Samsung
TECH INSTITUTE

MÓDULO 2. PROGRAMACIÓN ORIENTADA A OBJETOS CON JAVA

Relación de Problemas Nº 4

Proyecto prTest (arrays, scanner, equals)

Se va a crear una aplicación para anotar las calificaciones obtenidas por alumnos en exámenes tipos test de una asignatura. Para ello se crearán las clases `Test` y `TestAsignatura` en un paquete denominado `tests`.

1) La clase `Test` mantiene información sobre el examen tipo test de un estudiante, así como el nombre del estudiante (`String`), y el número de aciertos y errores (ambos de tipo `int`) en las respuestas contestadas del test. La clase incluirá:
   a) Un constructor con tres argumentos que proporcionan nombre del estudiante, aciertos y errores en el examen, respectivamente.
   b) Métodos para obtener el nombre del estudiante, el número de aciertos y el número de errores.
   c) Dos tests se consideran iguales cuando corresponden al mismo estudiante, independientemente de que el nombre incluya mayúsculas o minúsculas.
   d) La representación textual de un test vendrá dada por el nombre del estudiante (todo en mayúsculas), seguido del número de aciertos y errores, separados por comas y encerrados entre corchetes. Por ejemplo,
      ```
      NOMBRE DEL ALUMNO [aciertos,errores]
      ```
   e) El método `public double calificación(double valAc, double valErr)`, devolverá la calificación del test, considerando el número de aciertos por la valoración de cada uno de ellos más el número de errores por la valoración de cada uno. La valoración de cada acierto y error se proporcionan como parámetros. Se supone que la valoración del acierto debe ser estrictamente mayor que 0, mientras que la valoración del error deber negativo o 0. De no ser así, se lanzará una excepción de tipo `RuntimeException`.

2) La clase `TestAsignatura` deberá incluir información sobre el nombre de la asignatura (`String`), y una colección de tests. En este caso, la clase incluirá:
   a) Dos constructores. Uno con cuatro argumentos donde el primero determine el nombre de la asignatura, el segundo la valoración de cada acierto, el tercero la valoración de cada error y el cuarto argumento es un array de cadenas de caracteres donde cada posición incluye información sobre el resultado (aciertos y errores) de un estudiante. El formato de cada posición será:
      ```
      Apellidos, Nombre: aciertos + errores
      ```
   b) El segundo constructor solo incluirá dos argumentos, uno con el nombre de la asignatura y otro con el array de cadenas de caracteres. En este caso, se supondrá que

valoración de aciertos será 1, y la valoración de errores, 0. En ambos casos, a partir del array de cadenas de caracteres habrá que generar el array de objetos `Test`.

c) Un método `public double notaMedia()` para calcular la nota media de todos los exámenes.

d) Un método `public int aprobados()` para devolver el número de aprobados, considerando como tal al examen que supere la calificación de 5.

A continuación se presenta un ejemplo de uso de estas clases y la salida correspondiente:

```
import prTest.TestAsignatura;

public class PruebaTest {
	static String[] examenes = {
			"El hombre enmascarado:6+4",
			"La mujer Maravilla:9+1",
			"El hombre invisible:4+6",
			"Dr. Jekyll y Mr. Hide:5+5",
			"El increible hombre menguante:7+3",
			"El llanero solitario:7+3",
			"La chica invisible:8+2",
			"La princesa guerrera:2+8" };

	public static void main(String[] args) {
		TestAsignatura poo = new TestAsignatura(
				"Programación Orientada a Objetos", examenes);

		System.out.println("Asignatura: " + poo.nombre());
		System.out.println("Aprobados en el test: " + poo.aprobados());
		System.out.println("Nota media en el test: " + poo.notaMedia());
	}
}
```

La salida al ejecutar el programa anterior es

```
Asignatura: Programación Orientada a Objetos
Aprobados en el test: 6
Nota media en el test: 6.0
```

Proyecto prBancoB (arrays, herencia, excepciones)

Modifica el ejercicio del proyecto `prBanco` de la siguiente forma:

