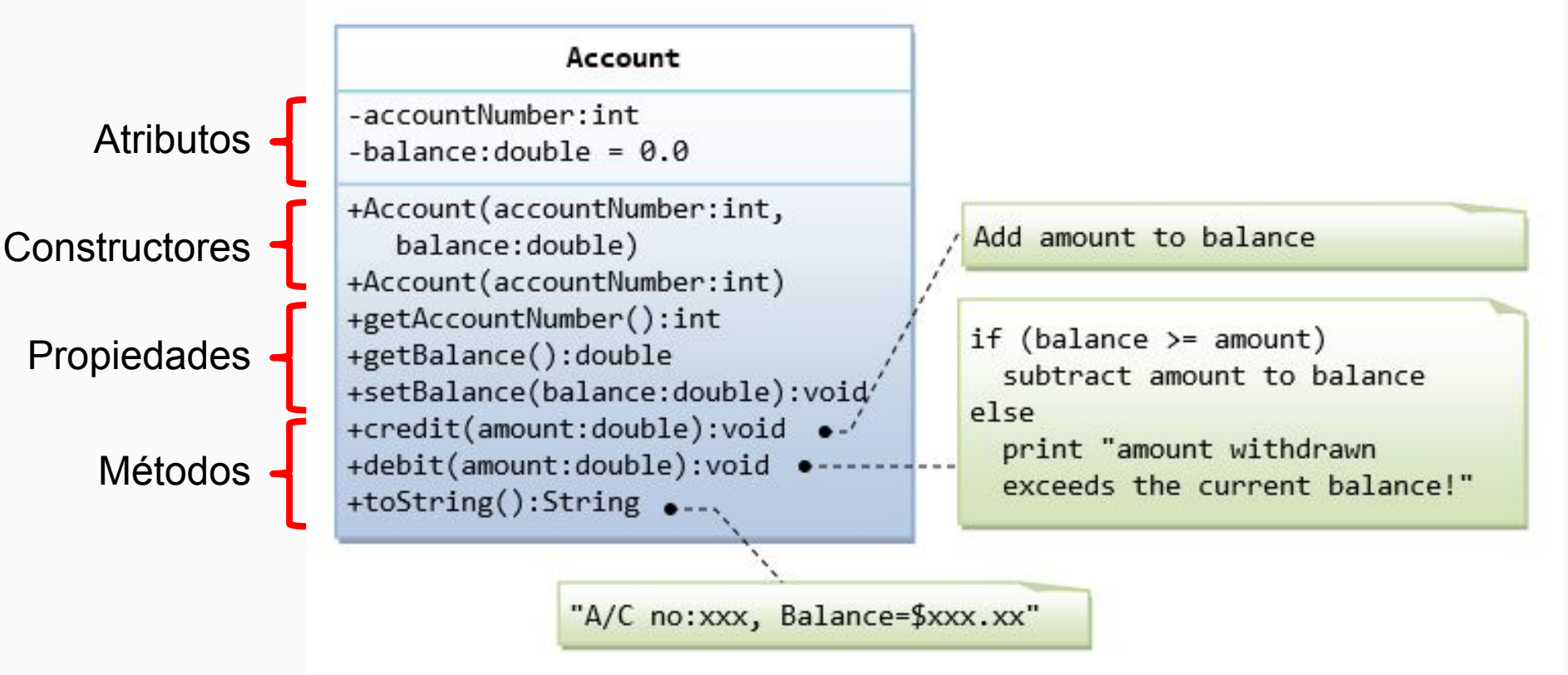
Tipos de Métodos

- Métodos de Instancia (No Estáticos)
  - Se definen dentro de una Clase.
  - Pueden acceder a otros métodos y variables de la clase.
  - Pueden acceder a métodos y variables estáticas.
- Métodos Estáticos o de Clase
  - Son métodos que se definen dentro de una Clase, pero pueden ser llamados sin la necesidad de crear un Objeto de la Clase (instancia).
  - Se referencian utilizando el Nombre de la Clase y el nombre del Método en cuestión.
  - Estos métodos no poseen Polimorfismo.
  - Se diseñaron para compartir comportamiento a todos los Objetos de la Clase.
  - Se debe asignar la palabra reservada **STATIC** al nombre del método.
  - **No** pueden acceder a otros métodos y variables de la clase.
  - Pueden acceder a métodos y variables estáticas.

Constructor

- Un constructor es un método especial de una clase o estructura en la programación orientada a objetos que inicializa un objeto de ese tipo.
- Un constructor es un método de instancia que generalmente tiene el mismo nombre que la clase y se puede usar para establecer los valores de los miembros de un objeto, ya sea por defecto o por valores definidos por el usuario.

POO - Conceptos fundamentales - Atributos



Asociación

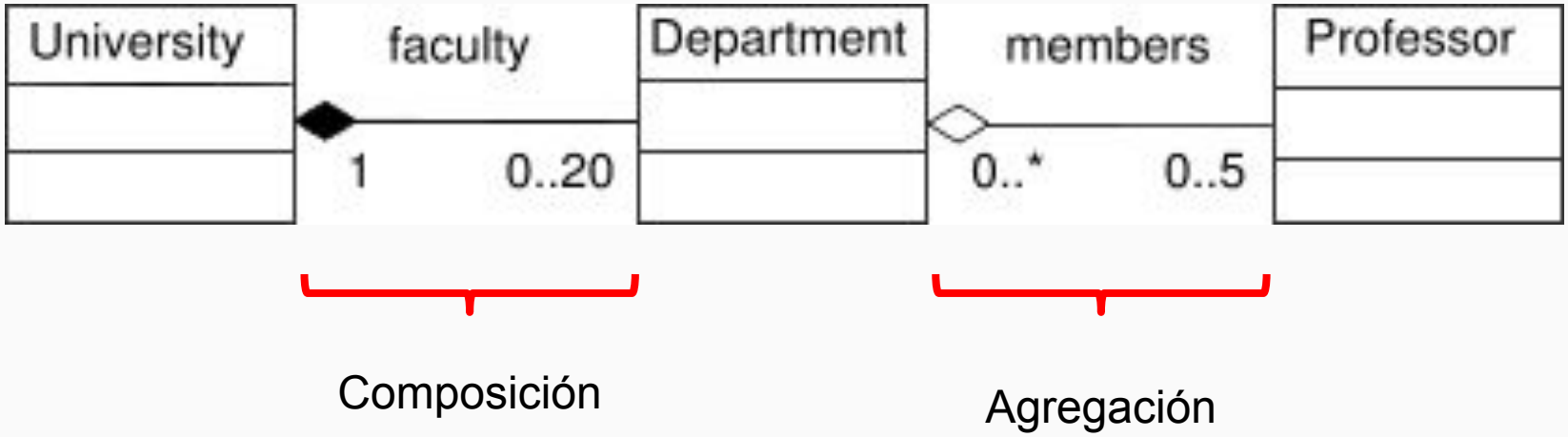
- Define una relación estructural entre clases, que permite que una instancia de objeto haga que otra realice una acción en su nombre.
- Esta relación específica que los objetos de un tipo están conectados a objetos de otro y no representan el comportamiento.

Tipos de Asociaciones

- Agregación:
  - Es un tipo de asociación que indica que una clase es parte de otra clase (composición débil).
  - La destrucción del compuesto no conlleva la destrucción de los componentes.

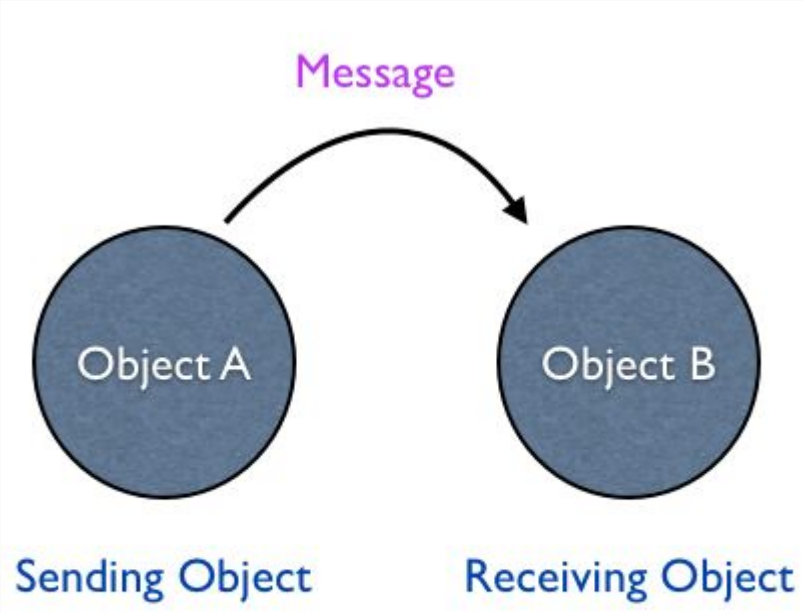
- *Composición*:
  - la composición es un tipo de relación dependiente en dónde un objeto más complejo es conformado por objetos más pequeños.
  - Es una forma fuerte de composición donde la vida de la clase contenida debe coincidir con la vida de la clase contenedor.
  - Los componentes constituyen una parte del objeto compuesto.

