



Laboratório 03

Como usar esse guia:

- Leia atentamente cada etapa
- Quadros com dicas tem leitura opcional, use-os conforme achar necessário
- Preste atenção nos trechos marcados como importante (ou com uma exclamação)

Sumário

Acompanhe o seu aprendizado	2
Conteúdo sendo exercitado	2
Objetivos de aprendizagem	2
Perguntas que você deveria saber responder após este lab	2
Para se aprofundar mais...	2
Agenda de Contatos	3
1. Exibir Menu	4
2. Cadastrar Contato	4
Implementando Contato	5
3. Exibir Contato	6
Implementando exibição do Contato	6
4. Listar Contatos	7
Implementando a listagem dos contatos	7
5. Adicionar e Remover Favoritos	7
6. Testar Agenda	10
Criando testes com o JUnit	13
Preparando o ambiente	13
Configurando o JUnit	14
Escrevendo o primeiro teste (Classe Contato)	14
Entendendo um pouco mais sua classe de teste	14
Criando o objeto a ser usado no teste	14
Bônus 1. Tratar Entradas Inválidas	15
Parâmetros Nulos	16
Parâmetros Inválidos	17
E a posição inválida...?	18
Bônus 2. Novas Funcionalidades.	18
Entrega	19

Acompanhe o seu aprendizado

Conteúdo sendo exercitado

- Classe básica de composição
- Uso do equals
- Introdução a testes de unidade com JUnit
- Introdução a tratamento de erros com exceção

Objetivos de aprendizagem

Ao final desse lab você deve conseguir:

- Reconhecer composição na relação entre objetos em código Java
- Usar delegação para implementar composição entre objetos
- Implementar métodos de igualdade entre objetos (equals em Java)
- Criar testes para programas que lhe ajudem a confiar na sua implementação e a ganhar tempo quando estiver programando
- Usar exceções para tratar situações inesperadas em programas

Perguntas que você deveria saber responder após este lab

- Como se caracteriza o relacionamento entre objetos via composição?
- O que significa dizer que o método equals é um método padrão de Java?
- Em que situações é necessário sobrescrever o método equals?
- O que é uma exceção?
- De que forma podemos usar exceções para lidar com entradas inválidas?
- Toda exceção deve fazer o programa parar?
- Os testes de unidade devem testar a unidade básica de um programa em Java. Liste algumas boas práticas para escrever testes de unidade.
- Em um cenário de composição, como separamos os testes da classe base (composite) e da classe composta?

Para se aprofundar mais...

- Referências bibliográficas incluem:
 - material de referência desenvolvido por professores de p2/lp2 em semestres anteriores ([ONLINE](#))
 - o livro Use a cabeça, Java
 - o livro Java para Iniciantes
 - os livros:
 - Core Java
 - <https://plataforma.bvirtual.com.br/Acervo/Publicacao/1238>
 - Java, Como programar (Deitel)
 - <https://plataforma.bvirtual.com.br/Acervo/Publicacao/39590>
- Tudo que você precisava saber sobre o framework JUnit (<http://junit.org/junit5/>)
- Um olhar mais pontual: javadoc da api do JUnit (<https://junit.org/junit5/docs/5.8.1/api/>)
- Artigo sobre sobrescrever o método [equals](#).

Agenda de Contatos

Neste laboratório, você irá trabalhar no contexto de um sistema para gerenciar seus **contatos**. O sistema deve permitir o cadastro e visualização desses contatos. Um contato é representado por um *nome*, *sobrenome* e um telefone. Deve ser possível listar todos os contatos, exibindo o nome completo do contato e sua posição na lista. Também deve ser possível ver detalhes de um contato (a partir da posição do contato na lista).

Além da funcionalidade de listagem, há a funcionalidade de cadastro onde é passado os detalhes a serem inseridos e a posição que ele deve ser inserido. O sistema está **limitado em 100 contatos**.

A seguir descreveremos as funcionalidades do projeto.

Dicas - O que fazer nas situações que NÃO ESTÃO especificadas?!

Resposta: Se não foi especificado, não precisa fazer. Faça o mais simples

Quando não está especificado o que fazer, você é livre para fazer o que quiser. A dica que podemos dar é... não implemente. Ser preguiçoso tem sua vantagem. Imagine que você está desenvolvendo uma Agenda para um cliente, e você colocou o *email* nos contatos da Agenda. Três coisas podem acontecer:

- O cliente não gostou da ideia, e você terá perdido tempo programando o email;
- O cliente gostou da ideia, mas quer que você faça de um jeito diferente;
- O cliente gostou da ideia e gostou do jeito que você fez.

A última alternativa é praticamente impossível de acontecer... As outras duas implicam em retrabalho. Caso você acabe o projeto antes, dedique seu tempo a: testar, melhorar a qualidade do código e documentar o que foi feito.

