Programación II
Herencia en Java (Parte 1)

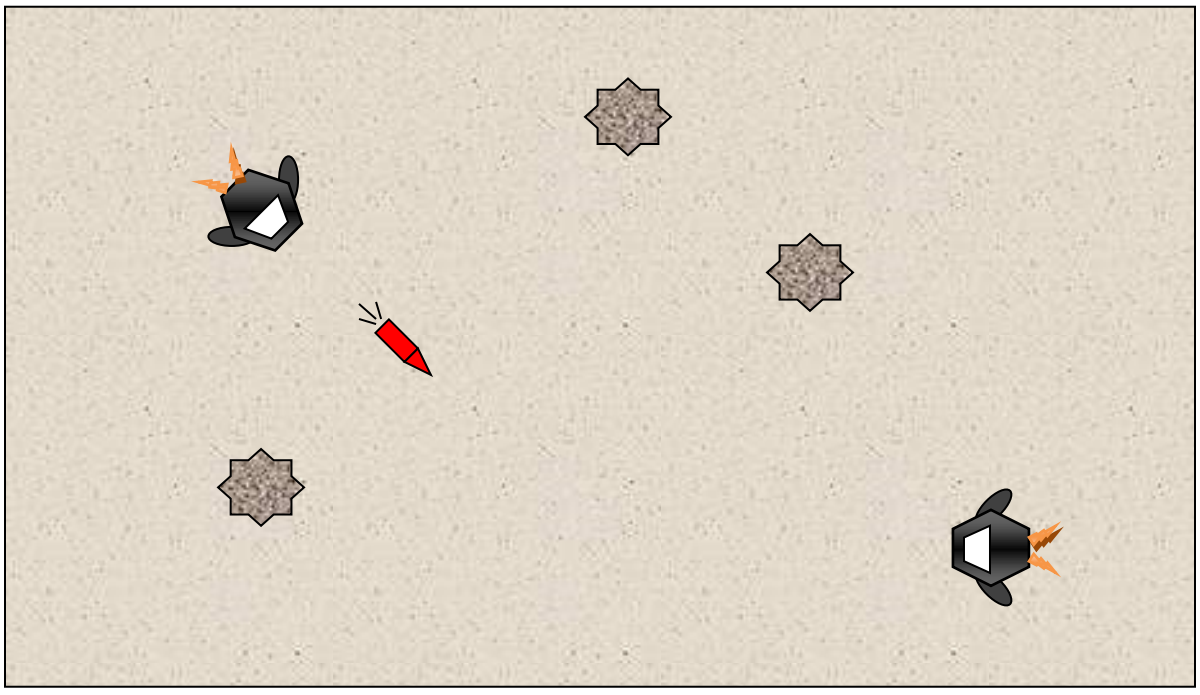
Queremos modelar un videojuego en el que dos naves intentan destruirse mutuamente.

1. Es un juego de dos jugadores (dos personas o bien una persona contrala computadora), cada uno controla una **nave**.

2. Las naves pueden disparar **misiles**. Cuando un misil impacta contra una nave, la nave pierde una unidad de energía. Cuando una nave pierde todas sus unidades de energía, es destruida y pierde el juego.

3. En el espacio de juego hay asteroides moviéndose libremente. Si un misil choca con un asteroide, el misil desaparece. Si una nave choca con un asteroide, pierde una unidad de energía.

¿Cómo podríamos modelar este juego? ¿Qué objetos tenemos? ¿Qué propiedades tienen? ¿Qué acciones pueden realizar?



Tenemos:

NAVES

Tienen:
- una posición en la pantalla
- una velocidad y un ángulo con el que se mueven
- una energía

Pueden:
- moverse
- chocar contra otros objetos
- disparar un misil
- aumentar o disminuir su velocidad
- cambiar su ángulo de movimiento
- perder energía
- desaparecer (cuando su energía llega a cero).

