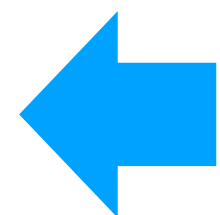


# Programação Orientada a Objetos

Victor Machado da Silva, MSc  
victor.silva@professores.ibmec.edu.br

# Índice

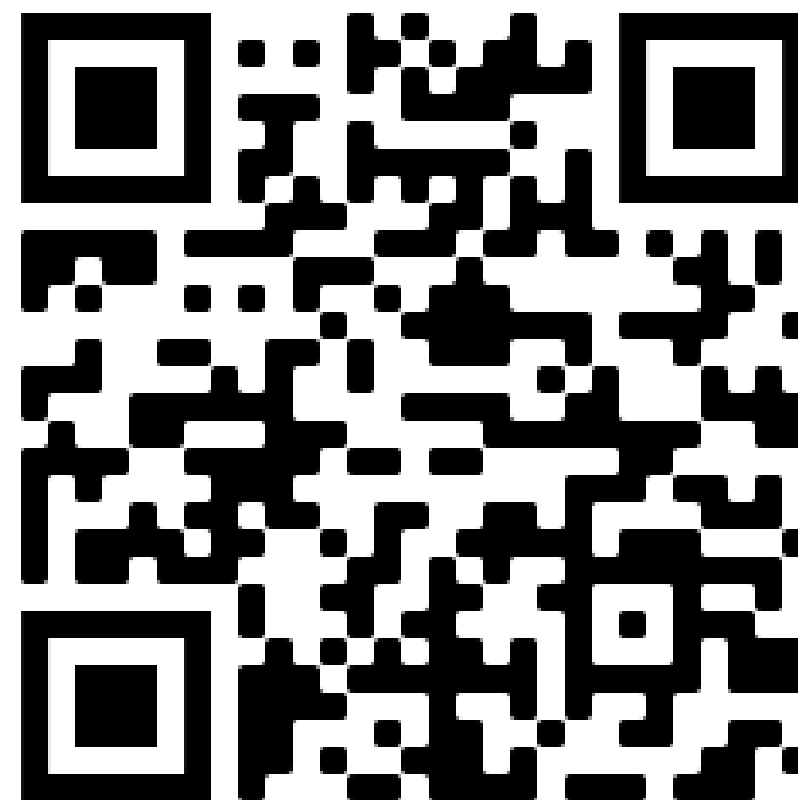
- [Apresentação do curso](#)
- [Configurando o ambiente](#)
- [Sobre paradigmas de programação](#)
- [Git](#)
- [Fundamentos da OO](#)
- [Introdução a Java](#)
- [Conceitos estruturais](#)



# Apresentação do curso

# Apresentação do curso

- Contato: [victor.silva@professores.ibmec.edu.br](mailto:victor.silva@professores.ibmec.edu.br)
- Aulas às segundas e quartas-feiras, de 7:30 às 9:20
- Grupo no Whatsapp: <https://chat.whatsapp.com/BCQDk3VSc4f0BjN2vbtpsa>
- Material no GitHub: <https://github.com/victor0machado/2021.1-progoo>



# Apresentação do curso

Aula	Dia	Dia sem.	Tópico
01	22/02/2021	seg	Introdução à disciplina
02	24/02/2021	qua	Sobre paradigmas de programação / Introdução ao Git
03	01/03/2021	seg	Fundamentos de OO: abstração, reuso, encapsulamento
04	03/03/2021	qua	Conceitos iniciais de Java: parte 1
05	08/03/2021	seg	Conceitos iniciais de Java: parte 2
06	10/03/2021	qua	Conceitos iniciais de Java: parte 3
07	15/03/2021	seg	Conceitos estruturais: classe, atributo, método
08	17/03/2021	qua	Conceitos estruturais: objeto
09	22/03/2021	seg	Noções de UML: introdução
10	24/03/2021	qua	Noções de UML: diagramas de classes
11	29/03/2021	seg	Conceitos relacionais: herança e polimorfismo
12	31/03/2021	qua	Conceitos relacionais: associação
13	05/04/2021	seg	Conceitos relacionais: interface
14	07/04/2021	qua	SEM AULA (SEMANA AP1)
15	12/04/2021	seg	SEM AULA (SEMANA AP1)
16	14/04/2021	qua	SEM AULA (SEMANA AP1)
17	19/04/2021	seg	Conceitos organizacionais: pacotes
18	21/04/2021	qua	SEM AULA (TIRADENTES)
19	26/04/2021	seg	Conceitos organizacionais: visibilidades
20	28/04/2021	qua	Noções de UML: diagramas de caso de uso

# Apresentação do curso

Aula	Dia	Dia sem.	Tópico
21	03/05/2021	seg	Noções de UML: diagramas de atividades
22	05/05/2021	qua	Noções de UML: diagramas de sequência
23	10/05/2021	seg	Boas práticas da OOP: parte 1
24	12/05/2021	qua	Boas práticas da OOP: parte 2
25	17/05/2021	seg	Boas práticas da OOP: parte 3
26	19/05/2021	qua	Princípios SOLID: parte 1
27	24/05/2021	seg	Princípios SOLID: parte 2
28	26/05/2021	qua	Princípios SOLID: parte 3
29	31/05/2021	seg	Princípios SOLID: parte 4
30	02/06/2021	qua	Princípios SOLID: parte 5
31	07/06/2021	seg	Design smells
32	09/06/2021	qua	Métricas de código
33	14/06/2021	seg	Frameworks Java
34	16/06/2021	qua	Frameworks Java
35	21/06/2021	seg	Frameworks Java
36	23/06/2021	qua	SEM AULA (SEMANA AP2)
37	28/06/2021	seg	SEM AULA (SEMANA AP2)
38	30/06/2021	qua	SEM AULA (SEMANA AP2)
39	05/07/2021	seg	SEM AULA (SEMANA AS)
40	07/07/2021	qua	SEM AULA (SEMANA AS)

# Apresentação do curso

## Avaliação

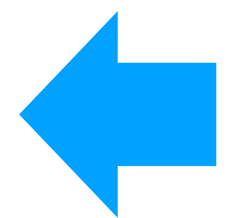
- Proporção:
  - Exercícios periódicos (AC): 20%
  - Projeto (AP1): 40%
  - Projeto (AP2): 40%
- Detalhes das entregas:
  - Exercícios da AC devem ser individuais
  - Projetos de AP1 e AP2 em grupos de no mínimo 2 e no máximo 3 pessoas
- AS será uma prova com consulta, que substituirá a menor nota entre AP1 e AP2.
- Os trabalhos serão *commitados* no GitHub em um *branch* separado do *master*, em um repositório privado criado pelos alunos, e um *pull request* deverá ser enviado para avaliação.



# Sugestões de materiais para estudo

- Thiago Leite e Carvalho - Orientação a Objetos: Aprenda seus Conceitos e suas Aplicabilidades de Forma Efetiva (Casa do Código, 2016)
- Apostila Java e Orientação a Objetos (Caelum - <https://www.caelum.com.br/apostilas>)
- Curso Java Completo (DevDojo - [https://www.youtube.com/watch?v=kkOSweUhGZM&list=PL62G310vn6nHrMr1tFLNOYP\\_c73m6nAzL](https://www.youtube.com/watch?v=kkOSweUhGZM&list=PL62G310vn6nHrMr1tFLNOYP_c73m6nAzL))
- Joshua Bloch - Java Efetivo: As Melhores Práticas para a Plataforma Java (Alta Books, 2019)
- Maurício Aniche - Orientação a Objetos e SOLID para Ninjas: Projetando Classes Flexíveis (Casa do Código, 2015)



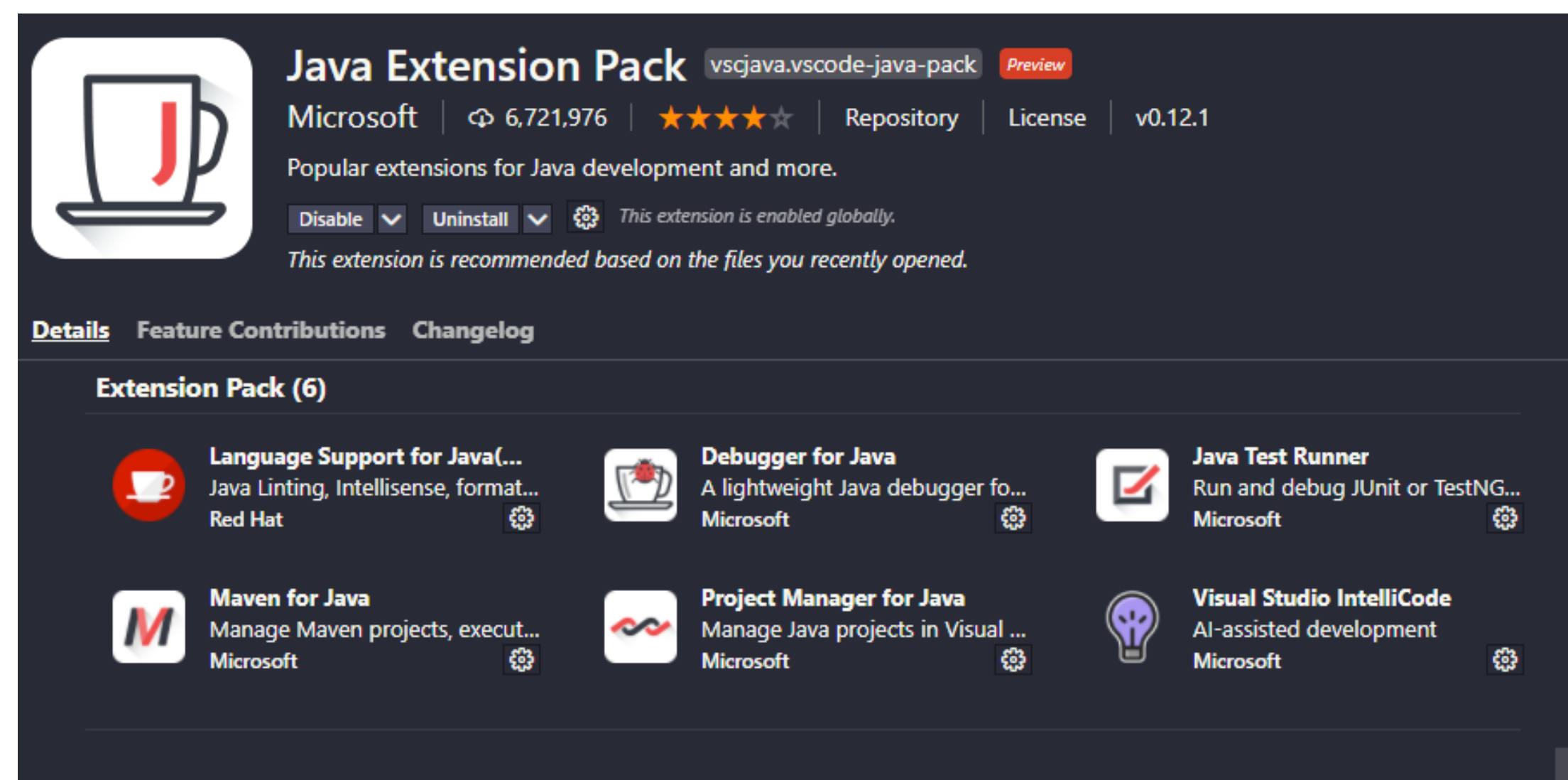


# Configurando o ambiente

# Configurando o ambiente

## IDE:

- Neste curso usaremos o **VSCode** como IDE principal para o desenvolvimento de código. Caso você ainda não tenha, sugiro dar uma olhada nos slides da disciplina de Programação (link [aqui](#)) e fazer a instalação e configuração inicial do software.
- Dentro do VSCode, será necessário instalar a *Java Extension Pack*, um pacote organizado pela própria Microsoft, com diversas extensões específicas para Java.



