UNIVERSITAT ROVIRA i VIRGILI

FINAL EXAMINATION

Técnicas Avanzadas de Programación

Examen Final de Conceptos (Variación 1S)

INSTRUCTORES:

DAVID REBOLLO MONEDERO • DANIEL BARCELONA PONS

Departament d'Enginyeria Informàtica i Matemàtiques
Universitat Rovira i Virgili

david.rebollo@urv.cat • daniel.barcelona@urv.cat

Examen

Curso: Técnicas avanzadas de programación

Versión: 21 ene. 20

Temario:
Java
Scala
herencia y composición
polimorfismo
lambdas y *streams*
diseño de patrones
reflexión
programación funcional
inmutabilidad
recursión

Instrucciones

Este examen final del curso de técnicas avanzadas de programación en Java y Scala permite el uso de apuntes, libros, código, o cualquier recurso que el alumno desee, así como acceso a Internet para cualquier consulta. Recomendamos tener a mano el código empleado en la asignatura, disponible en github.com/davidrm4/URV-TAP (contraseña tap101) y github.com/pedrotgn/TAP. También recomendamos el uso del propio equipo, con una configuración e IDE familiares[1].

Está terminantemente prohibido cualquier tipo de colaboración, incluyendo mensajería por móvil, o el uso de e-mail antes del momento de la entrega.

Rellena tu nombre, apellidos y NIF o similar, así como las respuestas al cuestionario de conceptos, en el fichero **Questionnaire.txt** proporcionado.

Finalmente, envía los resultados por **email a los dos instructores** de la asignatura, es decir, tanto a david.rebollo@urv.cat como a daniel.barcelona@urv.cat, independientemente del grupo en el que estés, con el **asunto "Final *Exam* 1S"** (o el número de variación correspondiente), sin necesidad de incluir nada más en el cuerpo de mensaje, adjuntando un **único fichero TXT** de respuestas al cuestionario mencionado.

El email deberá recibirse **antes del tiempo límite disponible, con el asunto requerido y no otro**. No todas las preguntas del cuestionario presentan la misma dificultad ni te llevarán el mismo tiempo, pero contarán igualmente para la nota, salvo que se indique lo contrario.

Responde a las preguntas siguientes en el fichero `Questionnaire.txt` proporcionado a tal efecto. Por favor, asegúrate de confirmar que estás trabajando sobre la variación asignada y rellena cuidadosamente tus apellidos, tu nombre, y tu NIF o similar. **Insistimos en que es crucial escoger la variación que coincida con el presente enunciado, variación que te será asignada antes de comenzar el examen**. Por favor, no cambies el número de variación.

Como respuesta en el cuestionario, siempre indica una única opción de entre las 4 disponibles. Si varias opciones pudieran considerarse correctas, o ninguna fuera estrictamente correcta, escoge aquélla que mejor responda a la pregunta correspondiente. Por defecto, se entenderá que la opción correcta es la **verdadera**, pero alternativamente, se puede pedir cuál de las opciones es **falsa**. Las respuestas acertadas cuentan positivamente (+1) y las demás, negativamente (−1/3), pero no responder ni bonifica ni penaliza (0).

Lee cuidadosamente el enunciado de la pregunta. No obstante, el propósito de la redacción extensa de algunas opciones es la claridad e ilustración, así como el convertir este examen en una actividad didáctica, y no realmente tender trampas. De hecho, algunas preguntas y sus respuestas pueden proporcionar pistas para otras preguntas. Si dos afirmaciones son incompatibles, lógicamente, una de ellas es falsa y la otra verdadera. No obstante, puede ser útil leer, al menos por encima, el resto de respuestas.

Algunas de las preguntas del cuestionario hacen referencia a una simple jerarquía de clases, ilustrada en la Fig. 1 de la pág. 2, en la que la superclase `Aircraft` contiene las dos subclases `FighterJet` y `PassengerJet`, y la superclase `Pilot` incluye la subclase `AirlinePilot`.

Pregunta 1. Sobre el polimorfismo ad hoc o de sobrecarga (*overloading*), **¿cuál de las siguientes afirmaciones es correcta?**

[1] Aunque puedes emplear la documentación de Java en docs.oracle.com/javase/8/docs/api/, probablemente sea más cómodo buscar en el código comentado en clase, en tu IDE o usar sus herramientas de navegación. Escoge el IDE con el que estés familiarizado. Por ejemplo, en IntelliJ, Edit>Find>Find in Path (Ctrl+Shift+F) busca en todos los ficheros del proyecto. Para navegar a través de clases y otros elementos, puedes usar View>Quick Documentation (Ctrl+Q), Navigate > Declaration or Usages (Ctrl+B), o Navigate > Class (Ctrl+N), o incluso All, marcando include non-project items.

a. No hay distinción entre su implementación en Java y otros lenguajes de programación orientados a objetos, por lo que en todos aquellos lenguajes que incorporan este tipo de polimorfismo, se permite la sobrecarga no sólo de métodos, sino también de operadores.

b. No garantiza la consistencia de tipos ni estática ni dinámicamente.

c. Es estático, en el sentido de que garantiza la consistencia en el uso de tipos desde la compilación.

d. Es dinámico, en el sentido de que garantiza la consistencia en el uso de tipos únicamente en tiempo de ejecución.