MISILES

Tienen:
- una posición en la pantalla
- una velocidad y un ángulo con el que se mueven

Pueden:
- moverse
- chocar contra otros objetos
- aparecer (cuando son disparados por una nave)
- desaparecer (cuando chocan contra otro objeto o se van de la pantalla)

ASTEROIDES

Tienen:
- una posición en la pantalla
- una velocidad y un ángulo con el que se mueven

Pueden:
- moverse
- chocar contra otros objetos
- cambiar su ángulo de movimiento

JUGADORES

Tienen una nave. Pueden enviar comandos a la nave, para que dispare un misil, y cambie su velocidad o su ángulo de movimiento.

JUEGO

Tiene una pantalla, y objetos que se mueven en esa pantalla

Marca el ritmo del juego, haciendo mover los objetos de acuerdo a la velocidad y el ángulo de cada uno de ellos. Verifica a cada paso si hay choque de objetos.

¿Hay algo que podamos inferir de esta lista de propiedades y acciones?

- ✓ Los tres tipos de objetos tienen posición (x; y) en el plano y un ángulo.
- ✓ Los tres se mueven de acuerdo con su posición, su velocidad y su ángulo.
- ✓ Cualquiera de los tres tipos de objetos puede chocar contra otro objeto en la pantalla.

Estos tres tipos de objetos tienen un comportamiento muy similar en algunos aspectos, aunque difieran un poco en otros. Esto nos lleva a pensar si estos tres tipos de objetos no pueden ser vistos también como miembros una clase común.

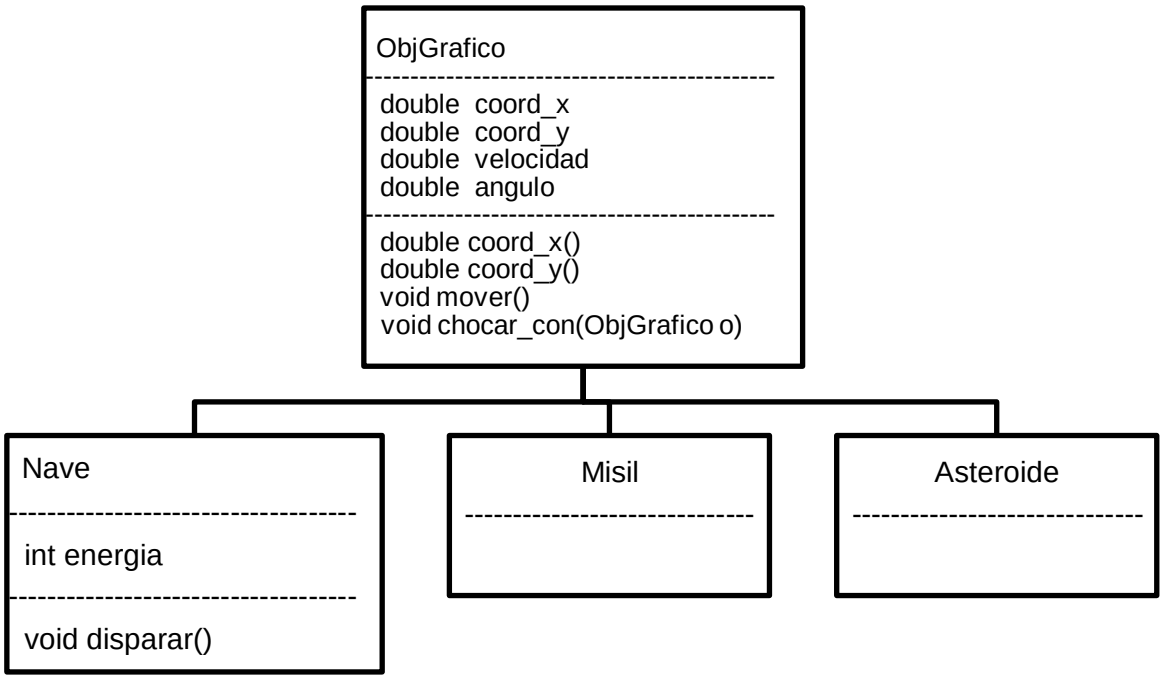
Y entonces, tiene sentido pensar tanto a *Nave*, como a *Asteroide* y a *Misil*, como un tipo particular de *Objeto Gráfico*.