# Configurando o ambiente

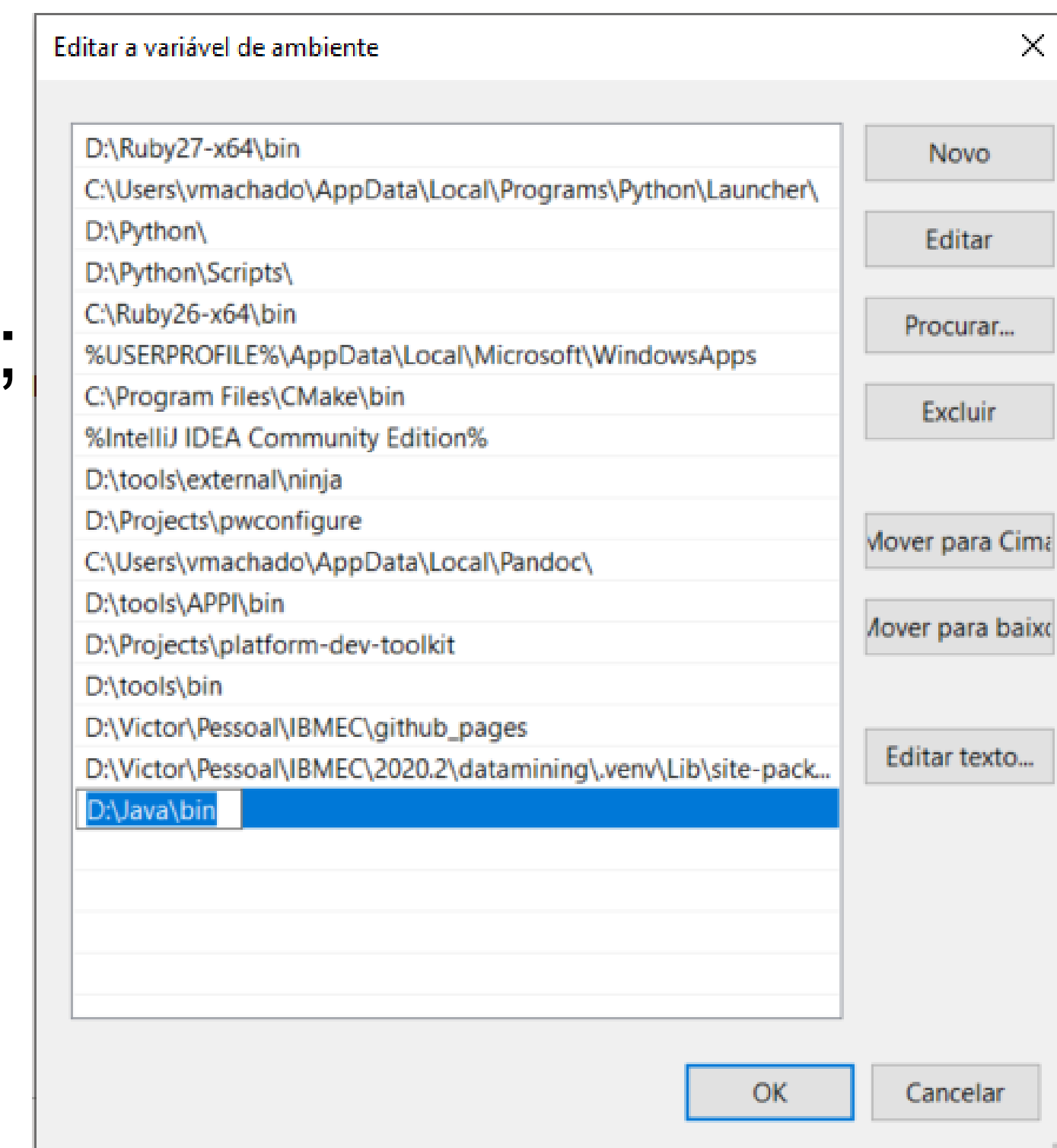
## Java:

- Normalmente, todo mundo possui o Java instalado no computador. No entanto, caso você não tenha, clique [aqui](#) para acessar o site do Java, baixar o JRE (*Java Runtime Environment*) e instalá-lo;
- Além do JRE, será necessário instalar o JDK (*Java Development Kit*), plataforma de desenvolvimento para Java. Acesse [esse link](#), baixe a versão mais recente do Java SE (atualmente, é a versão 15) e instale o software;
- Como sugestão, recomendo instalar em um caminho curto, como por exemplo *D:\Java*, para facilitar o acesso. Após a instalação, coloque a pasta “bin” do diretório Java no seu PATH, para poder acessar o compilador pela linha de comando (ver mais informações no próximo slide).

# Configurando o ambiente

Como adicionar um caminho ao PATH:

- Aperte a tecla do Windows e pesquise por “Editar as variáveis de ambiente para sua conta”;
- Na janela que abrir, procure pela linha com “Path”, selecione-a e clique em “Editar...”;
- Clique em “Novo” e insira o caminho desejado;
- Aperte “Ok” e “Ok” novamente para fechar tudo;
- Para confirmar se está tudo certo, abra um prompt de comando novo e chame um arquivo ou executável que está contido na pasta (no caso da pasta “bin”, chame “javac”)



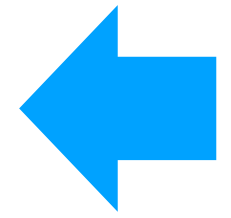
# Configurando o ambiente

## Git:

- Para o desenvolvimento dos trabalhos e acompanhamento da disciplina, vamos utilizar o Git para versionar os materiais;
- Para instalar o Git na máquina, acesse [esse site](#) e baixe o instalador. O processo de instalação é direto, basta clicar em “Next” para concluir a instalação;
- Também será necessário criar uma conta no GitHub ([www.github.com](https://www.github.com)). Caso queira, já pode me procurar por lá (victor0machado).

## Outros programas:

- Recomendo instalar o [notepad++](#) para um bom editor de texto simples.

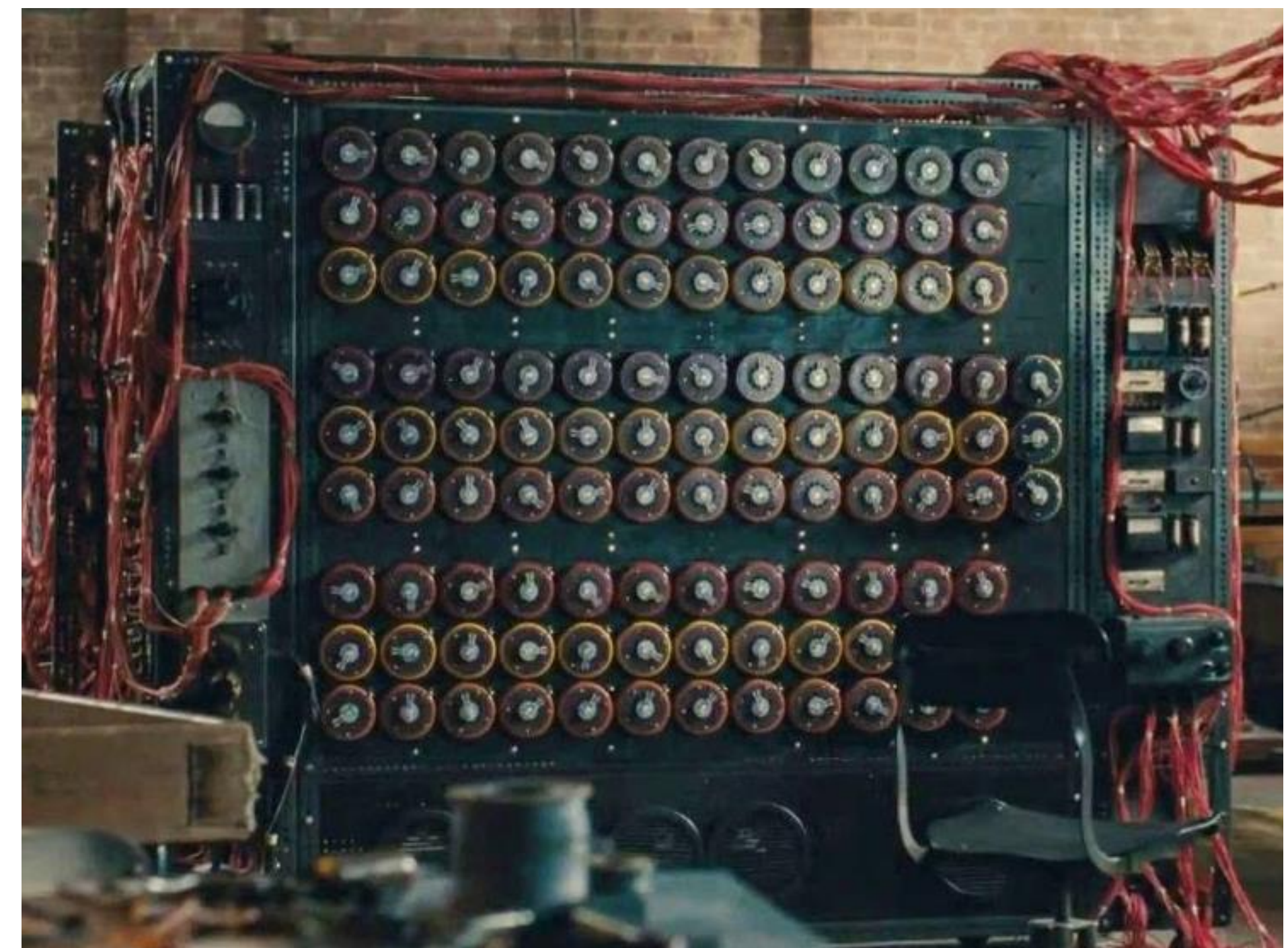


# Sobre paradigmas de programação



# Um pouco de história...

Apesar de termos registros históricos de computadores mecânicos e eletromecânicos há bastante tempo, os primeiros computadores próximos aos que conhecemos hoje começaram a surgir na década de 1930, com Alan Turing (1912-1954) e sua definição de uma **máquina universal**, ou **máquina de Turing**, um conceito abstrato de um computador, no qual pode-se modelar qualquer computador digital.





# Um pouco de história...

Durante as décadas de 1930 e 1940 surgiram diversos computadores a válvula, que ocupavam o espaço de salas inteiras e chegavam a pesar toneladas. Kits para computadores pessoais e computadores comerciais começaram a surgir na década de 1950, todos eles programáveis através de cartões que eram perfurados pelo usuário e lidos pelas máquinas, que então faziam as operações indicadas.

Até esse ponto, a memória interna de um computador era extremamente limitada. O UNIVAC, um dos primeiros computadores comerciais do mundo, tinha memória interna de 1.000 palavras de 12 caracteres. Essa memória ainda não era eletrônica, sendo utilizados tubos de mercúrio para armazenar as informações.



# Um pouco de história...

A primeira linguagem de programação de alto nível implementada para um computador foi o FORTRAN, em 1957, inicialmente para o IBM 704, que já possuía memória magnética.

O primeiro compilador para FORTRAN possuía 25 mil linhas de código (de máquina). Toda máquina IBM 704 vinha com uma fita magnética contendo o compilador e um manual de 51 páginas.

Como esperado, essa linguagem era muito limitada para nossos padrões atuais. Variáveis só podiam possuir um ou dois caracteres.

O FORTRAN evoluiu e foi durante mais de 30 anos a principal linguagem de programação científica e ainda hoje é usada em muitas áreas onde se necessita de alta precisão e eficiência.



# Um pouco de história...

Com o surgimento dos circuitos integrados, de computadores com cada vez mais memória e a popularização dos dispositivos para além das grandes universidades e empresas, foram surgindo outras linguagens, como:

- COBOL (1960)
- BASIC (1964)
- Pascal (1970)
- Smalltalk (1971)
- C (1972)
- PROLOG (1972)



# Um pouco de história...

A linguagem Java surgiu bem depois, em 1995, depois até de Python, que surgiu pela primeira vez em 1991. Quando a linguagem surgiu, os computadores pessoais e os sistemas operacionais já estavam difundidos no mundo inteiro.

O projeto surgiu em 1991, com a perspectiva de que a próxima grande tendência em desenvolvimento de sistemas seria a junção de computadores com dispositivos digitais. Não havia ainda linguagens adequadas para esse tipo de desenvolvimento.

A nova linguagem foi demonstrada em um dispositivo interativo portátil projetado para a indústria de TV a cabo. Esse conceito de dispositivos móveis ainda demoraria muitos anos até se tornar economicamente viável, portanto pode-se dizer que Java foi uma linguagem criada muito à frente do seu tempo.