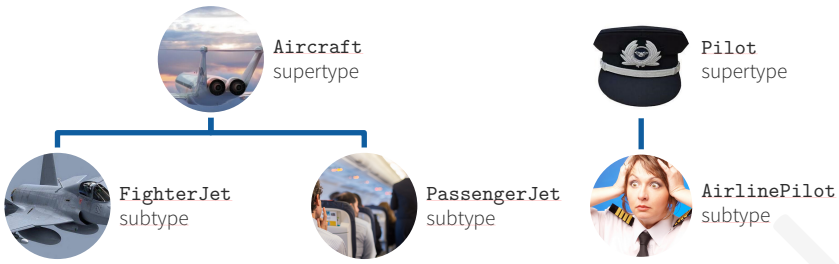

Fig. 1. Simples jerarquías de clases empleadas en alguna de las preguntas, en la que la superclase `Aircraft` contiene las dos subclases `FighterJet` y `PassengerJet`, y la superclase `Pilot` incluye la subclase `AirlinePilot`.

Pregunta 2. Considera el ejemplo de la Fig. 1. Supongamos que las subclases `FighterJet` y `PassengerJet` sobrescriben (*override*) el método de instancia (no estático) `fuelConsumption(distance)` de la superclase `Aircraft`, para el cálculo del consumo de combustible en un trayecto de distancia conocida, cálculo que dependerá del tipo específico de aeronave. En Scala, creamos una lista `aircraftList` de la clase padre `Aircraft`, que contiene objetos de la misma clase o de cualquiera de sus subclases:

```
val aircraftList: List[Aircraft] = List(new Aircraft(…), new FighterJet(…), new PassengerJet(…))
```

Mostramos el resultado del método mediante

```
aircraftList.foreach(aircraft => println(aircraft.fuelConsumption(1000)))
```

¿Cuál de las siguientes afirmaciones es falsa?

a. El almacenamiento de objetos de subtipos de una clase padre común en esta misma lista es un claro ejemplo de polimorfismo de subtipo.

b. La parametrización de esta lista, es decir, la especificación del tipo de elementos permisibles dentro de ella, es un claro ejemplo de polimorfismo paramétrico.

c. Scala empleará ligadura dinámica o tardía (*dynamic or late binding*), es decir, el método `fuelConsumption()` empleado para cada elemento corresponderá al tipo específico del objeto, ya sea `Aircraft`, `FighterJet`, o `PassengerJet`, y no al tipo general de la lista, `Aircraft`.

d. Scala empleará ligadura estática o temprana (*static or early binding*), es decir, el método `fuelConsumption()` empleado para cada elemento corresponderá al tipo general de la lista, `Aircraft`, y no al tipo específico del objeto, ya sea `Aircraft`, `FighterJet`, o `PassengerJet`.

Pregunta 3. Sobre el principio de substitución de Liskov, **¿cuál de las siguientes afirmaciones es falsa?** Considera el ejemplo ilustrativo de la Fig. 1.

a. De acuerdo con el principio de substitución de Liskov, en un contexto en el que se requiere un objeto de tipo T, dicho objeto puede remplazarse por otro de un tipo más específico o subtipo S, de modo que se pueden invocar sobre el nuevo objeto de subtipo S los campos y métodos requeridos originalmente para T, sin perjuicio en la compilación, ejecución o lógica del programa. En el ejemplo de la Fig. 1, donde se requiera `Aircraft`, podríamos proporcionar `PassengerJet`, que incorpora todos los miembros que podríamos necesitar de `Aircraft`, además de otros propios.

b. De acuerdo con el principio de substitución de Liskov, en un contexto en el que se requiere un objeto de tipo S, dicho objeto puede remplazarse por otro de un tipo más genérico o supertipo T, de modo que se pueden invocar sobre el nuevo objeto de supertipo T los campos y métodos requeridos originalmente para S, sin perjuicio en la compilación, ejecución o lógica del programa. En el ejemplo de la Fig. 1, donde se requiera `PassengerJet`, podríamos proporcionar `Aircraft`, que posee todos los miembros que podríamos necesitar de `PassengerJet`, además de otros propios.

c. Se asume que al substituir un objeto por otro, se suministran al menos todos los campos y métodos requeridos, vía herencia, sobrescritos (*overridden*) o no, además de otros más específicos, propios del objeto que substituye al original.

d. Este principio es la clave del polimorfismo de subtipo, y permite asignar objetos de subtipo S a variables de supertipo T, aunque la variable requiera a priori objetos del supertipo T.