POO - Conceptos fundamentales - Tipos de Asociaciones

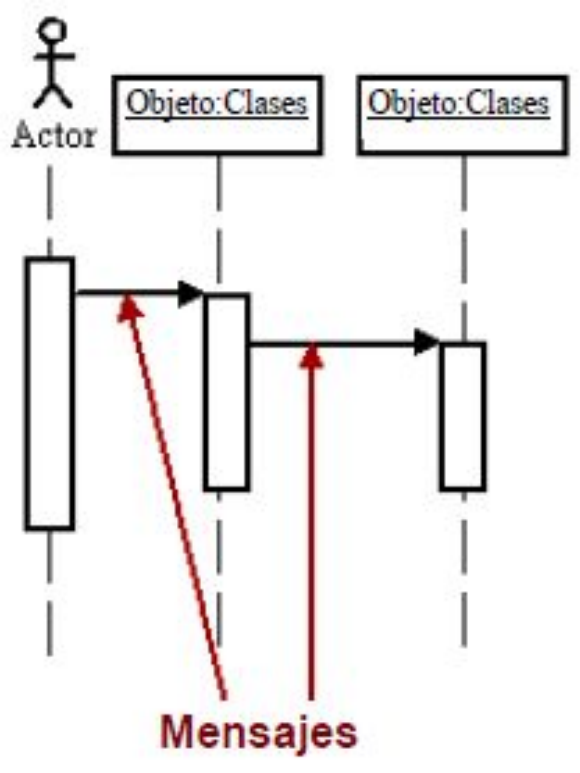


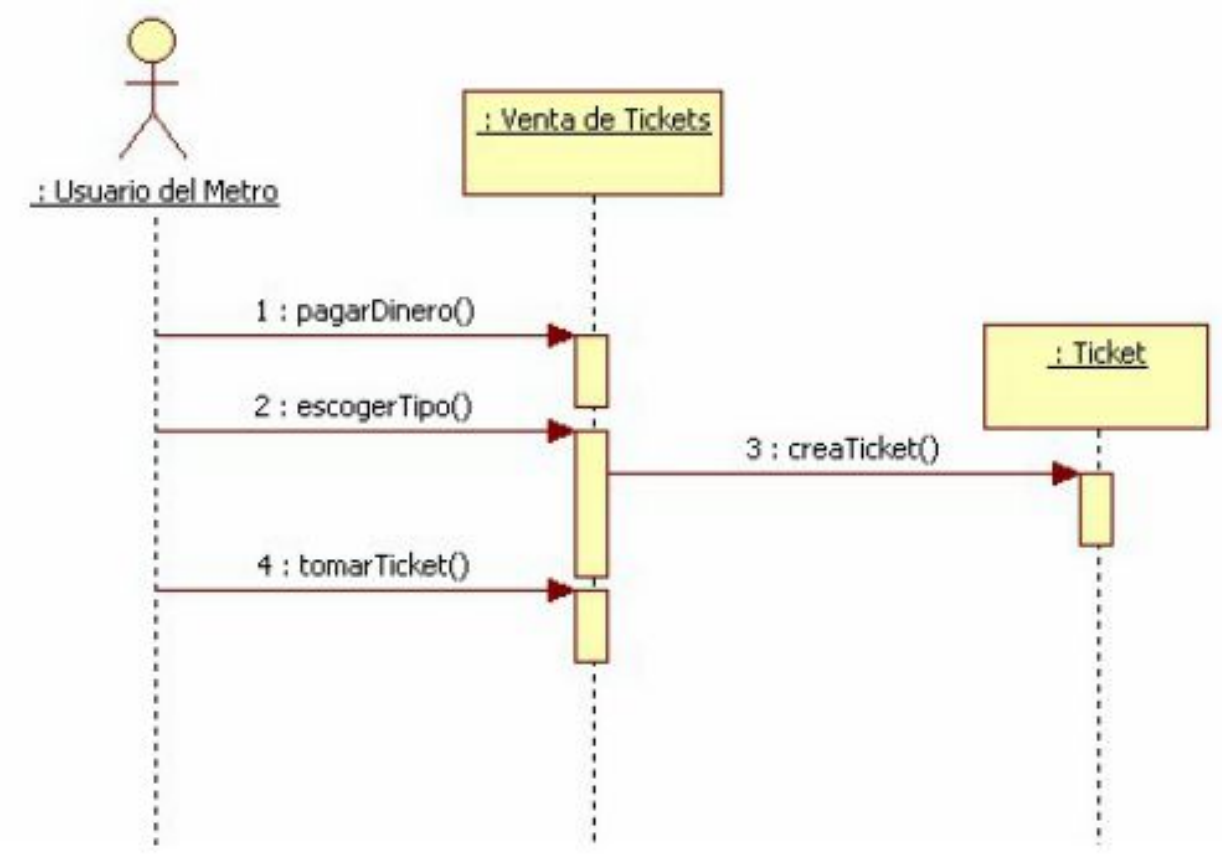
Mensajes

- Una comunicación dirigida a un objeto, que le ordena que ejecute uno de sus métodos con ciertos parámetros asociados al evento que lo generó.

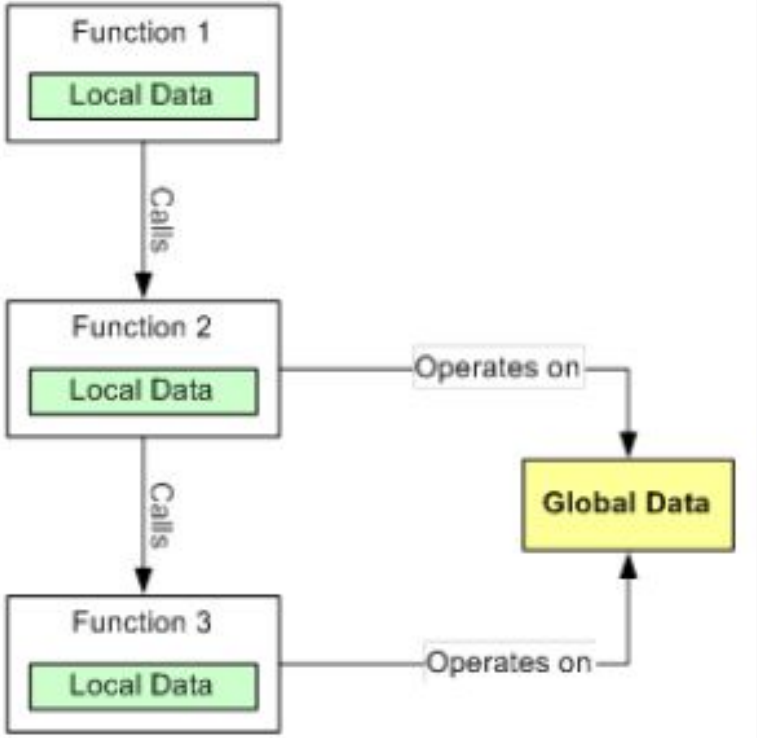


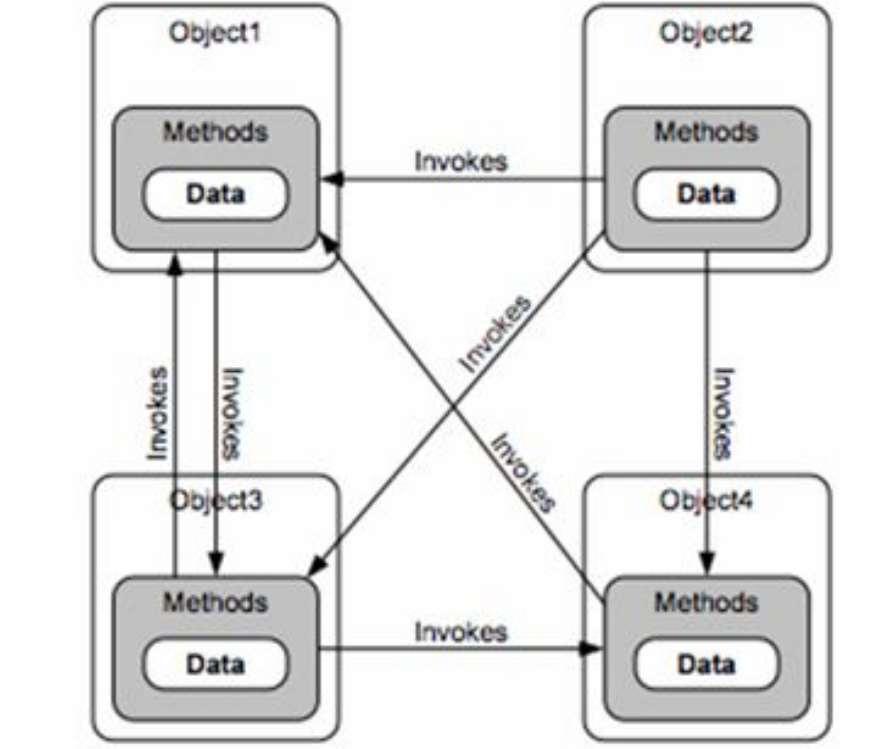
POO - Conceptos fundamentales - Mensajes