# Um pouco de história...

Java foi, de fato, a linguagem base que permitiu o surgimento dos dispositivos móveis. Até 2017, Java era a linguagem oficial do sistema Android, e o Kotlin, atual linguagem oficial, foi desenvolvido para que tenha uma interoperabilidade com Java.

De acordo com o índice TIOBE, principal medidor de uso de linguagens de programação, inclui Java em primeiro ou segundo lugar há 20 anos, dividindo o pódio com C, C++ e, desde 2019, Python.



# Paradigmas de programação

Um paradigma, por definição, é um conceito que define um exemplo típico ou modelo de algo. É a representação de um padrão a ser seguido.

Um **paradigma de programação** é uma forma de classificação de linguagens de programação, baseada nas suas funcionalidades. Um paradigma é, portanto, um tipo de estruturação ao qual a linguagem deverá respeitar.

Cada paradigma surgiu de necessidades diferentes. Dado isso, cada um apresenta maiores vantagens sobre os outros dentro do desenvolvimento de determinado sistema. Sendo assim, um paradigma pode oferecer técnicas apropriadas para uma aplicação específica.

# Paradigmas de programação

Paradigma imperativo ou procedural:

- As instruções devem ser passadas ao computador na sequência em que devem ser executadas;
- Como linguagens mais conhecidas temos COBOL, FORTRAN e Pascal;
- O código programado através desse paradigma é uma espécie de passo-a-passo dos procedimentos que a máquina deverá executar;
- A solução do problema será muito dependente da experiência e criatividade de quem trabalha com a programação - foco no “como”;
- Recomendado em projetos onde não se espera que haja mudanças significativas com o tempo;
- É eficiente e permite uma modelagem tal qual o mundo real, porém o código tende a ser de difícil legibilidade.

# Paradigmas de programação

Paradigma declarativo:

- Foca mais no “quê” deve ser resolvido, ao invés do “como”;
- Nível de abstração é maior;
- O foco deixa de ser como o resultado vai ser computado, e sim em como funciona a sequência lógica e qual o resultado esperado;
- Como exemplos de linguagens declarativas, temos Lisp, Prolog e Haskell.

# Paradigmas de programação

## Paradigma funcional:

- O uso de funções é destaque;
- O programa é dividido em blocos e, para sua resolução, são implementadas funções que definem variáveis em seu escopo e retornam algum resultado;
- São exemplos de linguagens o Lisp, o Scheme e o Haskell;
- É bastante indicado quando a solução requerida é fortemente dependente de uma base matemática;
- O paradigma funcional tem alocação de memória automática, eliminando possíveis “efeitos colaterais” nos cálculos matemáticos das funções.

# Paradigmas de programação

## Paradigma lógico:

- Deriva do paradigma declarativo;
- Utiliza formas de lógica simbólica como padrões de entrada e saída, para realizar inferências para produzir os resultados;
- Entre as linguagens que usam esse paradigma, podemos citar Mercury e Prolog;
- São utilizadas na solução de problemas que envolvem inteligência artificial, criação de programas especialistas e comprovação de teoremas.



# Paradigmas de programação

Paradigma orientado a objetos:

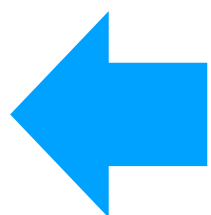
- Foi popularizado na década de 1990 com a linguagem Java, apesar de existir há mais tempo, em linguagens como Smalltalk e C++;
- Permite uma programação multiplataforma de uma mesma maneira;
- Apoia-se nas abstrações de classes e objetos ao tentar retratar a programação tal qual se enxerga o mundo real;
- Todos os objetos têm determinados estados e comportamentos. Esses estados são descritos pelas classes como atributos, e a forma como os objetos se comportam é definida por meio de métodos, que são equivalentes às funções do paradigma funcional;
- Três alicerces básicos: herança, polimorfismo e encapsulamento.



# Paradigmas de programação

Paradigma orientado a eventos:

- Usado por toda linguagem de programação que tem uso de recursos gráficos, como jogos e formulários;
- Dessa forma, a execução do programa se dá à medida que determinados eventos são disparados pelo usuário;
- Como exemplos, temos Visual Basic, C# e Delphi.



# Git

# Introdução

- Talvez você reconheça esse nome por causa de sites como *github* ou *gitlab*. **Git** é um sistema *open-source* de controle de versionamento.
- Versionar projetos é uma prática essencial no mundo profissional e, em particular, na área de tecnologia, para manter um histórico de modificações deles, ou poder reverter alguma modificação que possa ter comprometido o projeto inteiro, dentre outras funcionalidades que serão aprendidas na prática.
- Os projetos são normalmente armazenados em **repositórios**.
- Vamos aqui falar de alguns conceitos principais sobre como funciona o sistema, independente da plataforma que vamos utilizar (p.ex., *github*). Em seguida vamos aplicar alguns desses conceitos na prática.

# Para mais informações...

- O tempo que temos em aula não é suficiente para conseguirmos discutir todos os detalhes por trás dessa tecnologia. Portanto, seguem abaixo algumas sugestões de conteúdos extras para estudarem:
  - <http://git-scm.com/book/en/Getting-Started-About-Version-Control>
  - <http://git-scm.com/book/en/Getting-Started-Git-Basics>
  - <http://learngitbranching.js.org/>
  - <http://try.github.io/levels/1/challenges/1>

# O que é Controle de Versões?

- Controle de Versões é um sistema de grava mudanças aplicadas em um arquivo ou um conjunto de arquivos ao longo do tempo, para que o usuário possa lembrar ou recuperar depois.
- Na área de tecnologia o controle de versões já é algo consolidado, uma vez que inúmeras alterações são aplicadas em um software durante a sua implementação e manutenção. No entanto, adotar um sistema de controle de versões (VCS, da sigla em inglês) é algo recomendado para qualquer área.
- Quando se quer adotar um VCS, normalmente o primeiro passo é trabalhar com um controle local, fazendo cópias dos arquivos e os renomeando com algum padrão (p.ex., incluindo a data ao final do nome do arquivo).
- O Git veio para otimizar esse controle de versões, permitindo inúmeras alterações que seriam muito complicadas se fossem feitas manualmente.

# Uma breve história do Git

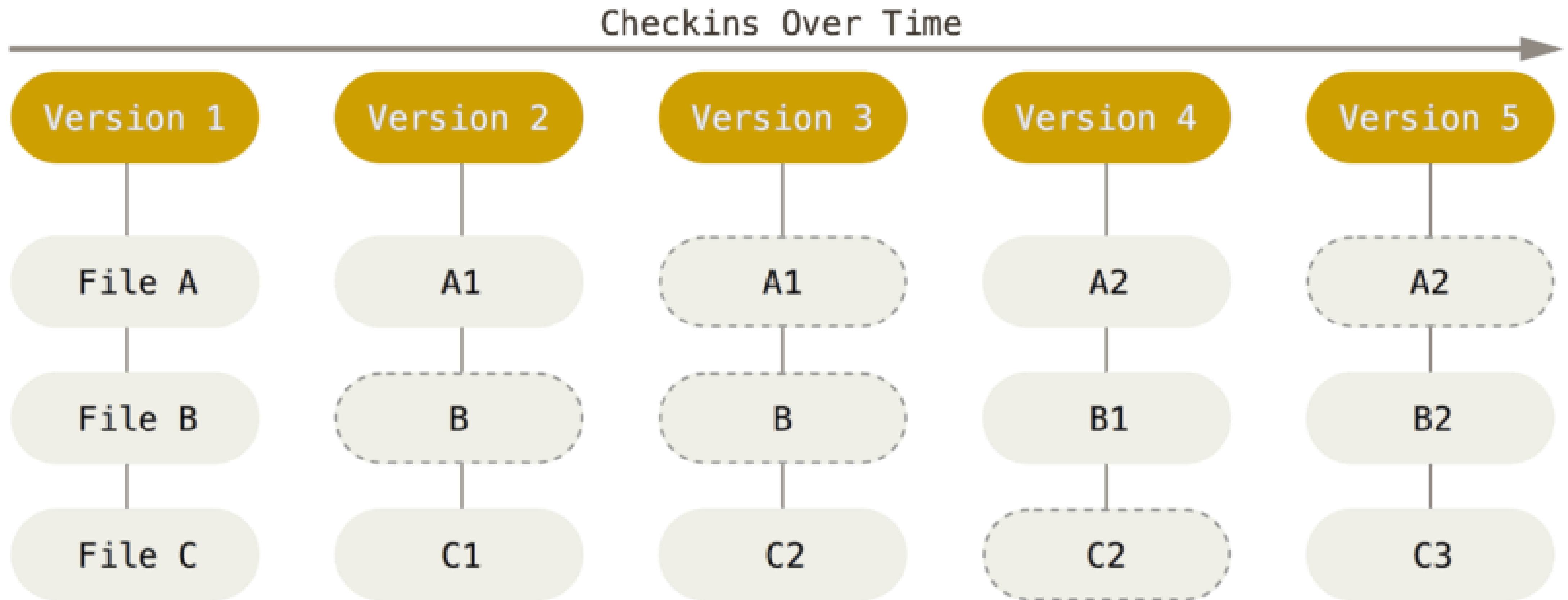
- A criação do Git está atrelada à criação do Linux, quando o time responsável utilizou uma solução de controle de versão que, eventualmente, tornou-se paga.
- A comunidade Linux, portanto, começou a planejar um sistema que aproveitasse algumas das características da solução anterior, porém evoluindo diversos aspectos.
- A comunidade tinha os seguintes objetivos para o novo sistema:
  - Velocidade
  - Design simples
  - Suporte forte para desenvolvimento não-linear, com milhares de atividades sendo realizadas em paralelo
  - Completamente distribuído, permitindo o acesso de qualquer lugar
  - Eficiente no suporte a grandes projetos



# O que é Git?

- **Git** funciona pensando que os dados de um repositório compõem uma série de “fotografias” de um sistema de arquivos em miniatura.
- No Git, a cada vez que você aplica uma alteração (ou *commit*), ou salva o estado do seu projeto, o Git basicamente tira uma “foto” de como os arquivos do repositório estão naquele momento e então ele armazena uma referência a essa foto.
- Para ser eficiente, se os arquivos não foram alterados, o Git não altera os arquivos novamente, apenas um link para a última versão do arquivo armazenada.
- Sendo assim, o Git trabalha os dados como um **fluxo de fotografias**.