1. Añade a la clase `Cuenta` el método publico `void cierreEjercicio()` que incremente el saldo de la cuenta en un 2 por ciento por los intereses del ejercicio (define una constante de clase para almacenar ese porcentaje, 0.02).
2. Crea la clase `CuentaVip` que se comporta igual que la clase `Cuenta` (es decir que extiende la clase `Cuenta`), pero que además:
    a. Dispone de un atributo privado de tipo `double` denominado `incentivo`, que determinará una cantidad que el banco ingresará en esa cuenta al cierre del ejercicio por tratarse de una cuenta vip.
    b. Dispone de un constructor en el que además del titular, el saldo y el número de cuenta se proporciona el valor del incentivo, `CuentaVip(String tit, double s, int n, incentivo double)`
    c. Dispone de un método público `void cierreEjercicio()`. El cierre de ejercicio de una cuenta vip es como el de un objeto de la clase `Cuenta`, pero tras incrementar el saldo con los intereses se ingresa o añade en la cuenta el valor de `incentivo`.
    d. Redefine el método público `String toString()` para que además de mostrar la información como un objeto de la clase `Cuenta`, muestre el incentivo entre símbolos '$' (mira el resultado en el ejemplo proporcionado).
3. Añade a la clase `Banco` el método público `public int abrirCuentaVip(String tit, double saldoInicial, double incentivo)`, que crea una `cuentaVip` con titular `tit`, saldo `saldoInicial` e incentivo `incentivo`, la mete en el array `ctas` y devuelve el número de cuenta asignada. Su comportamiento es idéntico al método `abrirCuenta` (esto es también duplica el tamaño del array en caso de que este se llene).
4. Añade a la clase `Banco` el método público `void cierreEjercicio()` que cierre el ejercicio de todas sus cuentas.
5. Añade a la clase `Banco` el método público `int [] abrirCuentas(String [] titulares)` que añada una cuenta de la clase `Cuenta` con saldo 0 por cada titular que aparece en el array `titulares`. Este métodos debe devolver un array con los números de cuenta de todas las cuentas que se han creado.
6. Añade a la clase `Banco` el método público `void cerrarCuentas(int [] numCuentas)` que elimine todas los objetos de la clase `Cuenta` cuyos números de cuenta aparecen en el array `numCuentas`.
7. Añade a la práctica la excepción `BancoException` no comprobada. Sustituye todas las llamadas a `RuntimeException` por llamadas a `BancoException`.
8. Modifica el método `cerrarCuentas` para que si una cuenta de las que se pasa en el array argumento no existe en el banco, la ignore y siga con el resto.

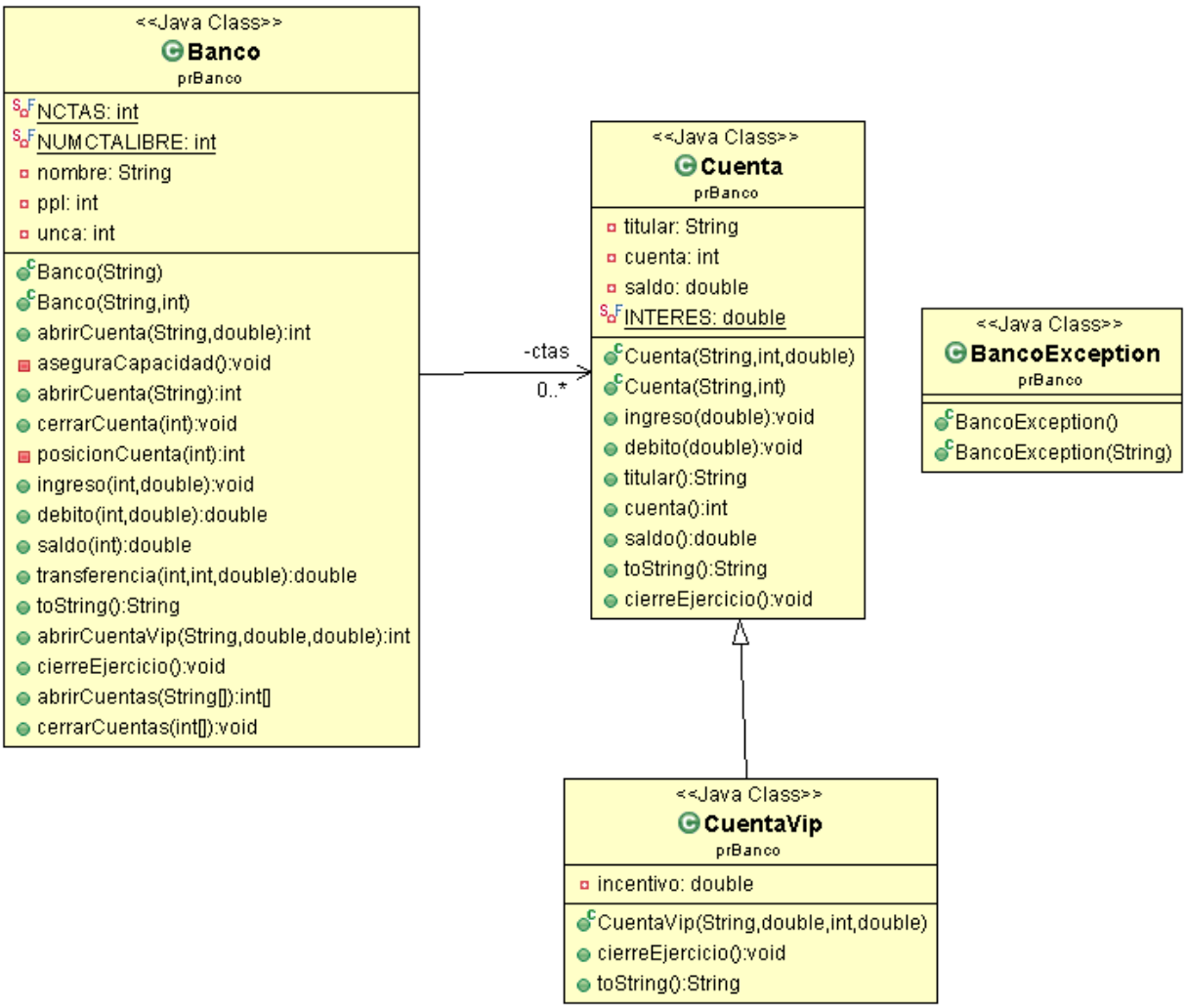
Utiliza el siguiente programa de prueba para verificar tu implementación (ponlo en el paquete por defecto):

```
import prBancoB.Banco;

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

		Banco b = new Banco("TubbiesBank", 5);
		int nPo = b.abrirCuentaVip("Po", 500, 300);
		int nDixy = b.abrirCuenta("Dixy", 500);
		int nTinkyWinky = b.abrirCuentaVip("Tinky Winky", 500,100);
		int nLala = b.abrirCuenta("Lala", 500);
		System.out.println(b);
		b.ingreso(nPo, 100);
		b.debito(nDixy, 100);
		b.transferencia(nTinkyWinky, nLala, 100);
```

```
                System.out.println(b);
                b.cerrarCuenta(nPo);
                b.ingreso(nDixy,200);
                System.out.println(b);

                /// Abre tres cuentas nuevas con los titulares proporcionados
                /// en el array nombres

                String [] nombres={"Dora", "Botas", "Pedro"};
                int [] cuentas = b.abrirCuentas(nombres);
                System.out.println (b);
                System.out.println(java.util.Arrays.toString(cuentas));

                /// Cierra las tres cuentas abiertas

                b.cerrarCuentas(cuentas);
                System.out.println (b);

                /// Hace un cierre del ejercicio de las cuentas que hay en
                /// en el banco b

                b.cierreEjercicio();
                System.out.println (b);


                int [] nCuentas = new int[2];
                nCuentas[0] = nDixy;
                nCuentas[1] = cuentas[0];
                b.cerrarCuentas(nCuentas);
                System.out.println (b);

        }
}
```

