Instituto Superior de Engenharia de Lisboa

LEIC

Técnicas de Desenvolvimento de Software

Teste Final de Época Normal, Inverno de 2022/2023, 13 janeiro 2023

Nome: Número:

Grupo I

Considerando os temas tratados nas aulas da disciplina, responda às perguntas seguintes assinalando de forma inequívoca a opção correta. Não responda arbitrariamente: cada resposta incorreta desconta 1/3 da cotação da pergunta ao total obtido no grupo. A cotação total do grupo é 7 valores e todas as perguntas têm a mesma cotação.

1. Considere a seguinte definição: `fun Path.lines() = this.readText().split("\n")`
   que tem um **erro de compilação** na chamada à *suspend function* `readText()`.

   Escolha a opção que corrige esse erro:

   - [ ] `fun Path.lines() = thread { this@lines.readText().split("\n") }`
   - [ ] `suspend fun Path.lines() = this.readText().split("\n")`
   - [ ] `fun Path.lines():List<String> { return suspend this.readText().split("\n") }`
   - [ ] `fun Path.lines():List<String> { return await this.readText().split("\n") }`

2. Seleccione a ferramenta fundamental para a **execução** de um software compilado e desenvolvido em Kotlin Compose Desktop:

   - [ ] intelliJ
   - [ ] Gradle
   - [ ] java (Java Runtime Environment)
   - [ ] mvn (Maven)

3. Dadas as seguintes definições:

`interface I { fun foo() = print("I") }`
 	
`fun I.bar() = print("I")`
	`class A : I`
	`class B : I { override fun foo() = print("B") }`
	`fun B.bar() = print("B")`

   Qual o output da execução de: fun main() = listOf(A(), B()).forEach { it.foo(); it.bar(); }

   - [ ] IIBB
   - [ ] IIII
   - [ ] IIBI
   - [ ] Nenhuma das opções

4. Dada a função:
```
fun  List<String>.maxLength() : Int = this.map { return it.length }.sorted().last()
```

   Note que a função `sorted()` ordena uma lista de inteiros de forma **crescente**.
   Qual o output da execução de:

```
fun main() = println(listOf("isel", "portugal", "tds").maxLength())
```

   - [ ] 4
   - [ ] 8
   - [ ] 3
   - [ ] Excepção

5. Dada a definição: `class X<T>(src: T){ fun <R> foo(t: T, r: R): R = r}`

   escolha a opção que **<u>não</u>** está correcta e dá erro de compilação:

   - [ ] `val n1 = X<String>("pt").foo("isel", 7)`
   - [ ] `val n2 = X("pt").foo("isel", 7)`
   - [ ] `val n3 = X<String>("pt").foo<String, Int>("isel", 7)`
   - [ ] `val n4 = X("pt").foo<Int>("isel", 7)`

6. Dada a definição: `fun Z.copy() = Z()`

   escolha a opção que tem *output* `true` para a execução de:

```
fun main() { val z = Z();  println(z === z.copy()) }
```

   - [ ] `class Z`
   - [ ] `data class Z`
   - [ ] `object Z { operator fun invoke() = Z }`
   - [ ] `sealed class Z`

7. A relação entre A e B representada num diagrama de classes na forma B ➔ A corresponde à seguinte definição de B:

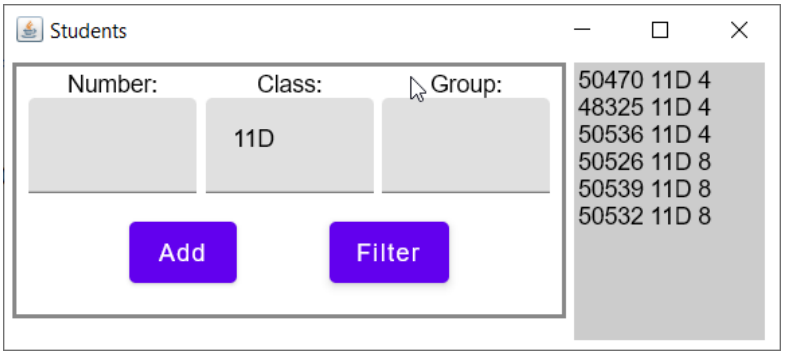
   - [ ] `class B : A()`
   - [ ] `class B(val a: A)`
   - [ ] `class B(a: A)`
   - [ ] `class B() { fun a() = A()}`