- **Objetos Gráficos:** Tienen una posición en la pantalla, una velocidad y un ángulo con el que se mueven. Pueden moverse. Pueden chocar contra otros objetos gráficos. Pueden aumentar o disminuir su velocidad. Pueden cambiar su ángulo de movimiento.

Esta forma de encontrar una nueva clase, a partir de encontrar un comportamiento común entre otras clases ya existentes, se denomina habitualmente como mecanismo de **abstracción**. Veremos después que no es el único mecanismo.

Las clases *Nave*, *Misil* y *Asteroide* pasan a ser subclases de *Objeto Gráfico*. Y si la clase *ObjetoGrafico* tiene definidas determinadas propiedades y métodos, los objetos de las subclases también los tendrán.

Veamos cómo podríamos representar este modelado:



Decimos entonces, que las clases *Nave*, *Misil* y *Asteroide* **heredan de** la clase *ObjetoGrafico* sus métodos y datos.

Sintaxis de la herencia

Class *ObjetoGrafico*
private doublecoord_x
private doublecoord_y
public double coord_x(){ return coord_x}
public double coord_y(){ return coord_y}
…
Class Nave ***extends*** ObjetoGrafico
Class Misil ***extends*** ObjetoGrafico
Class Asteroide ***extends*** ObjetoGrafico

Nave, *Misil* y *Asteroide* heredan los métodos *coord_x()* y *coord_y()*, los cuales acceden a los atributos privados *coord_x*, y *coord_y*, respectivamente.

Nave, además de comportarse como un *ObjetoGrafico* (por los métodos que hereda), tiene también un comportamiento adicional propio. En nuestro ejemplo, una nave tiene una energía y puede disparar, características que no pertenecen a todos los objetos gráficos.

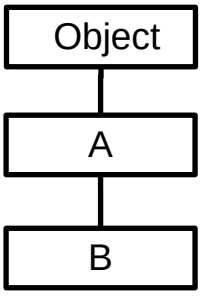
Definiciones

Se llama **subclase** a una clase B derivada de otra clase A. La clase A se llama la **superclase** o **clase base**.

Al derivar una clase B desde otra clase A, la clase B **hereda** automáticamente todos los miembros (datos y métodos) de la clase A.

Con excepción de la clase ***Object***, todas las clases en Java tienen exactamente una superclase. Esta restricción se conoce con el nombre de **herencia simple**.

Si para una clase no se declara explícitamente su superclase, se adopta implícitamente la clase ***Object*** como **superclase**.



Polimorfismo

Pensemos ahora un poco en algún posible algoritmo del juego. Muy probablemente, en la clase **Juego** tendremos que almacenar la lista de todos los objetos que se encuentran en la pantalla, no importa de qué tipo sean (*Nave*, *Misil* o *Asteroide*).

En Java, esto puede hacerse de la siguiente manera:

```
ArrayList<ObjetoGrafico> objetos;
Nave nave1, nave2;
Asteroide asteroide1, asteroide2, asteroide3;
...
...
objetos.add(nave1);
objetos.add(nave2);
objetos.add(asteroide1);
objetos.add(asteroide2);
objetos.add(asteroide3);
```

Luego, en cada iteración del ciclo principal del juego, podríamos tener algo similar a:

```
for (i=0;i<objetos.size();i++) {
      objetos.get(i).mover();
      ...
      ...
}
```

¿Qué pueden observar en este fragmento de código?

Si bien se define un ArrayList cuyos elementos son de la clase ObjetoGrafico, luego no se le agregan objetos de esa clase, sino de las clases Nave y Asteroide. Esto es posible gracias a la herencia, ya que cualquier objeto de una subclase, también puede ser visto como un objeto de la superclase.

