HERENCIA, POLIMORFISMO E INTERFACES

Adrián A. Alarcón O.

HERENCIA

HERNECIA

Es el proceso donde una clase adquiere las propiedades de otra clase

- extends

EJEMPLO

```
class Vehicle{
   public String color:
}

class Car extends Vehicle{

}

class Truck extends Vehicle{

}

class MainClass{
   public static void main(String[] args) {
     Car car = new Car();
     Truck truck = new Truck();

     car.color = "red";
     truck.color = "blue";
   }
}
```

RELACIÓN "ES UN"

```
public class Animal {

}

public class Mammal extends Animal {

}

public class Reptile extends Animal {

}

public class Dog extends Mammal {

}
```

RELACIÓN "TIENE UN"

```
public class Vehicle{}
public class Speed{}
public class Seat{}

public class Van extends Vehicle {
   private Speed speed;
   private Seat seat;
}
```

POLIMORFISMO

POLIMORFISMO

Es una relación de herencia, un objeto de la clase padre puede almacenar un objeto de cualquiere de las clases hijas (heredadas)

La clase padre es compatible con todos los tipos que derivan de ella

POLIMORFISMO

```
class Vehicle{
  String brand;
  String model

  protected void showInfo(){
    System.out.println("brand: " + brand);
    System.out.println("model: " + model);
  }
}

class Car extends Vehicle{
  int wheels;
}

class Boat extends Vehicle{
  int anchor;
}
```

POLIMORFISMO

```
class MyApplication{

  public static void main(String[] args) {

    Vehicle vehicle = new Vehicle();
    vehicle.brand = "Generic";
    vehicle.model = "2018";

    Car car = new Car();
    car.brand = "Nisan";
    car.model = "2017";
    car.wheels = 4;
    car.showInfo();

    Boat boat = new Boat();
    boat.brand = "Galeon";
    boat.model = "2016";
    boat.anchor = 1;
    boat.showInfo();
  }
}
```

POLIMORFISMO

```
class MyApplication{

  public static void main(String[] args) {

    Vehicle a;

    Vehicle b;

  }
}
```

POLIMORFISMO

```
class MyApplication{

  public static void main(String[] args) {

    Vehicle a = new Car();

    Vehicle b = new Boat();

  }
}
```

POLIMORFISMO

```
class MyApplication{

  public static void main(String[] args) {

    Vehicle a = new Car();
    a.brand = "Nisan";
    a.model = "2017";

    Vehicle b = new Boat();
    b.brand = "Galeon";
    b.model = "2016";
  }
}
```

OVERRIDING

```
class Vehicle{
  String brand;
  String model

  protected void showInfo(){
    System.out.println("brand: " + brand);
    System.out.println("model: " + model);
  }
}

class Car extends Vehicle{
  int wheels;

   protected void showInfo(){
     System.out.println("brand: " + brand);
     System.out.println("model: " + model);
     System.out.println("wheels: " + wheels);
  }
}
```

OVERRIDING

```
class MyApplication{

  public static void main(String[] args) {

    Vehicle car1 = new Vehicle();
    car1.brand = "Nisan";
    car1.model = "2017";
    car1.showInfo();

    Vehicle car2 = new Car();
    car2.brand = "Galeon";
    car2.model = "2016";
    car2.showInfo();
  }
}
```

OVERRIDING

```
class Animal {
  public void move() {
    System.out.println("Animals can move");
  }
}

class Dog extends Animal {
  public void move() {
    System.out.println("Dogs can walk and run");
  }
  public void bark() {
    System.out.println("Dogs can bark");
  }
}
```

OVERRIDING

```
public class TestDog {

  public static void main(String args[]) {
    Animal animal = new Animal();
    Animal dog = new Dog();

    animal.move();
    dog.move();
    dog.bark();
  }
}
```

OVERLOADING

La sobrecarga permite tener multiples metodos con el mismo nombre pero con diferentes parametros y tipos

Nota: La sobrecarga solo aplica a los parametros y no al tipo de retorno

OVERLOADING

```
public class Sum {

  public int sum(int x, int y) {
    return (x + y);
  }

  public int sum(int x, int y, int z) {
    return (x + y + z);
  }

  public double sum(double x, double y) {
    return (x + y);
  }

}
```

OVERLOADING

```
public class MainClass{

  public static void main(String args[]) {
    Sum s = new Sum();
    System.out.println(s.sum(10, 20));
    System.out.println(s.sum(10, 20, 30));
    System.out.println(s.sum(10.5, 20.5));
  }

}
```

OVERLOADING

```
public class MainClass{

  public int foo() {
    return 10;
  }

  public char foo() {  // error de compilación
    return 'a';
  }

}
```

ENCAPSULAMIENTO

ENCAPSULAMIENTO

Consiste en ocultar del estado de un objeto, de forma que sólo es posible modificarlo mediante los métodos definidos para dicho objeto.