Neste laboratório você partirá do código de um colega *que começou a implementação das funcionalidades descritas aqui, mas não acabou*. O [código está disponível aqui](#). A ideia deste código inicial é facilitar seu desenvolvimento e praticar um pouco a leitura e entendimento de programas. O código contém 3 classes:

- Agenda.java: uma versão bem simples de uma agenda onde os contatos são, de fato, strings. Você deve modificar essa classe tanto para refletir todas as funcionalidade de agenda descritas neste lab, quanto para refletir uma representação mais adequada para os contatos
- MainAgenda.java: uma versão bem simples de uma interface com usuário para a classe Agenda. Note que existe um código base bem interessante sobre manipulação de menus aqui. Entretanto, ele está incompleto, especialmente, no que se refere às funcionalidades da Agenda que essa classe irá usar.
- LeitorDeAgenda.java: essa classe lê dados de um arquivo .csv que contém dados de um contato. Observe que esse arquivo não contém todas as informações que desejamos para um contato, mas somente dados iniciais como nome, sobrenome e telefone. O leitor vai carregar esses dados do arquivo e pedir para serem cadastrados na nossa Agenda. A ideia é que você não precise mudar nada nessa classe.

1. Exibir Menu

O sistema deve exibir um menu para o usuário com as opções existentes nesse sistema, como descrito abaixo.

```
(C)adastrar Contato
(L)istar Contatos
(E)xibir Contato
(S)air

Opção>
```

Caso o usuário entre com qualquer valor diferente dos possíveis, deve exibir uma mensagem de opção inválida e exibir novamente o menu e o pedido por uma opção, como no exemplo abaixo.

```
(C)adastrar Contato
(L)istar Contatos
(E)xibir Contato
(S)air

Opção> X
OPÇÃO INVÁLIDA!

(C)adastrar Contato
(L)istar Contatos
(E)xibir Contato
(S)air

Opção>
```

Por fim, a escolha da opção S simplesmente encerra a execução do programa. A funcionalidade de cadastro e listagem serão descritas posteriormente.

2. Cadastrar Contato

O sistema deve permitir o cadastro de contatos, como especificado no exemplo abaixo.

```
(C)adastrar Contato
(L)istar Contatos
(E)xibir Contato
(S)air

Opção> C

Posição> 1
Nome> Ouvidoria
Sobrenome> UFCG
Telefone> (83) 21011585
CADASTRO REALIZADO
```

```
(C)adastrar Contato  
(L)istar Contatos  
(E)xibir Contato  
(S)air  
  
Opção>
```

Importante! Caso o usuário selecione uma posição que já exista, o contato existente será substituído.

Fique atento as seguintes situações de erro:

1. O sistema deve permitir apenas posições válidas (entre 1 e 100, inclusive). O sistema deve exibir a mensagem “POSIÇÃO INVÁLIDA” e exibir novamente o menu de opções caso uma posição inválida seja colocada.
2. Caso o usuário tente cadastrar um contato com nome e sobrenome já existente no sistema, o cadastro deve ser negado, e a mensagem "CONTATO JA CADASTRADO" deve ser exibida. Isto deve acontecer mesmo que o usuário tente cadastrar em uma posição diferente daquela de onde o contato de mesmo nome e sobrenome já está.
3. Caso o usuário tente cadastrar um contato com nome vazio, o cadastro deve ser negado e a mensagem “CONTATO INVALIDO” deve ser exibida.
4. Caso o usuário tente cadastrar um contato com telefone vazio, o cadastro deve ser negado e a mensagem “CONTATO INVALIDO” deve ser exibida.

Implementando Contato

Existem diferentes formas de estruturar e implementar os contatos do sistema. Na disciplina de LP2 você deve pensar mais nas diferentes alternativas que existem entre as diferentes implementações e escolher aquela que seja mais adequada (mais legível, mais fácil de manter, mais barata a curto e longo prazo). Por exemplo, para implementar contatos, você poderia:

- Ter 5 arrays `String[100]`, um para nomes, outro para sobrenomes e outros para telefones
- Ter uma matriz `String[100][5]`, onde cada linha é um contato e as colunas representam nome, sobrenome e telefones
- Ter um `String[500]`, onde para o contato N, a posição $5*N$ representa o nome, $5*N+1$ sobrenome, $5*N+2$ telefone para cada contato na posição
- Criar a classe `Contato`. Nessa alternativa, a Agenda tem um array de contatos (`Contato[100]`) e o `Contato` passa a ser o responsável por ter o seu próprio nome, sobrenome e demais dados.

⚠⚠⚠ Cada uma dessas soluções resolvem o problema, entretanto, é preciso escolher uma delas e esse é o maior desafio de programar grandes sistemas. De acordo com o conteúdo trabalhado na disciplina até o momento, esperamos que você já leve em consideração os conceitos estudados de orientação a objetos e opte pela quarta alternativa => ⚠⚠⚠

Uma vez decidido como representar os contatos, resta implementar o cadastro em si dos contatos quando solicitado pelo usuário.

