Programación II
Herencia en Java

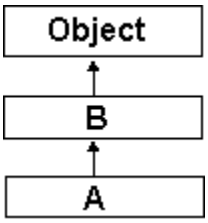
Definición

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

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

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.



Ejemplo

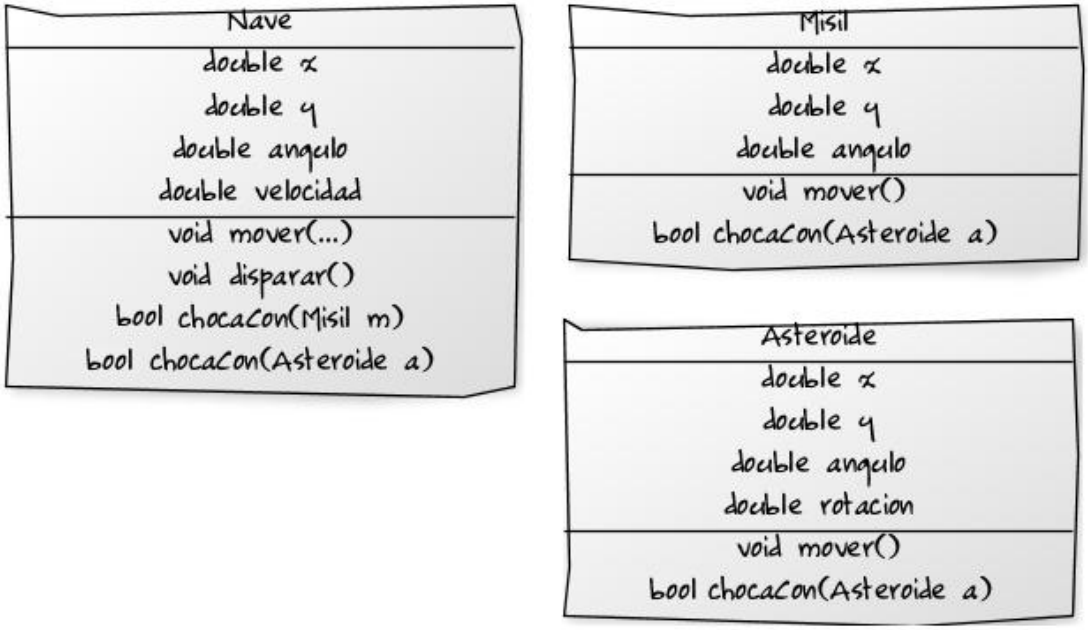
Estamos implementando un videojuego en el que dos naves intentan destruirse mutuamente.

1. Es un juego de dos jugadores (dos personas o bien una persona contra la 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.

Ejemplo1



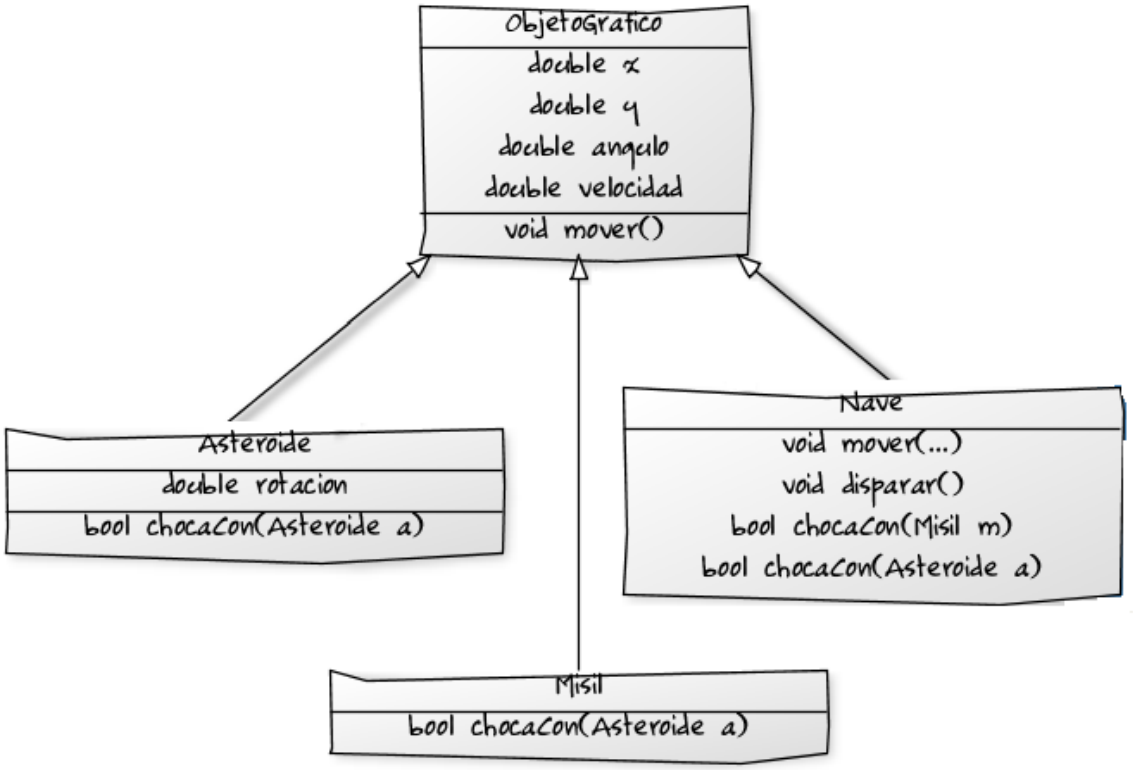
Las tres clases tienen **datos y métodos repetidos**!