Y dentro del ciclo, ¿cuál es la clase que atiende la invocación del método mover()?

Cuando en el ciclo invocamos al método *mover()*sobre un objeto gráfico, no nos estamos preocupando de qué clase o subclase es realmente el objeto. En tiempo de ejecución, el método *mover()*que se invoque dependerá de la clase del objeto sobre el que se vaya aplicando. Esta característica es lo que llamamos **polimorfismo**.

Definición

El **polimorfismo** se refiere a la propiedad por la que es posible enviar mensajes sintácticamente iguales a objetos de clases distintas.

En nuestro ejemplo, **las referencias a ObjetoGrafico son polimórficas**, dado que pueden ser de cualquiera de las tres subclases.

Sobreescritura

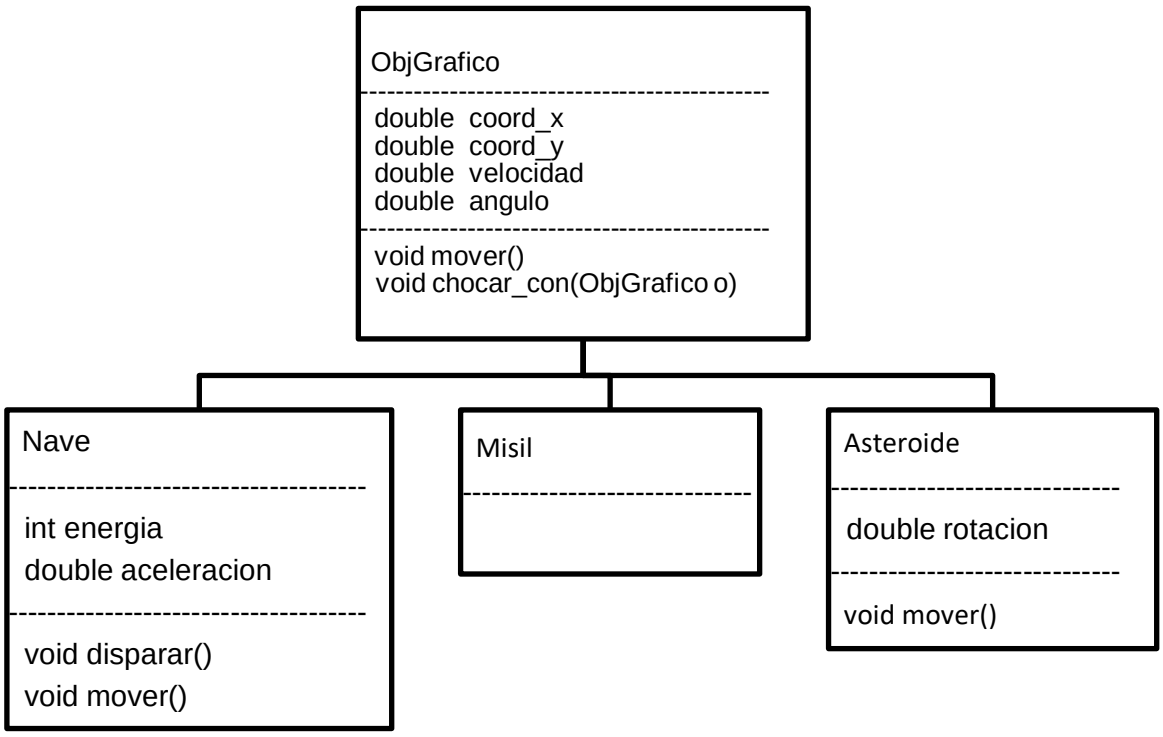
Ya vimos que una subclase hereda los métodos definidos en la superclase. Por lo que, en nuestro ejemplo, la invocación al método *mover()* de cualquiera de las tres subclases no haría más que terminar ejecutando el mismo método de la clase *ObjetoGrafico*.

Sin embargo, también podemos pensar en otra situación, en la que para algunas de las subclases el método *mover()* se comporte de manera diferente.