Um método importante a ser considerado para contato é o *equals*, que permitirá que verifiquemos se dois contatos cadastrados são iguais. Para essa atividade, vamos considerar que dois contatos são iguais se tiverem o mesmo nome (nome e sobrenome). Por exemplo, caso a classe `Contato` tenha o método *equals*, esse método deveria funcionar como descrito no código abaixo:

```

Contato meuContatoUfcg = new Contato("Livia", "Campos", "2101-9999");
Contato meuContatoCel = new Contato("Livia", "Campos", "9973-2999");
Contato outroContatoCel = new Contato("Matheus", "Gaudencio",
"9973-1999");
if( meuContatoUfcg.equals(meuContatoCel)){
    System.out.println("Sou eu, Livia!");
}
if( !meuContatoUfcg.equals(outroContatoCel)){
    System.out.println("Nao eh Livia!");
}
if( !meuContatoUfcg.equals("Oi....")){
    System.out.println("Definitivamente nao eh Livia!");
}

```

3. Exibir Contato

A opção de exibir o contato deve exibir o contato desejado com todos os seus detalhes que tiverem algum dado associado. Caso não haja contato na posição em questão, deve apenas exibir a mensagem “POSIÇÃO INVÁLIDA!” e exibir novamente o menu de opções.

```

(C)adastrar Contato
(L)istar Contatos
(E)xibir Contato
(S)air

```

```

Opção> E
Contato> 1

```

```

Ouvidoria UFCG
(83) 21011585

```

```

(C)adastrar Contato
(L)istar Contatos
(E)xibir Contato
(S)air

```

```

Opção>

```

Implementando exibição do Contato

Aqui, novamente, você deve escolher entre diferentes possibilidades de implementações:

- O código do *main* usa os atributos do contato para gerar a mensagem a ser imprimida
- O contato passa a ter um método que imprime na saída a mensagem adequada representado o contato
- O contato passa a ter um método que retorna o que deve ser imprimido e o menu imprime o que foi retornado pelo contato
- Uma nova classe será criada. Objetos dessa classe recebem um contato e imprimem a saída desejada.
- ...

!!! Qual a solução mais adequada? Quais as vantagens e desvantagens de cada solução? Existem outras soluções melhores? Quando estiver desenvolvendo um código, será bastante comum ter diferentes alternativas de implementação. A melhor solução geralmente é a que vai oferecer uma manutenção mais fácil. Por exemplo, se o usuário decidir mudar a mensagem impressa, onde seria mais fácil modificar? Uma regra boa é não imprimir nada com System.out dentro das classes que não são o **main**. Dessa maneira passamos a ter mais flexibilidade uma vez que podemos usar o valor retornado tanto para ser impresso quanto para qualquer outra operação necessária. !!!

4. Listar Contatos

Seu sistema deve listar todos os contatos existentes na agenda.

```
(C)adastrar Contato
(L)istar Contatos
(E)xibir Contato
(S)air

Opção> L

1 - Ouvidoria UFCG
2 - Coordenacao Computacao UFCG
10 - MC Pedrinho
22 - Fabio Moraes

(C)adastrar Contato
(L)istar Contatos
(E)xibir Contato
(S)air

Opção>
```

Implementando a listagem dos contatos

Todo objeto em Java pode gerar uma representação em String através da implementação do método **public String toString()**. Se sua classe implementa esse método, todo objeto pode ser convertido para String naturalmente pela linguagem Java. Por exemplo, caso a classe Contato tenha o método **toString**, esse método será naturalmente invocado ao realizarmos uma operação como descrita no código abaixo:

```
Contato meuContato = new Contato("Matheus", "Rego", "2101-9999");
System.out.println("Contato " + meuContato);
// a linha acima é equivalente a:
System.out.println("Contato " + meuContato.toString());
```

5. Adicionar e Remover Favoritos

Considere agora que é possível favoritar (ou desfavoritar) contatos para criar uma lista rápida de acesso de 10 posições. A lista de favoritos permite outra forma de acessar seus contatos. Quando você exibe um contato que está na lista de favoritos, é preciso informar que aquele contato é favorito com um coração (❤️).

Para permitir a manipulação de favoritos, é preciso:

- Adicionar a funcionalidade adicionar favorito
- Adicionar a funcionalidade de listar favoritos
- Adicionar a funcionalidade de remover favorito
- Alterar a função de exibição de contato

Veja o exemplo abaixo de funcionamento:

```
(C)adastrar Contato
(L)istar Contatos
(E)xibir Contato
(F)avoritos
(A)dicionar Favorito
(R)emover Favorito
(S)air

Opção> L

1 - Ouvidoria UFCG
2 - Coordenacao Computacao UFCG
10 - MC Pedrinho
22 - Fabio Moraes

(C)adastrar Contato
(L)istar Contatos
(E)xibir Contato
(F)avoritos
(A)dicionar Favorito
(R)emover Favorito
(S)air

Opção> E
Contato> 1

Ouvidoria UFCG
(83) 21011585

(C)adastrar Contato
(L)istar Contatos
(E)xibir Contato
(F)avoritos
(A)dicionar Favorito
(R)emover Favorito
(S)air

Opção> A
Contato> 1
Posicao> 1
```


CONTATO FAVORITADO NA POSIÇÃO 1!

