

Sesión 06: Juegos

Hoja de problemas


Programación 2

Ángel Herranz

aherranz@fi.upm.es

Universidad Politécnica de Madrid

Febrero 2019

-  **Ejercicio 1.** En clase hemos llegado a la conclusión que, de momento, basta con disponer de dos atributos para representar el palo y el valor de la carta. Algo como

```
public class Naipe {  
    private String palo;  
    private String valor;  
    ...  
}
```

Usar atributos de la clase `String` no es un mal punto de inicio especialmente si no conocemos la existencia `enum`.

Hemos realizado algunos supuestos sobre el API que queremos que la clase exponga (API expuesto = métodos públicos que permiten crear, modificar y observar objetos de dicha clase):

- **No** queremos el constructor por defecto.
- Queremos un constructor con dos argumentos que sólo admita palos y valores válidos en la construcción (isi no, que se rompa todo mi programa!).
- No ha habido más orientación pero estaría bien disponer de observadores para saber el palo y el valor de la carta.
- Y cómo no, un método para “dibujar” una carta.

El resultado final debería permitirnos tener un programa principal que haga cosas como estas:

```
class Texas {  
    public static void main(String[] args) {  
        // Declaro una variable array de naipes  
        Naipe[] cartas;  
  
        // Creo un array que va a contener 5 naipes
```

```

cartas = new Naipe[5];

// Creo "la mejor mano" del poquer
cartas[0] = new Naipe("picas", "as");
cartas[1] = new Naipe("picas", "rey");
cartas[2] = new Naipe("picas", "dama");
cartas[3] = new Naipe("picas", "valet");
cartas[4] = new Naipe("picas", "diez");
for (int i = 0; i < cartas.length; i++) {
    cartas[i].imprimirPalo();
}
}
}

```

Y que me impide hacer cosas como esta (no compila):

```

class Texas {
    public static void main(String[] args) {
        Naipe n = new Naipe();
        n.imprimirPalo();
    }
}

```

O como esta (se rompe en tiempo de ejecución):

```

class Texas {
    public static void main(String[] args) {
        Naipe n = new Naipe("oros", "as");
        n.imprimirPalo();
    }
}

```

Puedes ver la primera versión ideada entre “todos” durante la sesión:

```

class Naïpe {
    private String palo;
    private String valor;

    /**
     * No quiero que nadie use new Naïpe();
     */
    private Naïpe() {
    }

    /**
     * Comproueba el palo y el valor antes de crear
     * la instancia.
     */
    public Naïpe(String palo, String valor) {
        int i;
        String[] palosValidos =
            {"corazones", "picas", "treboles", "diamantes"};
        String[] valoresValidos =
            {"as", "dos", "tres", "cuatro", "cinco",
            "seis", "siete", "ocho", "nueve", "diez",
            "valet", "dama", "rey"};
        i = 0;
        while (i < palosValidos.length &&
            palosValidos[i].equals(palo) ) {
            i++;
        }
        if (i == palosValidos.length) {
            System.err.println("Palo no válido: " + palo);
            System.exit(-1);
        }
        i = 0;
        while (i < valoresValidos.length &&
            valoresValidos[i].equals(valor) ) {
            i++;
        }
        if (i == valoresValidos.length) {
            System.err.println("Valor no válido: " + valor);
            System.exit(-1);
        }
        this.palo = palo;
        this.valor = valor;
    }

    /**
     * Imprime el palo.
     */
    void imprimePalo() {
        System.out.println(palo);
    }
}

```

- 📄 **Ejercicio 2.** En clase hemos aprendido que podemos crear unas “cosas” que Herranz ha llamado “enumerados”. Los enum son clases que definen una enumeración de objetos, ni más, ni menos. Así por ejemplo podemos hacer:

```
public enum Palo {  
    TREBOLES, DIAMANTES, CORAZONES, PICAS;  
}
```

Y de repente, podemos hacer esto en un programar principal:

```
class Texas {  
    public static void main(String[] args) {  
        Palo p;  
        p = Palo.PICAS;  
        System.out.println(p);  
    }  
}
```

El objetivo es cambiar por completo la clase Naipe para hacer uso de los enumerados Palo y Valor (todavía por definir). Buscamos poder escribir este programa principal:

```
class Texas {  
    public static void main(String[] args) {  
        // Declaro una variable array de naipes  
        Naipe[] cartas;  
  
        // Creo un array que va a contener 5 naipes  
        cartas = new Naipe[5];  
  
        // Creo "la mejor mano" del poquer  
        cartas[0] = new Naipe(Palo.PICAS, Valor.AS);  
        cartas[1] = new Naipe(Palo.PICAS, Valor.REY);  
        cartas[2] = new Naipe(Palo.PICAS, Valor.DAMA);  
        cartas[3] = new Naipe(Palo.PICAS, Valor.VALET);  
        cartas[4] = new Naipe(Palo.PICAS, Valor.DIEZ);  
        for (int i = 0; i < cartas.length; i++) {  
            cartas[i].imprimirPalo();  
        }  
    }  
}
```

Debería quedarte algo parecido a esto:

```

        }
    }
    void imprimirPalo() {
        System.out.println(palo);
    }
    /**
     * Imprime el palo.
     */
}

public Naipe(Palo palo, Valor valor) {
    this.palo = palo;
    this.valor = valor;
}

/**
 * Construye un nuevo naipe con valores enumerados.
 */
private Naipe() {
    //
    * No quiero que nadie use new Naipe();
    //
    private Palo palo;
    private Valor valor;
}

```

📖 **Ejercicio 3.** Nos gustaría conservar las dos versiones de los constructores: la de strings y la de enumerados. ¿Cómo debemos modificar la implementación del constructor con strings para que mi clase siga funcionando con atributos internos enumerados (Palo y Valor)?