Comparación de Programación Procedural y Orientación a Objetos

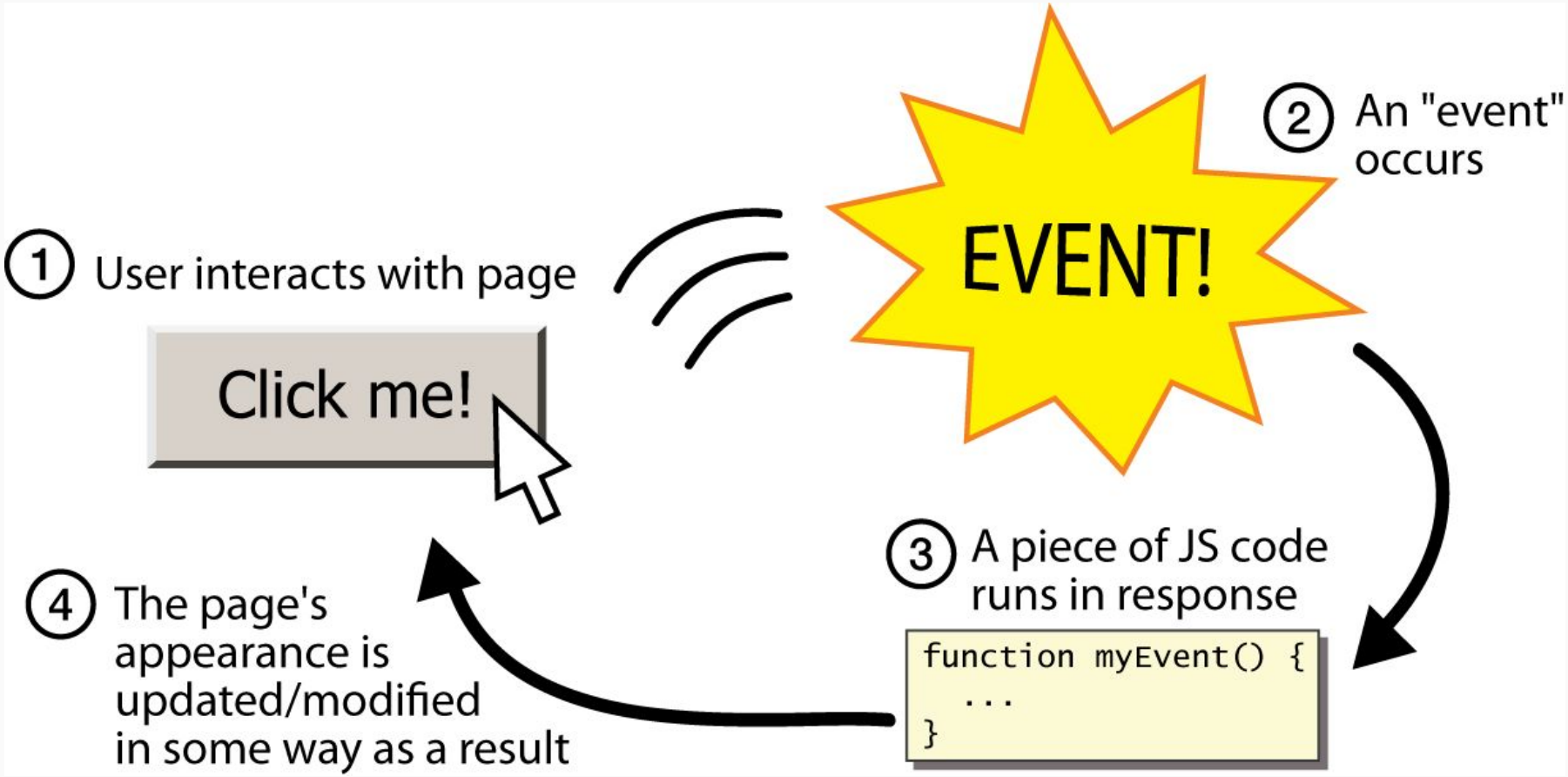




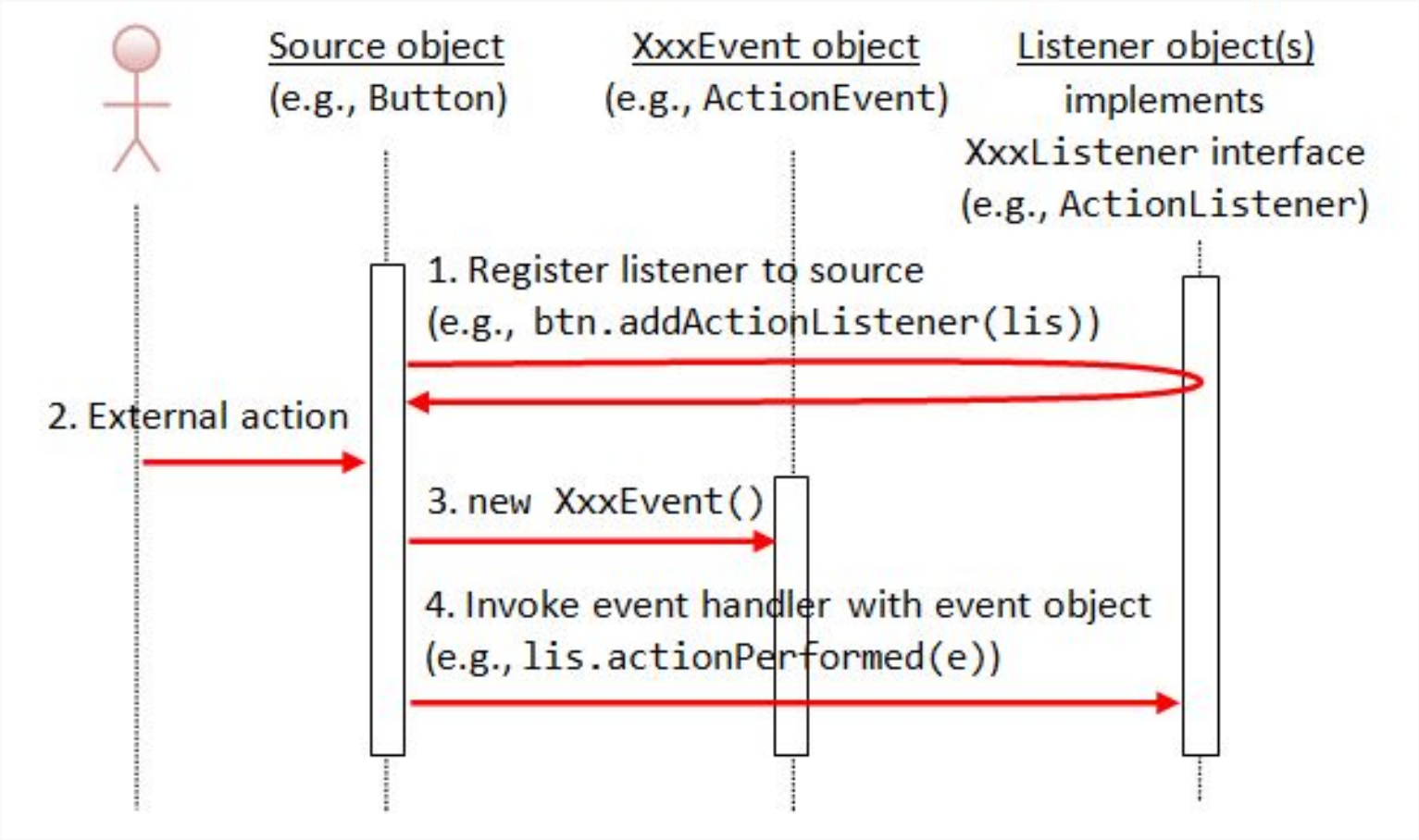
Evento

- Es un suceso en el sistema (tal como una interacción del usuario con la máquina, o un mensaje enviado por un objeto).
- El sistema maneja el evento enviando el mensaje adecuado al objeto pertinente.
- También se puede definir como evento la reacción que puede desencadenar un objeto; es decir, la acción que genera.

POO - Conceptos fundamentales - Eventos



POO - Conceptos fundamentales - Eventos



Control de Acceso: Accesibilidad

- Son palabras clave en los lenguajes orientados a objetos que establecen la accesibilidad de clases, métodos y otros miembros.
- Los modificadores de acceso son una parte específica de la sintaxis del lenguaje de programación utilizada para facilitar la encapsulación de componentes.

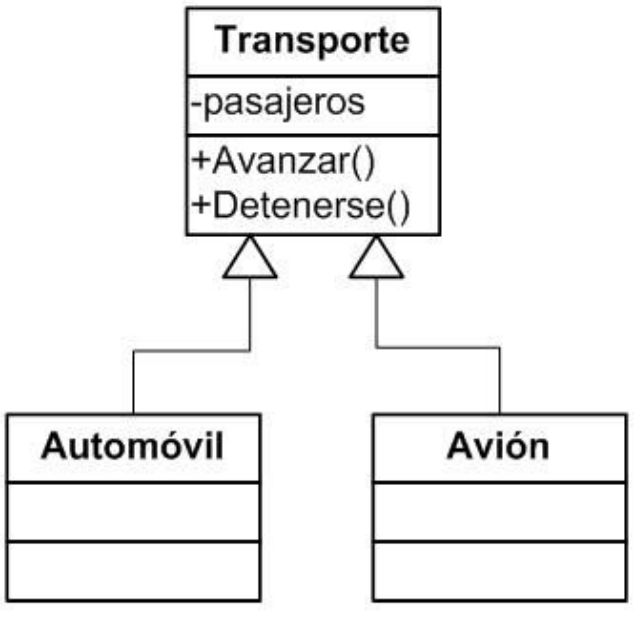
Control de Acceso: Accesibilidad

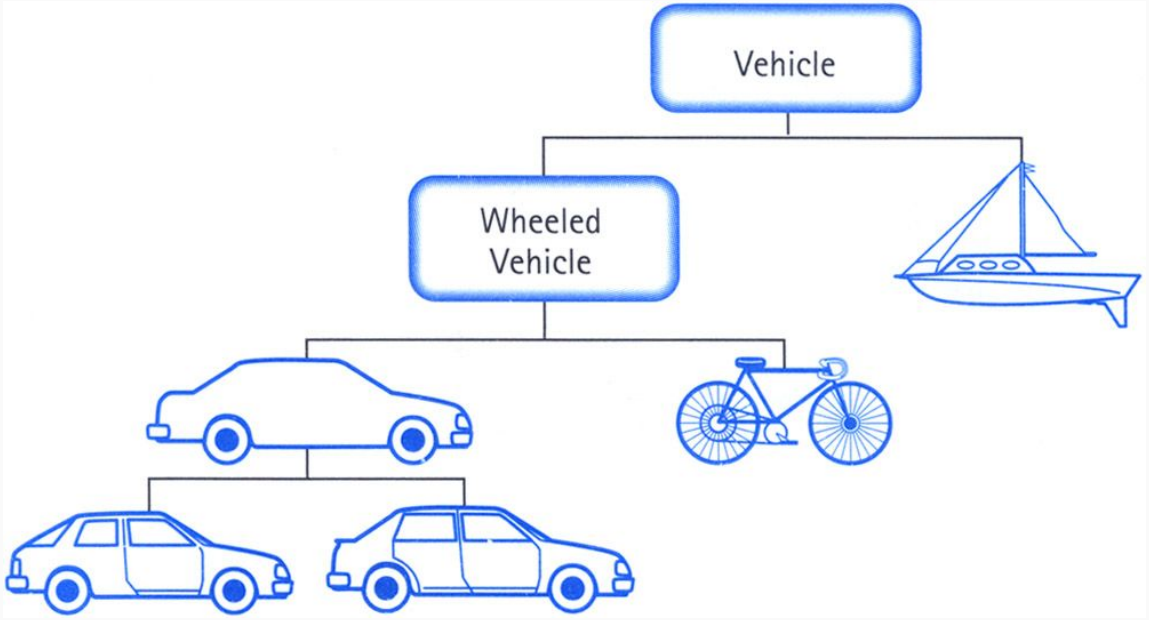
- *Público:* significa que el elemento está disponible para todos.
- *Privado:* significa que nadie puede acceder a ese elemento excepto el creador del tipo, dentro de los métodos de ese tipo.
- *Protegido:* actúa como privado, con la excepción de que una clase heredada tiene acceso a miembros protegidos, pero no a miembros privados.
- *Java*
  - Tiene un acceso "predeterminado", que entra en juego si no utiliza uno de los especificadores antes mencionados.
  - Esto se suele llamar acceso a paquetes porque las clases pueden acceder a los miembros de otras clases en el mismo paquete, pero fuera del paquete, esos mismos miembros parecen ser privados.

Herencia

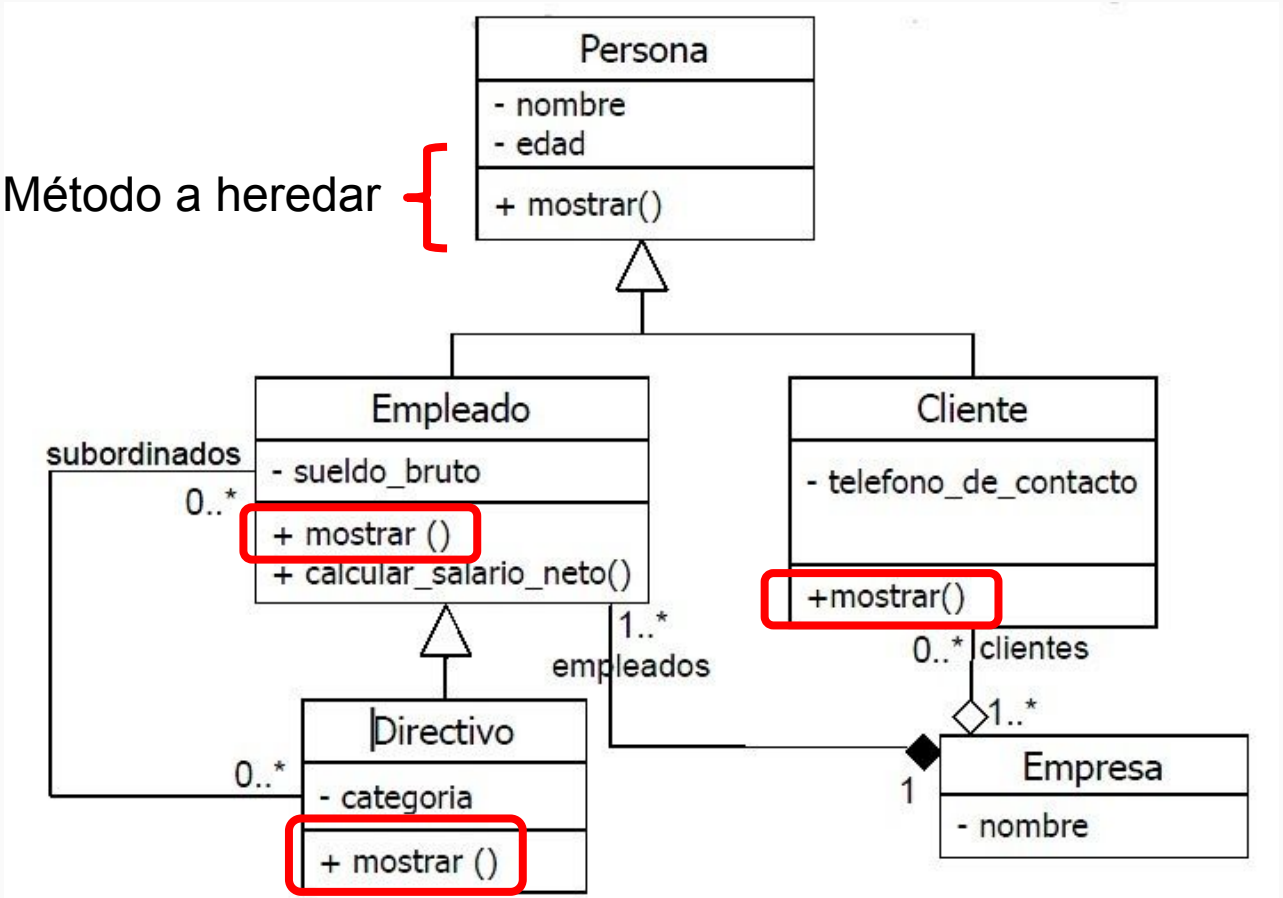
- La herencia es uno de los mecanismos de los lenguajes de programación orientada a objetos basados en clases, por medio del cual una clase se deriva de otra de manera que extiende su funcionalidad.
- Es la relación entre una clase general y otra clase más específica.
- La herencia facilita la creación de objetos a partir de otros ya existentes e implica que una subclase obtiene todo el comportamiento (métodos) y eventualmente los atributos (variables) de su superclase.
- La clase de la que se hereda se suele denominar clase base, clase padre, superclase, clase ancestro (el vocabulario que se utiliza suele depender en gran medida del lenguaje de programación).