(C)adastrar Contato
(L)istar Contatos
(E)xibir Contato
(F)avoritos
(A)dicionar Favorito
(R)emover Favorito
(S)air

Opção> F

1 - Ouvidoria UFCG

(C)adastrar Contato
(L)istar Contatos
(E)xibir Contato
(F)avoritos
(A)dicionar Favorito
(R)emover Favorito
(S)air

Opção> E

Contato> 1

♥ Ouvidoria UFCG
(83) 21011585

Opção> R

Posicao> 1

(C)adastrar Contato
(L)istar Contatos
(E)xibir Contato
(F)avoritos
(A)dicionar Favorito
(R)emover Favorito
(S)air

Opção> E

Contato> 1

Ouvidoria UFCG
(83) 21011585

Fique atento a dois detalhes de uso da lista de favoritos:

- Se um novo contato for inserido na lista de favoritos em uma posição que já tenha um contato, o antigo contato deixa de ser um favorito.
- O contato só pode aparecer uma vez na lista de favoritos, ou seja, não é possível cadastrar um contato que já exista em alguma posição na lista de favoritos.

6. Testar Agenda

Nosso sistema tem 3 funcionalidades básicas: cadastrar, exibir e listar contatos. Para garantir que você implementou o programa corretamente, **é preciso garantir que cada uma dessas funcionalidades faça o que foi especificado (validação) e garantir que tudo que o software procura fazer, ele faz corretamente (verificação).**

!!! Testar o software é uma das maneiras de garantir a sua corretude. Testar um software é verificar se o software a ser executado com determinadas entradas produz a saída esperada. Até agora costumamos sempre receber essas entradas prontas, mas um bom desenvolvedor deve ser capaz de produzir testes adequados para seu programa. !!!

Um bom teste é aquele que:

- É capaz de encontrar erros no programa;
- É simples;
- Não é redundante.

Para testar a Agenda, nós podemos criar um plano de testes. O plano de testes deve ter: casos de testes, as entradas a serem usadas em cada caso e as saídas esperadas. Um testador desse sistema que esteja focando nos testes de uma classe Agenda que tivesse as funcionalidades sobre controle de contatos. Vejamos um caso de teste para o cadastro de um contato na agenda:

Especificação do Teste	Exemplo de código de teste
<p>1. Cadastrar um novo contato em posição vazia</p> <ul style="list-style-type: none">○ Cadastrar o usuário na posição 1 (vazia)○ Colocar nome "Matheus", sobrenome "Gaudencio" e telefone "(83) 99999-0000"○ A agenda deve ter cadastrado com sucesso	<pre>Agenda agenda = new Agenda(); // o método abaixo não lança exceções agenda.cadastraContato(1, "Matheus", "Gaudencio", "(83) 99999-0000");</pre>

Essa não é a única forma de testar esse código. Caso o cadastroContato retorne um valor booleano ou um inteiro indicando sucesso, é possível ter outras maneiras de testar a especificação acima, veja os exemplos abaixo que faz uso do assert para essa verificação.

<pre>Agenda agenda = new Agenda(); // considerando que o método cadastraContato retorna true caso bem sucedido assert agenda.cadastraContato(1, "Matheus", "Gaudencio", "(83) 99999-0000");</pre>
<pre>Agenda agenda = new Agenda();</pre>

```
// considerando que o método cadastraContato retorna a posição do contato em caso de sucesso
assert agenda.cadastraContato(1, "Matheus", "Gaudencio", "(83) 99999-0000") == 1;
```

!!! O **assert** é um comando reservado em java para verificar se uma expressão é verdadeira. Caso não seja, o código irá falhar. No entanto, 1) o assert só é verificado se foi verdadeiro ou falso quando a jvm é executada com o parâmetro **-ea**; 2) testar códigos apenas usando asserts é bem complicado e não é uma prática comum! !!!

Com um código pronto assim, o desenvolvedor não precisa interagir com a linha de comando, colocar comandos, ou algo do tipo para garantir que seu código está funcionando. Se o desenvolvedor testar o código acima ele consegue, rapidamente, identificar se o código funciona ou não. Se ele fizer alguma alteração nos atributos de Agenda, o desenvolvedor sabe onde está o erro.

TESTES SÃO EXTREMAMENTE IMPORTANTES PARA IDENTIFICAR PROBLEMAS, E GARANTIR QUE O QUE FOI FEITO FUNCIONA COMO VOCÊ ESPERAVA!

Vejamos agora um exemplo de uma boa descrição de casos de teste para o Cadastrar contato focados na classe Agenda. Você vai precisar colocar esses testes em código!