El resultado debe ser el siguiente:

```
TubbiesBank: [[(Po/1001) -> 500.0]$300.0$ [(Dixy/1002) -> 500.0] [(Tinky Winky/1003) -> 500.0]$100.0$
[(Lala/1004) -> 500.0] ]
[[(Po/1001) -> 600.0]$300.0$ [(Dixy/1002) -> 400.0] [(Tinky Winky/1003) -> 400.0]$100.0$ [(Lala/1004) ->
600.0] ]
[[(Dixy/1002) -> 600.0] [(Tinky Winky/1003) -> 400.0]$100.0$ [(Lala/1004) -> 600.0] ]
[[(Dixy/1002) -> 600.0] [(Tinky Winky/1003) -> 400.0]$100.0$ [(Lala/1004) -> 600.0] [(Dora/1005) -> 0.0]
[(Botas/1006) -> 0.0] [(Pedro/1007) -> 0.0] ]
[1005, 1006, 1007]
[[(Dixy/1002) -> 600.0] [(Tinky Winky/1003) -> 400.0]$100.0$ [(Lala/1004) -> 600.0] ]
[[(Dixy/1002) -> 612.0] [(Tinky Winky/1003) -> 508.0]$100.0$ [(Lala/1004) -> 612.0] ]
[[(Tinky Winky/1003) -> 508.0]$100.0$ [(Lala/1004) -> 612.0] ]
```


<<Java Class>>
Banco
prBanco
NCTAS: int
NUMCTALIBRE: int
nombre: String
ppl: int
unca: int
Banco(String)
Banco(String,int)
abrirCuenta(String,double):int
aseguraCapacidad():void
abrirCuenta(String):int
cerrarCuenta(int):void
posicionCuenta(int):int
ingreso(int,double):void
debito(int,double):double
saldo(int):double
transferencia(int,int,double):double
toString():String
abrirCuentaVip(String,double,double):int
cierreEjercicio():void
abrirCuentas(String[]):int[]
cerrarCuentas(int[]):void
-ctas
0..*
<<Java Class>>
Cuenta
prBanco
titular: String
cuenta: int
saldo: double
INTERES: double
Cuenta(String,int,double)
Cuenta(String,int)
ingreso(double):void
debito(double):void
titular():String
cuenta():int
saldo():double
toString():String
cierreEjercicio():void
<<Java Class>>
BancoException
prBanco
BancoException()
BancoException(String)
<<Java Class>>
CuentaVip
prBanco
incentivo: double
CuentaVip(String,double,int,double)
cierreEjercicio():void
toString():String