Pregunta 4. Sobre subtipos, considera el ejemplo ilustrativo de la Fig. 1. Supongamos que `PassengerJet` es una subclase de `Aircraft`, ya sea en Java o Scala, y que la subclase sobrescribe (*overrides*) el método general de instancia (no estático) `fuelConsumption(distance)`, para el cálculo del consumo de combustible en un trayecto de distancia conocida, cálculo que dependerá del tipo específico de aeronave, pero disponible para `Aircraft` en general y heredado por cualquier subtipo en particular. Adicionalmente, `PassengerJet` introduce el método específico `passengerLoad()`, que calcula el número máximo de pasajeros para un vuelo cómodo y seguro. **¿Cuál de las siguientes afirmaciones es falsa?**

a. Almacenamos un objeto de tipo `PassengerJet` en una variable de tipo `Aircraft`. Como `PassengerJet` es un subtipo de `Aircraft`, cualquier método aplicable a `Aircraft` es heredado y aplicable a `PassengerJet`, como `fuelConsumption()`, por lo

que podríamos aplicar dicho método a la variable. La implementación escogida para el método en cuestión sería asignada en tiempo de ejecución, para el tipo de objeto almacenado.

b. Por otro lado, `passengerLoad()` sería aplicable a la variable de tipo `Aircraft`, pero requeriría *downcasting*.

c. La situación descrita en el apartado (a) constituye un ejemplo de polimorfismo de subtipo y ligadura dinámica (*dynamic binding*). Además, es un ejemplo del principio de substitución de Liskov, ya que en un contexto en el que se espera o requiere `Aircraft`, se proporciona `PassengerJet`, que tiene todos los miembros que podríamos necesitar de `Aircraft`, además de otros.

d. Introducimos un método estático `increaseManufacturedUnits()` en `Aircraft`, que incrementa el número de aeronaves manufacturadas, sea cual sea su tipo específico. Dicho número es naturalmente almacenado en un campo (*field*) también estático. En Java, una invocación a dicho método estático sobre una variable de tipo `Aircraft` usará ligadura dinámica (*dynamic binding*), es decir, Java escogerá en tiempo de ejecución, inteligentemente, la implementación del método contador dependiendo del tipo de aeronave específico almacenado en la variable general.

Pregunta 5. Sobre listas de tipo genérico, ¿cuál de las siguientes afirmaciones es falsa?

a. En Java, las listas *no* son covariantes. Para ilustrar este hecho, considera que aunque `Integer` es un subtipo de `Number`, `LinkedList<Integer>` no es un subtipo de `LinkedList<Number>`, ya que la operación `add(number)` funcionaría en la segunda lista, pero no en la primera, violando el principio de substitución de Liskov.

b. En Scala, las listas inmutables `List` *sí* son covariantes. La razón es que no se permite mutar la lista, sólo se permite el acceso —por ejemplo contar el número de elementos que cumplen cierta condición, sin modificarlos—, por lo que *no* existe el riesgo descrito de añadir un objeto de un supertipo a una lista de un subtipo.

c. En Scala, `ListBuffer` implementa una lista mutable, y por lo tanto, una construcción de tipo contravariante.

d. En Scala, `ListBuffer` implementa una lista mutable, y por lo tanto, una construcción de tipo invariante.

Pregunta 6. Sobre *"currying"* y aplicación parcial de funciones. Recuerda que en Scala `S <: T`, o equivalentemente, `T >: S`, indica que `S` es un subtipo de `T`. Por otro lado, una función que toma un argumento de tipo `A` y retorna un valor de tipo `B` se considera un objeto de tipo compuesto `A => B`. ¿Cuál de las siguientes afirmaciones es falsa?

a. Sea la función `f` de tipo `(X, Y) => R` para ciertos tipos `X`, `Y` y `R`. La versión *"curried"* `fC` de la función sería de tipo `X => (Y => R)`. La evaluación parcial de `fC` en el argumento `x` de tipo `X` resultaría en una función de tipo `Y => R` para cada valor de `x`. En general, *"currying"* es la técnica de traducir la evaluación de una función que toma argumentos múltiples en la evaluación de una secuencia de funciones, cada una con un solo argumento. Así, si `f` es de tipo `(X, Y, Z) => R`, entonces la versión *"curried"* `fC` de la función sería de tipo `X => (Y => (Z => R))`.

b. Para una función `f` de tipo `(X, Y) => R`, el tipo `(X => Y) => R` también se podría obtener a partir de una versión "*curried*" de la función original, donde se tomaría como entrada una función de tipo `X => Y`, y se retornaría un valor de tipo `R`. (Para verificar la validez o invalidez de esta afirmación, quizá te sea útil considerar un ejemplo concreto para la función de partida, digamos `def f(x: Int, y: Int): Boolean = x == y`, que simplemente compara dos enteros.)

c. Considera una función con varias listas de parámetros, como por ejemplo `def add(x: Int)(y: Int)(z: Int): Int = x + y + z`. La invocación de la función incluyendo sólo parte de las listas iniciales, por ejemplo `val addPart = add(1)(2)_` daría como resultado una función, que aplicada a argumentos consistentes con las listas finales arrojaría el resultado final, por ejemplo `addPart(3)`. Esto se conoce como aplicación parcial de funciones (*partial function application*).

d. Una función es covariante en el tipo de retorno, es decir, se puede emplear en un contexto en el que se invoca la función, una función que devuelva un resultado más específico del esperado, por lo que si `B' <: B`, entonces `A => B' <: A => B`, lo cual a su vez implica que `C => (A => B') <: C => (A => B)`.

Pregunta 7. Recuerda que en Scala `S <: T`, o equivalentemente, `T >: S`, indica que `S` es un subtipo de `T`. Por otro lado, una función que toma un argumento de tipo `A` y retorna un valor de tipo `B` se considera un objeto de tipo compuesto `A => B`. De acuerdo con el principio de substitución de Liskov, podemos entender un subtipo específico S como un tipo que puede substituir a su supertipo general T en un contexto en el que se esperaría dicho supertipo T. En términos de funciones, deseamos que se cumplan todos los requisitos exigidos en la entrada y retorno de la función, quizá sobradamente.