- Para fazer os testes, considere os dados do contato MATHEUS como:
 - Nome: Matheus
 - Sobrenome: Gaudencio
 - Telefone: (83) 99999-0000
- 1. Cadastrar um novo contato em posição vazia
 - Cadastrar os dados de MATHEUS na posição 1 (vazia)
 - A agenda deve ter cadastrado com sucesso
- 2. Cadastrar um novo contato em posição existente
 - Cadastrar os dados de MATHEUS na posição 1 (vazia)
 - Cadastrar os dados "Pedro", "Silva", "(84) 98888-1111" na posição 1
 - A agenda deve ter cadastrado com sucesso
- 3. Cadastrar um novo contato com nome e sobrenome já cadastrados em outra posição
 - Cadastrar os dados de MATHEUS na posição 1 (vazia)
 - Cadastrar os dados de MATHEUS na posição 3 (vazia)
 - A agenda não deve ter cadastrado com sucesso
- 4. Cadastrar um novo contato na posição limite
 - Cadastrar os dados de MATHEUS na posição 100 (vazia)
 - A agenda deve ter cadastrado com sucesso
- 5. Cadastrar um novo contato em uma posição acima do limite
 - Cadastrar os dados de MATHEUS na posição 101
 - A agenda não deve ter cadastrado com sucesso
- 6. Cadastrar um novo contato em uma posição abaixo do limite
 - Cadastrar os dados de MATHEUS na posição 0
 - A agenda não deve ter cadastrado com sucesso
- 7. Cadastrar um novo contato com telefone vazio
 - Cadastrar os dados "Matheus", sobrenome "Gaudencio" e telefone "" na posição 1
 - A agenda não deve ter cadastrado com sucesso
- 8. Cadastrar um novo contato com nome vazio

- Cadastrar os dados "", sobrenome "Gaudencio" e telefone "(83) 99999-0000" na posição 1
- A agenda não deve ter cadastrado com sucesso

É importante observar que, para uma funcionalidade simples como "Cadastrar um novo contato na agenda", temos pelo menos 7 casos de teste diferentes! Algumas observações importantes:

- **TESTE APENAS AQUILO QUE FOI ESPECIFICADO!** Precisa testar se o sobrenome for vazio? Se há letras no telefone? Não. Se a especificação não dita, não é um comportamento que precisa existir.
- **TODO TESTE É INDEPENDENTE!** Sempre comece cada teste do zero. E teste apenas aquilo que é o propósito do teste. Não tente testar 4 funcionalidades diferentes em um único caso de teste pois, em caso de falha, pode dificultar identificar onde é o erro.

Veja agora exemplos de casos de testes esperados da funcionalidade de exibir contato ainda da classe Agenda:

1. Exibir um contato cadastrado com todos os dados
 - a. Cadastrar os dados de MATHEUS na posição 1 (vazia)
 - b. A representação do contato obtido da agenda na posição 1 deve ser:
Matheus Gaudencio
(83) 99999-0000
2. Exibir um contato cadastrado sem o telefone
 - a. Cadastrar os dados de MATHEUS na posição 1 (vazia)
 - b. A representação do contato obtido da agenda na posição 1 deve ser:
Matheus Gaudencio
3. Exibir um contato em uma posição sem contato
 - a. Ao pegar a representação da agenda na posição 100, o sistema deve dar um erro (ou retornar nada).
4. Exibir um contato em uma posição inválida (limite inferior)
 - a. Ao pegar a representação da agenda na posição 0, o sistema deve dar um erro (ou retornar nada)
5. Exibir um contato em uma posição inválida (limite superior)
 - a. Ao pegar a representação da agenda na posição 101, o sistema deve dar um erro (ou retornar nada)
6. Exibir um contato favoritado
 - a. Cadastrar os dados de MATHEUS na posição 1 (vazia)
 - b. Favoritar o usuário da posição 1
 - c. A representação do contato obtido da agenda na posição 1 deve ser:
 Matheus Gaudencio
(83) 99999-0000

Importante: estes são apenas alguns testes para a classe Agenda. Ainda existem outras funcionalidades que poderiam ser testadas, como adicionar e remover favoritos. Além disso, mostramos testes de uma única classe: Agenda. Você ainda precisa testar cada classe individualmente. Seria necessário bolar casos de teste para a classe Contato, por exemplo. Isto é importante pois, se o teste da agenda falhar, pode ser difícil identificar se o erro existe na classe Agenda, ou uma classe usada por Agenda (como a classe Contato). Se Contato tiver seus próprios testes, e eles estiverem passando, provavelmente o problema estará em Agenda. Isto significa que

pode ser necessário testar algo já testado em Agenda, mas agora para Contato. Por exemplo, é interessante testar a exibição/toString de Contato, mesmo que isso seja o valor retornado pela Agenda ao exibir contato.