ENCAPSULAMIENTO

```
public class Employee {

  public int age;

}
```

ENCAPSULAMIENTO

```
public class MainApp {
  Employee e = new Employee();

  e.age = 40;
  e.age = -10;
  e.age = 200;

}
```

ENCAPSULAMIENTO

```
public class Employee {

  private int age;

  public int getAge() {
    return age;
  }

  public void setAge(int newAge) {
    if (newAge >= 0 && newAge <= 100){
      age = newAge;
    }
  }

}
```

INTERFACES

INTERFACES

Es similar a una clase pero no implementa ningún método, solo los define.

Todos sus metodos son abstractos.

- abstract
- implements

INTERFACES

```
public interface Animal {

  public abstract void eat();
  public abstract void sleep();

}
```

INTERFACES

```
public class Cat implements Animal {

   public void eat() {
      System.out.println("Cat eating");
   }

   public void sleep() {
      System.out.println("Cat sleeping");
   }

}
```

INTERFACES

```
class MainClass{

  public static void main(String args[]) {
     Cat c = new Cat();
     c.eat();
     c.sleep();
  }

}
```

INTERFACES

```
class MainClass{

  public static void main(String args[]) {
     Animal c = new Cat();
     c.eat();
     c.sleep();
  }

}
```

HERENCIA EN INTERFACES

```
public interface Sports {
   public void setHomeTeam(String name);
   public void setVisitingTeam(String name);
}

public interface Football extends Sports {
   public void homeTeamScored(int points);
   public void visitingTeamScored(int points);
   public void endOfQuarter(int quarter);
}

public interface Hockey extends Sports {
   public void homeGoalScored();
   public void visitingGoalScored();
   public void endOfPeriod(int period);
   public void overtimePeriod(int ot);
}
```

HERENCIA MULTIPLE INTERFACES

```
public interface Hockey extends Sports, Event{

}
```

HERENCIA MULTIPLE INTERFACES

```
public interface Printer{
  public void print();
}

public interface Scanner{
  public void scan();
}

public class MultiFunction implements Printer, Scanner{
  public void print(){
    //printer implementation here
  }

  public void scan(){
    //scan implementation here
  }

}
```

EJERCICIO: LISTA DE PROFESORES

EJERCICIO: GAME OF THRONES

CLASES ABSTRACTAS

Este tipo de clases nos permiten crear “método generales”, que recrean un comportamiento común, pero sin especificar cómo lo hacen

- Tiene métodos abstractos
- No puede ser instanciada

EJEMPLO

```
abstract class Character{

  protected Weapon weapon;
  protected String name;

  Character(Weapon weapon, String name){
    this.weapon = weapon;
    this.name = name;
  }

  public void changeWeapon(Weapon weapon){
    this.weapon = weapon;
    System.out.println("Cambio de arma");
  }

  public abstract void fight();

}
```

EJERCICIO: EL CONGRESO MEXICANO

COHESIÓN

COHESIÓN

Se refiere al grado en que una clase tiene un único y bien definido papel o responsabilidad

- Alta cohesión es el estado optimo, donde una clase esta diseñada con un solo objetivo

ACOPLAMIENTO

ACOPLAMIENTO

Se refiere al grado en que una clase conoce los miembros de otra clase

- Bajo acoplamiento es el grado deseable, se trata de reducir al minimo el uso de un API
- El uso de una interfaz en una variable de referencia es un ejemplo de bajo acoplamiento

CASTING

CASTING

Consiste en la conversión de un tipo de objeto a otro tipo de objeto

- Downcasting
- Upcasting

CASTING

```
class Person{

  String name;

  void walk(){
    //...
  }

  void talk(){
    //...
  }
}

class Professor extends Person{

  void teach(){
    //...
  }

}
```

UPCASTING

```
void main(){

  Profesor professor = new Professor();

  Person person = new Person();

  person = professor;

}
```

DOWNCASTING

```
void main(){

  Profesor professor = new Professor();

  Person person = new Person();

  professor = (Profesor)person;

}
```