Si `A' >: A` y `B' <: B`, entonces `A' => B' <: A => B`, es decir, una función es covariante en el tipo de retorno y contravariante en el de entrada[2]. Por consiguiente, una función puede substituirse por otra que retorne un tipo más específico y acepte un argumento más general. Por ejemplo, en un contexto en el que se requiera una función `AirlinePilot => Aircraft`, podríamos suministrar en su lugar una función `Pilot => PassengerJet`, que funcionaría para cualquier `Pilot`, no sólo para `AirlinePilot`, y retornaría el subtipo `PassengerJet`, que también es un `Aircraft`.

Por otro lado, también en la notación de Scala, recuerda que si `f` es de tipo `(X, Y) => R`, entonces la versión *"curried"* `fC` de la función sería de tipo `X => (Y => R)`. ¿Cuál de las siguientes afirmaciones es correcta?

a. Si `X' <: X`, `Y' <: Y` y `R' >: R`, entonces `Y' => R' <: Y => R`, y finalmente, `X' => (Y' => R') <: X => (Y => R)`.

b. Si `X' >: X`, `Y' >: Y` y `R' <: R`, entonces `Y' => R' <: Y => R`, y finalmente, `X' => (Y' => R') <: X => (Y => R)`.

c. Si `X' <: X`, `Y' >: Y` y `R' <: R`, entonces `Y' => R' <: Y => R`, y finalmente, `X' => (Y' => R') <: X => (Y => R)`.

d. Si `X' >: X`, `Y' <: Y` y `R' <: R`, entonces `Y' => R' <: Y => R`, y finalmente, `X' => (Y' => R') <: X => (Y => R)`.

Pregunta 8. Sobre la sobrescritura (*overriding*) y sobrecarga (*overloading*) de métodos en Java, ¿cuál de las siguientes afirmaciones es falsa?

[2] Una de las preguntas de una de las variaciones del examen parcial de Scala contenía un error tipográfico. A pesar de que la redacción era intencionadamente redundante y correcta, así como el ejemplo ilustrativo, dicho error tipográfico podría técnicamente inducir a confusión. A pesar de que no recibimos consulta alguna al respecto, hemos preferido corregirlo en las soluciones publicadas, y la corrección pertinente se incorpora en el enunciado de esta pregunta.

a. La anotación `@Override` es opcional pero conveniente si nuestro propósito es sobrescribir y no sobrecargar el método. La razón de su conveniencia es que indica al compilador que verifique para nosotros que la firma (*signature*) del método sea la misma. Por un lado, se comprobará que el nombre del método se ha escrito correctamente. Por el otro, que la lista de parámetros —tipo, orden y número— coincide, de modo que no incurramos en una sobrecarga accidental.

b. Si en una subclase se declara un método con el mismo nombre que en otro de la clase padre, pero la lista de parámetros —tipo, orden y número— es diferente, el método en cuestión se considerará sobrecargado (*overloaded*), y se empleará aquella versión en la que la lista de parámetros de la declaración y la lista de argumentos de la invocación encajen consistentemente.

c. Al sobrescribir (*override*) un método, la firma (*signature*) debe ser idéntica, pero el tipo de retorno puede cambiarse arbitrariamente sin que afecte de manera adversa a la compilación o la ejecución. De hecho, según convenga para la implementación, el tipo de retorno podría ser más general, más específico, o incluso completamente distinto.

d. En la sobrescritura (*overriding*), el modificador de acceso (`private`, *package-protected*, `protected`, `public`) debe ser igual o más permisivo —en otras palabras, igual o menos estricto— que el del método en la clase padre. Por esta razón, un método abstracto en una interfaz, que es implícitamente público, al sobrescribirse, la implementación resultante también será pública.

Pregunta 9. Sobre interfaces en Java, ¿cuál de las siguientes afirmaciones es falsa?

a. Tanto los campos (*fields*) como los métodos de una interfaz son implícita y obligatoriamente públicos. Además, los campos son estáticos y finales.

b. Sin modificador alguno y sin cuerpo, un método se interpreta como abstracto, y se supone que su implementación será suministrada por una clase que implemente dicha interfaz.

c. Una interfaz también puede incorporar otros dos tipos de métodos: métodos estáticos y métodos "default". Para ellos sí se suministra implementación, a diferencia de los métodos abstractos. Sin embargo, los métodos estáticos *no* se heredan, y cualquier referencia a los mismos en una clase que implemente dicha interfaz requiere anteponer el nombre de la interfaz, por ejemplo, `InterfaceName.staticMethod()`. Como la herencia múltiple de interfaces sí está permitida en Java, de esta forma evitamos ambigüedad sobre un método con un mismo nombre implementado de forma distinta en varias interfaces. Excluyendo el caso particular de interfaces, es decir, para clases convencionales que *no* sean interfaces, un método estático sí se hereda y, en principio, es posible invocarlo directamente sin que sea obligatorio el nombre de la clase padre como prefijo.

d. Además de que en clases convencionales que *no* sean interfaces un método estático se hereda, también se permite su sobrescritura (*override*). Obsérvese que el término sobrescritura (*override*) se emplea en un sentido estricto, y no se trataría de una mera ocultación de método (*method hiding*). Esto quiere decir que al invocar el método estático en cuestión, el mecanismo de ligadura dinámica (*dynamic binding*) operará exactamente como si se tratase de un método dinámico sobrescrito, es decir, se escogerá apropiadamente en tiempo de ejecución el método estático que corresponda al tipo o subtipo de objeto referido, y se hará independientemente de la clase de la variable que lo refiera.