Resumo:

- **Teste todas as unidades (classes) do seu sistema!**
- **Teste cada funcionalidade de uma classe, mesmo que ela tenha sido usada (e testada) em outra classe!**

Um bom caso de teste é o que testa as situações que podem revelar um erro no programa. Um bom testador é aquele que é capaz de identificar as situações de código que podem gerar erros no programa. São exemplos dessas situações no cadastro: “O cadastro normal de um contato”, “A substituição de um contato já existente”, “O cadastro em posição inválida”.

⚠⚠⚠ Nós desenvolvemos testes que operam nas posições 0, 1, 100 e 101. Essas posições representam VALORES LIMITE da especificação. Um valor limite é aquele que está na borda e representa situações extremas da execução do programa. Pense da seguinte forma: “se o programa funciona para posição 1, ele vai funcionar para posição 2, 3, 4, 5...”. Da mesma forma, as situações que ele provavelmente poderia ter erros seriam aquelas situações limite (posições como 100, 101.. para o nosso programa). Pense em quantas vezes você confundiu o operador “>=” com “>”. ⚠⚠⚠

Programas, mesmo que simples, podem ter 10 ou mais casos de teste por funcionalidade! Nossa agenda poderia ter facilmente 40 casos de teste. Toda vez que alteramos o programa, mesmo que seja para alterar o nome de uma variável, estamos potencialmente inserindo um erro. É importante executar todos os testes cada vez que o programa é alterado.

Felizmente, você não precisa testar manualmente cada um dos casos de teste. Existem bibliotecas e programas que permitem que o programa seja automaticamente testado!

Criando testes com o JUnit

O [JUnit](#) é uma biblioteca (conjunto de códigos) que permite a execução automática de testes de classes. Esses testes mais básicos são conhecidos como testes de unidade.

-- USE SEMPRE O JUNIT 5 --

Preparando o ambiente

O primeiro passo é definir onde no projeto vão ficar os nossos testes. Tipicamente, as classes de teste (chamado de código de teste) ficam em um diretório diferente dos usados para armazenar as classes com o código da aplicação (também chamado de código de produção). Digamos que o programa que você está fazendo está na pasta “agenda” e que o diretório com os pacotes e classes do programa esteja em uma pasta chamada “src” dentro de “agenda” (ou seja, agenda/src). Seria natural colocar os testes em uma pasta testes (ou seja: agenda/testes).

Para isso, no eclipse, clique com o botão direito no projeto e selecione **New > Source Folder**. Nomeie o seu novo diretório fonte para “testes”. O ideal é que a mesma hierarquia de pacotes que existe no pacote do seu projeto exista também no seu diretório de testes. Se, por exemplo, você tem

o pacote principal, com a classe Menu, então o teste dessa classe se chamará MenuTest e ficará no pacote principal dentro da pasta testes. (Não crie esta classe agora.)

Configurando o JUnit

Vá em "build path" do projeto criado (clcando com o botão direito do mouse sobre o projeto, escolha a opção "Build path > Add libraries". Adicione a biblioteca JUnit. Você vai precisar escolher a versão do JUnit. Trabalhe sempre com a versão estável mais recente do JUnit, que no caso é a 5. Finalize esta configuração. O "build path" define o classpath a ser utilizado na compilação do projeto.

Escrevendo o primeiro teste (Classe Contato)

⚠⚠⚠ Clicando com o botão direito sobre o diretório de testes, escolha "**New > JUnit Test Case**". Você vai ter que oferecer informação para que o esqueleto da sua classe de teste seja criado com pouco esforço. A sua classe sob teste (class under test) é a classe Contato. A boa prática de programação sugere que o nome de sua classe de teste seja o mesmo nome da classe sendo testada seguido do nome Test: ContatoTest. Clique em "Next" para continuar a configuração de seu esqueleto de teste. Você vai agora definir que métodos da classe Contato você quer testar. Você quer testar todos os métodos que não sejam muito triviais (exemplos de métodos triviais: um método getNome que retorna nome ou métodos gerados automaticamente pelo eclipse). ⚠⚠⚠

Você já pode rodar o teste que você escreveu clicando com o botão direito do mouse sobre a classe e selecionando "**Run as > JUnit Test**". O esqueleto da classe de teste criada automaticamente vai sempre falhar. Falhas são representadas por uma barra vermelha no término da execução do teste. O próximo passo é implementar os testes para testar cada método da classe.

Entendendo um pouco mais sua classe de teste

Cada método de teste na sua classe de teste começa com uma anotação @Test. Essa anotação diz à JVM que cada método da classe de teste deve testar um aspecto "pequeno" da classe sob teste. Por exemplo, deve haver um método de teste para testar cada método da classe Contato separadamente. Cada método de teste deve ser pequeno e específico.

Os métodos de teste JUnit se utilizam de asserções ("assertions"), que são declarações que checam se uma condição é verdadeira ou falsa. Se a condição é falsa, o teste falha. Quando todas as asserções feitas em um método de teste forem verdadeiras, vai aparecer uma barra verde ao final da execução do caso de teste. "Passar" e "Falhar" são veredictos de um caso de teste.