Proyecto prMasterMind (herencia, scanner, equals, excepciones)

Se desea implementar un juego (`Mastermind`) en el que el usuario ha de adivinar una combinación secreta de cifras no repetidas creada aleatoriamente por el programa. El usuario tendrá la posibilidad de hacer varios intentos para adivinar la combinación secreta, respondiendo el programa a cada uno de estos intentos con el número de cifras acertadas, indicando cuántas de éstas están en la posición correcta y cuántas aparecen en la combinación secreta aunque en una posición diferente. Por ejemplo, si la combinación secreta es 4235 y el jugador intenta 4350 el programa responderá con que el número de cifras colocadas es 1 (la cifra 4) y el de descolocadas es 2 (las cifras 3 y 5). Para ello:

1) Cread la clase `Movimiento` cuyas instancias mantengan información sobre un intento (como un `String`) y dos números enteros, los cuales indicarán el número de cifras bien colocadas y descolocadas. Se pide:

- Un constructor que crea una instancia dada una combinación secreta (`String`) y los enteros que debe mantener.
- Métodos `int colocadas()`, `int descolocadas()` y `String cifras()` que permiten conocer cada uno de los componentes de su estado.
- Redefine el método `public boolean equals(Object o)` de modo que dos movimientos sean iguales si lo son las cifras que contienen. (Modifica también el método `public int hashCode()`).
- Redefine el método `toString()` de forma que un movimiento se represente con el formato:

```
[cifras, cifras_colocadas, cifras_descolocadas]
```

Ejemplo: `[4350, 1, 2]`

2) Definid la clase `MasterMindException` que en lo sucesivo se utilizará para crear excepciones en caso de que se produzca algún tipo de error. Esta excepción será de no obligado tratamiento.

3) Cread la clase `MasterMind`, la cual contendrá una combinación de cifras secreta (un `String`), de manera que cualquier usuario de esta clase puede intentar averiguar dicha combinación. Ninguna combinación de cifras podrá contener cifras repetidas. Se pide:

- Un constructor que crea una instancia del mastermind dado el tamaño de las combinaciones de cifras. Habrá un segundo constructor sin argumentos que creará una instancia del juego con combinaciones de un tamaño por defecto (en este caso, 4). La combinación secreta se creará en el momento de la creación del mastermind. Un argumento no positivo o mayor que el número de cifras disponibles (10) producirá el lanzamiento de la excepción `MasterMindException`.
- Definid el método `int longitud()` que proporcione la longitud de la combinación en este juego.
- Definid un método `private boolean validaCombinacion(String cifras)` que compruebe si una cadena de caracteres corresponde a un movimiento válido. Una cadena es un movimiento válido si tiene la longitud correcta, es decir, las cifras que exige el juego, todos los caracteres son números y no contiene cifras repetidas.
- Definid un método `public Movimiento intento(String cifras)` tal que dada una cadena de cifras comprueba si es una cadena válida y entonces

devuelva un objeto `Movimiento` que contenga las cifras, el número de cifras que aparecen en la combinación secreta en la misma posición y el número de cifras en posiciones distintas. Si la cadena de cifras no es válida lanzará la excepción `MasterMindException`.