# O que é Git?



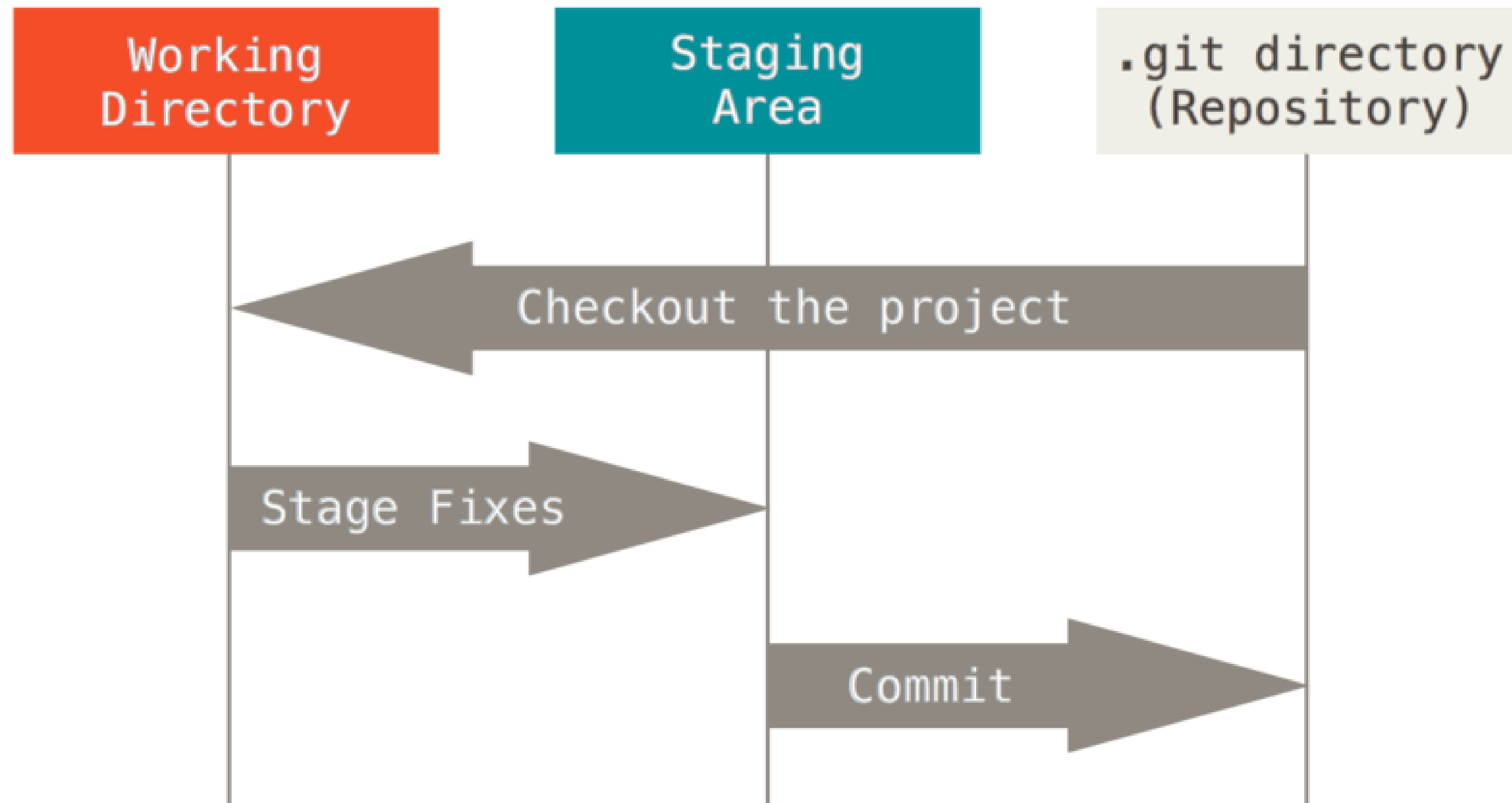
# O que é Git?

- A maior parte das operações no Git precisa apenas de arquivos e recursos locais para operarem, e normalmente nenhuma informação é necessária de outro computador na rede. Isso fornece uma velocidade de operação que outros VCS não possuem. Como cada usuário possui todo o histórico do projeto no computador, a maioria das operações aparenta ser quase instantânea.
- Para todos os efeitos, na prática, o Git normalmente só acrescenta informações ao seu banco de dados, nunca removendo informações. É muito difícil, e não recomendado, gerar operações que removam informações, já que essas operações podem afetar o histórico do seu projeto.

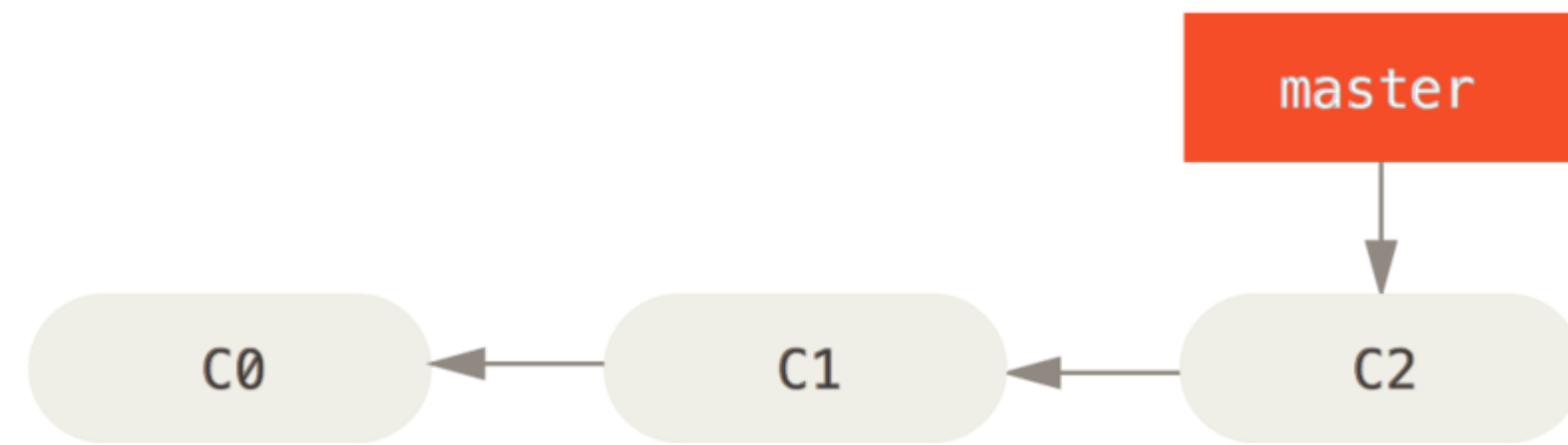
# Os três estados

- O Git tem três estados principais nos quais os arquivos de um repositório podem se encontrar: **modificado**, **preparado** e “**commitado**”:
  - **Commitado** significa que os dados estão armazenados de forma segura em seu banco de dados local;
  - **Modificado** significa que você alterou o arquivo, mas ainda não fez o commit no banco de dados;
  - **Preparado** significa que você marcou a versão atual de um arquivo modificado para fazer parte do seu próximo commit.
- Isso leva a três seções principais de um projeto Git: o diretório Git, o diretório de trabalho e área de preparo.