JUnit oferece muitos métodos de assertion.

Criando o objeto a ser usado no teste

Em todo teste (método anotado com @Test) que exercita uma classe, um ou mais objetos da classe sob teste precisam ser criados. É com base nesse(s) objetos que as asserções são avaliadas. É comum ter um método que cria esses objetos. É o método anotado com @BeforeEach. Veja a seguir:

```
import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.junit.jupiter.api.Assertions.assertNotEquals;
```

```

import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;

public class ContatoTest {

    private Contato contatoBase;

    @BeforeEach
    void preparaContatos() {
        this.contatoBase = new Contato("Matheus", "Gaudencio", "555-5551");
    }

    @Test
    void testNomeCompleto() {
        String msg = "Esperando obter o nome completo";
        assertEquals("Matheus Gaudencio",
this.contatoBase.nomeCompleto(), msg);
    }
}

```

Antes de executar cada método anotado com o `@Test`, o JUnit executa o método anotado com `@BeforeEach` de forma que o contatoBasico é criado a cada teste executado.

Agora é a sua vez de implementar seus testes!

Faça testes para as classes Contato e Agenda.

Dicas - O que não precisa ser testado?	
getAtributo	Em geral, esses métodos não tem lógica complexa (só retornam valor), então não precisam ser testados diretamente.
Métodos privados	Não são testados diretamente, nem devem se tornar métodos públicos para que possam ser testados.

Dicas - Vários casos de testes pressupõem a existência do método equals	
O <i>equals</i> em Contato	Considere que dois contatos são iguais se tiverem o mesmo nome (nome e sobrenome).

Bônus 1. Tratar Entradas Inválidas

Até agora permitimos que o usuário possa colocar qualquer nome, telefone e qualquer entrada. Entretanto, nunca devemos confiar que o usuário sempre vai usar o sistema da maneira correta.

Essa afirmação é forte, mas significa que, independente do sistema, há grande chance do usuário fazer bobagem. Então, devemos tomar o cuidado de validar os dados recebidos.

Por exemplo, o usuário pode acabar não colocando um nome para o contato (nome vazio). Considerando essa situação, você pode pensar em diferentes designs para impedir que isso aconteça:

- Na classe principal, ao receber a entrada de nome, você verifica se o nome é vazio e, se for, não se deve criar o objeto contato;
- Na classe Contato, ao construir o objeto, devemos verificar se o nome é vazio. Se for, o objeto não deve permitir sua criação.

Para saber o que fazer nessa situação, vamos ver primeiro o que é feito em Java.

Parâmetros Nulos

Veja a saída da execução do código abaixo quando criamos um objeto Scanner com um parâmetro nulo:

```
import java.util.Scanner;

public class ExemploConstrutorInvalido {

    public static void main(String[] args) {
        Scanner sc = new Scanner(null);
        System.out.println("O programa vai fechar...");
    }
}
```

Saída:

```
// Error: // Uncaught Exception: Typed variable declaration : Object constructor : at Line: 3 : in file:
<unknown file> : new Scanner ( null )
```

```
Target exception: java.lang.NullPointerException
```

```
java.lang.NullPointerException
    at java.io.StringReader.<init>(StringReader.java:50)
    at java.util.Scanner.<init>(Scanner.java:702)
```

Ao construir um objeto Scanner com o parâmetro null, o Java INTERROMPE a execução e “lança” uma exceção. Observe que a mensagem “O programa vai fechar...” não aparece pois o programa não chega a executar essa linha de código.

É muito comum encontrar no código do Java o seguinte código em construtores e métodos:

```
public String next(Pattern pattern) {
    if (pattern == null) {
        throw new NullPointerException();
    }
    ...
}
```


!!! Uma exceção representa uma situação de erro no sistema. O throw é a palavra chave em Java que “lança” uma exceção. Quando uma exceção dessa natureza acontece, é porque o usuário tentou fazer algo com o sistema que, caso ele continuasse executando, apenas ocasionaria mais erros ao sistema. O sistema, ao lançar uma exceção, para de executar e imprime uma mensagem com esse erro. É importante observar que uma exceção também é um objeto (new `NullPointerException()`). !!!

Altere seu programa de forma que o mesmo não aceite argumentos null no construtor de `Contato`. Caso um argumento `null` seja passado, seu programa deve lançar uma exceção `NullPointerException`. Crie o teste associado para garantir que a exceção está sendo de fato lançada. Para isso, basta usar uma notação especial do JUnit, como mostra o código abaixo:

```
@Test
public void testNomeNull() {
    try {
        Contato contatoInvalido = new Contato(null, "Gaudencio",
        "21010000");
        fail("Era esperado exceção ao passar código nulo");
    } catch (NullPointerException npe) {

    }
}
```

Você pode melhorar a mensagem que aparece durante uma exceção, bastando para isso criar o objeto `NullPointerException` com a mensagem como parâmetro. Exemplo: “`throw new NullPointerException(“Nome nulo”);`”.