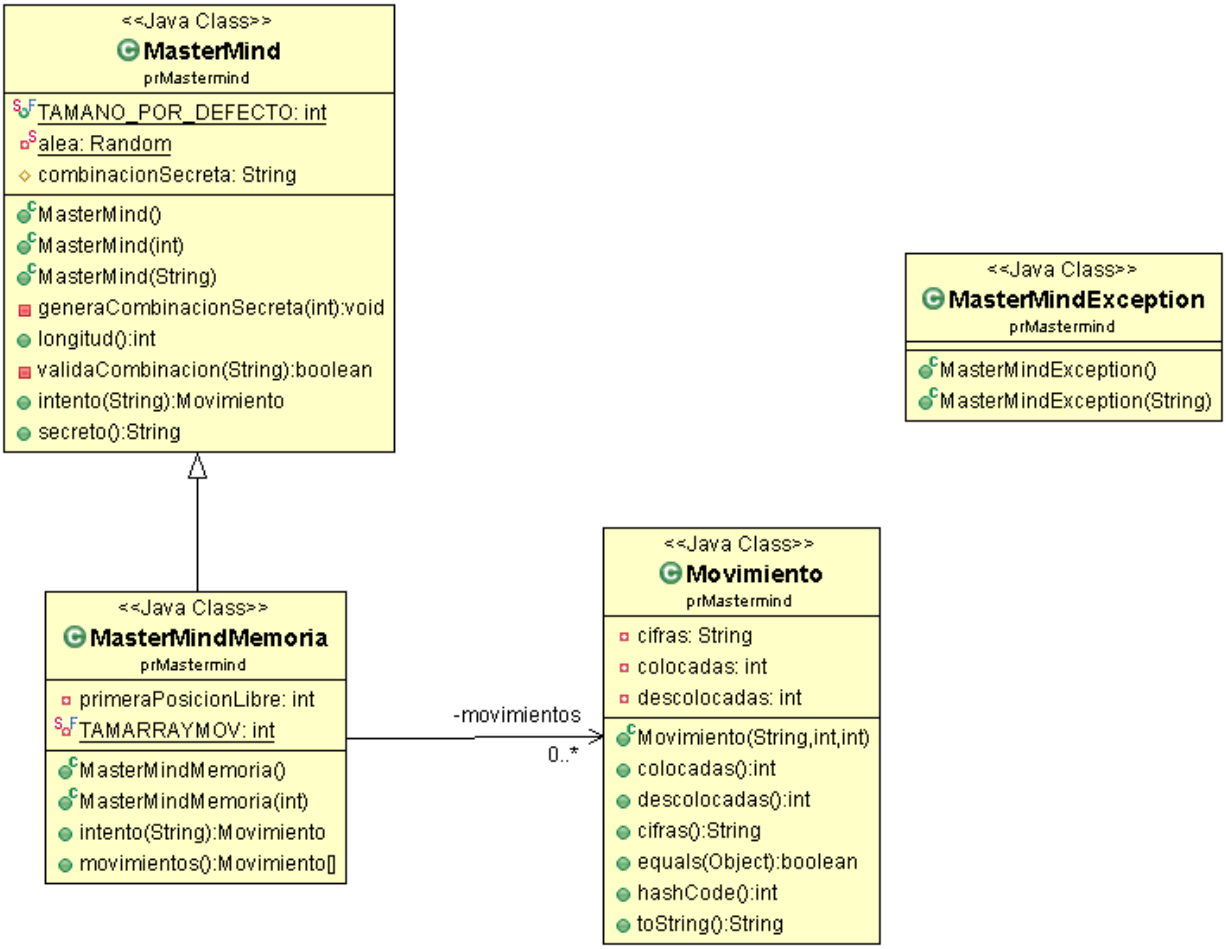
- Definid un método `public String secreto()` que permita conocer la combinación secreta. Devolverá una cadena de caracteres con la combinación secreta.

4) Cread la clase `MasterMindMemoria`, cuyos objetos se comportan igual que los de la clase `MasterMind` excepto que éstos recuerdan todos los movimientos válidos que se han intentado hasta el momento, no permitiendo que el usuario repita combinaciones de cifras. Las combinaciones repetidas provocan el lanzamiento de la excepción `MasterMindException`. Se pide:

- Constructores con argumentos como los de la clase `MasterMind`.
- Redefine el método `Movimiento intento(String cifras)` de forma que tenga el comportamiento esperado.
- Definid un método `Movimiento[] movimientos()` que proporcione un array con los movimientos válidos ya realizados.

Probad las clases con el programa siguiente:

```
import java.util.Scanner;

import prMastermind.MasterMind;
import prMastermind.MasterMindException;
import prMastermind.Movimiento;

public class TestMasterMind {

	public static void main(String[] args) throws MasterMindException {
		MasterMind j = new MasterMind();
	    // System.out.println(j.secreto());
		try (Scanner sc = new Scanner(System.in)) {
			boolean acertado = false;
			String cifras = null;
			int intento = 1;
			while (!acertado) {
				try {
					System.out.print("Intento " + intento
							+ ". Introduce cifra: ");
					cifras = sc.next();
					Movimiento mov = j.intento(cifras);
					System.out.println("Intento " + intento + " " + mov);
					acertado = mov.colocadas() == j.longitud();
					intento++;
				} catch (MasterMindException e) {
					System.out.println(cifras + " no válidas: "
							+ e.getMessage());
				}
			}
			System.out.println("Correcto en " + (intento - 1) + " intentos");
		}
	}
}
```