# Os três estados

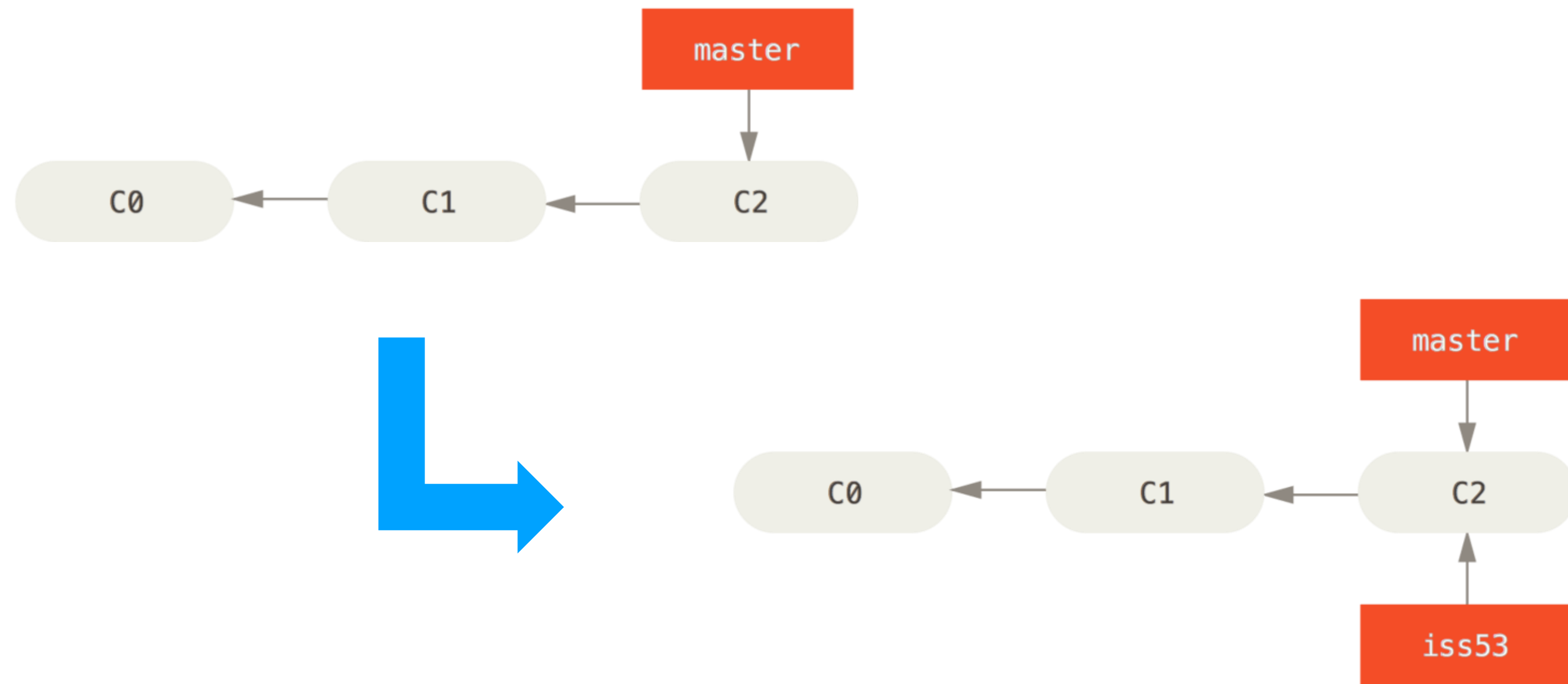


# Branches no Git

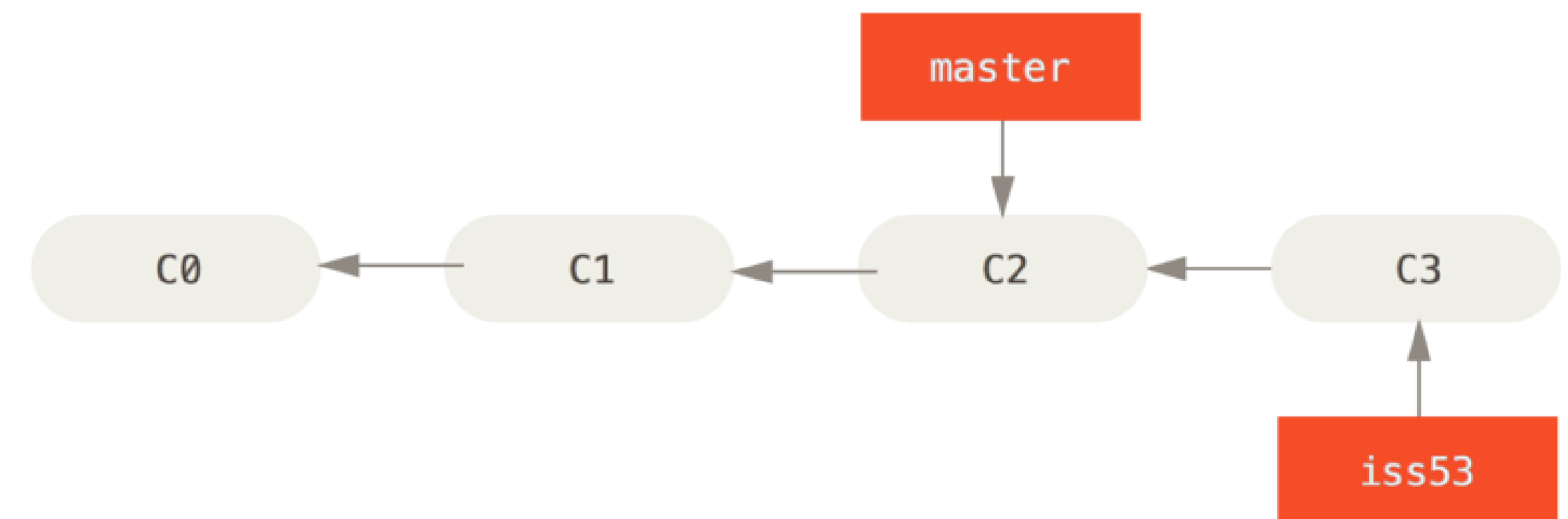
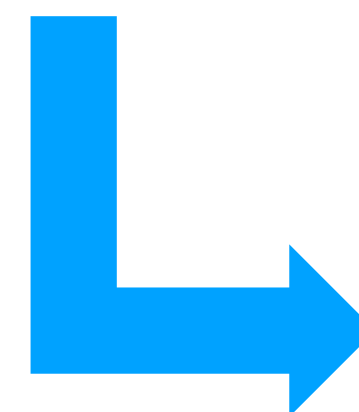
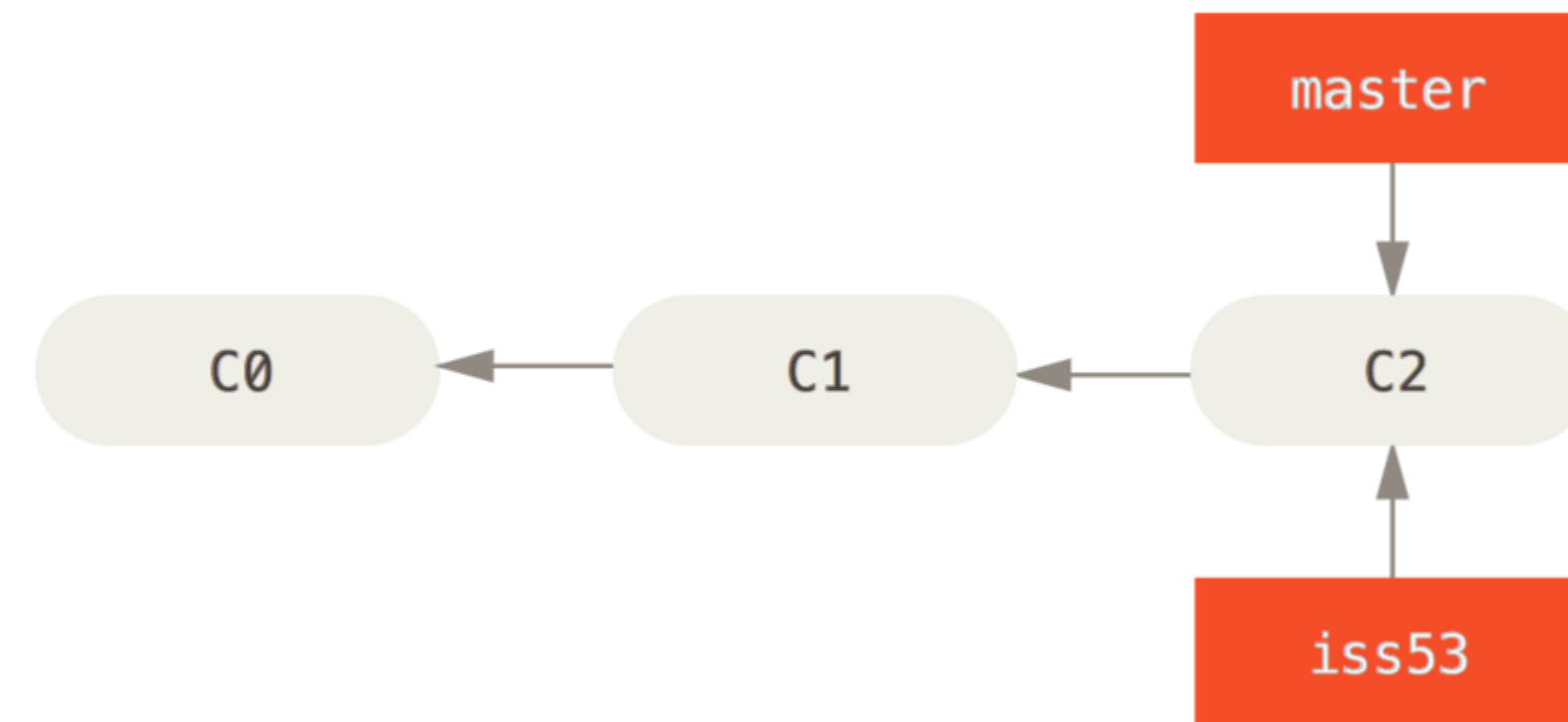
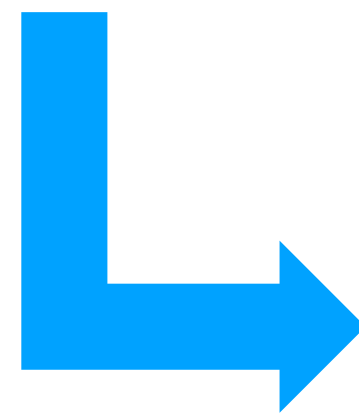
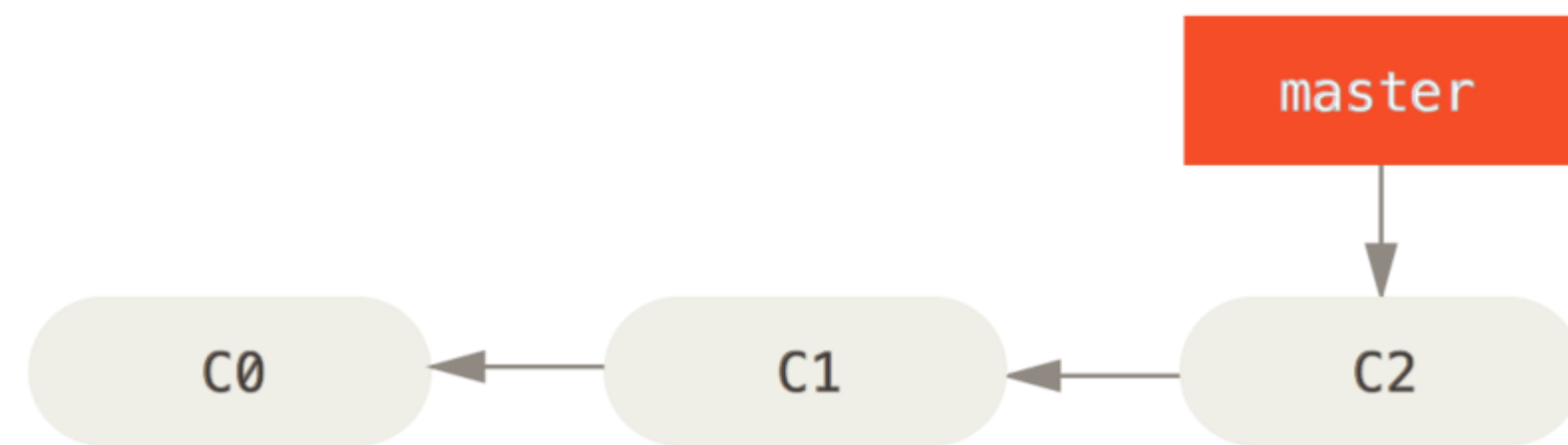




