Paradigmas de la Programación
Práctico 6: Programación Orientada a Objetos (2)
privacidad, herencia

Laura Alonso Alemany Ezequiel Orbe

22 de mayo de 2015

Seleccione la(s) respuesta(s) correcta(s)

1. La diferencia entre interfaces de Java y clases abstractas de C++ es que...
   a) las clases abstractas de C++ permiten herencia múltiple.
   b) las interfaces de Java no pueden heredar de ninguna clase.
   c) las interfaces de Java permiten herencia múltiple.
   d) las clases abstractas de C++ no pueden ser instanciadas.
2. En programación orientada a objetos, la interfaz es la parte de un objeto...
   a) que se ve en la pantalla.
   b) que se comunica con el resto del programa.
   c) que puede ser heredada por más de una clase.
3. Tenemos un *name clash* cuando...
   a) una clase hereda dos miembros con el mismo nombre de dos de sus madres.
   b) las signaturas de dos clases cualquiera tienen un miembro en común.
4. Se dice que en Java no hay funciones porque...
   a) no es un lenguaje funcional.
   b) no se pueden encontrar funciones fuera de los objetos.
   c) se llaman métodos.
5. La diferencia entre `extends` e `implements` en Java consiste en que...
   a) `extends` crea relaciones de herencia, mientras que `implements` crea relaciones de implementación.
   b) `extends` se aplica para heredar de clases, mientras que `implements` se aplica para heredar de interfaces.
   c) `extends` sólo permite herencia simple, mientras que `implements` permite herencia múltiple.

Ejercicios prácticos

1. Identifique en el siguiente código en C++ un problema con la herencia del miembro `meow`.

```
class Felino {
public:
 void meow() = 0;
};

class Gato : public Felino {
public:
 void meow() { std::cout << "miau\n"; }
};

class Tigre : public Felino {
public:
 void meow() { std::cout << "ROARRRRRR\n"; }
};

class Ocelote : public Felino {
public:
 void meow() { std::cout << "roarrrrr\n"; }
};
```

2. A partir del siguiente código Ruby hemos tratado de escribir un código Java con la misma semántica, pero los resultados no son iguales. Cuál es la diferencia y por qué? Cómo deberíamos modificar el programa en Java para que haga lo mismo que el programa en Ruby?

```
#!/usr/bin/ruby

class Being

    @@count = 0

    def initialize
        @@count += 1
        puts "creamos un ser"
    end

    def show_count
        "Hay #{@@count} seres"
    end

end

class Human < Being

   def initialize
       super
       puts "creamos un humano"
   end
end

class Animal < Being
```

```
    def initialize
        super
        puts "creamos un animal"
    end
end

class Dog < Animal

    def initialize
        super
        puts "creamos un perro"
    end
end

Human.new
d = Dog.new
puts d.show_count
```

```
class Being {

    private int count = 0;

    public Being() {
        count++;
        System.out.println("creamos un ser");
    }

    public void getCount() {
        System.out.format("hay %d seres%n", count);
    }
}
class Human extends Being {

    public Human() {
        System.out.println("creamos un humano");
    }
}
class Animal extends Being {

    public Animal() {
        System.out.println("creamos un animal");
    }
}
class Dog extends Animal {

    public Dog() {
        System.out.println("creamos un perro");
    }
}

public class Inheritance2 {

    @SuppressWarnings("ResultOfObjectAllocationIgnored")
    public static void main(String[] args) {
```

```
        new Human();
        Dog dog = new Dog();
        dog.getCount();
    }
}
```

3. A partir de la siguiente `template` en C++, escriba una clase de C++ con la misma semántica pero específica para `int`.

```
template <class A_Type> class calc
{
  public:
    A_Type multiply(A_Type x, A_Type y);
    A_Type add(A_Type x, A_Type y);
};
template <class A_Type> A_Type calc<A_Type>::multiply(A_Type
   x,A_Type y)
{
  return x*y;
}
template <class A_Type> A_Type calc<A_Type>::add(A_Type x,
   A_Type y)
{
  return x+y;
}
```

4. Los `mixins` son una construcción de Ruby que permite incorporar algunas de las funcionalidades de la herencia múltiple, ya que Ruby es un lenguaje con herencia simple. Con un `mixin` se pueden incluir en una clase miembros de otra clase, con la palabra clave `include`. Los *name clashes*, si los hay, se resuelven por el orden de los `include`, de forma que la última clase añadida prevalece, y sus definiciones son las que se imponen en el caso de conflicto. Teniendo esto en cuenta, describa el comportamiento del siguiente pedazo de código.

```
module EmailReporter
    def send_report
      # Send an email
    end
  end

  module PDFReporter
    def send_report
      # Write a PDF file
    end
  end

  class Person
  end

  class Employee < Person
    include EmailReporter
    include PDFReporter
  end

  class Vehicle
```

```
  end

  class Car < Vehicle
    include PDFReporter
    include EmailReporter
  end
```