DOWNCASTING

```
void main(){

  Profesor professor = new Professor();

  Person person = new Profesor();

  professor = (Profesor)person;

}
```

INSTANCEOF

INSTANCEOF

Es un operador que nos sirve para probar de que tipo de instancia es un objeto

INSTANCEOF

```
void main(){

  Person person = new Profesor();

  if (person instanceof Profesor){
    System.out.prinln("Efectivamente person es un Profesor");
  }

}
```

INSTANCEOF

```
void main(){

  Person[] persons = new Person[10];

  // aqui llenamos el arreglo de personas

  for(int i =0; i <= persons.length; i++){
    if (persons[i] instanceof Profesor){
      System.out.prinln("Buenos dias querido profesor!");
    }
  }

}
```

GARBAGE COLLECTOR

GARBAGE COLLECTOR

Es un mecanismo de gestión de memoria y se encarga de liberar y compactar el espacio en memoria

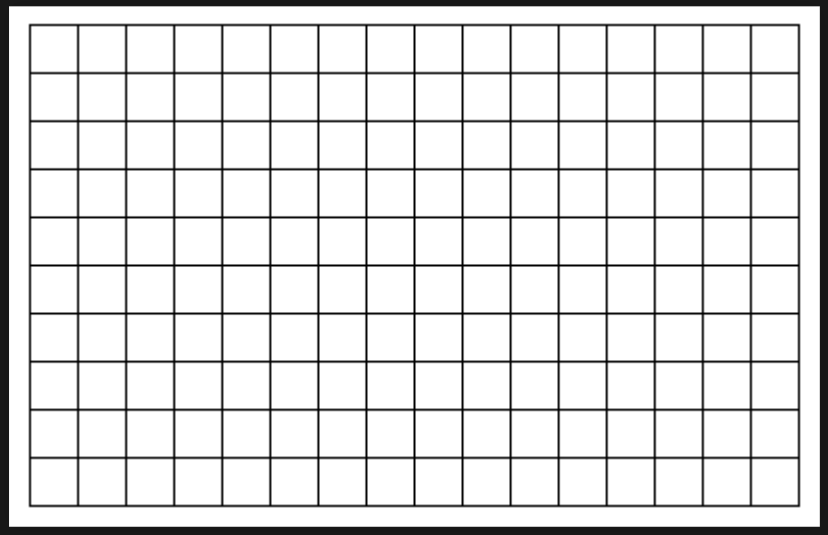
- System.gc()
- Runtime.getRuntime().gc()

GARBAGE COLLECTOR

¿Que recolecta el GC?

- Solo instancias de clases son sujetas al GC, no primitivos.
- La JVM primero verifica si realmente hay un exceso de memoria antes de ejecutar su recolector de basura

GARBAGE COLLECTOR



GARBAGE COLLECTOR

Un objeto se convierte en elegible para lel GC cuando es inalcanzable

- Cuando explicitamente el objeto es seteado null
- Cuando la referencia que indicaba a este objeto, apunta hacia cualquier otro

GARBAGE COLLECTOR

```
class GarbageTruck {

  public static void main(String[] args) {

    String s = new String("hello");
    System.out.println(s);

    s = null;

  }

}
```

GARBAGE COLLECTOR

```
class GarbageTruck {

  public static void main(String[] args) {

    StringBuffer s1 = new StringBuffer("hello");
    StringBuffer s2 = new StringBuffer("goodbye");

    System.out.println(s1);

    s1 = s2;

  }

}
```

GARBAGE COLLECTOR

```
// Lo objetos que son creados dentro de un método,
// también son considerados por el gc

class GarbageTruck {

  public void doSomeThingCool() {

    Date date = new Date();
    StringBuffer now = new StringBuffer(date.toString());

    System.out.println(now);

  }

}
```

GARBAGE COLLECTOR

```
// Pero si un objeto creado dentro del método es retornado,
// entonces esta referencia es asignada en una variable,
// por lo tanto este objeto no es elegible por el gc

class GarbageTruck {

  public Date doSomeThingCool() {

    Date date = new Date();
    StringBuffer now = new StringBuffer(date.toString());

    return date
  }

}
```

FINALIZE

FINALIZE

Es un metodo de la clase Object que se mada ejecutar justo cuando el GC decide que ya no existen referencias hacia el objeto