POO - Conceptos fundamentales - Herencia

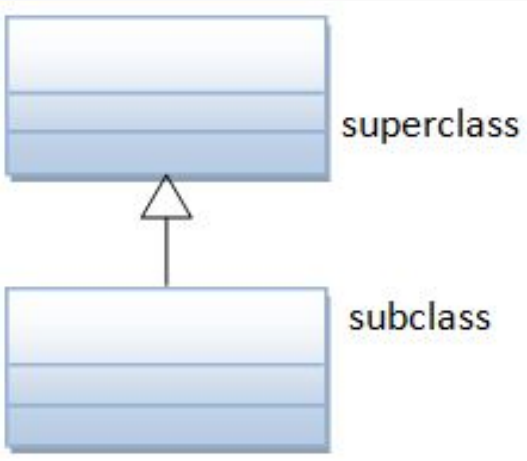


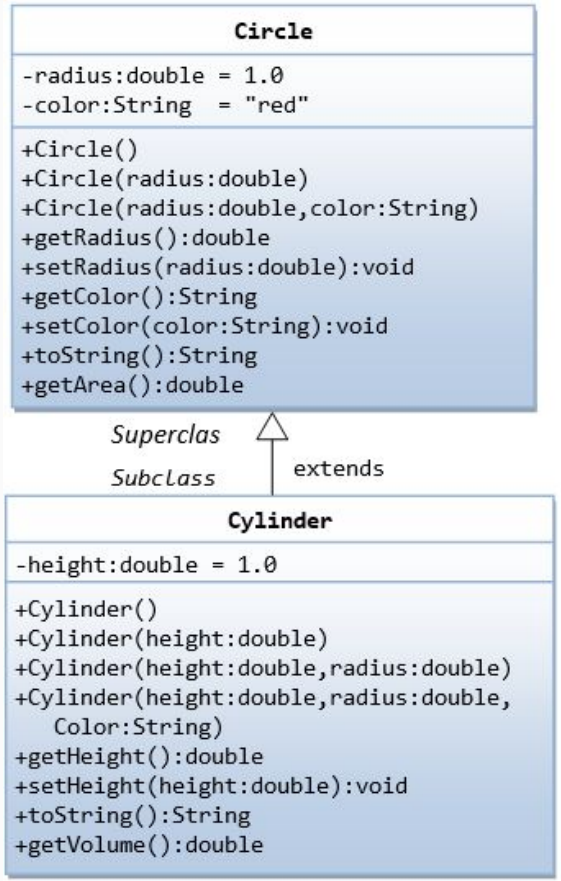


POO - Conceptos fundamentales - Herencia



POO - Conceptos fundamentales - Herencia: Superclase y subclase





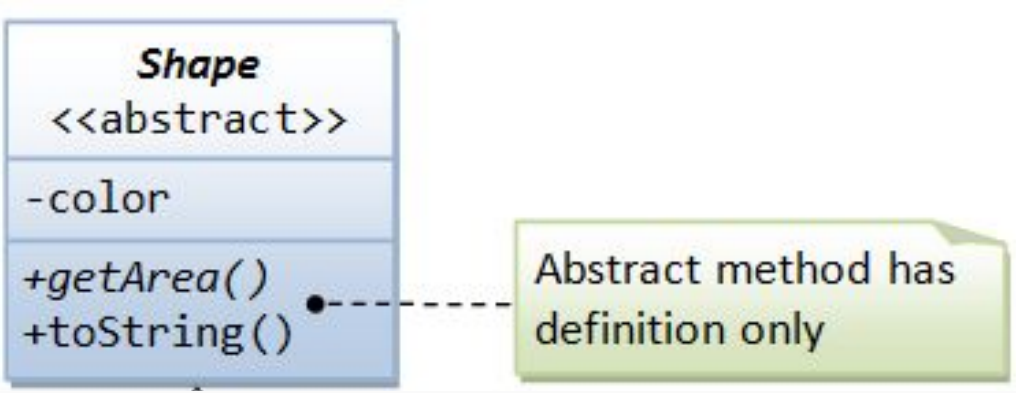
Tipos de Herencia

- Uno de los problemas en OOP que se ha vuelto especialmente prominente desde la introducción de C ++ es si todas las clases deberían ser heredadas de una sola clase base. En Java y otros lenguajes se optó por esta idea.
- *Herencia Simple*
  - Todos los objetos tienen una interfaz en común, por lo que todos son, el mismo tipo fundamental.
  - Se puede garantizar que todos los objetos tengan cierta funcionalidad.
  - Se puede realizar ciertas operaciones básicas en cada objeto.
  - Se facilita la implementación de la “Recolección de Basura” o “garbage collector” (que es un ventaja de JAVA sobre C++)
- *Herencia Multiple*
  - La herencia se realiza desde varias clases bases.
  - Ofrece mayor flexibilidad, pero se necesita más control sobre la jerarquía de clases.

Clases Abstractas

- Una clase abstracta es una clase que está diseñada para usarse específicamente como clase base con el fin de proporcionar detalles de implementación a sus hijos.
- No permite crear instancias de su clase.
- Puedo o no contener uno o más métodos abstractos.
- Un método abstracto es un método que se declara, pero no contiene ninguna implementación (no tiene cuerpo)

POO - Conceptos fundamentales - Clases Abstractas

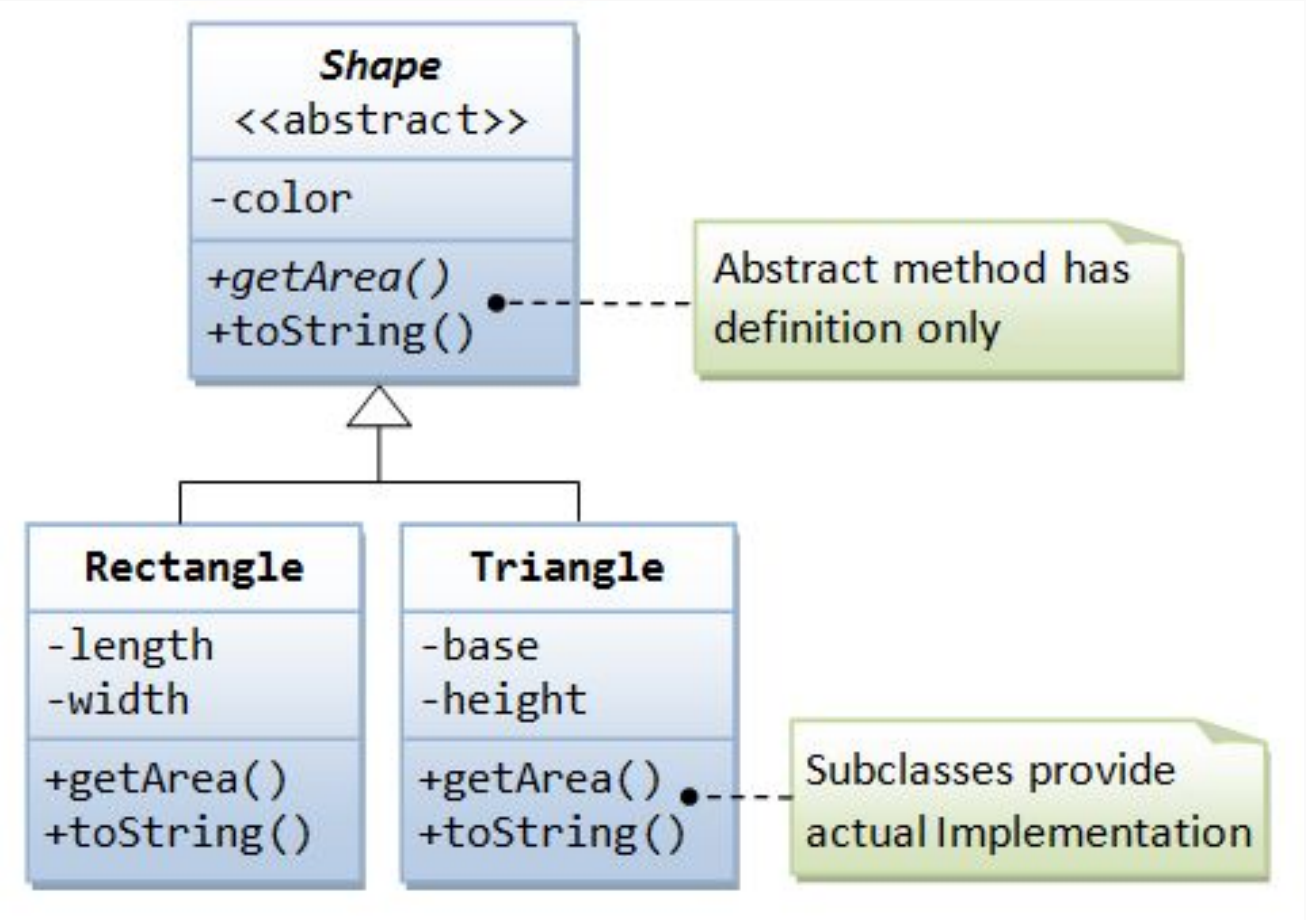



```
public abstract class Shape {

    private int color;

    public abstract double getArea();
}
```