1. Los tres tipos de objetos tienen posición (x; y) en el plano y ángulo.
2. Los tres se mueven de acuerdo con su posición y su ángulo, y además en un caso se debe tener en cuenta la velocidad.
3. Hay muchos métodos para chequear si un objeto choca contra otro!

Más aún, el código de todos ellos seguramente será muy similar.

Cuando sucede esto, es importante preguntarse si estos objetos no son en realidad **miembros de una misma clase**, cada uno con sus características.

Ejemplo2



Sintaxis de la herencia

Class ObjetoGrafico
private doublé x
private doublé y
public double x(){ return x}
public double y(){ return y}
…

Class Asteroide ***extends*** ObjetoGrafico

Class Nave ***extends*** ObjetoGrafico

Nave y Asteroide heredan el métodos x() e y(), los cuales acceden a los atributos privados x e y respectivamente.

Polimorfismo

La función principal ahora tiene una lista de ObjetoGrafico, que mantiene todos los objetos en la pantalla.

```
ArrayList<ObjetoGrafico> objetos;
Nave nave;
Asteroide asteroide;

objetos.add(nave)
objetos.add(asteroide)
```

Ejemplo2

El polimorfismo se refiere a la propiedad por la que es posible enviar mensajes sintácticamente iguales a objetos de tipos distintos.

En el ejemplo2, las referencias a ObjetoGráfico son polimórficas, dado que pueden ser de cualquiera de las tres subclases.

Cuando se ejecuta un método sobre un ObjetoGráfico, no es relevante en ese momento de qué tipo es realmente el objeto.
El método mover va tomando la forma del ObjetosGrafico que se este referenciando.

Signatura(Firma)

La firma de una función se compone de 4 componentes que la hacen única:
-Nombre
-Tipo de retorno(Ej. String, Boolean, etc.)
-Modificadores(Ej. Public, Protected, Final, etc.)
-Parametros. Para cada parámetro:
- Nombre
- Posición
- Tipo

Por ejemplo

(1) Public String toString(), tiene distinta signatura que
(2) Private String toString()

Mientras que (1) sobreescribe Object.toString().
En el caso (2) solamente se sobrecarga (1).

Sobreescritura (*overriding*) de métodos

Un método de instancia en una subclase con la misma **signatura** y el mismo **tipo de retorno** que un método de instancia de la superclase **sobrescribe** (overrides) el método original.

La subclase Nave **sobrescribe** el método mover() de la clase base.

Cuando se llama a este método sobre un ObjetoGráfico, se ejecuta el método sobrescrito en la subclase en cuestión.

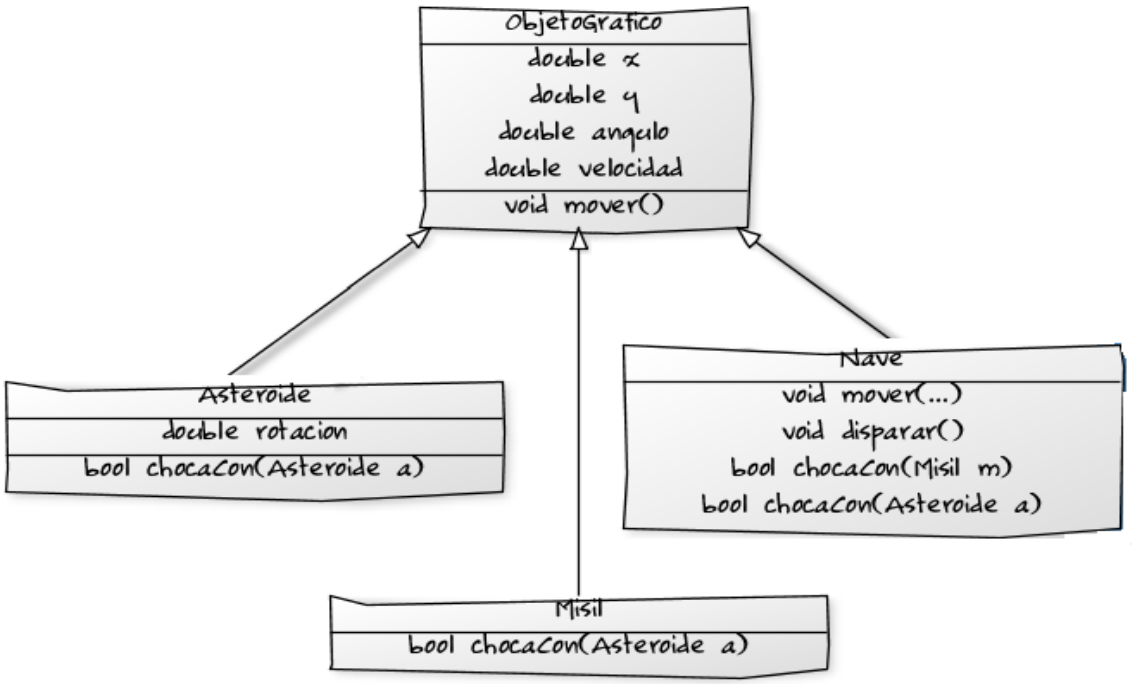
Se puede llamar al método de la superclase desde el método sobrescrito, en este caso con `ObjetoGráfico.mover()`.

