

Sesión 06: Modelizando Póquer

Hoja de problemas

Programación 2

Ángel Herranz

aherranz@fi.upm.es

Universidad Politécnica de Madrid

2022-2023

-  **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 del naipes. Algo como

```
public class Naipes {  
    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`. Otras alternativas son **int** para `valor` o incluso **char** para `palo`.

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 del naipes.
- Y cómo no, un método para “dibujar” un naipes (¿qué mejor candidato que `toString()`?).
- Seguro que se te ocurren cosas un poco locas, como por ejemplo un método que “imprima” directamente el palo de un naipes: `imprimirPalo()`.

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
```

```

Naipes[] cartas;

// Creo un array que va a contener 5 naipes
cartas = new Naipes[5];

// Creo "la mejor mano" del poquer
cartas[0] = new Naipes("picas", "as");
cartas[1] = new Naipes("picas", "rey");
cartas[2] = new Naipes("picas", "dama");
cartas[3] = new Naipes("picas", "valet");
cartas[4] = new Naipes("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) {
        Naipes n = new Naipes();
        n.imprimirPalo();
    }
}

```

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

```

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

```

Puedes ver la primera versión ideada entre “todos” durante una sesión en el curso 2018-2019.

```

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 en 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 Naipes:

- Observador `palo()`: devuelve el palo del naipе (del tipo `Palo`, no `String`).
- Observador `valor()`: devuelve el valor del naipе (del tipo `Valor`, no `String`).
- Observador `toString()`: devuelve un `String` que pinte el naipе “bonito”. Algo como esto:

```

/**
 * Devuelve un String para pintar el naipe "bonito".
 */
public String toString() {
    String bonita =
        "+" + "-" + "\n" +
        "|\" + valorBonito() + \"|\" + paloBonito() + \"\n\" +
        "+\" + \"\n\" +
        "+\" + \"\n\" +
        "+\" + \"\n\" +
        ";
    return bonita;
}

/**
 * Método privado que devuelve un string que representa el palo en bonito.
 */
private String paloBonito() {
    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.** La propuesta en las transparencias es hacer un programa que:

- reparta dos manos aleatorias de Póquer,
- una para *Player 1*,
- otra para *Player 2*,
- y diga quien es el ganador.

Para ello, parece una buena idea modelizar la clase *baraja*. Su API debería ser algo como esto:

- Un constructor `Baraja()` para crear una baraja.
- Al menos un *observador/modificador* para sacar un naipes aleatoria: `Naipes repartir()`: ese método debería devolver un naipes aleatorio de la baraja de tal forma que cuando se vuelva a ejecutar no devuelva más veces ese naipes.
- Puede que tenga interés implementar un observador `int naipes()` que diga cuantos naipes quedan en la baraja.

Internamente, una forma de representar una baraja, puede ser un array de longitud 52 de naipes. La estrategia, muy probablemente planificada en clase: llenar el array en el constructor creando todas los naipes de la baraja, y en `repartir()` generar un número aleatorio entre 0 y 51 para sacar un naipes.

📄 **Ejercicio 6.** Herranz dijo en clase que no le gustaban los métodos *observadores/modificadores*. ¿Qué alternativa hay a `repartir()`?

Es posible hacer un método observador que te diga la carta que hay arriba en la baraja, digamos `Naipes siguiente()`, y un método modificador que la quite, digamos `quitarSiguiente()`.

📄 **Ejercicio 7.** El método `repartir()` (o `siguiente()`/`quitarSiguiente()`) tiene una dificultad que habrá que manejar:

¿cómo vamos a evitar repartir dos veces el mismo naipes?

📄 **Ejercicio 8.** Ya sólo nos queda el empujón final: sacar 5 naipes para *player 1*, 5 naipes para *player 2* y comparar las dos jugadas de acuerdo a las reglas que se pueden ver en Wikipedia: https://en.wikipedia.org/wiki/List_of_poker_hands.

Quizás una buena idea puede ser crear la clase *Mano*. Básicamente es una agrupación de 5 naipes que podrían pasarse en un constructor junto con un método observador en el que se implementa la *lógica de la comparación* de manos, pongamos que lo llamamos `boolean mejorQue(Mano m)`.