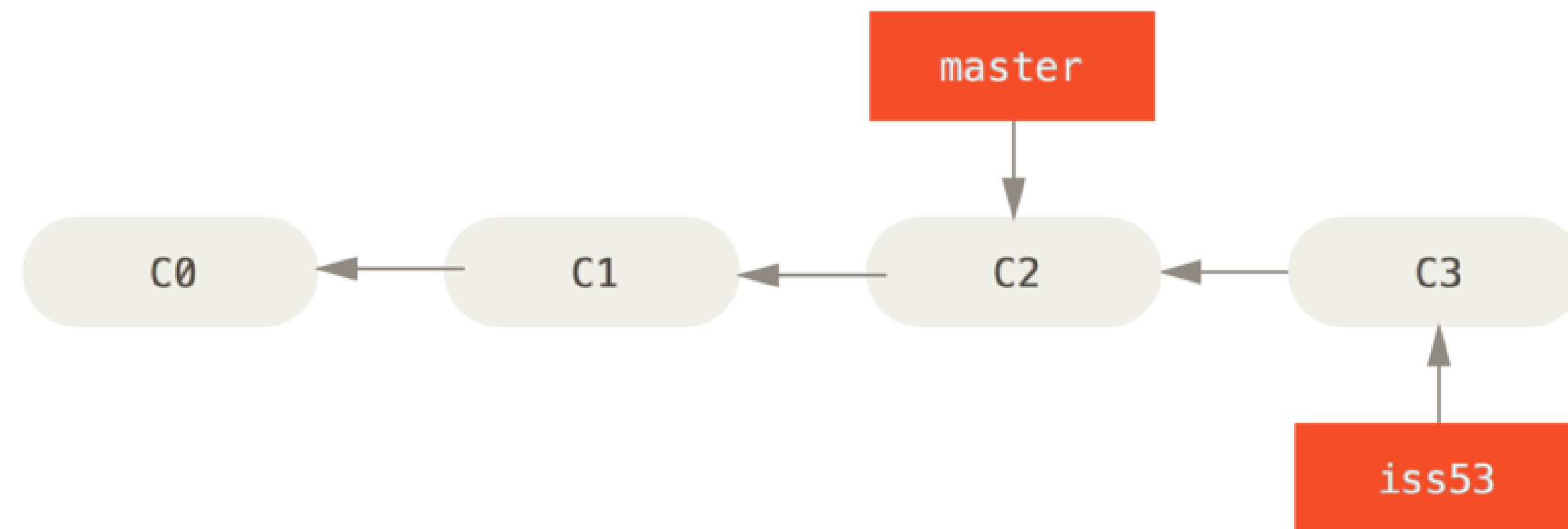
# Branches no Git



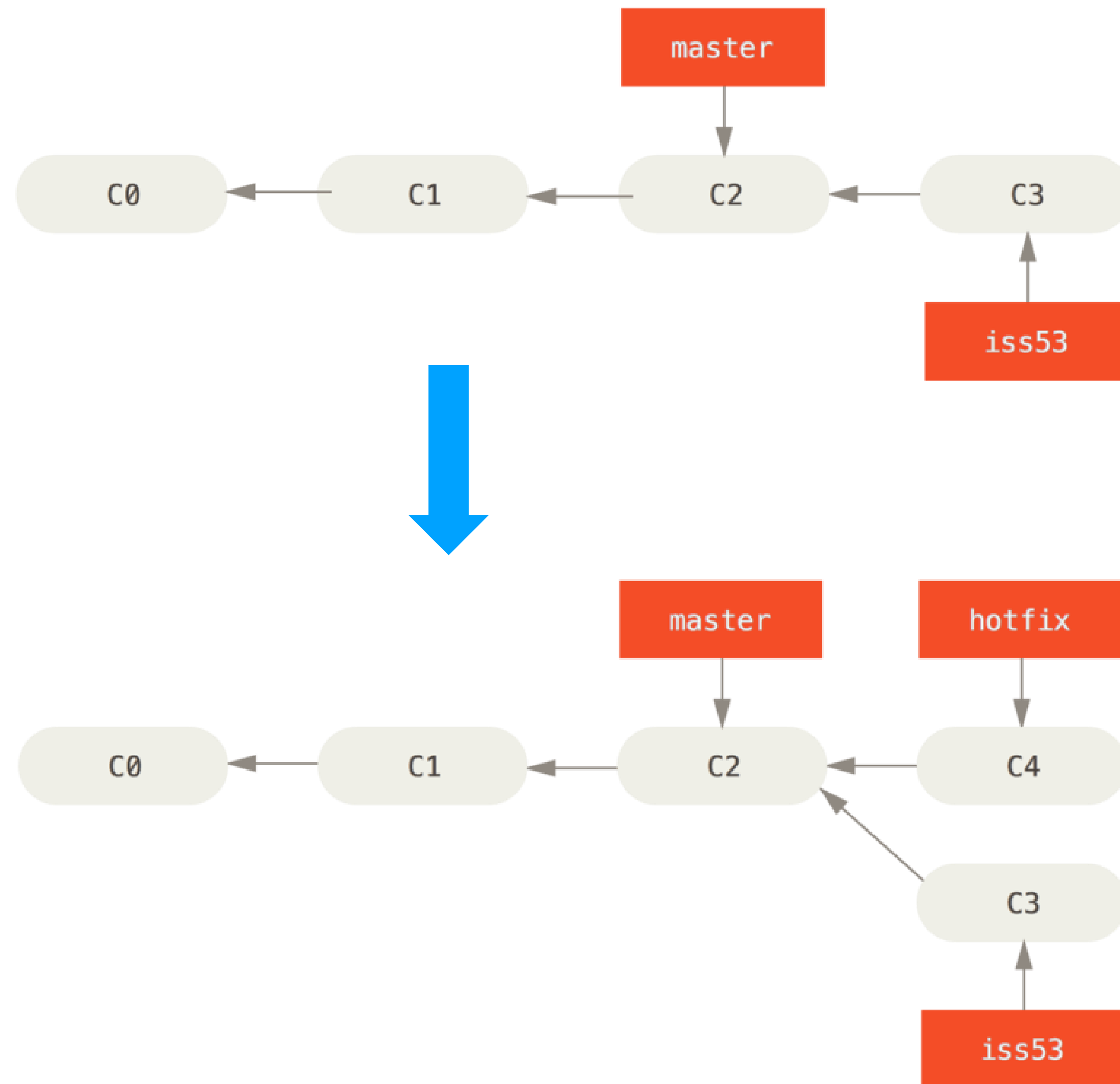
# Branches no Git



# Branches no Git

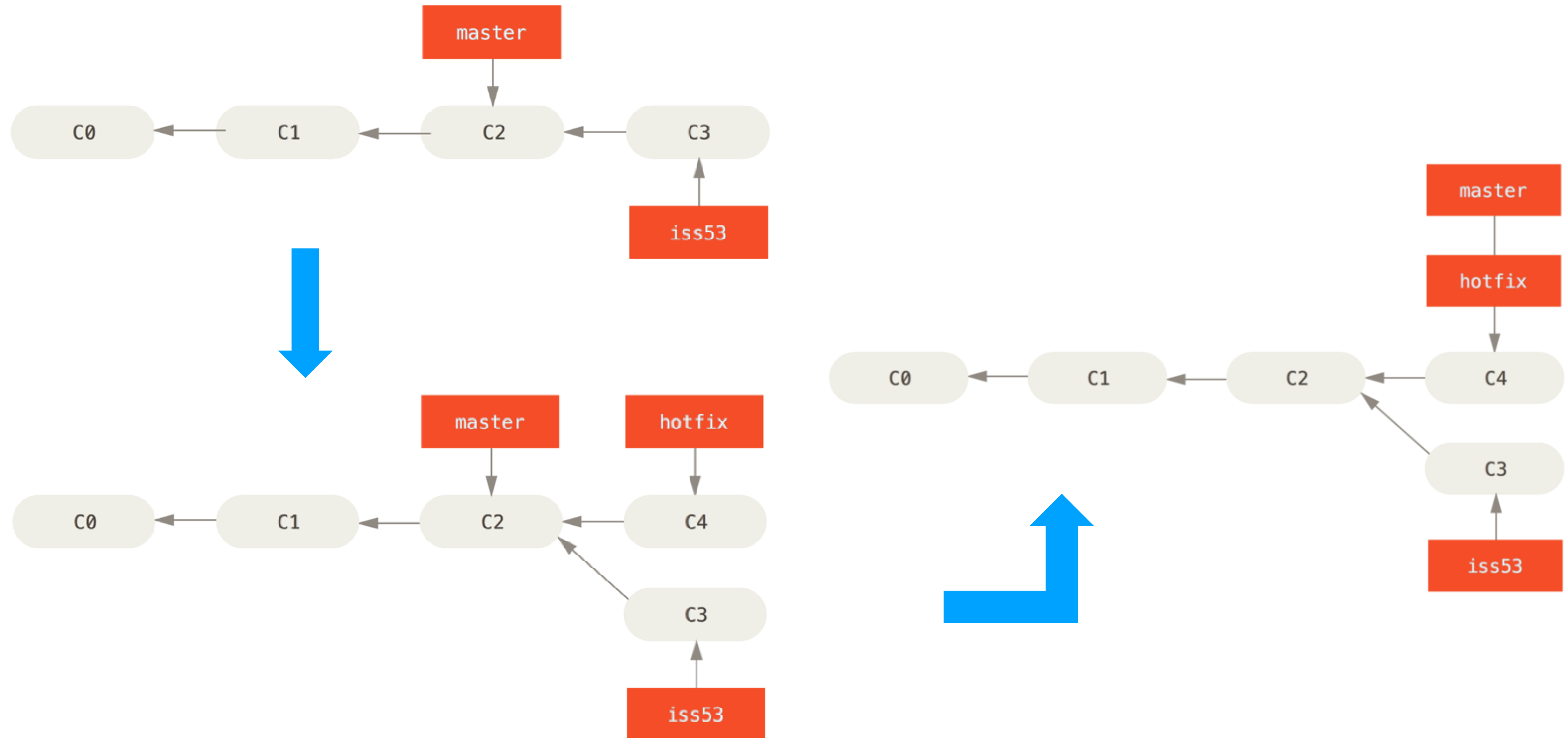


# Branches no Git

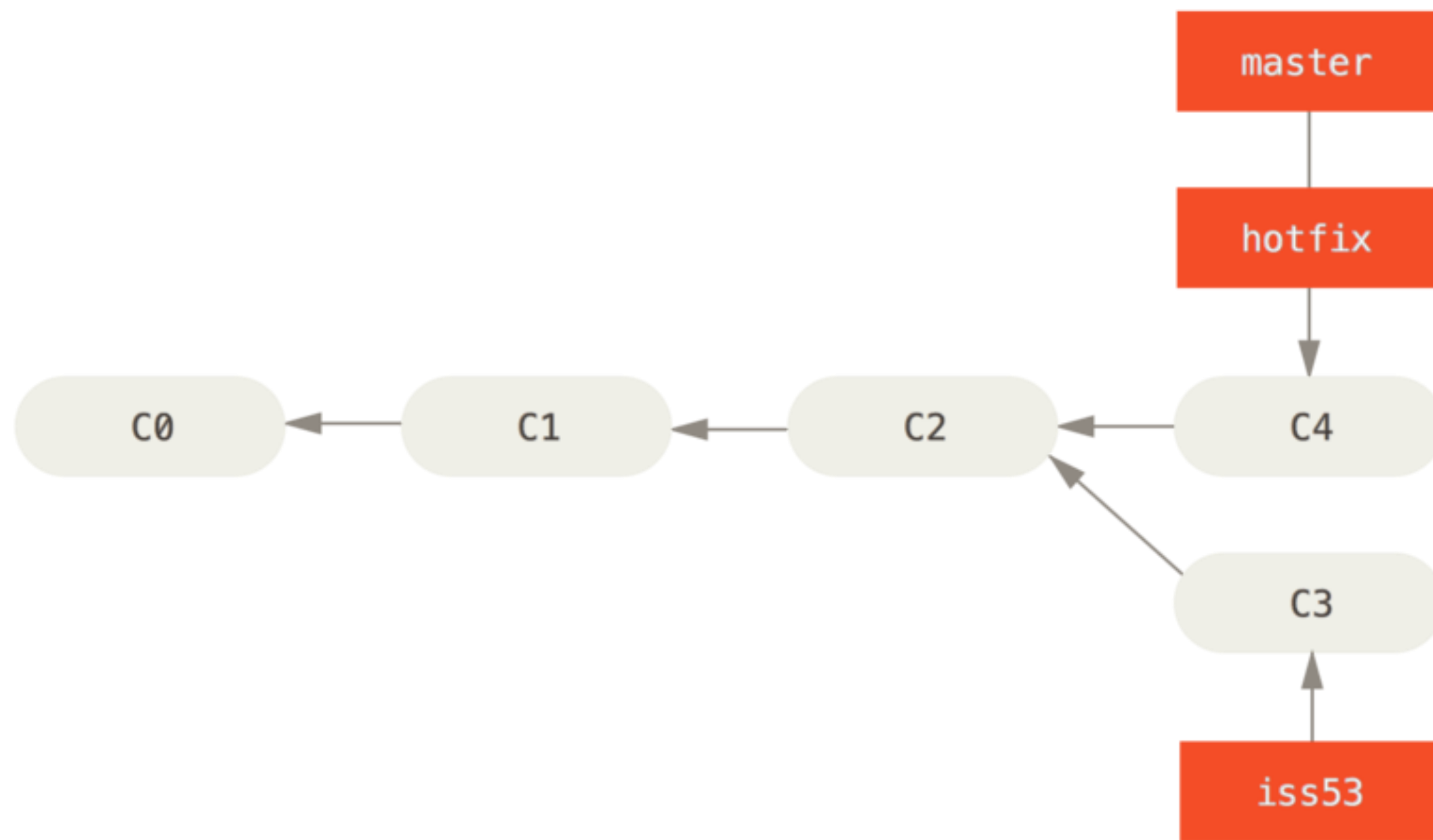




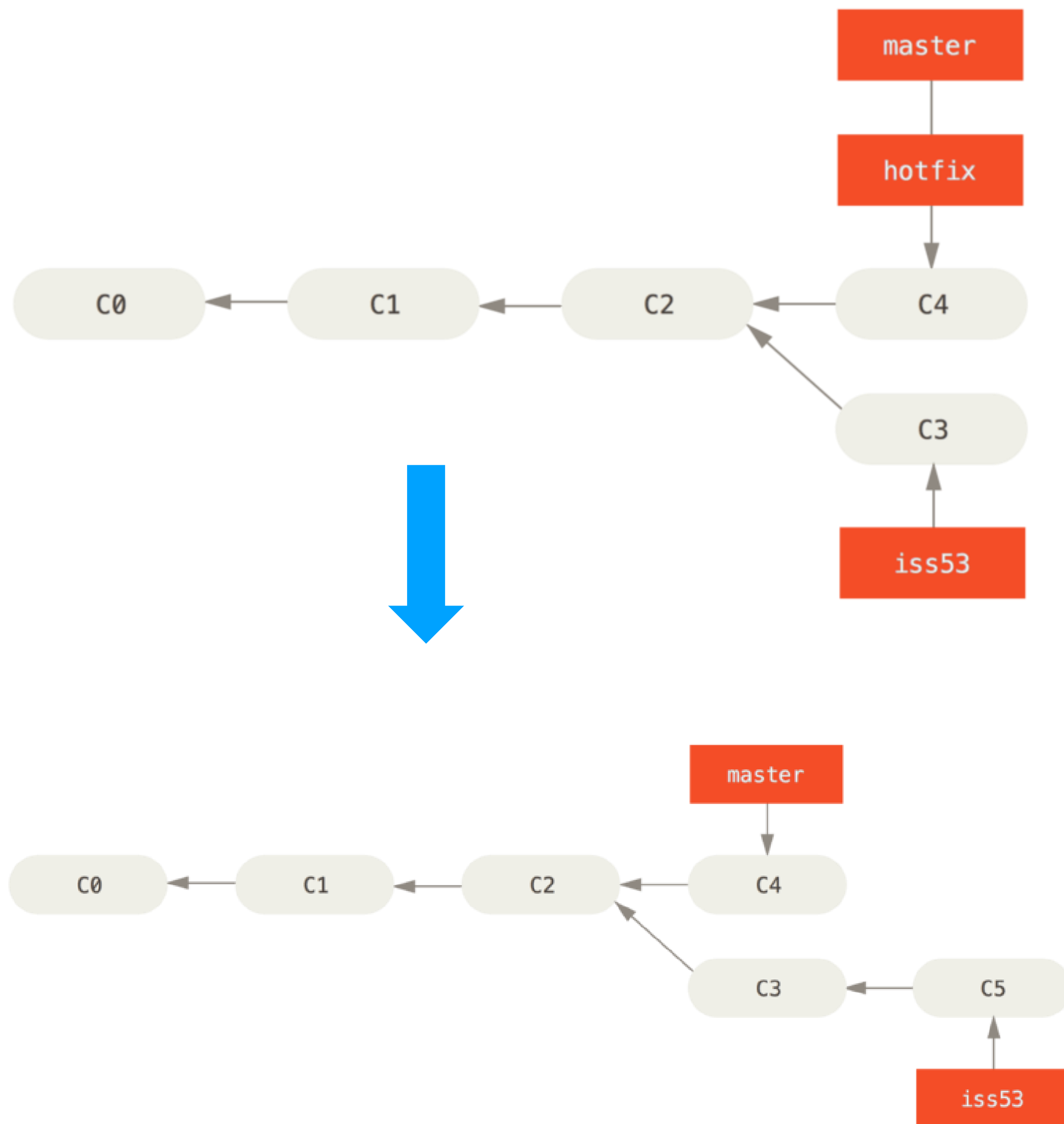
# Branches no Git



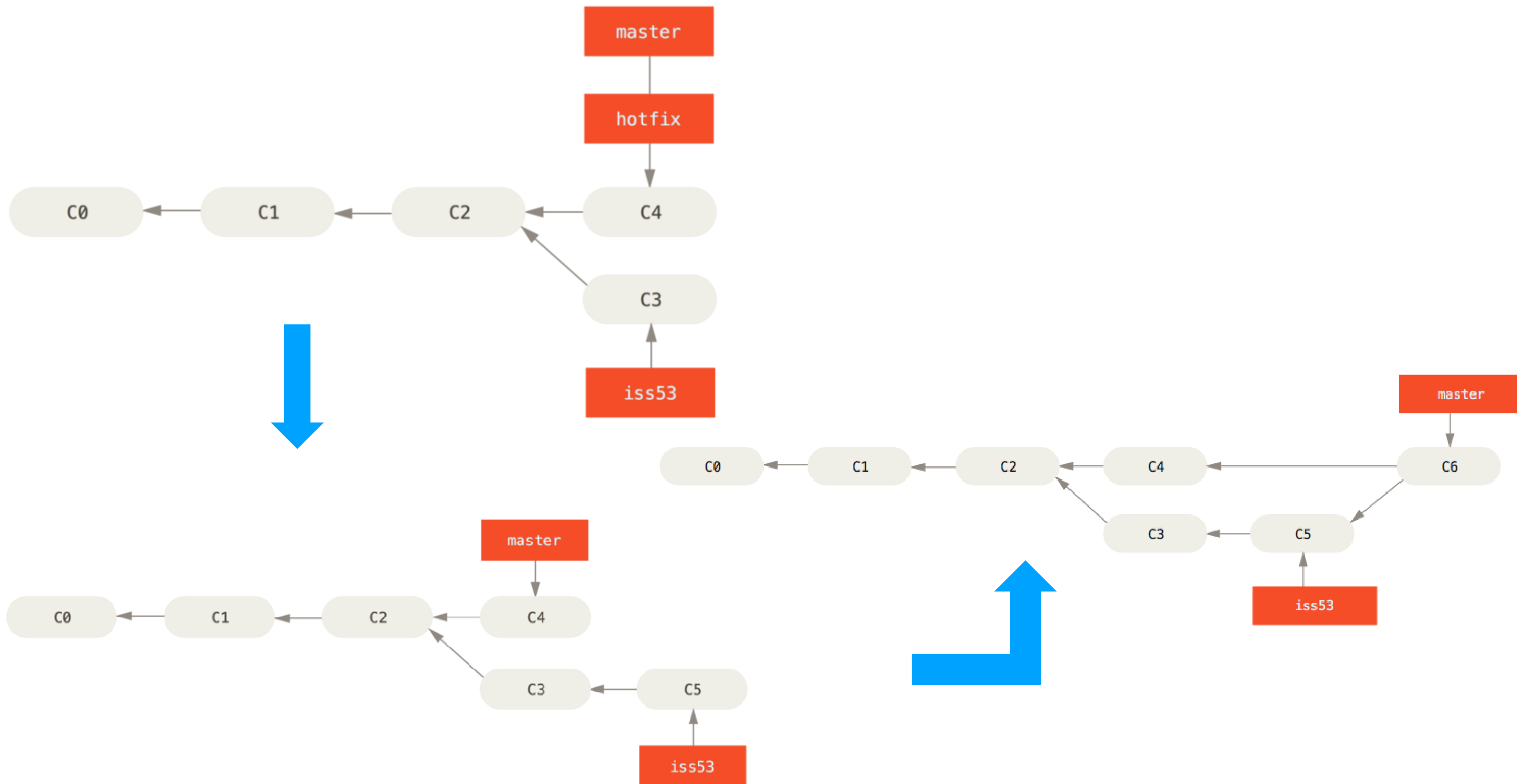
# Branches no Git



# Branches no Git



# Branches no Git



# Iniciando o uso de Git com GitHub

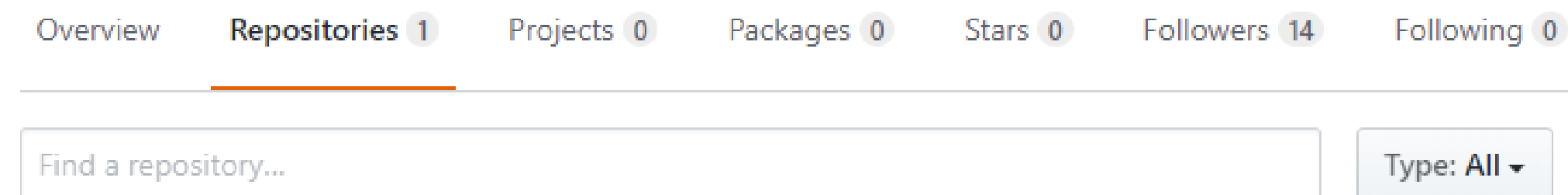
- Antes de mais nada, precisamos instalar o sistema Git na nossa máquina.
- <https://git-scm.com/downloads>
- O GitHub ([www.github.com](http://www.github.com)) é uma plataforma de versionamento que utiliza Git como base.
- Não é o nosso objetivo ensinar a plataforma a fundo, mas vamos utilizar algumas funcionalidades. Caso tenha interesse em se aprofundar no assunto, recomendo algumas páginas:
  - <https://github.com/culturagovbr/primeiros-passos>;
  - <https://help.github.com/en>;
  - [https://rogerdudler.github.io/git-guide/index.pt\\_BR.html](https://rogerdudler.github.io/git-guide/index.pt_BR.html).
- Antes de começar a configurar, crie uma conta no site.



# Algumas tarefas no GitHub

- **Criando um novo repositório:**

- No canto superior direito, clique no ícone do seu usuário e, em seguida, em **Your Profile**;