Supongamos que una nave también pudiera contar con aceleración, y que su movimiento tenga en cuenta esta aceleración. Y que un asteroide tenga un cierto grado de rotación, que también modifique su movimiento.

Pero el método *mover()* heredado de *ObjetoGrafico* no nos permite considerar ni la aceleración ni la rotación, ya que la superclase no conoce ninguna de las dos propiedades. *¿Cómo se resuelve entonces esta necesidad?*

Lo que hacemos, es reescribir el método *mover()*, tanto en la clase *Nave* como en la clase *Asteroide*. Dentro del método de la clase *Nave* podemos considerar la aceleración, y dentro del método de la clase *Asteroide*, la rotación.



En este caso, se dice que las subclases *Nave* y *Asteroide* **sobrescriben** el método *mover()* de la clase base.

Cuando en el ciclo principal del juego se llama a este método sobre un *ObjetoGrafico*, si el objeto es una nave o un asteroide se ejecuta el método sobrescrito en la subclase en cuestión. En cambio, si el objeto es un misil, se ejecuta el método heredado de *ObjetoGrafico*.

Sintaxis de la sobrescritura

Cuando se sobrescribe un método, se puede poner antes la **anotación "**Override", para indicarle al compilador que nuestra intención es sobrescribir un método de la superclase.

```
@Override
 void mover()
 {
 ...
 }
```

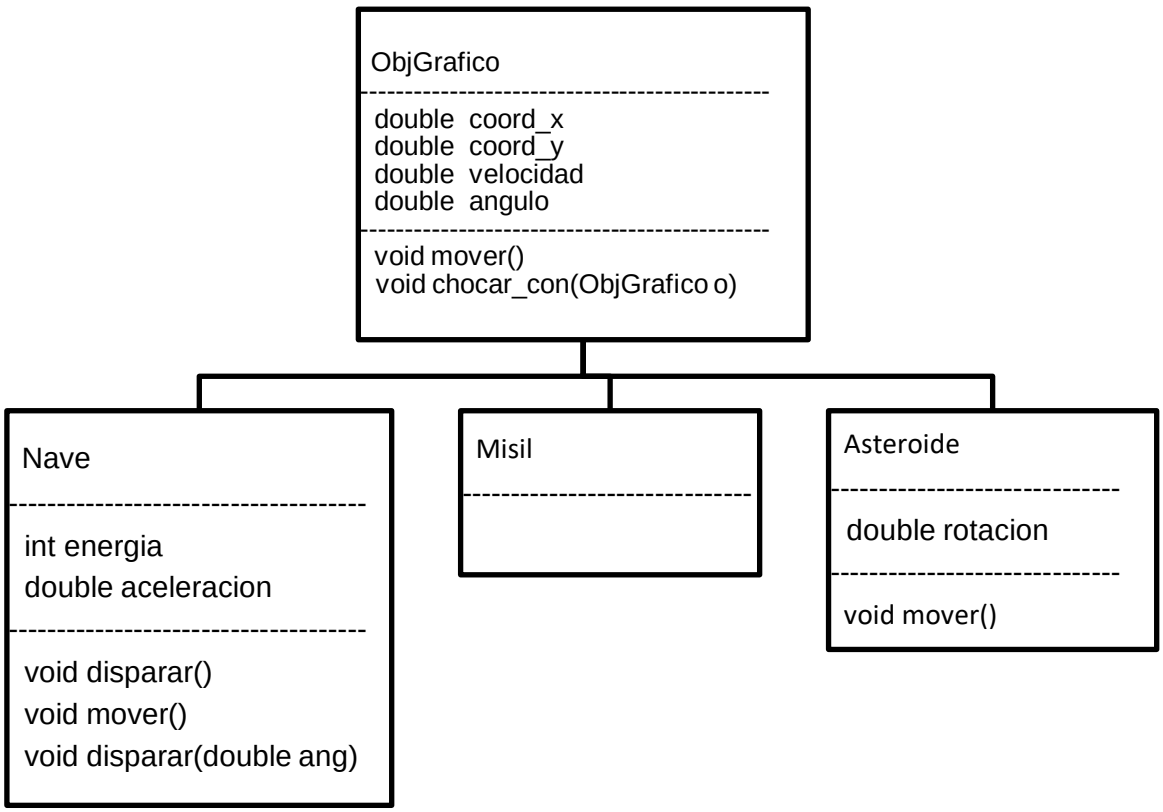
Si el método no existe en ninguna de las superclases, se genera un error.