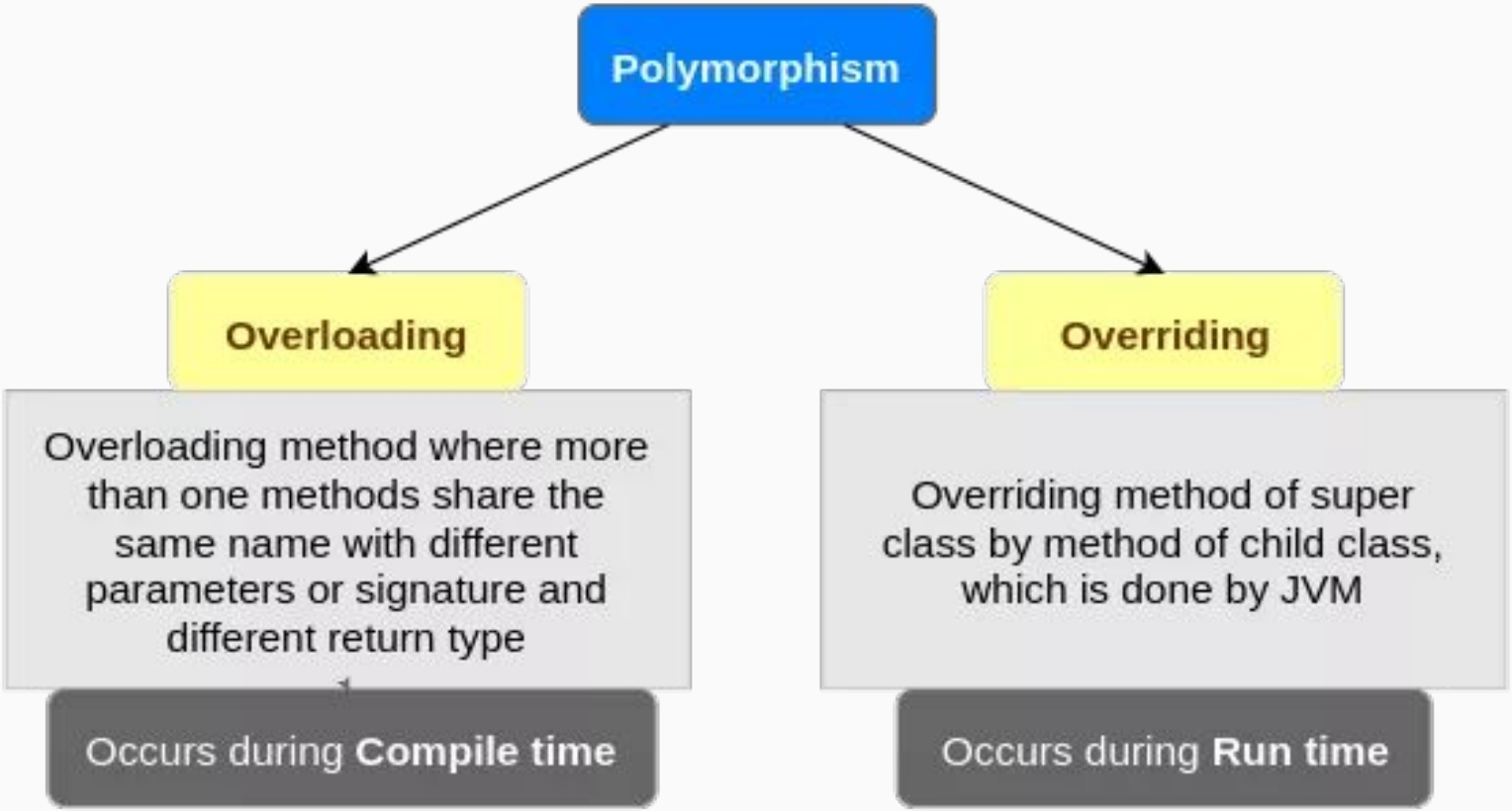
POO - Conceptos fundamentales - Clases Abstractas



Polimorfismo en Java

- Es la provisión de una interfaz única para entidades de diferentes tipos o el uso de un solo símbolo para representar múltiples tipos diferentes.
- JAVA presenta 2 variaciones del Polimorfismo
  - Overloading
  - Overwriting

POO - Conceptos fundamentales - Polimorfismo en JAVA



POO - Conceptos fundamentales - Polimorfismo en JAVA

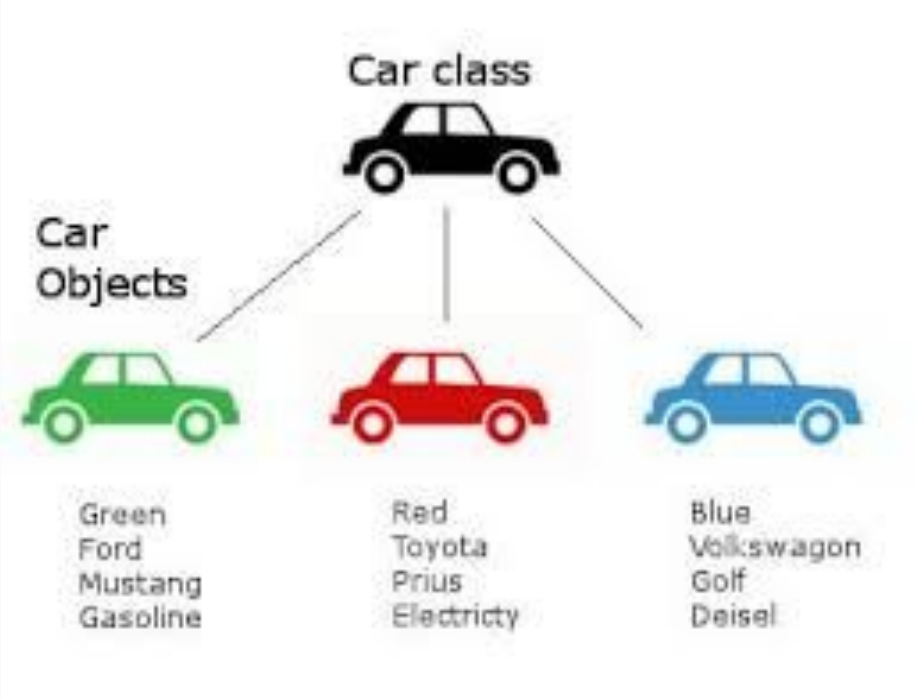
SOBRECARGA	SOBREESCRITURA
Se implementa dentro de la misma clase.	Se implementa entre 2 clases que tienen una relación de herencia.
Los parámetros del método deben ser diferentes.	Los parámetros siempre deben ser los mismos.
El tipo de retorno del método no importa, puede ser el mismo o diferente.	El tipo de retorno siempre debe ser el mismo.
Se puede modificar los modificadores de acceso.	Se pueden modificar siempre y cuando el nuevo modificador de acceso no sea más restrictivo que el original.
Se utiliza esta técnica para incrementar la legibilidad del programa.	Se utiliza esta técnica para proveer de implementaciones específicas de métodos que son provistos por la clase padre.

POO - Conceptos fundamentales - Objetos

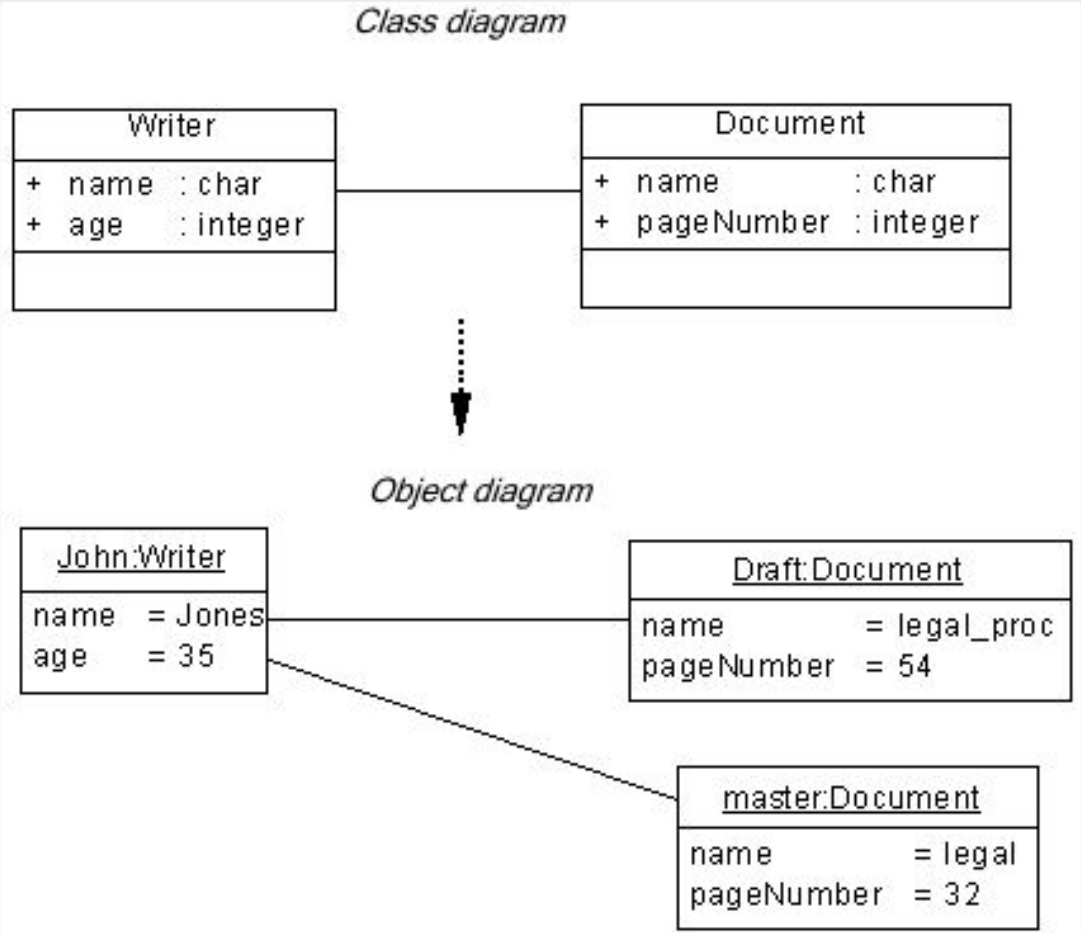
Objeto

- Instancia de una clase.
- Entidad provista de un conjunto de propiedades o atributos (datos) y de comportamiento o funcionalidad (métodos), los mismos que consecuentemente reaccionan a eventos.
- Se corresponden con los objetos reales del mundo que nos rodea, o con objetos internos del sistema (del programa).