<<Java Class>>
MasterMind
prMastermind
TAMANO_POR_DEFECTO: int
alea: Random
combinacionSecreta: String
MasterMind()
MasterMind(int)
MasterMind(String)
generaCombinacionSecreta(int):void
longitud():int
validaCombinacion(String):boolean
intento(String):Movimiento
secreto():String
<<Java Class>>
MasterMindException
prMastermind
MasterMindException()
MasterMindException(String)
<<Java Class>>
MasterMindMemoria
prMastermind
primeraPosicionLibre: int
TAMARRAYMOV: int
MasterMindMemoria()
MasterMindMemoria(int)
intento(String):Movimiento
movimientos():Movimiento[]
-movimientos
0..*
<<Java Class>>
Movimiento
prMastermind
cifras: String
colocadas: int
descolocadas: int
Movimiento(String,int,int)
colocadas():int
descolocadas():int
cifras():String
equals(Object):boolean
hashCode():int
toString():String

Proyecto prGenetico (interfaces, herencia, copia)

Los algoritmos genéticos se utilizan para resolver problemas de optimización de forma aproximada cuando no es posible hacerlo mediante algoritmos exactos; están inspirados en la teoría de la evolución y trabajan manipulando un conjunto de soluciones tentativas para un problema. A dichas soluciones se les denomina *individuos* y al conjunto de soluciones *población*. En el caso más simple, que será el que nosotros trataremos, un individuo está formado por una cadena binaria (secuencia de ceros y unos) y tiene asociado un valor real que representa, de algún modo, su aproximación a una solución óptima, conocido como su *aptitud* (*fitness*) para el problema. A cada dígito de la secuencia binaria se le llama *gen* y a la secuencia completa *cromosoma*. La única información que necesitan estos algoritmos sobre el problema que han de resolver es la forma de calcular la aptitud de una solución, lo que se conoce como la función de fitness.

Un algoritmo genético procede del siguiente modo. En primer lugar, todos los individuos de la población son inicializados con secuencias aleatorias de ceros y unos. A continuación, el algoritmo entra en un bucle cuyo cuerpo se repite un número determinado de veces (*generaciones*). Dentro de este bucle, dos individuos (padres) son elegidos aleatoriamente de la población, se *recombinan* sus cromosomas para dar a luz a un nuevo cromosoma que, seguidamente, es *mutado*. Con el cromosoma resultante se forma un nuevo individuo que se compara con el peor individuo de la población y, si su fitness es más alto, lo sustituye; de lo contrario, se desecha.

En los casos en que se presente alguna situación excepcional se deberá lanzar una excepción del tipo `RuntimeException` (o de alguna clase heredera de ella que se adecue mejor a la situación en cuestión).

Se construirán las siguientes clases e interfaces:

1) La clase `Cromosoma` con los métodos que aparecen en el diagrama adjunto, teniendo en cuenta lo siguiente:

Para almacenar los datos de un cromosoma se debe usar un array de enteros, cada una de cuyas posiciones representará un gen. Dicho array debe ser visible en subclases y clases dentro del paquete.

El constructor de la clase `Cromosoma` tomará como parámetros la longitud del mismo (número de genes que contiene) y un valor booleano que indica si debe asignar de forma aleatoria un valor cero o uno a cada gen (caso true), o inicializarlos con el valor por defecto 0 (caso false).

Los métodos con nombre `gen` permitirán consultar y establecer el valor de cada gen del cromosoma, y `longitud()` permitirá consultar el número de genes.

El método `mutar(double)` debe recorrer todos los genes del cromosoma que se le pasa como parámetro invirtiendo su valor (de 0 a 1 ó de 1 a 0) de acuerdo con la probabilidad de mutación que recibe como parámetro, y que deberá aplicar a cada gen por separado.

El método `copia()` deberá construir y devolver una copia del cromosoma.

2) La interfaz `Problema` con un único método, `evalua(Cromosoma)`, que debe devolver el valor de fitness asociado al cromosoma que recibe como argumento en el problema que representa.

3) La clase `Individuo`, formada por un `Cromosoma` y su valor de `fitness`, teniendo en cuenta lo siguiente:

a. Esta clase deberá tener dos constructores: el primero con dos parámetros, la longitud del cromosoma y el problema a resolver; y el segundo, también con dos parámetros, que ahora son el cromosoma que debe asociarse al individuo y el problema. El primero debe crear un cromosoma de forma aleatoria, y el segundo debe generar una copia del cromosoma que recibe como parámetro antes de asociarlo al objeto que está creando. En ambos constructores el objeto de clase `Problema` se usará únicamente para evaluar el cromosoma y asociarle el correspondiente valor de `fitness` (en dicho problema).

b. El método `Cromosoma cromosoma()` debe devolver una copia del objeto de la clase `Cromosoma` asociado al objeto de clase `Individuo`.

c. El valor de `fitness` de un objeto `Individuo` puede consultarse con el método `fitness()`.

4) La clase `OneMax` que implementa la interfaz `Problema` sabiendo que, en este problema, el `fitness` de un individuo viene dado por el número de cifras "1" que contiene su cromosoma.