Parâmetros Inválidos

Entretanto, existem parâmetros inválidos além de nulos. Por exemplo, e se o nome do contato for criado com uma string vazia? Ou se for uma string só composta por espaços? Nesta situação, o objeto em questão não é nulo! Entretanto, esse parâmetro não representa o nome de uma pessoa.

Nesta situação, os objetos em Java costumam lançar uma exceção chamada `IllegalArgumentException`. Por exemplo, no método abaixo, utilizado durante a seleção de um intervalo de um array (classe `Arrays`), o Java verifica se o índice inicial (`fromIndex`) é menor ou igual ao índice final do intervalo (`toIndex`). Quando esta situação não é respeitada, uma exceção é lançada.

```
private static void rangeCheck(int length, int fromIndex, int toIndex) {
    if (fromIndex > toIndex) {
        throw new IllegalArgumentException(
            "fromIndex(" + fromIndex + ") > toIndex(" + toIndex + ")");
    }
    ...
}
```

Faça que seu programa lance `IllegalArgumentException` quando os contatos forem construídos com objetos `Strings` não-nulos, porém inválidos (nessa situação, strings vazias ou composta apenas por espaços).

E a posição inválida...?

Observe que quando o usuário coloca uma posição inválida, nós não interrompemos a execução do programa! Ou seja, o sistema não para sua execução quando o usuário coloca uma posição inválida.

Essa é uma situação esperada e que permite recuperação. Nessa situação, não lançamos uma exceção, mas simplesmente inserimos essa situação dentro do fluxo do programa (condição a ser tratada num else, por exemplo).

Existem ainda situações que parte do código pode lançar exceções, mas que o programador não quer que o programa pare de executar. Nestas situações, nós precisamos capturar e tratar as exceções lançadas. Exploraremos isto em situações futuras.

Dicas - Algumas outras exceções de Java e seus significados...	
ArithmeticException	Operação aritmética inválida (divisão por zero)
ClassCastException	O objeto não é da classe adequada
IllegalArgumentException	O parâmetro do método/construtor não é válido
IndexOutOfBoundsException	O índice utilizado foi maior ou menor que os limites do array
NoSuchElementException	O elemento desejado não existe
NumberFormatException	O formato do número em questão é inválido
UnsupportedOperationException	A operação desejada não é suportada/permitida.

Bônus 2. Novas Funcionalidades.

Vamos deixar a brincadeira mais divertida...

- Incrementando a classe Contato
 - O telefone pode ser editado agora (Mudar Telefone)
 - Você pode adicionar tags a um ou mais contatos; considere que cada contato terá um número máximo de 5 tags; a mesma tag pode ser adicionada a mais de um contato por vez.
- Incrementando Agenda
 - Você pode agora remover um contato da agenda
 - A remoção do contato da agenda, remove-o também da lista de favoritos
 - Remover o contato significa deixar a posição que o contato ocupa, na agenda, vazia
 - Você pode ter outras formas de consultar um contato na agenda
 - Pelo nome: retorna uma representação textual de todos os contatos que apresentam o mesmo nome que o especificado
 - Pelo sobrenome: retorna uma representação textual de todos os contatos que apresentam o mesmo sobrenome que o especificado

Entrega

Faça um programa de Agenda que:

- Cadastre contatos
- Exiba detalhes de um contato
- Imprima a lista de contatos
- Adicione, remova e imprima favoritos

conforme o que está descrito nas seções 1-5 da especificação acima e tenha testes de unidade feitos com JUnit.

Bônus: Seu programa deve parar de executar e lançar uma exceção quando o contato for criado com uma entrada inválida (nulo ou espaço vazio para qualquer um dos campos).

É importante que todo código esteja devidamente documentado, à exceção das classes de testes (mas se quiser documentar, e recomendamos, pode ficar à vontade).

Ainda, você deve entregar um programa com testes para as classes com lógica testável (todas as classes menos a classe de interface com o usuário). **IMPORTANTE! NÓS IREMOS AVALIAR SEU CÓDIGO A PARTIR DOS TESTES!** Nós não executaremos a sua interface por linha de comando várias vezes, mas pelo contrário, avaliaremos se você fez bons testes, e qual o resultado da execução desses testes!

Faça bons testes, que explorem as condições limite.

Para a entrega, faça um **zip** da pasta do seu projeto. Coloque o nome do projeto para: LAB3_SEU_NOME e o nome do zip para LAB3_SEU_NOME.ZIP. Exemplo de projeto: LAB3_MATHEUS_GAUDENCIO.ZIP. Este **zip** deve ser submetido pelo Canvas.

Seu programa será avaliado pela corretude e, principalmente, pelo DESIGN do sistema. É importante:

- Usar nomes adequados de variáveis, classes, métodos e parâmetros.
- Fazer um design simples, legível e que funciona. É importante saber, apenas olhando o nome das classes e o nome dos métodos existentes, identificar quem faz o que no código.