Pregunta 10. Supongamos que se requiere implementar una pila (*stack*) a partir de una lista enlazada (*linked list*) disponible. Se prefiere que la implementación de la clase representando la pila sea transparente, es decir, que los campos (*fields*) y métodos de la clase correspondiente a la lista enlazada no sean visibles a la hora de usar la pila. Esta cuestión se consideró para Java, pero, ¿cuál de las afirmaciones siguientes sería correcta en Scala?

a. No es posible implementar una pila a partir de una lista enlazada, ya que son objetos diferentes. Esto es cierto, indistintamente, en Java y Scala.

b. En este caso, para Scala, es preferible emplear herencia, ya que la filosofía de ambos lenguajes, Java y Scala, es completamente distinta, y un conocimiento avanzado de ambos lenguajes conduce inmediatamente a esta conclusión.

c. El principio de diseño aplica igualmente, por lo que en el caso descrito es preferible emplear composición. Los métodos de la pila, en general, delegarán en los de la lista enlazada. Gran parte de lo aprendido en Java, y por supuesto también en Scala, es útil a la hora de abordar el aprendizaje de cualquier otro lenguaje orientado a objetos.

d. Ninguna de las anteriores respuestas es correcta.

Pregunta 11. Considera la declaración del método genérico siguiente en Java, para calcular el *mínimo* entre dos objetos comparables:

```
public static <T extends Comparable<? super T>> T genericMin(T x, T y) {
   T min = x;
   if (y.compareTo(min) < 0) min = y;
   return min;
}
```

¿Cuál de las siguientes afirmaciones es falsa?

a. La interfaz `Comparable` declara el método abstracto `compareTo()`. Por convención, `x.compareTo(y)`, para dos objetos `x` e `y`, retorna un valor `int` que será positivo si se considera que `x` es menor o antecede a `y`, cero si se considera que `x` e `y` son iguales o equivalentes, y un valor negativo si se considera que `x` es mayor o es posterior a `y` de acuerdo con el orden establecido.

b. Por un lado, el parámetro de tipo T está acotado superiormente (*upper bounded*) por la interfaz `Comparable`, es decir, que la cláusula `extends` indica que se permitiría emplear el método genérico para `Comparable`, o para cualquier subtipo de `Comparable`. El procedimiento de borrado de tipo (*type erasure*) crearía una única versión en el código, remplazando T con su cota `Comparable`. Por otro lado, el uso del comodín (*wildcard*) y la cláusula `super` en `Comparable<? super T>` es un ejemplo de contravariancia, según el cual se admitiría un método de comparación que aceptase el tipo T, o un supertipo.

c. El método `genericMin()` descrito podría invocarse con argumentos de cualquier tipo que implemente la interfaz `Comparable`, incluyendo `Integer` o `Double`, pero también `String`. Aunque en el caso de `String`, se emplearía un orden lexicográfico —como el que se emplearía en un diccionario, no numérico— para encontrar el mínimo.

d. También sería aplicable el método a una clase `Student` que representase alumnos, y que quisiéramos ordenar, por ejemplo, por apellido, o bien por nota en este examen, siempre y cuando se implementase el método `compareTo()` y se citase la interfaz implementada al declarar la clase. Además, si quisiéramos ordenar o minimizar simultáneamente primero por nota y después por apellido, una opción sería implementar `compareTo()` mediante condicionales, de forma que cuando la nota sea igual, se decida el signo del método a partir del apellido, en orden lexicográfico.

Pregunta 12. Considera el siguiente código en Scala, que implementa una estructura de datos `StructureX` con operaciones `operationA()` y `operationB()`.

```
class StructureX[E] {
  private var elements: List[E] = Nil

  def operationA(x: E): Unit = elements = x :: elements

  def operationB(): E = {
    val x = elements.head
    elements = elements.tail
    x
  }
}
```

¿Cuál de las siguientes afirmaciones es falsa?

a. Se trata de una estructura de datos genérica, que manipula elementos de tipo no acotado ni superior ni inferiormente —en otras palabras, de cualquier subclase de `Any` (incluyéndose a sí misma)—, parametrizado por `E`.

b. El método `operationA(x)` actúa como la operación *push* de una pila (*stack*), añadiendo el elemento `x`. El método `operationB()` actúa como la operación *pop* de una pila (*stack*), extrayendo y suministrando el elemento superior de la pila, es decir, aquél que se añadió más recientemente. Por tanto, se trata de una estructura de tipo *last-in first-out* (LIFO).

c. Ciertamente, las operaciones `operationA()` y `operationB()` añaden y extraen respectivamente, pero el código mostrado se comporta como una cola (*queue*) de tipo *first-in first-out* (FIFO), en la que el primer elemento añadido es el primer elemento que se extraerá.

d. El código mostrado implementa una estructura de datos a partir de una lista `List`, mediante composición (*composition*), no mediante herencia (*inheritance*), de modo que la implementación concreta y la estructura interna no son directamente visibles.