5) La clase `Poblacion`, teniendo en cuenta que una población está constituida por un número fijo de individuos y se implementará usando un array de `Individuo`; además:

a. El constructor recibirá como parámetros: el tamaño de la población (número de individuos), la longitud de los individuos (número de genes de sus cromosomas) y el problema a resolver (un objeto que implementa la interfaz `Problema`) y, con estos datos, deberá crear una población de individuos generados de forma aleatoria.

b. El método `numIndividuos()` debe devolver el número de individuos de la población (tamaño de la población).

c. El método `individuo(int)` devolverá el *i*-ésimo individuo de la población.

d. El método `mejorIndividuo()` devuelve el individuo con mayor valor de fitness de la población.

e. El método `reemplaza(Individuo)` sustituirá el peor individuo de la población (el de menor valor de fitness) por el individuo que se pasa como parámetro; pero sólo en el caso de que este último sea mejor (mayor fitness).

6) La clase abstracta `AlgoritmoGenetico`, sabiendo que cada algoritmo genético almacenará información necesaria sobre un problema, una población de soluciones tentativas,

la probabilidad de mutación de los genes de los individuos y el número de pasos que debe realizar. Además, debe tenerse en cuenta lo siguiente:

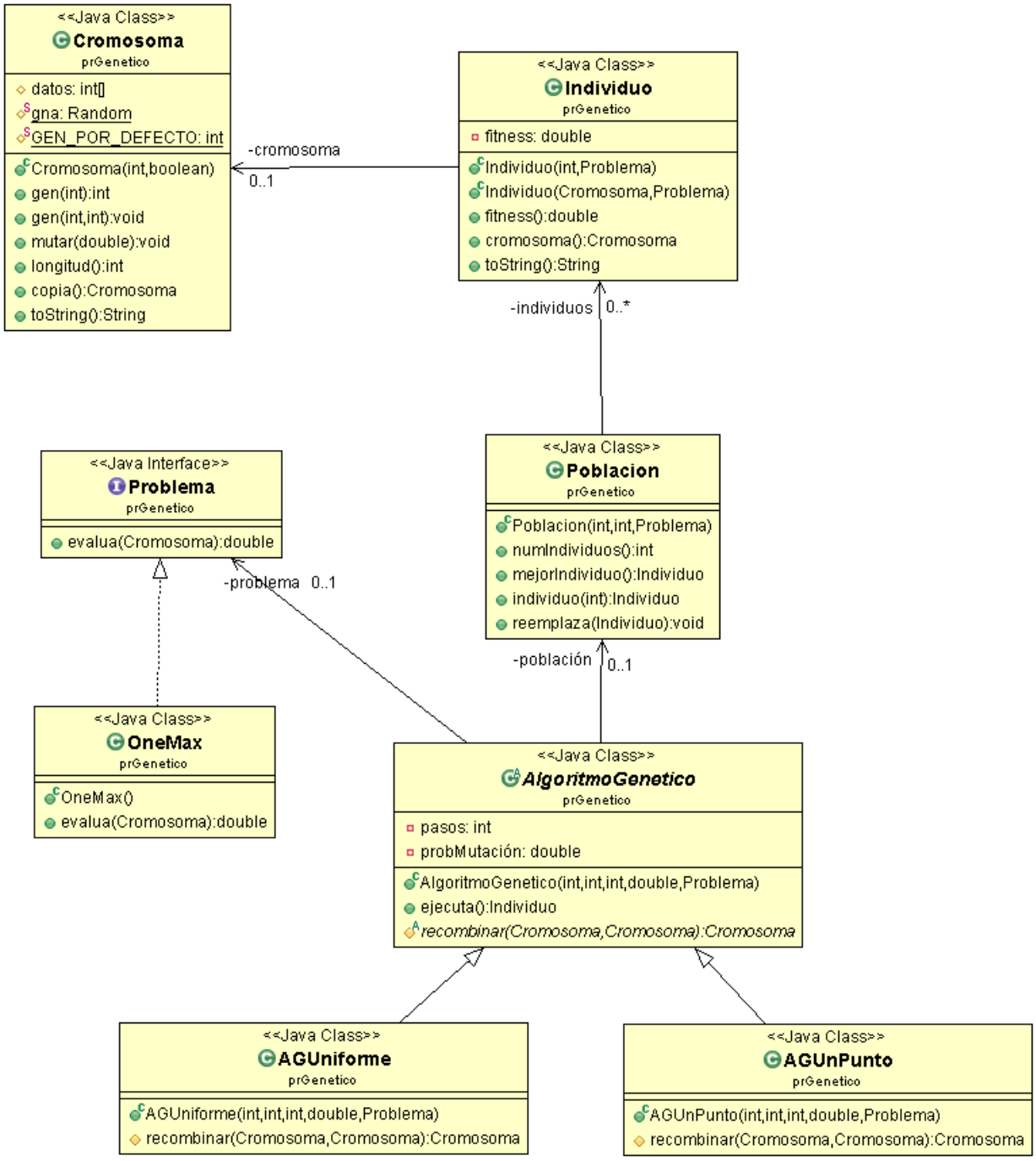
a. El constructor de esta clase debe recibir el tamaño de la población que va a utilizar, la longitud de los individuos de dicha población, el número de pasos del algoritmo (generaciones), la probabilidad de mutar un gen en el cromosoma y el problema que se debe resolver. El constructor deberá crear la población de individuos.