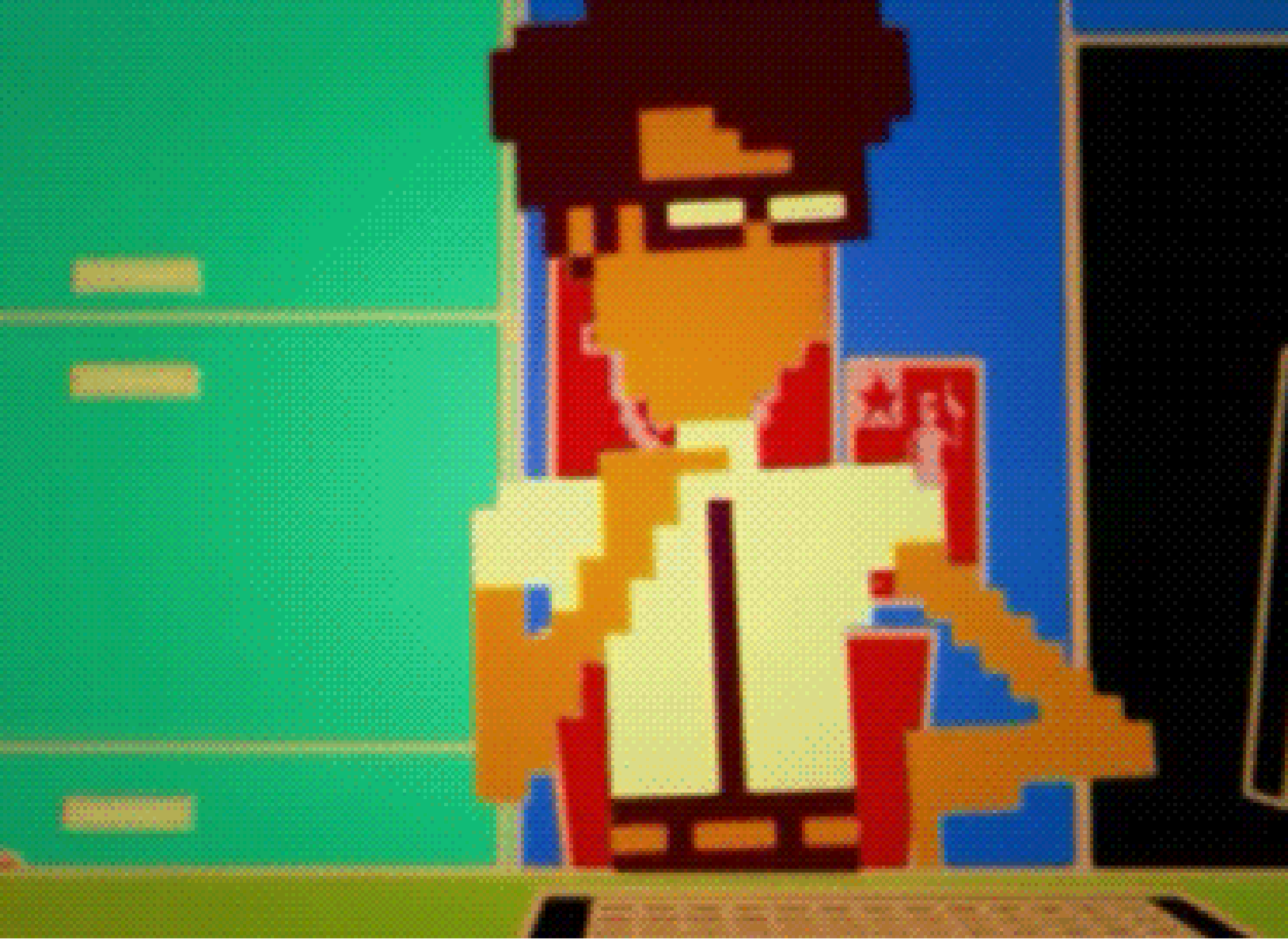
FINALIZE

```
class MyClass {

  int i = 50;

  protected void finalize() throws Throwable {
    System.out.println("From Finalize Method");
  }

}
```

EJERCICIOS

- Cuenta Bancaria
- Control de Peso
- Contraseñas
- Electrodomésticos


5 MINUTES TO GO...



CONSTANTE

```
class MyClass {

        int x = 0;

        int y = 1;


}
```

CONSTANTE

```
class MyClass {

        final int MALE = 0;
        final int FEMALE = 1;

        int gender = MALE;

        if (gender == MALE){

        }

}
```