Pregunta 13. ¿Cuál de las siguientes expresiones lambda en Java implementaría correctamente tanto la interfaz funcional `IntUnaryOperator`, como la interfaz `UnaryOperator<Integer>`?

a. `n -> n / 2`

b. `n -> n % 2 == 0`

c. `(m, n) -> m * n`

d. `s -> s.toUpperCase()`, o equivalentemente, su referencia de método `String::toUpperCase`.

Pregunta 14. Sobre el método intermedio de *stream* `reduce()` en Java, asume que `intArray` es de tipo `int[]`, y considera el código siguiente:

```
System.out.printf("%d%n", IntStream.of(intArray).reduce(identity, intBinOp));
```

¿Cuál de las siguientes afirmaciones es falsa?

a. Estableciendo `identity = 0` y `intBinOp = (x, y) -> x + 2 * y`, el resultado sería el doble de la suma acumulada de los elementos de `intArray`. Una manera de ahorrar operaciones consistiría en calcular la suma primero mediante `intBinOp = (x, y) -> x + y`, y multiplicar una sola vez el resultado por 2.

b. Estableciendo `identity = 0` y `intBinOp = (x, y) -> x + y * y`, el resultado sería la suma acumulada de los cuadrados de los elementos de `intArray`, es decir, el cuadrado de la norma euclídea del vector correspondiente.

c. Estableciendo `identity = 1` y `intBinOp = (x, y) -> x * (2 * y)`, el resultado sería el doble del producto acumulado de los elementos de `intArray`.

d. Estableciendo `identity = 1` y `intBinOp = (x, y) -> x * (2 * y)`, el resultado sería el producto acumulado de los elementos de `intArray`, multiplicado por 2 elevado a la longitud del arreglo (*array*).

Pregunta 15. Sobre el patrón de diseño *singleton*, ¿cuál de las siguientes afirmaciones es falsa?

a. Su objetivo es restringir la instanciación de la clase a un único objeto, accesible a través de una referencia posiblemente global. Sin entrar en detalles de cada aplicación concreta, en principio, una funcionalidad similar podría obtenerse considerando, en lugar del *singleton*, una serie de variables y métodos estáticos accesibles globalmente.

b. Dicho objeto podría representar y gestionar el acceso a un recurso compartido, como por ejemplo una base de datos o un controlador de dispositivo, para el que se preferiría un único punto de acceso, quizá por seguridad o eficiencia.

c. La variación perezosa (*lazy*) del patrón de diseño *singleton* pretende retrasar la instanciación del objeto hasta que es realmente necesaria, posiblemente tras ciertos procesos de inicialización, mientras que la variación impaciente (*eager*) instanciaría el objeto inmediatamente, en cuanto la clase se cargase (*class loading*), tras una invocación dinámica o estática.

d. La variación perezosa (*lazy*) contiene una autorreferencia (*self-reference*) a una instanciación de la propia clase, es decir, un campo (*field*) estático del mismo tipo que la clase en cuestión, en cuya declaración se inicializará al único objeto instanciable. La variación impaciente (*eager*), sin embargo, se inicializa a `null`.

Pregunta 16. Sobre patrones de diseño, ¿cuál de las siguientes afirmaciones es falsa?

a. Se requiere implementar una clase `A` que representa una colección de información de interés, a la que varios agentes correspondientes a subclases concretas `B1`, `B2`, etc. de una clase abstracta `B` desearían subscribirse. Dichas subclases `B1`, `B2` desearían poder registrarse a una lista para recibir actualizaciones sobre la información recogida por `A`. Basándose en estos requerimientos, un patrón de diseño apropiado para ellos sería *observer*.

b. Supongamos que tenemos código antiguo para una clase que implementa una cierta interfaz que no se ajusta perfectamente a la interfaz requerida actualmente. Deseamos crear una nueva clase basada en la clase antigua —ya sea mediante composición o herencia—, para proporcionar una funcionalidad similar a través de la implementación de la interfaz actual. Basándose en esta información, en principio, el patrón de diseño más apropiado para tal fin sería *adapter*.

c. Consideremos una clase abstracta `Duck`, con subclases como `MallardDuck`, `RedheadDuck` o `RubberDuck`. Algunos métodos de `Duck` como `quack()` pueden ser concretos, pero otros podrían ser abstractos, como `display()`, e implementados específicamente para cada subclase. Hay un cierto procedimiento, que podría representarse como el método `fly()`, que pueden realizar algunas subclases, pero no todas, y entre las que pueden, muchas pueden reutilizar la misma implementación. Supongamos que uno pudiera tener 20 subtipos de pato, y quizás 5 formas de volar, además de la posibilidad de no volar. Optamos por definir la interfaz funcional `FlyBehavior`, con el método abstracto `fly()`. Las clases como `FlyWithWings` y `FlyNoWay` (la segunda para aquellos que no pueden volar) implementan `FlyBehavior` de forma modular y reutilizable. Agregamos un campo de instancia `FlyBehavior` a la clase `Duck`, y un método de instancia `performFly()` que invocará el método `fly()` en este campo de instancia, delegando así la acción. El comportamiento apropiado se invocará para una subclase específica gracias al enlace tardío o dinámico. La inicialización de una subclase dada creará una instancia de la clase de comportamiento y la asignará al campo de instancia `FlyBehavior` heredado. Por ejemplo, el constructor de `MallardDuck` podría instanciar el comportamiento `FlyWithWings`. A continuación, la acción `performFly()` heredada ejecutará el método `fly()` apropiado. En el caso descrito, estaríamos empleando el patrón de diseño *strategy*.