POO - Conceptos fundamentales - Objetos



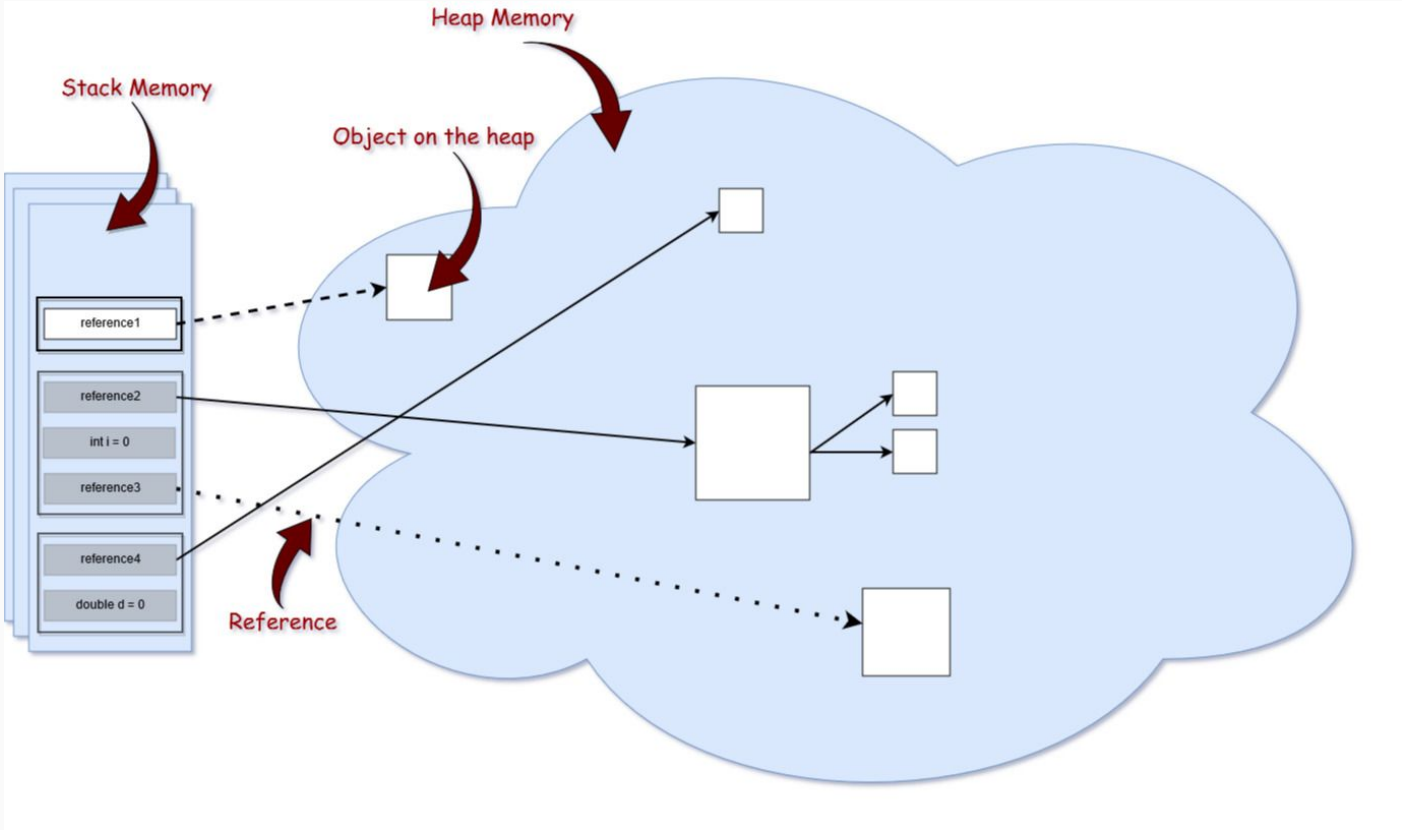
POO - Conceptos fundamentales - Diagrama de Objetos



Clases y Objetos: Ciclos de Vida

- Un problema fundamental cuando se trabaja con objetos es la forma en que se crean y se destruyen.
- Cada objeto requiere recursos, especialmente memoria, para existir. Cuando un objeto ya no se necesita, debe limpiarse para que estos recursos se liberen para su reutilización.
- Un enfoque es crear objetos dinámicamente en un conjunto de memoria llamado HEAP (montón).
- En este enfoque, no sabe hasta el tiempo de ejecución cuántos objetos se necesitan, cuál es su duración o cuál es su tipo exacto. Esos se determinan de forma espontánea mientras el programa se está ejecutando. Si necesita un objeto nuevo, simplemente lo hace en el montón en el punto que lo necesita.
- Cada vez que desee crear un objeto, utilice el nuevo operador para crear una instancia dinámica de ese objeto.

Administración de la Memoria



Administración de la Memoria

- *Stack*
  - Es responsable de mantener referencias a objetos del HEAP. y de almacenar valores de los tipos primitivos.
  - Las variables en el Stack tienen una cierta visibilidad, también llamada alcance (scope).
  - Si el compilador ejecuta el cuerpo de un método, solo puede acceder a los objetos de la pila que se encuentran dentro del cuerpo del método. No puede acceder a otras variables locales, ya que están fuera del alcance.
- *Heap*
  - Esta parte de la memoria almacena el objeto real en la memoria. Esas son referidas por las variables de la pila. Por ejemplo, analicemos lo que sucede en la siguiente línea de código.
  - La palabra clave NEW es responsable de garantizar que haya suficiente espacio libre en el HEAP.

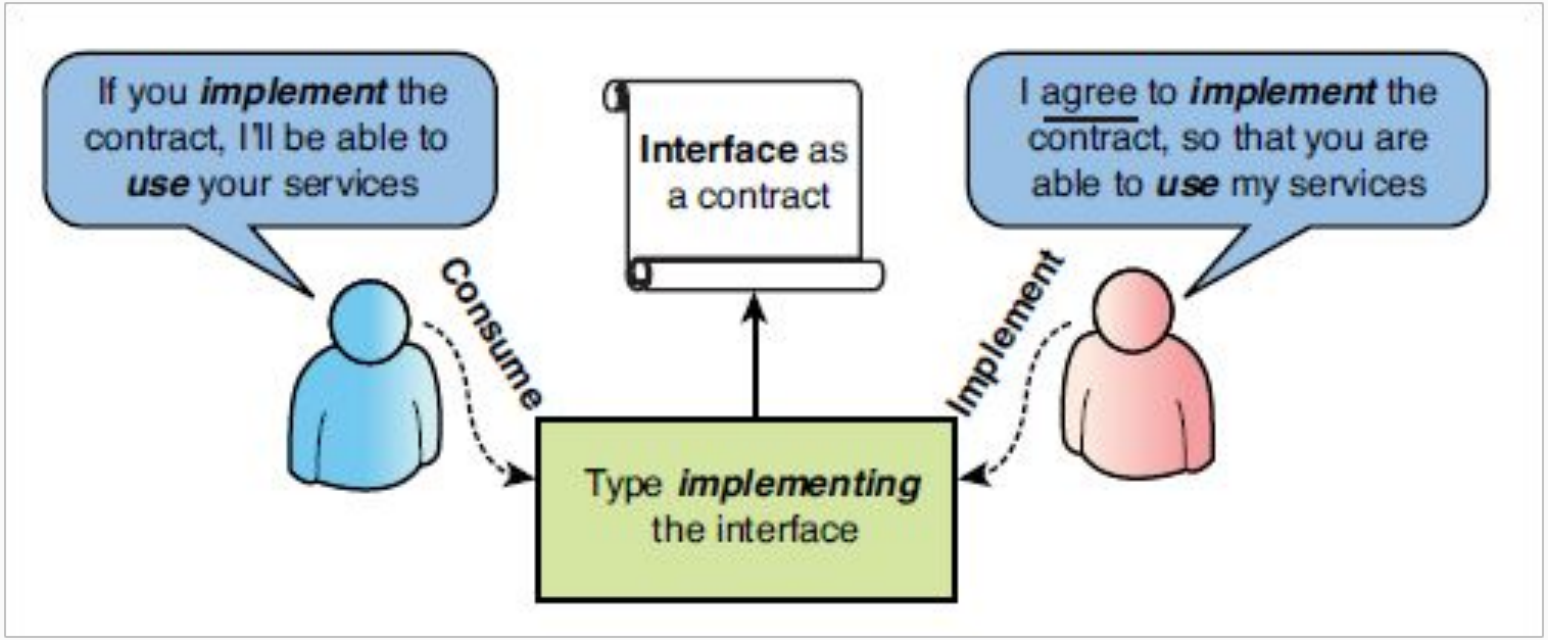
Tipos por Referencia

- Se denominan así a todos los objetos en poseen un puntero en la memoria Stack que “apunta” al “valor” del objeto en el Heap.
- El objeto en el Heap no se limpia (garbage collection) mientras haya una fuerte referencia que lo señale, o si es muy accesible a través de una cadena de fuertes referencias.
- En lenguajes como JAVA, existen diferentes tipos de referencias, lo cual permite seleccionar objetos para ser eliminados por el Garbage Collector.

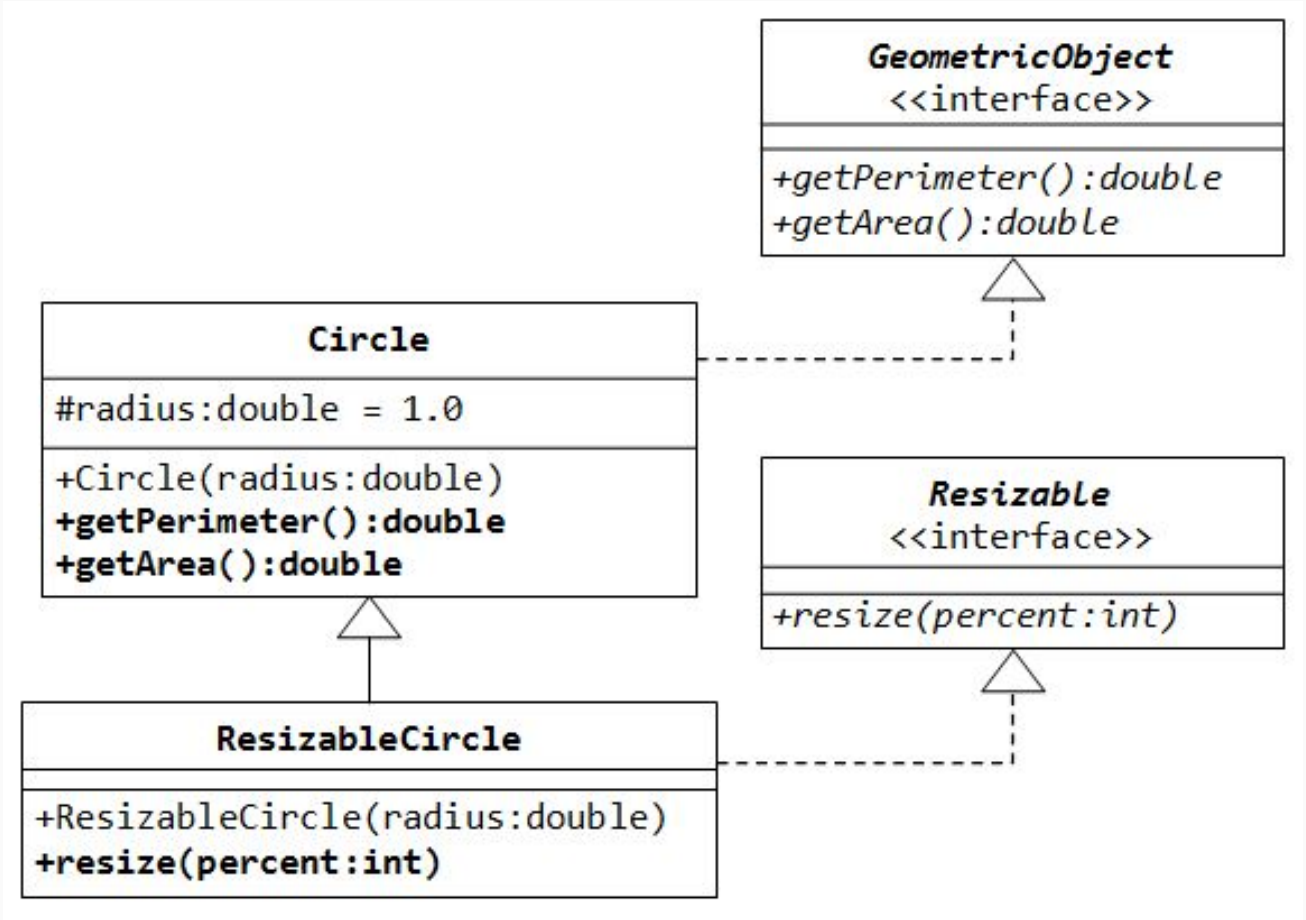
Interfaces

- Una interfaz es un conjunto de métodos que no definen un cuerpo o se dice que poseen un cuerpo vacío.
- Se definen formalmente como “contratos” que una clase debe “implementar” para exponer funcionalidad a otras clases.
- Una clase puede “implementar” los métodos de varias interfaces.
- Una interfaces posee la capacidad de “heredar” los métodos de una o varias interfaces.
- Las interfaces puede actuar como un medio de “comunicación” entre dos sistemas independientes.