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()
 {
 ...
 }
```

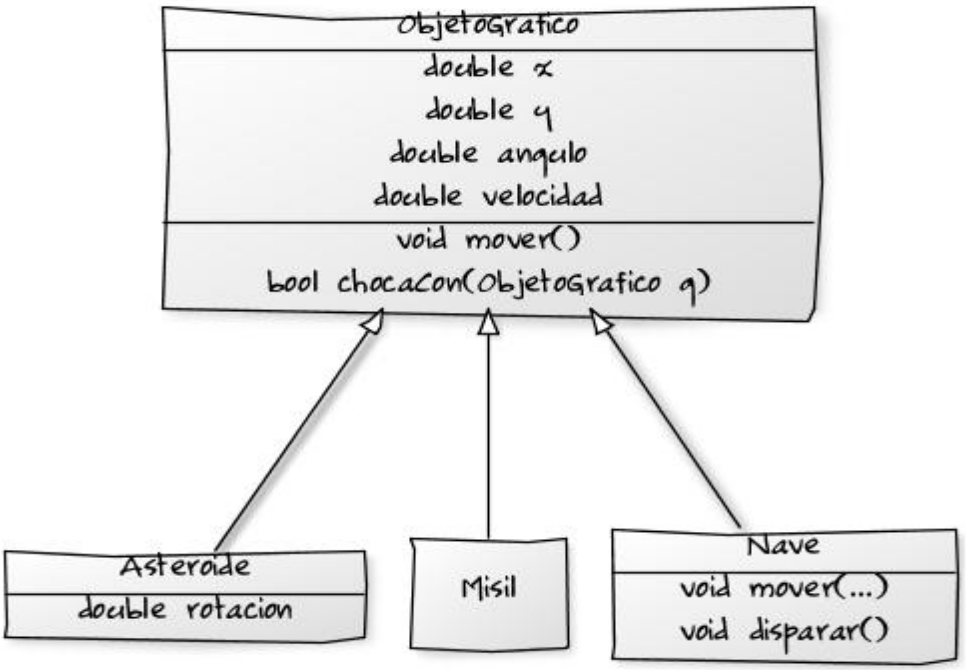
Si el método no existe en ninguna de las superclases, se genera un error.

Ejemplo3



¿Podemos hacer algo mas?

Ejemplo4



Sobrecarga

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 con parámetros diferentes. El compilador usará una u otra dependiendo de los parámetros usados.

Ejemplo: Articulo.java

Algunos métodos en una clase pueden tener el mismo nombre. Estos métodos deben contar con diferentes argumentos. El compilador decide qué método invocar comparando los argumentos. Se generara un error si los métodos tienen definidos los mismos parámetros.

```
public class Articulo {
   private float precio;
   public void setPrecio() {
        precio = 3.50;
   }
   public void setPrecio(float nuevoPrecio) {
        precio = nuevoPrecio;
   }
}
```

Sobreescritura Vs. Sobre-Carga

Sobreescritura	**Sobrecarga**
Diferentes clases que se heredan	Misma clase
Misma signatura	Diferente signatura
Diferente Algoritmo	Mismo Algoritmo para otro caso.
Ej: int sumar(int x, unt y)	void sumar(int x, int y). El resultado se acumula sobre x.

Clases abstractas

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, ObjetoGráfico 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();
      }
```

Interfaces

Una **interface** es un tipo de referencia (similar a una clase) que contiene solamente constantes, signaturas de métodos y tipos anidados.

En una interface no hay código y no puede ser instanciada! Es similar a una clase abstracta en la que todos los métodos son abstractos.

Se dice que una clase **implementa** una interface, y se utiliza la palabra ***implements*** en el código para hacerlo explícito

```
public class Persona implements Serializable
 {
 ...
 }
```

Clases abstractas Vs. Interfaces

Las clases abstractas pueden contener **métodos implementados**.
Las interfaces no contienen código.

Una clase **implementa** una interface (palabra clave *implements*), mientras que una subclase **extiende** (palabra clave ***extends***) una clase base.

Una clase puede heredar de una única clase base, mientras que puede implementar más de una interface.

Una interface específica un **contrato** que dice qué métodos debe implementar la clase. En cambio, el mecanismo de herencia permite **compartir código común** a varias clases.