d. En el caso descrito anteriormente en el apartado (c), si `fly()` se definiera como un método abstracto en una interfaz `Flyable`, solo aquellas clases que pueden volar podrían declarar que implementan `Flyable`, e incluir una implementación para `fly()`. A pesar de que muchas de estas implementaciones pudieran ser idénticas, esta alternativa sería idónea en términos de reutilización y modularidad.

Pregunta 17. Consideremos los escenarios de aplicación del patrón de diseño proxy, para la implementación de un representante de una clase sin modificación del código original, así como de la versión reflexiva (*reflective*) conocida como proxy dinámico (*dynamic proxy*). ¿Cuál de los siguientes escenarios no constituiría una aplicación directa y clara de un proxy?

a. Representación local de un objeto remoto.

b. Acceso perezoso (*lazy*) a recursos de inicialización costosa, posponiendo procesos no inmediatamente necesarios según convenga.

c. Validaciones de entrada y salida adicionales, posiblemente incluyendo el registro (*logging*) o auditoría del acceso a un recurso.

d. Acceso a una colección de objetos de manera secuencial, sin necesidad de conocer la estructura interna, conforme a la interfaz `Iterator` en Java o a la característica (*trait*) del mismo nombre en Scala.

Pregunta 18. Sobre programación funcional y conceptos relacionados. ¿Cuál de las siguientes afirmaciones es falsa?

a. Una función de orden superior (*higher-order function*) es aquélla que acepta una función como argumento de entrada y/o retorna una función como valor de salida. Un lenguaje con funciones de primera clase (*first-order functions*) considera las funciones como entidades computables, es decir, datos o valores. Por ello, las propias funciones tienen tipos, se permite su almacenamiento y manipulación mediante variables, y son posibles las funciones de orden superior (*higher-order functions*).

b. El efecto principal (*main effect*) de una función es su valor de retorno. El efecto secundario (*side effect*) de una función es todo efecto observable persistente debido a su ejecución que no esté ceñido estrictamente a su valor de retorno. Esto incluye, cualquier alteración en memoria —incluyendo cambios en estructuras de datos mutables accesibles a través de referencias proporcionadas como argumentos de entrada y cambios de estado en variables globales— cualquier operación de entrada/salida (*I/O*) en disco que conlleve modificaciones de ficheros, e incluso comunicación externa a través de la red.

c. Una función pura (*pure function*) es una función libre de efectos secundarios (*side-effect free*). Suele entenderse también, aunque no siempre, que una función pura es, además de libre de efectos secundarios, determinística, en el sentido de que la misma entrada produce el mismo retorno.

d. El uso del término "función pura" en la literatura siempre es perfectamente claro, y la definición en sí de "determinística" tampoco da lugar a ambigüedad posible. En el caso particular de un argumento efectivamente pasado por referencia, por ejemplo una lista mutable, no importa si cuando hablamos de "determinística" nos referimos a mantener constante como argumento de entrada la referencia solamente, léase, su posición en memoria, o también los datos referenciados de una estructura mutable.

Pregunta 19. Sobre optimización en programación funcional. ¿Cuál de las siguientes afirmaciones es falsa?

a. Las funciones puras en el sentido habitual de libres de efectos secundarios y deterministicas, se prestan a pruebas unitarias (*unit testing*), es decir, a validaciones de una función aisladas de posibles efectos colaterales del sistema integral, para una lista predeterminada de pares de entrada y salida. Si la función no fuera determinística e incorporase efectos aleatorios, pero por lo menos sí fuera libre de efectos secundarios (*side-effect free*), seguiría pudiéndose validar de forma aislada de otras funciones y del sistema global. No obstante, la comprobación de entrada y salida ya no sería determinística, sino que debería analizar el comportamiento estadístico de la salida de la función —por ejemplo media, varianza, mediana, distribución— y verificar que se ajusta a las directrices de diseño.

b. Consideremos una expresión repetida que evalúa la misma función para el mismo argumento de entrada, posiblemente en varias partes del código o dentro de un bucle. Recordemos que la optimización conocida como eliminación de subexpresión común (*common subexpression elimination*, CSE) consiste en calcular la expresión una vez, manteniendo el resultado en una variable y remplazando expresiones repetidas por simples accesos a dicha variable. Claramente, una función que *no* estuviera libre de efectos secundarios, retornaría un resultado posiblemente dispar de una ejecución a otra, dependiendo del estado de la memoria, el disco, o incluso la red, por lo que *no* sería una candidata ideal para la optimización descrita. Supongamos que otra función distinta *sí* está

completamente libre de efectos secundarios (*side-effect free*) pero *no* es determinística, sino que el resultado depende de números aleatorios generados internamente. A pesar de ello, al ser libre de efectos secundarios, esta última función sería una candidata idónea para la optimización descrita, y a pesar del remplazo, el comportamiento del programa optimizado sería completamente idéntico en todos los aspectos funcionales, exceptuando su velocidad de ejecución.