La idea es poder crear instancias de Naipe con cualquiera de las dos versiones del constructor. Así

```
Naipe n = new Naipe(Palo.CORAZONES, Valor.AS);
```

o así:

```
Naipe n = new Naipe("corazones", "as");
```

Y que el objeto construido sea idéntico.

Para ello, los enumerados de Java nos hacen un regalo muy interesante:

- Nos regalan un método `ordinal()` que podemos invocar sobre cualquier enumerado y nos devuelve la posición que ocupa en la enumeración.
- Nos regala una *función* `values()`

Veamos un ejemplo ilustrativo de uso:

```

class Texas {
    public static void main(String[] args) {

```

```

Palo p;
int i;
p = Palo.PICAS;

// Uso de ordinal() (devuelve un entero)
i = p.ordinal();
System.out.println(i);
System.out.println(Palo.TREBOLES.ordinal());
System.out.println(Palo.DIAMANTES.ordinal());
System.out.println(Palo.CORAZONES.ordinal());

// Uso de values() (devuelve un array de instancias de tipo Palo)
Palo[] palos;
palos = Palo.values();
System.out.println(palos[0]);
}
}

```

📄 **Ejercicio 4.** Pues ya solo nos queda terminar de implementar todas las operaciones que creemos que vamos a utilizar. Nuestro API para Naïpe:

- Observador palo(): devuelve el palo de la carta (del tipo Palo, no String).
- Observador valor(): devuelve el valor de la carta (del tipo Valor, no String).
- Observador toString(): devuelve un String que pinte la carta “bonita”. Algo como esto:

```
/**  
 * Devuelve un String para pintar la carta "bonita".  
 */  
public String toString() {  
    String bonita =  
        "+" + "-" + "\n"  
        + "|      | \n" +  
        "+      + \n";  
    return bonita;  
}  
  
/**  
 * Método privado que devuelve un string que representa el palo en bonito.  
 */  
private String palobonito() {  
    String bonito;  
    switch (palo) {  
case TREBLES:  
return "♥";  
case DIAMANTES:  
return "♦";  
case CORAZONES:  
return "♡";  
case PICAS:  
return "♠";  
default:  
return "?";  
}  
}  
  
/**  
 * Método privado que devuelve un string que representa el valor en bonito.  
 */  
private String valorbonito() {  
    switch (valor) {  
case AS:  
return "A";  
case REY:  
return "K";  
case DAMA:  
return "Q";  
case VALET:  
return "J";  
case DIEZ:  
return "10";  
default:  
return (valor.ordinal() + 1) + " ";  
}  
}
```

- ☐ **Ejercicio 5.** Continuando con la modelización del juego de cartas *Texas hold'm*, implementa una nueva clase *Mano* que represente las dos cartas de un jugador.
- ☐ **Ejercicio 6.** Ahora te toca implementar una nueva clase *Mano* que represente las dos cartas de un jugador.
- ☐ **Ejercicio 7.** Si no se te ha ocurrido, puedes hacer que el constructor reparta cartas aleatoriamente.
- ☐ **Ejercicio 8.** Implementa una nueva clase *Comunitarias* que represente las cinco cartas comunitarias.
- ☐ **Ejercicio 9.** ¿Has representado las cuatro fases?

Flop tres cartas descubiertas.

Turn cuatro cartas descubiertas.

River las cinco cartas descubiertas.

- ☐ **Ejercicio 10.** Escribe una *función* *mejorJugada* en el programar principal que reciba unas cartas comunitarias y dos manos y decida qué mano es la ganadora. Úsala en un programa principal. Algo como esto:

```
public class Compara {
    /**
     * Devuelve 1 si m1 es ganadora, -1 si m2 es ganadora y 0 si son iguales
     */
    private static int mejorJugada(Comunitarias c,
                                    Mano m1,
                                    Mano m2) {
        ...
    }

    public static void main(String args) {
        Mano m1 = new Mano();
        Mano m2 = new Mano();
        Comunitarias c = new Comunitarias();
        switch (mejorJugada(c, m1, m2) {
            case 0:
                System.out(m1 + " es igual jugada que " + m2);
                break;
            case 1:
                System.out(m1 + " es mejor jugada que " + m2);
                break;
            case -1:
                System.out(m1 + " es peor jugada que " + m2);
                break;
        }
    }
}
```


}

}

}