POO - Conceptos fundamentales - Interfaces



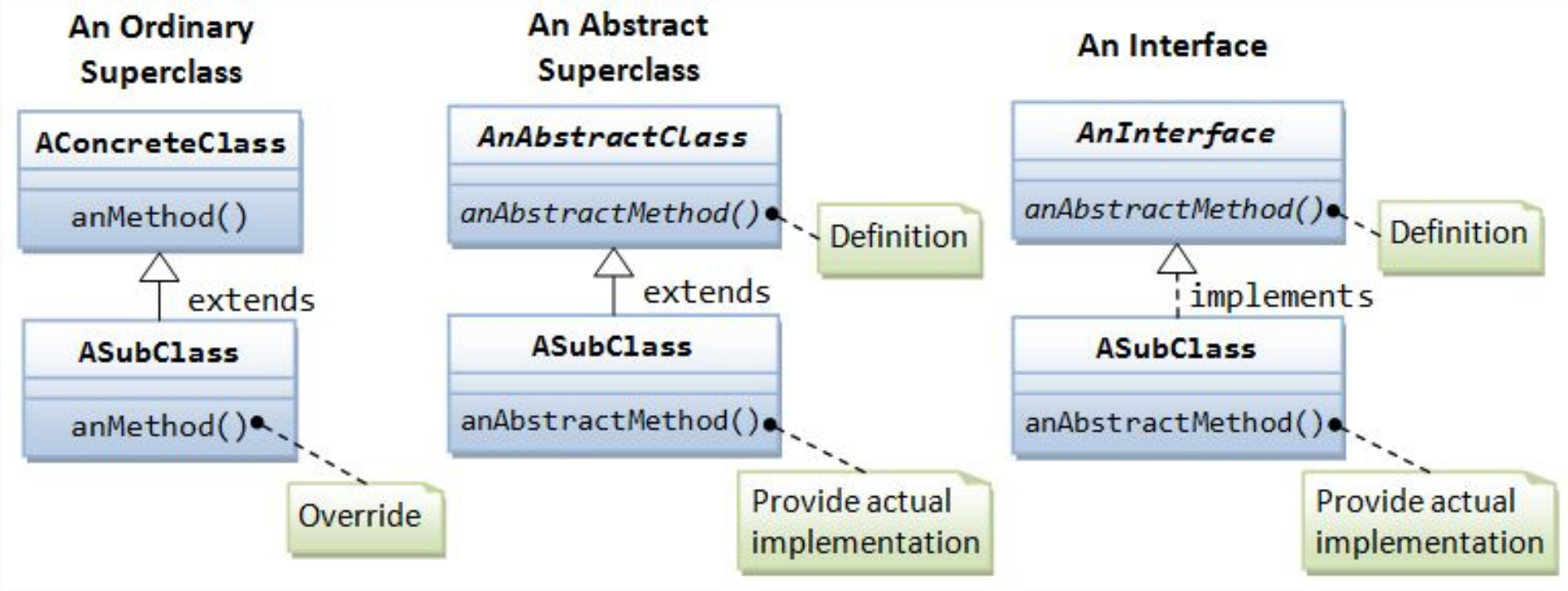
POO - Conceptos fundamentales - Interfaces



Diferencias entre Clases Abstractas e Interfaces

INTERFACES	CLASES ABSTRACTAS
Solo tienen Miembros incompletos: solamente define la firma de los métodos, pero no el cuerpo.	Pueden definir Miembros incompletos y completos.
No pueden tener modificadores de acceso, por defecto, se considera que todos los miembros son públicos.	Pueden definir modificadores de accesos para todos sus miembros: atributos, métodos, propiedades, etc.
No pueden incluir atributos.	Pueden definir atributos para almacenar información.
No poseen un Constructor.	Pueden definir uno o varios constructores.
Sus miembros no pueden ser Static.	Solo sus miembros completos puede ser Static.
Soportan la Herencia Múltiple.	No soportan la Herencia Múltiple (JAVA o C#)

Diferencias entre Clases vs Clases Abstractas vs Interfaces



POO - Conceptos fundamentales - Casting

Casting

- Significa tomar un Objeto de un tipo particular y "convertirlo" en otro tipo de Objeto.
- Este proceso se llama lanzar “castear” una variable.

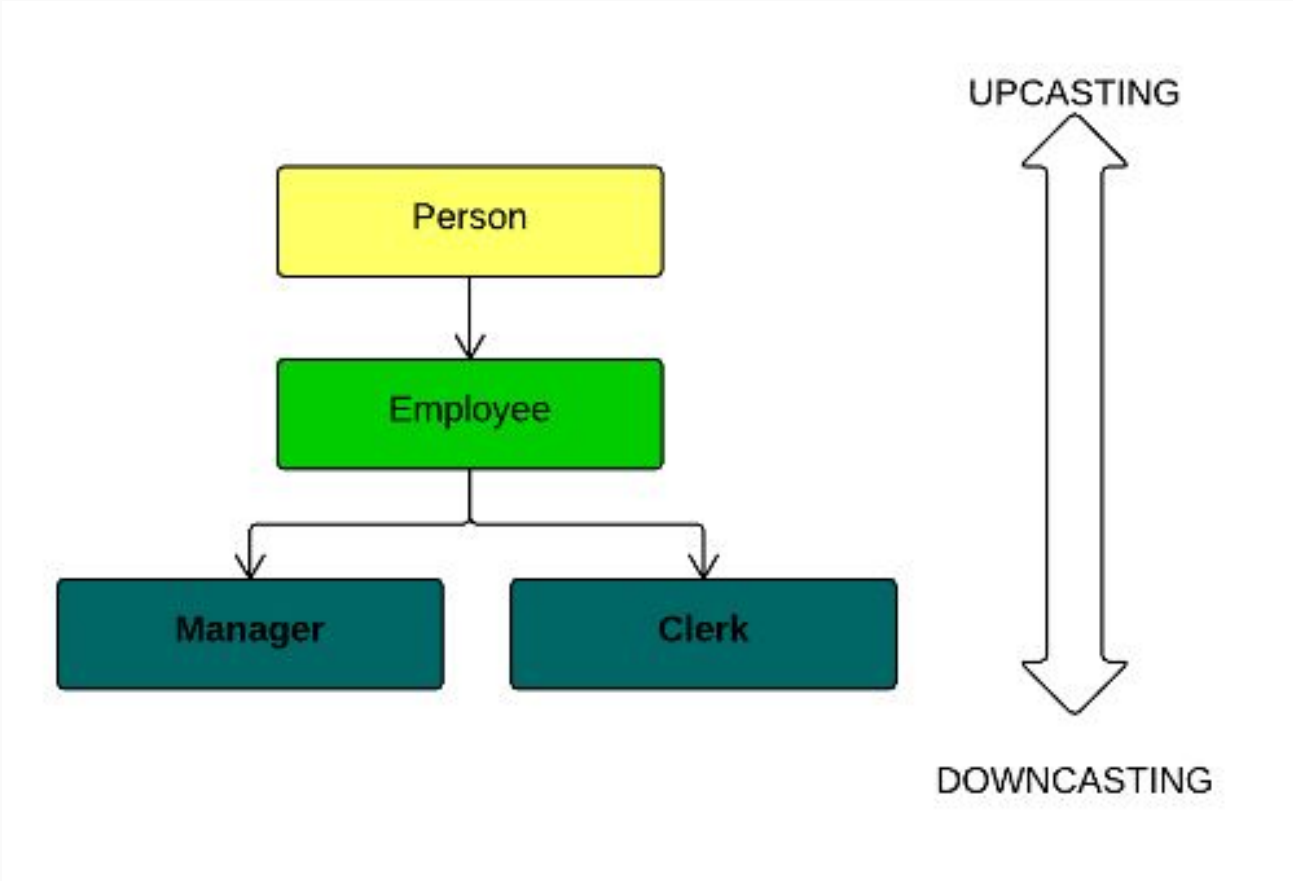
Upcasting

- Es el proceso de “Castear” desde una clase “hija” a una clase “base” (padre).
- Normalmente, se define a este proceso, como implícito.
- Involucra la “herencia” entre las clases.

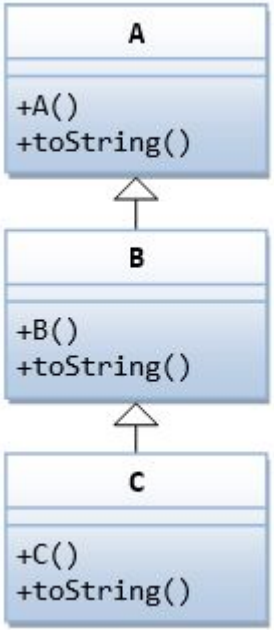
Downcasting

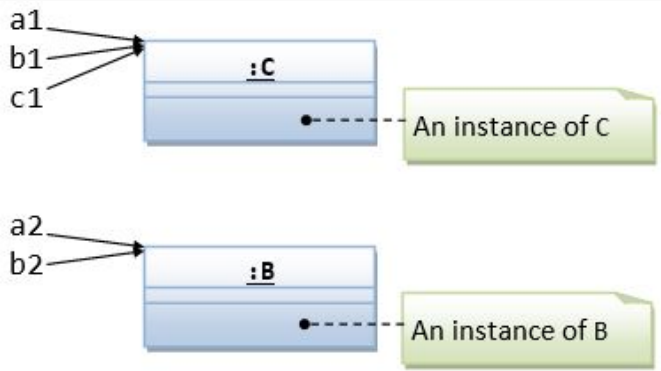
- Es el proceso inverso al Upcasting: “castear” una clase “base” a su clase “hija”.
- Este proceso es propenso a errores, debido a que la clase “padre” no siempre sabe específicamente a qué clase “hija” debe castearse.
- Se define como un proceso “Explícito”.