c. Sea `f` un procedimiento, función o método que invoca a otro `g`. Supongamos que la llamada a `g` es la última operación en `f`, o dicho de manera más precisa, que en el camino actual a lo largo del flujo de ejecución, al finalizar la ejecución de `g` no es necesario continuar con la ejecución de `f`. La llamada de `f` a `g` se consideraría una llamada terminal (*tail call*). En dicha llamada, se puede simplificar la gestión de la pila (*stack*), reutilizando la trama (*stack frame*) de `f` en lugar de añadir una nueva para `g`, con la dirección de retorno directo al contexto en el que se invocó `f`, ya que no es necesario retornar de `g` a `f`. Esta simplificación se conoce como optimización de llamadas terminales (*tail-call optimization*).

d. La implementación recursiva del factorial de un número natural mostrada a continuación es una recursión terminal (*tail recursion*), ya que la última operación es una invocación a la propia función factorial, con un argumento decreciente:

```
def factorialTailRecursive(n: Int): Int = {

@tailrec
def factorialAux(n: Int, p: Int): Int =
  if (n == 0)
    p
  else
    factorialAux(n - 1, n * p)

factorialAux(n, 1)
}
```

Pregunta 20. Sobre recursiones. La secuencia de Fibonacci $F_N = 0, 1, 1, 2, 3, 5, 8, \ldots$ se define habitualmente[3] a partir del estado inicial $F_0 = 0$ y $F_1 = 1$, mediante la recursión $F_N = F_{N-1} + F_{N-2}$ para todo $N \geqslant 2$. (Una convención similar consiste en partir del estado inicial $F_1 = F_2 = 1$, definiendo la sucesión comenzando en $n = 1$ en lugar de $n = 0$, pero con la misma recursión para $N \geqslant 3$.)

Vimos en clase un famoso ejemplo de pregunta de entrevista[4] de Amazon, en el que el número de formas w_N de subir N escalones de 1 en 1 o saltando de 2 en 2 se podía calcular recursivamente, a partir del problema de subir $N-1$ escalones, si acabamos subiendo el último peldaño sin saltar, y el de subir $N-2$ escalones, si acabamos subiendo dos peldaños de una vez. Concretamente, vimos que $w_N = F_{N+1}$. Por ejemplo, existen $w_5 = F_6 = 8$ maneras de subir $N = 5$ peldaños, que podrían representarse como 11111, 1112, 1121, 1211, 122, 2111, 212, 221.

Considera ahora un problema que, como verás, guarda cierta relación. Tenemos un **texto con N páginas**, que deseamos enviar a ciertos revisores voluntarios. Consideramos **dividir el texto en porciones contiguas de 1, 2 ó 4 páginas (pero no 3, por simplicidad)**, de forma que cada porción representará una cantidad de trabajo distinta para nuestros voluntarios. Insistimos en que las porciones serán contiguas, es decir, podemos crear una porción de dos páginas con las páginas 3 y 4, pero no con las páginas 3 y 5, que no sería contigua y resultaría más fácil perder el hilo. Seguramente vendrá bien considerar porciones de distinto tamaño, pues es previsible que nuestros revisores tengan distinta disponibilidad.

Estamos interesados en calcular el nuevo **número de formas w_N de dividir el trabajo**, y nos planteamos emplear una definición recursiva, con un estado inicial que produzca valores coherentes para $N \geqslant 1$, aunque nuestra definición informática funcione para valores artificiales como $N = 0$, o incluso negativos, que a pesar de no ser reales, quizá resulten en un código más compacto y elegante, aunque nos contentamos con que sea válido para los N necesarios.

Para esta nueva definición de w_N correspondiente al problema de interés, **¿cuál de las siguientes afirmaciones es falsa?**

a. $w_1 = 1, w_2 = 2, w_3 = 3, w_4 = 6$. En particular, para $N = 4$, las $w_4 = 6$ opciones pueden representarse mediante la lista de listas 1111, 112, 121, 211, 22, 4. Por otro lado, $w_N = w_{N-1} + w_{N-2} + w_{N-4}$ para $N > 4$. La recursión sería válida incluso para $N = 4$ simplemente definiendo $w_0 = 1$.

b. $w_5 = 10, w_6 = 18, w_{10} = 169$.

c. $w_{20} = 46\,754$.

d. $w_{20} = 82\,047$.

Puedes resolver este problema como desees, programando o no en Scala, pues únicamente pedimos escoger la respuesta adecuada, no solicitamos el código.

[3] Al tratarse de una recursión lineal, tiene una forma cerrada conocida, llamada fórmula de Binet, relacionada con el número áureo, que si tienes curiosidad y tiempo —obviamente después del examen— puedes encontrar en https://en.wikipedia.org/wiki/Fibonacci_number#Binet's_formula.

[4] Puedes encontrar el vídeo de YouTube en el que se explica el problema original de los escalones en https://www.youtube.com/watch?v=5o-kdjv7FD0.