INSTITUTO METRÓPOLE DIGITAL

TESTE DE SOFTWARES I - **DIM0507**

Discente: Oziel Alves do Nascimento Júnior

Matrícula: 20170065711

Exercícios - Classes de equivalência e Valor limite

1. Mão na Massa 04:

Classes de Equivalência com base em Interface

Restrição de entrada	Classe Válida	Classe Inválida
`valor >= 1`	`true`	`false`

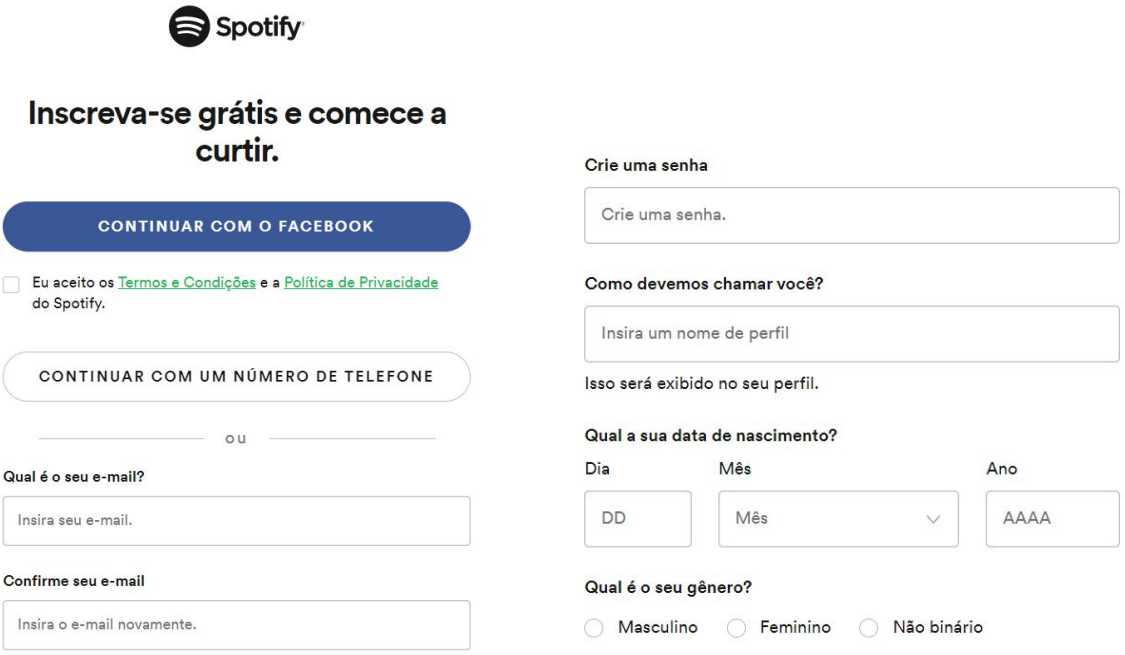
Classes de Equivalência com base em Funcionalidade

Imposto	Restrição de entrada	Classe Válida	Classe Inválida
0%	`1 <= valor <= 1200`	`true`	`false`
10%	`1201 <= valor <= 5000`	`true`	`false`
15%	`5001 <= valor <= 10000`	`true`	`false`
20%	`valor > 10000`	`true`	`false`
-	`valor <= 0`	`false`	`true`

Obs.: É possível encontrar os casos de teste implementados com base nestas classes de equivalência no projeto `./ExercíciosJUnit`, na pasta `./src_test/param`. Os testes parametrizados referentes ao **Mão na Massa 05** também podem ser encontrados na mesma pasta.