- Na nova janela, clique em **Repositories** e em **New**;

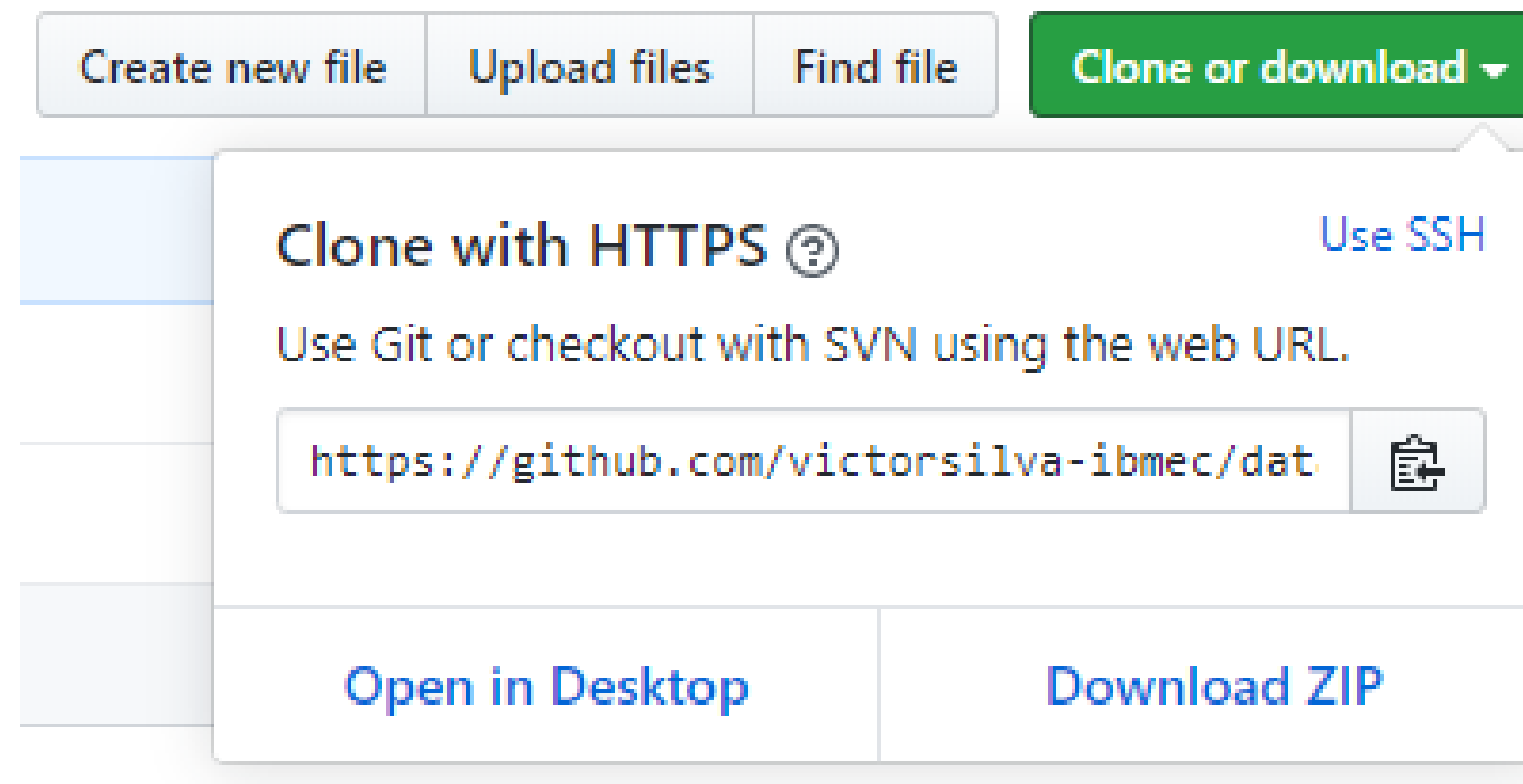


- Defina um nome (evite espaços e números), insira uma descrição se desejar e marque a opção **Public**, para que ele possa ser compartilhado. Por fim, marque a caixa **Initialize this repository with a README**;
- Clique em **Create repository**.

A screenshot of the GitHub "Create new repository" form. The "Owner" is set to "victorsilva-ibmec" and the "Repository name" is "data-mining" with a green checkmark. A note says "Great repository names are short and memorable. Need inspiration? How about upgraded-octo-bassoon?". The "Description (optional)" field contains "Repositório para curso de Data Mining com Python". The "Public" option is selected, with the description "Anyone can see this repository. You choose who can commit." The "Private" option is also visible. A note says "Skip this step if you're importing an existing repository." The "Initialize this repository with a README" checkbox is checked, with the description "This will let you immediately clone the repository to your computer." Below this are dropdowns for "Add .gitignore: None" and "Add a license: None" with an information icon. At the bottom is a green "Create repository" button.