Grupo II

Pretende-se desenvolver uma aplicação para gerir informação dos alunos quanto à turma e ao grupo. A aplicação permite adicionar mais alunos e filtrar os alunos por uma das suas propriedades (número, turma ou grupo).
A função `main` da aplicação é a seguinte:

```
fun main() {
   val allStudents: List<Student> =
      Path("students.txt").readLines().map { it.split(' ').toStudent() }
   application {
      val winState = WindowState(width = 450.dp, height = 300.dp)
      val state = remember { StudentsState(allStudents) }
      Window(onCloseRequest = ::exitApplication, state = winState, title = "Students") {
         Row(Modifier.padding(5.dp)) {
            FieldsEdit(
               labels = listOf("Number","Class","Group"),
               onAdd = { fields: List<String> -> state.add(fields) },
               onFilter = { fields: List<String> -> state.filter(fields) }
            )
            StringList(state.viewStudents.map { it.toString() } )
         }
      }
   }
}
```

A figura apresenta a janela da aplicação depois de ser preenchido o campo "Class" (Turma) e premido o botão "Filter", ficando apenas visíveis os alunos da turma "11D".



O componente `FieldsEdit`, apresentado com rebordo, é configurável e permite editar as propriedades com os nomes indicados em `labels`, admitindo que os conteúdos são *strings*. Quando é premido o botão "Add" ou "Filter" é chamada a função correspondente passando como parâmetro a lista com os conteúdos editados.
O componente `StringList` apresenta uma lista de *strings* numa coluna com fundo cinzento.

O botão "Filter" filtra os alunos apresentados pelos campos preenchidos, ou apresenta todos os alunos se todos os campos estiverem vazios.
O botão "Add" acrescenta um aluno à lista, caso todos os campos estejam preenchidos e válidos. Sendo apresentados todos os alunos (sem filtro).

1. [2] Tendo em atenção a função `main`, defina o tipo *imutável* `Student`, a função `toString()` e a função `toStudent()`. `Student` que representam a informação de um estudante com as propriedades `number`, `classId` e `group`. Para simplificar, o número e o grupo são valores inteiros sem restrições e a turma (`classId`) é uma `String` qualquer. A função `toStudent()` deve lançar `IllegalArgumentException` se a lista de `String` não tiver exatamente 3 elementos.

2. [3] Defina o tipo `StudentsState` com as operações `add` e `filter`, chamadas na função `main`. A propriedade `viewStudents` tem os alunos a apresentar e a sua alteração deve provocar o refrescamento da apresentação da lista de alunos.
   A operação `add` adiciona um aluno à lista principal (não filtrada) se todos os campos forem válidos para os valores das propriedades de `Student`. Caso já exista um aluno com o número indicado, esse será previamente removido.
   A operação `filter` filtra os alunos apresentados pelos campos que estiverem preenchidos.

3. [1] Crie um teste automático para verificação da correção da operação `add` de `StudentsState`.

4. [1] Implemente o *Composable* `StringList`, usado na função `main`, para apresentação de uma lista de *strings*, com aspeto semelhante ao da parte direita da janela da figura.

5. [2] Para facilitar a implementação de `FieldsEdit`, implemente o *Composable* `LabeledField(label: String, value: String, onChange: (String)->Unit)` que será responsável pela edição de cada campo. Nota: Este componente é *stateless*, ou seja, não mantém o estado do valor em edição..

6. [4] Implemente o *Composable* `FieldsEdit` usado em `main`, com o aspecto semelhante ao apresentado na figura. Este componente cria um componente `LabeledField` para cada elemento da lista `labels` e mantém o estado de edição de cada um (é *stateful*). Quando é premido um dos botões (Add ou Filter) é passado como argumento a lista dos estados de cada campo.

Duração: 90 minutos
ISEL, 13 de Janeiro de 2023