2. Mão na massa 06:

Para este exercício foi selecionado o formulário de cadastro do Spotify:



2.1 Classes de Equivalência quanto ao **e-mail** do usuário
Considere o seguinte regex para validação do E-mail:
`re(/^[a-z0-9.]+@[a-z0-9]+\.[a-z]+\.([a-z]+)?$/i)`

Classes de Equivalência

	Classe Válida	**Classes Inválidas**		
email	atende ao regex	atende ao regex	não atende ao regex	não atende ao regex
&	**&**	**&**	**&**	**&**
email_confirm	`e-mail = email_confirm`	`e-mail != email_confirm`	`e-mail = email_confirm`	`e-mail != email_confirm`

2.1.1 Casos de teste utilizando apenas o critério de classes de equivalência

```
validaEmail(teste@teste.com, teste@teste.com);
validaEmail(teste@teste.com, outro_teste@teste.com);
validaEmail(teste.teste.com, teste.teste.com);
validaEmail(teste.com, outro_teste.teste.com);
```

2.1.2 Casos de teste utilizando o critério de análise do valor limite

Neste caso não existe um range de valores a ser observado entre as partições consideradas. A definição de casos de testes utilizando o critério de análise do valor limite será melhor apresentada para as outras variáveis do formulário nos próximos itens.

2.2 Classes de Equivalência quanto a **senha** do usuário
Obs.: Empiricamente foi identificado que o número de caracteres mínimos é 6:

	Classe Válida	**Classe Inválida**
senha	`len(senha) >= 6`	`len(senha) < 6`

2.2.1 Casos de teste utilizando apenas o critério de classes de equivalência

```
validaSenha('123456');
validaSenha('abcd');
```

2.2.2 Casos de teste utilizando o critério de análise do valor limite

```
validaSenha('12345');
validaSenha('123456');
validaSenha('1234567');
validaSenha('abcde');
```

2.3 Classes de Equivalência quanto ao **apelido** do usuário

	Classe Válida	Classe Inválida
apelido	`len(apelido) >= 1`	`len(apelido) >= 1`

2.3.1 Casos de teste utilizando apenas o critério de classes de equivalência

```
validaSenha('Wanessa');
validaSenha('');
```

2.3.2 Casos de teste utilizando o critério de análise do valor limite

```
validaSenha('');
validaSenha('A');
validaSenha('Ab');
```

2.4 Classes de Equivalência quanto a **data de aniversário** do usuário

Obs.: Note que existe uma validação nativa para a data recolhida por inputs, considere a idade mínima para cadastro no Spotify = 10 anos e o formato de data `"dd-mm-yyyy"`.

	Classe Válida	Classes Inválidas
data_nasc	`data_nasc.ano < 2010`	`data_nasc.ano >= 2010`

2.4.1 Casos de teste utilizando apenas o critério de classes de equivalência

```
validaData('01-01-2007');

// Gera exceção de idade
validaData('01-01-2015');
```

2.4.2 Casos de teste utilizando o critério de análise do valor limite

```
validaData('01-01-2009');
validaData('01-01-2010');
validaData('01-01-2011');
```

2.5 Classes de Equivalência quanto a **gênero** do usuário

Obs.: Note que existe uma validação nativa para a data recolhida por inputs

	Classes Válidas			Classe Inválida
genero	`'feminino'`	`'masculino'`	`'nao-binario'`	`''`

2.5.1 Casos de teste utilizando apenas o critério de classes de equivalência

```
validaGenero('feminino');
validaGenero('masculino');
validaGenero('nao-binario');
validaGenero('');
```

2.5.2 Casos de teste utilizando o critério de análise do valor limite

Neste caso não foi observado ranges de valores na definição das partições.

2.6 Conjunto de testes para o formulário de cadastro do Spotify

A tabela abaixo representa um grupo de testes compondo um teste em que o cadastro é realizado e 6 outros testes que possuem informações não validadas.

Causa	**email**	'teste@teste.com'	'teste.com'	'angi@teste.com'	'andre@teste.com'	'Kaio@teste.com'	'amanda@teste.com'	'Joao@teste.com'
	email_confirm	'teste@teste.com'	'teste.com'	'@teste.com'	'andre@teste.com'	'Kaio@teste.com'	'amanda@teste.com'	'Joao@teste.com'
	senha	‘123456’	‘123456789’	‘abcdef’	‘1a2b3’	‘abcdefg’	‘202020’	‘1a2b3c’
	apelido	‘Oziel’	‘J’	‘Angélica’	‘André’	‘’	‘Amanda08’	‘João Pedro’
	data_nasc	'19-03-1999'	'29-03-2010'	'20-11-2009'	'18-10-1999'	'19-03-1999'	'12-06-2011'	'18-10-1990'
	genero	'masculino'	'masculino'	'feminino'	'masculino'	'nao-binario'	'feminino'	''
Efeito	**Cadastro Realizado**	X						
	Infomação não validada		X	X	X	X	X	X

3. Exercício a partir do capítulo 4 do Livro do Ammann e Offuf

3.1. Cite quais os os tipos de particionamento que são discutidos no capítulo?

Os tipos de particionamento discutidos no capítulo 4 do Livro do Ammann e Offut são o particionamento baseado em **interface** e o particionamento baseado em **funcionalidade**.