POO - Conceptos fundamentales - Casting



POO - Conceptos fundamentales - Casting

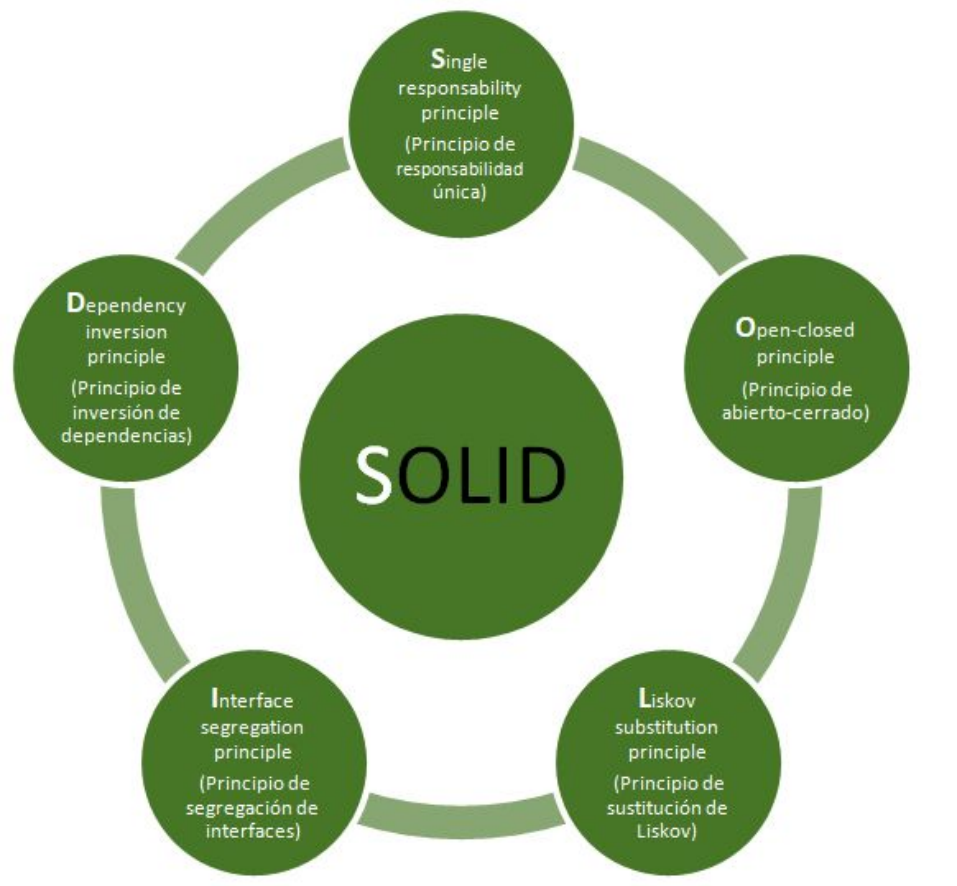





```
public class TestCasting {
   public static void main(String[] args) {
      A a1 = new C();    // upcast
      System.out.println(a1);  // run C's toString()
      B b1 = (B)a1;      // downcast okay
      C c1 = (C)b1;      // downcast okay

      A a2 = new B();   // upcast
      System.out.println(a2);  // run B's toString()
      B b2 = (B)a2;     // downcast okay
      C c2 = (C)a2;     // compilation okay, but runtime error ClassCastException
   }
}
```

POO - Principios SOLID



Solid, cinco principios básicos de diseño de clases

- Solid es un acrónimo inventado por Robert C.Martin para establecer los cinco principios básicos de la programación orientada a objetos y diseño.
- Este acrónimo tiene bastante relación con los patrones de diseño, en especial, con la alta cohesión y el bajo acoplamiento.
- El objetivo de tener un buen diseño de programación es abarcar la fase de mantenimiento de una manera más legible y sencilla así como conseguir crear nuevas funcionalidades sin tener que modificar en gran medida código antiguo.

S-Responsabilidad simple (Single responsibility)

- Este principio trata de destinar cada clase a una finalidad sencilla y concreta.
- En muchas ocasiones estamos tentados a poner un método reutilizable que no tienen nada que ver con la clase simplemente porque lo utiliza.

Ejemplo

Un algoritmo de formateo de números en una clase destinada a leer de la base de datos porque fue el primer sitio donde se empezó a utilizar. Esto conlleva a tener métodos difíciles de detectar y encontrar de manera que el código hay que tenerlo memorizado en la cabeza.

O-Abierto/Cerrado (Open/Closed)

- Principio que habla de crear clases extensibles sin necesidad de entrar al código fuente a modificarlo.
- El diseño debe ser abierto para poderse extender pero cerrado para poderse modificar.

Ejemplo

El uso más común de extensión es mediante la herencia y la implementación de métodos. Existe otra alternativa que consiste en utilizar métodos que acepten una interface de manera que podemos ejecutar cualquier clase que implemente ese interface. En todos los casos, el comportamiento de la clase cambia sin que hayamos tenido que tocar código interno.

L-Sustitucion Liskov (Liskov substitution)

- Este principio habla de la importancia de crear todas las clases derivadas para que también puedan ser tratadas como la propia clase base.
- Cuando creamos clases derivadas debemos asegurarnos de no re-implementar métodos que hagan que los métodos de la clase base no funcionasen si se tratasen como un objeto de esa clase base.

Ejemplo

Cada clase que hereda de otra puede usarse como su padre sin necesidad de conocer las diferencias entre ellas.

I-Segregacion del interface (Interface segregation)

- Este principio trata de algo parecido al primer principio.
- Cuando se definen interfaces estos deben ser específicos a una finalidad concreta.
- Si tenemos que definir una serie de métodos abstractos que debe utilizar una clase a través de interfaces, es preferible tener muchos interfaces que definen pocos métodos que tener un interface con muchos métodos.

Ejemplo

El objetivo de este principio es principalmente poder re-aprovechar los interfaces en otras clases. Si tenemos un interface que compara y clona en el mismo interface, de manera más complicada se podrá utilizar en una clase que solo debe comparar o en otra que solo debe clonar.

D-Inversión de dependencias (Dependency inversion)

- El objetivo de este principio conseguir desacoplar las clases.
- En todo diseño siempre debe existir un acoplamiento pero hay que evitarlo en la medida de lo posible.
- Un sistema no acoplado no hace nada pero un sistema altamente acoplado es muy difícil de mantener.

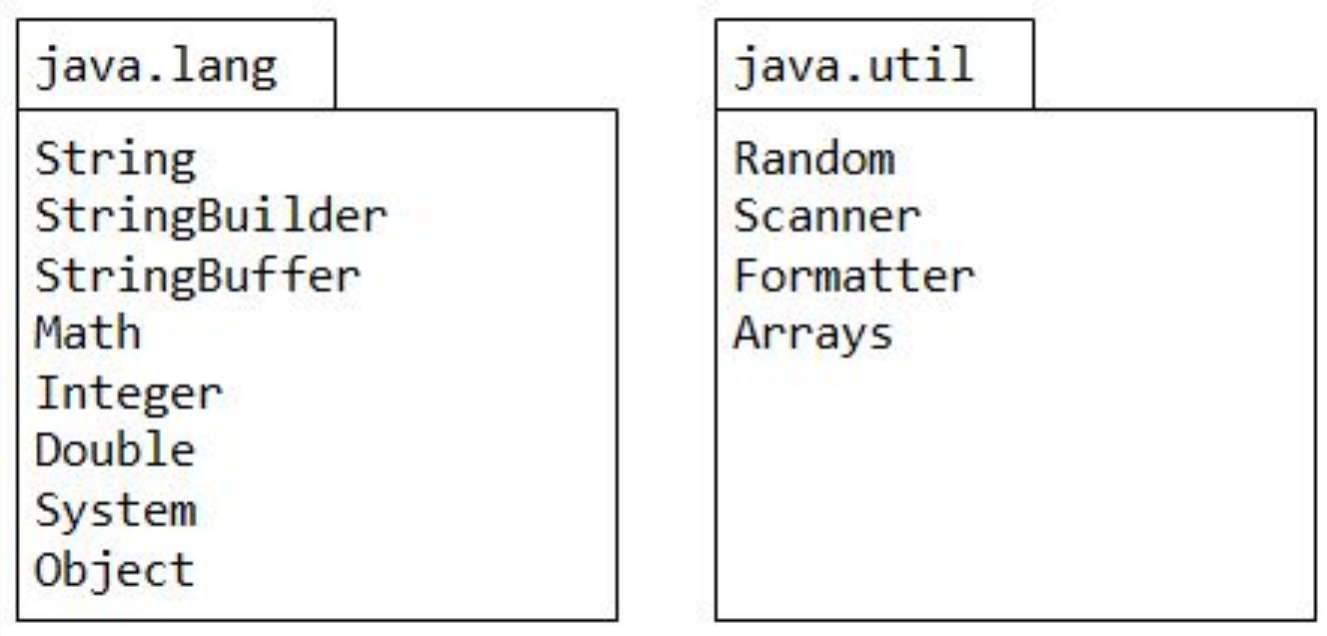
Ejemplo

El objetivo de este principio es el uso de abstracciones para conseguir que una clase interactúe con otras clases sin que las conozca directamente. Es decir, las clases de nivel superior no deben conocer las clases de nivel inferior. Dicho de otro modo, no debe conocer los detalles. Existen diferentes patrones como la inyección de dependencias o service locator que nos permiten invertir el control.

Paquetes y Namespaces

- Un paquete es una colección de clases y otros miembros relacionados, como interfaces, errores, excepciones, anotaciones y enumeraciones.
- Los paquetes se usan para:
  1. Organizar clases y otros miembros relacionados.
  2. Gestión de espacios de nombres (namespaces): cada paquete es un espacio de nombres.
  3. Resolución de conflictos de nombres. Por ejemplo, com.zzz.Circle y com.yyy.Circle se tratan como dos clases distintas. Aunque comparten el mismo nombre de clase Circle, pertenecen a dos paquetes diferentes: com.zzz y com.yyy. Estas dos clases pueden coexistir e incluso pueden usarse en el mismo programa a través de los nombres completos.
  4. Control de acceso: además de público y privado, puede otorgar acceso a una clase / variable / método a clases dentro del mismo paquete solamente.
  5. Distribución de clases Java: todas las entidades en un paquete se pueden combinar y comprimir en un solo archivo, conocido como archivo JAR (Java Archive), para su distribución.

POO - Conceptos fundamentales - Paquetes



Notación y uso de Paquetes: import

- Uso de la clase Scanner dentro del paquete java.lang.

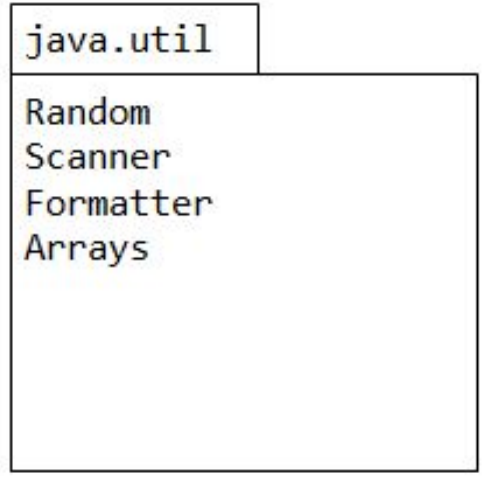
```
import java.util.Scanner;

public class Main {

    public static void main(String[] args) {

        Scanner s = new Scanner(System.in);

    }
}
```



- Uso de la clase Scanner usando el denominado nombre “**fully-qualified**”.

```
public class Main {

    public static void main(String[] args) {

        java.util.Scanner s = new java.util.Scanner(System.in);

    }
}
```

POO - Referencias y Material de Estudio

Referencias y Ejemplos

- Ejercicios
- Ejemplos

Referencias Extras

- Introduction to Java Programming
- Java Basis
- OOP Basis
- OOP - Composition, Inheritance and Polymorphism