b. El método `ejecuta()` deberá ejecutar, tantas veces como indique el número de pasos, la secuencia siguiente: seleccionar dos individuos de la población aleatoriamente, tomar sus cromosomas y recombinarlos usando el método abstracto `recombinar(Cromosoma, Cromosoma)`, mutar el resultado con la probabilidad indicada y, por último, crear un individuo con el cromosoma resultante que se insertará en la población reemplazando al peor individuo siempre y cuando sea mejor que éste. Finalmente, devolverá el mejor individuo de la población después de la terminación del bucle.

7) Existen muchos operadores de recombinación que pueden utilizarse para recombinar individuos. Dos de los más conocidos son la *recombinación de un punto* y la *recombinación uniforme*. En el primero se genera un número aleatorio *z* comprendido entre cero y la longitud del cromosoma. Los primeros *z* genes del individuo resultante se toman del primer padre y el resto del segundo. En la recombinación uniforme el valor de cada gen del individuo resultante se elige de forma aleatoria de uno de los padres. Dicho lo anterior, se deberá construir las subclases `AGUnPunto` y `AGUniforme` de `AlgoritmoGenetico` de forma que implementen la recombinación de un punto y la recombinación uniforme, respectivamente, en el método `recombinar`. El constructor de dichas clases debe tener la misma signatura que el de la clase `AlgoritmoGenetico`.

Para probar el funcionamiento de los dos tipos de algoritmos genéticos implementados se puede usar la siguiente clase `TestGenetico` que no pertenece al paquete.

```
import prGenetico.AGUnPunto;
import prGenetico.AGUniforme;
import prGenetico.AlgoritmoGenetico;
import prGenetico.Individuo;
import prGenetico.OneMax;
import prGenetico.Problema;

public class TestGenetico {

	static final int TAM_POBLACION = 20;
	static final int PASOS_GA = 400;
	static final int LONG_CROMOSOMA = 50;

	static final double PROB_MUT = 0.02;

	public static void main(String[] args) {
		Problema problema = new OneMax();

		AlgoritmoGenetico ga1 = new AGUnPunto(TAM_POBLACION, LONG_CROMOSOMA,
				PASOS_GA, PROB_MUT, problema);
		Individuo solucion1 = ga1.ejecuta();
		System.out.println("Solución 1:" + solucion1);

		AlgoritmoGenetico ga2 = new AGUniforme(TAM_POBLACION, LONG_CROMOSOMA,
				PASOS_GA, PROB_MUT, problema);
		Individuo solucion2 = ga2.ejecuta();
		System.out.println("Solución 2:" + solucion2);
	}
}
```


<<Java Class>>
Cromosoma
prGenetico
datos: int[]
gna: Random
GEN_POR_DEFECTO: int
Cromosoma(int,boolean)
gen(int):int
gen(int,int):void
mutar(double):void
longitud():int
copia():Cromosoma
toString():String
-cromosoma
0..1
<<Java Class>>
Individuo
prGenetico
fitness: double
Individuo(int,Problema)
Individuo(Cromosoma,Problema)
fitness():double
cromosoma():Cromosoma
toString():String
-individuos
0..*
<<Java Interface>>
Problema
prGenetico
evalua(Cromosoma):double
-problema
0..1
<<Java Class>>
Poblacion
prGenetico
Poblacion(int,int,Problema)
numIndividuos():int
mejorIndividuo():Individuo
individuo(int):Individuo
reemplaza(Individuo):void
-población
0..1
<<Java Class>>
OneMax
prGenetico
OneMax()
evalua(Cromosoma):double
<<Java Class>>
AlgoritmoGenetico
prGenetico
pasos: int
probMutación: double
AlgoritmoGenetico(int,int,int,double,Problema)
ejecuta():Individuo
recombinar(Cromosoma,Cromosoma):Cromosoma
<<Java Class>>
AGUniforme
prGenetico
AGUniforme(int,int,int,double,Problema)
recombinar(Cromosoma,Cromosoma):Cromosoma
<<Java Class>>
AGUnPunto
prGenetico
AGUnPunto(int,int,int,double,Problema)
recombinar(Cromosoma,Cromosoma):Cromosoma