3.2. O que caracteriza cada um dos tipos.

O **Particionamento baseado em interface** extrai características a partir dos parâmetros de um método, é de fácil abstração por isso leva a um particionamento mais direto e pode ser parcialmente automatizado, vale também ressaltar que esta técnica tem um baixo custo de tempo. Já o **Particionamento baseado em funcionalidade** define características de acordo com a funcionalidade, dessa forma, se torna mais complexa a identificação de fatores que influenciam no comportamento do código, no entanto, o este tipo de estratégia resulta em melhores testes e classes mais bem definidas.

3.3 Defina quais os casos de testes (apenas os valores, nao precisa implementar a classe JUnit) para o exemplo abaixo utilizando cada um dos tipos de particionamento do dominio de entrada:

```
public class RecursosHumanos {
      public static int NAO_EMPREGAR = 0;
```

```
      public static int EMPREGAR_TEMPO_PARCIAL = 1;
      public static int EMPREGAR_TEMPO_INTEGRAL = 2;

/*Este método indica se a pessoa cadastrada deve ser empregado ou não
pela empresa. E se sim indica se será tempo integral ou parcial. O
método lança a exceção IdadeInvalidaException caso seja passado como
parâmetro uma idade negativa. */

public int avaliaEmpregabilidade(int idade, boolean cursoSuperior)
throws IdadeInvalidaException {
   int status = -1;
   if (idade < 0){
      throw new  IdadeInvalidaException();
   }

   if(0 < idade < 17){
      status = NAO_EMPREGAR;
   }

   if(17=< idade < 22){
      status = EMPREGAR_TEMPO_PARCIAL;
   }

   if(22< idade < 60 && cursoSuperior == true){
      status =  EMPREGAR_TEMPO_INTEGRAL;

   }

   if(22< idade < 60 && cursoSuperior == false){
      status =  EMPREGAR_TEMPO_PARCIAL;
   }

   return status;
   }
}
```

3.3.1 Classe de equivalência com base em interface

	Classe Inválida		Classes Válidas	
idade	`idade < 0`	`idade < 0`	`idade > 0`	`idade > 0`
&	**&**	**&**	**&**	**&**
cursoSuperior	`false`	`true`	`true`	`false`

3.3.1 a) Casos de teste utilizando apenas o critério de classes de equivalência com base em interface

```
@Test (expected = IdadeInvalidaException.class)
public void testIdadeInvalida0() {
avaliaEmpregabilidade(0, false);
}
@Test (expected = IdadeInvalidaException.class)
public void testIdadeInvalida1() {
avaliaEmpregabilidade(-5, true);
}

@Test
public void testIdadeValida0() {
avaliaEmpregabilidade(1, false);
}
@Test
public void testTempoParcial0() {
avaliaEmpregabilidade(2, true);
}
```

3.3.2 b) Casos de teste utilizando o critério de análise do valor limite

```
@Test (expected = IdadeInvalidaException.class)
public void testIdadeInvalida1() {
avaliaEmpregabilidade(-1, true);
}
public void testIdadeInvalida1() {
avaliaEmpregabilidade(-1, false);
}
@Test (expected = IdadeInvalidaException.class)
public void testIdadeInvalida0() {
avaliaEmpregabilidade(0, true);
}
@Test (expected = IdadeInvalidaException.class)
public void testIdadeInvalida0() {
avaliaEmpregabilidade(0, false);
}

@Test
public void testIdadeValida0() {
avaliaEmpregabilidade(1, false);
}
@Test
public void testTempoParcial0() {
avaliaEmpregabilidade(1, true);
}
```

3.3.1 c) Tabela de decisão

Causa	**idade**	idade < 0	idade < 0	idade > 0	idade > 0
	cursoSuperior	false	true	true	false
Efeito	**Valida**			X	X