Sobrecarga

Imaginémonos ahora qué debería realizar el método *disparar()* de la Nave.

Podría, por ejemplo, crear un nuevo objeto *Misil*, en una posición próxima a la de la nave, y dotarlo de su misma velocidad y ángulo, para que a partir de ese momento se mueva en el ciclo junto a los demás objetos.

Pero también podemos pensar que el ángulo con que se mueva el misil no sea el mismo que el de la nave, para que pueda disparar en otra dirección. Esto nos lleva a crear otro método en la clase *Nave*, con el mismo nombre, pero incluyendo como parámetro el ángulo de disparo.



Los métodos *disparar()* y *disparar(ang)* son diferentes, y pueden coexistir sin ningún inconveniente dentro de la misma clase. Esto es así, porque el compilador reconoce que tienen diferente cantidad de parámetros, y eso le permite saber cuál de los dos métodos es el que se está invocando. En este caso, se dice que el método *disparar* está **sobrecargado**.

Existen 3 componentes de un método que lo hacen único. Variando cualquiera de estos componentes, se puede definir un método diferente, sobrecargándolo:

- Nombre del método
- Cantidad de parámetros
- Tipo de los parámetros

Al conjunto de estos 3 componentes, se lo denomina **signatura** o **firma** del método.

Definiciones

La **sobrecarga** se refiere a la posibilidad de tener dos o más funciones con el mismo nombre, con la misma funcionalidad, pero con diferente signatura. Es decir, dos o más funciones con el mismo nombre, pero con parámetros diferentes. El compilador usará una u otra dependiendo de los parámetros usados.

La **sobreescritura** se produce, en cambio, cuando un método de instancia en una subclase está definido con la misma **signatura** que un método de instancia de la superclase, con lo cual se dice que **sobrescribe** (overrides) el método original.

Un ejemplo, importante, para tener en cuenta. Si en cualquier clase tenemos definidos estos dos métodos (con diferente signatura):

(1) *public String toString()*
(2) *public void toString(String s)*

Mientras que (1) sobreescribe el método Object.toString(), en (2) solamente se está realizando una sobrecarga de (1).

Sobreescritura vs. Sobrecarga

Sobreescritura	**Sobrecarga**
Diferentes clases que se heredan	Misma clase
Misma signatura	Diferente signatura
Diferente algoritmo	Mismo algoritmo para otro caso

Clases abstractas

Volviendo a nuestro juego, vimos que contamos con objetos de las clases *Nave*, *Asteroide* y *Misil*. *¿Tendrá algún sentido contar con objetos de la clase ObjetoGrafico?*

Evidentemente, no. Si queremos asegurarnos que nadie pueda crear un objeto de una clase, lo que podemos hacer es definirla como una **clase abstracta**.

Una **clase abstracta** es una clase que no se puede **instanciar** (no se pueden crear objetos de esa clase), y se utiliza como base para subclases concretas.

En nuestro ejemplo, *ObjetoGrafico* puede ser una clase abstracta, dado que no tenemos objetos en la aplicación de esa clase.

Un **método abstracto** es un método que no tiene implementación, y que debe sobrescribirse en las subclases.

```
public abstract class ObjetoGráfico
{
 ...
      abstract void dibujar();
      }
```

En este ejemplo, ObjetoGrafico no implementa el método *dibujar()*, pero sí obliga a que lo implementen todas sus subclases.