3.3.2 Classe de equivalência com base em funcionalidade

	Classe I		**Classes V**					
idade	idade <= 0	idade <= 0	1 <= idade <= 16	1 <= idade <= 16	17 <= idade <= 22	17 <= idade <= 22	23 <= idade <= 60	23 <= idade <= 60
&	**&**	**&**	**&**	**&**	**&**	**&**	**&**	**&**
cursoSuperior	false	true	true	false	false	true	false	true

3.3.2 a) Casos de teste utilizando apenas o critério de classes de equivalência com base em interface

```
@Test (expected = IdadeInvalidaException.class)
public void testIdadeInvalida0() {
avaliaEmpregabilidade(0, false);
}
@Test (expected = IdadeInvalidaException.class)
public void testIdadeInvalida1() {
avaliaEmpregabilidade(0, true);
}
// idade <= 16 anos e curso superior?
// assertEquals(0, avaliaEmpregabilidade(15, true), 0.0);
@Test
public void testNaoEmpregavel() {
assertEquals(0, avaliaEmpregabilidade(15, false), 0.0);
}
@Test
public void testTempoParcial0() {
assertEquals(1, avaliaEmpregabilidade(20, false), 0.0);
}
@Test
public void testTempoParcial1() {
assertEquals(1, avaliaEmpregabilidade(20, true), 0.0);
}
@Test
public void testTempoParcial2() {
assertEquals(1, avaliaEmpregabilidade(40, false), 0.0);
}
@Test
```

```
public void testTempoIntegral() {
assertEquals(2, avaliaEmpregabilidade(40, true), 0.0);
}
```

3.3.2 b) Casos de teste utilizando o critério de análise do valor limite

```
@Test (expected = IdadeInvalidaException.class)
public void testIdadeInvalida0() {
   avaliaEmpregabilidade(-1, false);
}
@Test (expected = IdadeInvalidaException.class)
public void testIdadeInvalida1() {
   avaliaEmpregabilidade(0, true);
}
// idade <= 16 anos e curso superior?
@Test
public void testNaoEmpregavel0() {
   assertEquals(0, avaliaEmpregabilidade(1, true), 0.0);
}
@Test
public void testNaoEmpregavel1() {
   assertEquals(0, avaliaEmpregabilidade(2, false), 0.0);
}
@Test
public void testNaoEmpregavel2() {
   assertEquals(0, avaliaEmpregabilidade(15, true), 0.0);
}
@Test
public void testNaoEmpregavel3() {
   assertEquals(0, avaliaEmpregabilidade(16, false), 0.0);
}

@Test
public void testTempoParcial0() {
   assertEquals(1, avaliaEmpregabilidade(17, true), 0.0);
}
@Test
public void testTempoParcial1() {
   assertEquals(1, avaliaEmpregabilidade(18, false), 0.0);
}
@Test
public void testTempoParcial2() {
   assertEquals(1, avaliaEmpregabilidade(21, true), 0.0);
}
@Test
public void testTempoParcial3() {
   assertEquals(1, avaliaEmpregabilidade(22, false), 0.0);
}
```

```
@Test
public void testTempoIntegral0() {
   assertEquals(2, avaliaEmpregabilidade(23, true), 0.0);
}

@Test
public void testTempoParcial4() {
   assertEquals(1, avaliaEmpregabilidade(24, false), 0.0);
}

@Test
public void testTempoIntegral1() {
   assertEquals(2, avaliaEmpregabilidade(59, true), 0.0);
}

@Test
public void testTempoParcial4() {
   assertEquals(1, avaliaEmpregabilidade(60, false), 0.0);
}

@Test (expected = IdadeInvalidaException.class)
public void testIdadeInvalida2() {
   avaliaEmpregabilidade(61, true);
}
```

3.3.2 c) Tabela de Decisão

Causa	**idade**	idade <= 0	idade <= 0	1 <= idade <= 16	1 <= idade <= 16	17 <= idade <= 22	17 <= idade <= 22	23 <= idade <= 60	23 <= idade <= 60
	cursoSuperior	false	true	true	false	false	true	false	true
Efeito	**Não empregar**			X	X				
	Empregar em tempo parcial					X	X	X	
	Empregar em tempo Integral								X