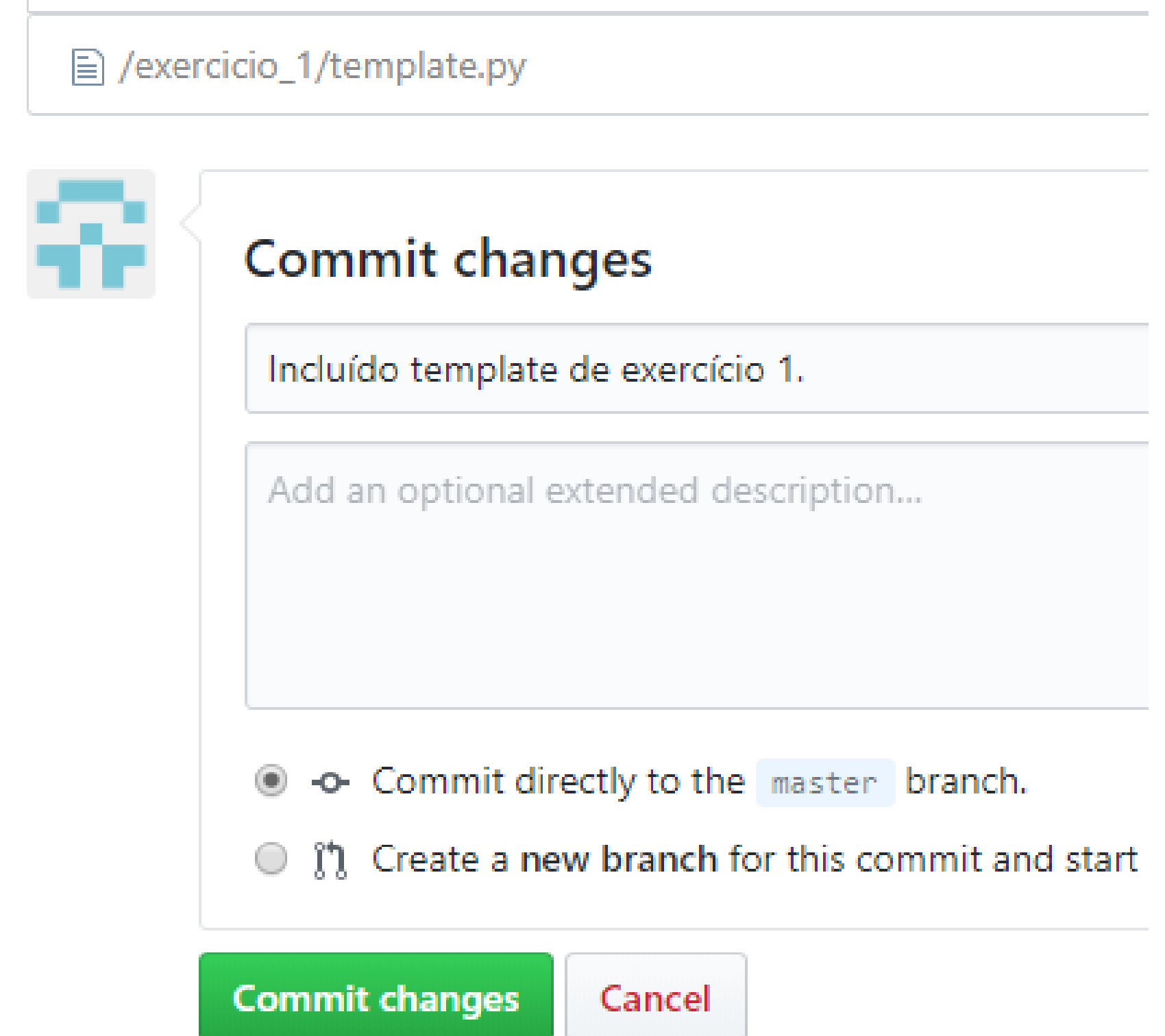
# Algumas tarefas no GitHub

- **Baixando manualmente um repositório para a máquina:**
  - Na tela do seu repositório, clique em **Clone or download**;
  - Em seguida, clique em **Download ZIP**;
  - Com o arquivo .zip baixado, descompacte-o na sua pasta de projeto, substituindo arquivos antigos se necessário.



# Algumas tarefas no GitHub

- **Fazendo um novo commit no seu repositório:**
  - Commits são alterações feitas no repositório. Podem conter um único arquivo ou vários. Caso um arquivo commitado já exista, o GitHub vai fazer um controle de versão, comparando as alterações entre a versão anterior e a que foi commitada;
  - Na tela do seu repositório, clique em **Upload files**;
  - Arraste para a tela os arquivos que deseja incluir;
  - Insira uma breve descrição do que está sendo commitado;
  - Deixe marcada a opção **Commit directly to the master branch**;
  - Clique em **Commit changes**.



The screenshot shows the GitHub 'Commit changes' interface. At the top, a file path `/exercicio_1/template.py` is displayed. Below it is a blue icon representing a commit. The main section is titled 'Commit changes' and contains a text box with the message 'Incluído template de exercício 1.' Below this is a larger text box with the placeholder 'Add an optional extended description...'. At the bottom, there are two radio button options: 'Commit directly to the master branch.' (which is selected) and 'Create a new branch for this commit and start'. At the very bottom are two buttons: 'Commit changes' (green) and 'Cancel' (grey).

# Algumas tarefas no GitHub

- **Submetendo um trabalho para revisão:**
  - Um **pull request** é o ato de submeter para aprovação as alterações ou inserções de código de um ou mais arquivos. A pessoa que abre um **pull request** sinaliza que gostaria de uma aprovação do conteúdo antes de ele ser, de fato, incorporado ao repositório;
  - No repositório desejado, clique na pasta em que você deseja incluir ou atualizar os arquivos;
  - Clique em **Upload files**;
  - Arraste para a tela o(s) arquivo(s) com a sua atualização, e na descrição explique o que está sendo feito. Em seguida, clique em **Commit changes**;
  - Na nova janela, insira um comentário e depois clique em **Create pull request**.

# Alguns comandos do Git

- Apesar do Github fornecer recursos para operar com Git direto pelo navegador, esses recursos são limitados. Por exemplo, fazer checkout de um novo branch apenas pelo navegador pode ser bem complicado.
- Uma forma mais usual de se usar o Git é através de programas específicos para o computador.
- É bem comum utilizar os comandos do Git por linha de comando, porém existem bons programas para uso do Git através de uma interface gráfica, como o [Sourcetree](#).
- No slide a seguir serão apresentados alguns comandos comuns para o uso do Git pela linha de comando. Eles podem ser executados pelo Git Bash (programa instalado junto com o Git), ou pelo terminal.



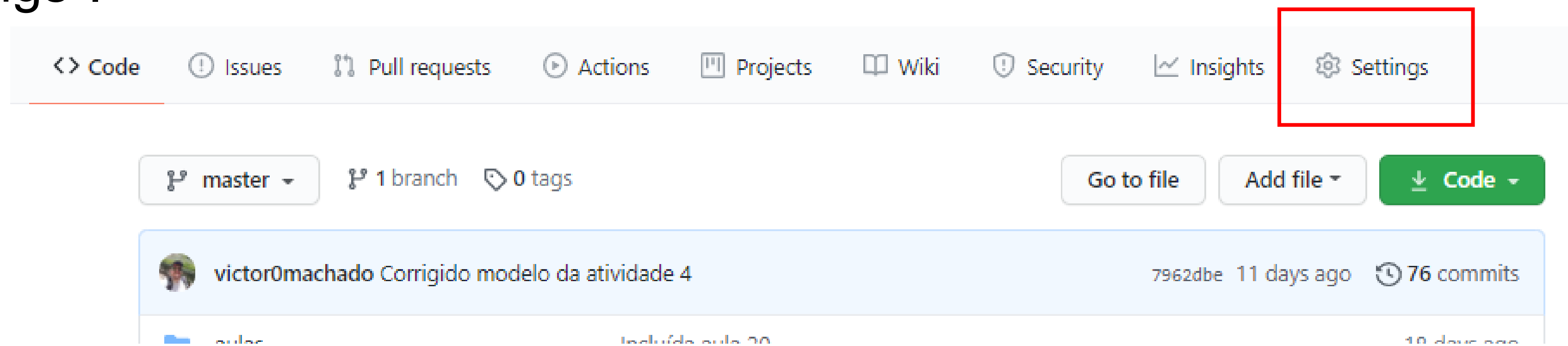
# Alguns comandos do Git

- Uma observação, para facilitar as operações, é trabalhar sempre na raiz do repositório.
- Com o terminal na raiz do repositório, insira os seguintes comandos para obter os efeitos apresentados:

Comando	Efeito
git init	Inicia um repositório Git no diretório em questão
git status	Indica os status de arquivos modificados, adicionados ou removidos, além de arquivos preparados (staged)
git add <arquivo>	Prepara o arquivo mencionado
git add -u	Prepara todos os arquivos modificados (porém não faz nada com arquivos novos)
git add .	Prepara todos os arquivos (incluindo arquivos novos)
git commit -m "Mensagem"	Faz um commit dos arquivos preparados, incluindo a mensagem de commit definida
git restore <arquivo>	Desfaz modificações do arquivo que não foi preparado
git restore --staged <arquivo>	Desfaz a preparação do arquivo (porém mantém modificações)
git pull	No branch escolhido, atualiza as informações com o repositório remoto
git branch	Lista todos os branches armazenados localmente
git branch --show-current	Lista o branch atual
git branch -m <novo_nome>	Renomeia o branch atual (cuidado ao fazer isso para branches que já estão no repositório remoto!)
git checkout <branch>	Dá checkout no branch mencionado
git checkout -b <branch>	Cria um novo branch, com o nome mencionado, e dá checkout nele
git push	Envia os commits realizados localmente para o branch remoto

# GitHub pages

- Uma boa forma de manter o seu portfólio sempre atualizado é com uma página pessoal, na qual você inclui seu currículo, projetos realizados, trabalhos, interesses pessoais e profissionais, e outras informações que achar pertinente.
- O Github possui uma forma muito simples de se criar um repositório que também serve como página pessoal.
- Para isso, crie um repositório normal, vá na página desse repositório e clique em “Settings”.



# GitHub pages

- Nas configurações, desça a página até encontrar a seção “GitHub Pages”.

GitHub Pages

GitHub Pages is designed to host your personal, organization, or project pages from a GitHub repository.

**Source**  
GitHub Pages is currently disabled. Select a source below to enable GitHub Pages for this repository. [Learn more.](#)

**Theme Chooser**  
Select a theme to publish your site with a Jekyll theme using the gh-pages branch. [Learn more.](#)

- No campo “Source”, indique o branch que você quer que seja a sua página principal (usualmente é o branch master). Se quiser, escolha um tema da lista de temas gratuitos disponíveis e, em seguida, clique em “Save”.

# GitHub pages

- A página terá uma atualização que mostrará a URL do site, além de incluir um campo no qual você pode inserir um domínio customizado, caso o tenha.

## GitHub Pages

GitHub Pages is designed to host your personal, organization, or project pages from a GitHub repository.

Your site is ready to be published at <https://victor0machado.github.io/2020.2-logprog/>.

### Source

Your GitHub Pages site is currently being built from the master branch. [Learn more.](#)

Branch: master ▾

/ (root) ▾

Save

### Theme Chooser

Select a theme to publish your site with a Jekyll theme. [Learn more.](#)

Choose a theme

### Custom domain

Custom domains allow you to serve your site from a domain other than victor0machado.github.io. [Learn more.](#)

Save

# GitHub pages

- O site funciona como um repositório normal. Todas as páginas devem ser escritas em Markdown, que já vimos ao longo do curso.

```
# Boas-vindas

Vou atualizar essa página com os materiais das disciplinas que leciono no
IBMEC/RJ.

## Disciplinas

* [Algoritmos e Programação de Computadores](/courses/algprog.md)
* [Lógica e Programação de Computadores](/courses/logprog.md)
* [Data Mining com Python](/courses/datamining.md)

## Meus contatos

* E-mail: <victor.silva@professores.ibmec.edu.br>
* [Linkedin](https://www.linkedin.com/in/victormachadodasilva/)
* [Lattes](http://lattes.cnpq.br/1584907276781609)
```

## Repositório público do Prof. Victor Machado

Material usado nas minhas disciplinas do  
IBMEC/RJ

[View My GitHub Profile](#)

## Boas-vindas

Vou atualizar essa página com os materiais das disciplinas que leciono no  
IBMEC/RJ.

## Disciplinas

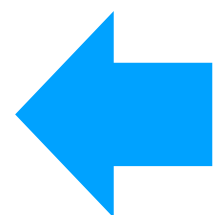
- Algoritmos e Programação de Computadores
- Lógica e Programação de Computadores
- Data Mining com Python

## Meus contatos

- E-mail: [victor.silva@professores.ibmec.edu.br](mailto:victor.silva@professores.ibmec.edu.br)
- [Linkedin](#)
- [Lattes](#)

- O único arquivo exigido para o site é o **index.md**, que deve ficar na raiz do repositório. Novos arquivos e pastas podem ser criados se necessário, e a navegação é sempre relativa à raiz do repositório.





# Fundamentos da Orientação a Objetos

# Por que usar OO?

Segundo o Paradigma Procedural, é possível representar todo e qualquer processo do mundo real a partir da utilização de **apenas** três estruturas básicas:

- Sequência: Os passos devem ser executados um após o outro, linearmente. Ou seja, o programa seria uma sequência finita de passos. Em uma unidade de código, todos os passos devem ser feitos para se programar o algoritmo desejado;
- Decisão: Uma determinada sequência de código pode ou não ser executada. Para isto, um teste lógico deve ser realizado para determinar ou não sua execução. A partir disto, verifica-se que duas estruturas de decisão (também conhecida como seleção) podem ser usadas: a if-else e a switch.
- Iteração: É a execução repetitiva de um segmento (parte do programa). A partir da execução de um teste lógico, a repetição é realizada um número finito de vezes. Estruturas de repetição conhecidas são: for, foreach, while, do-while, repeat-until, entre outras (dependendo da linguagem de programação).

# Por que usar OO?

Usar apenas essas três estruturas pode apresentar algumas limitações:

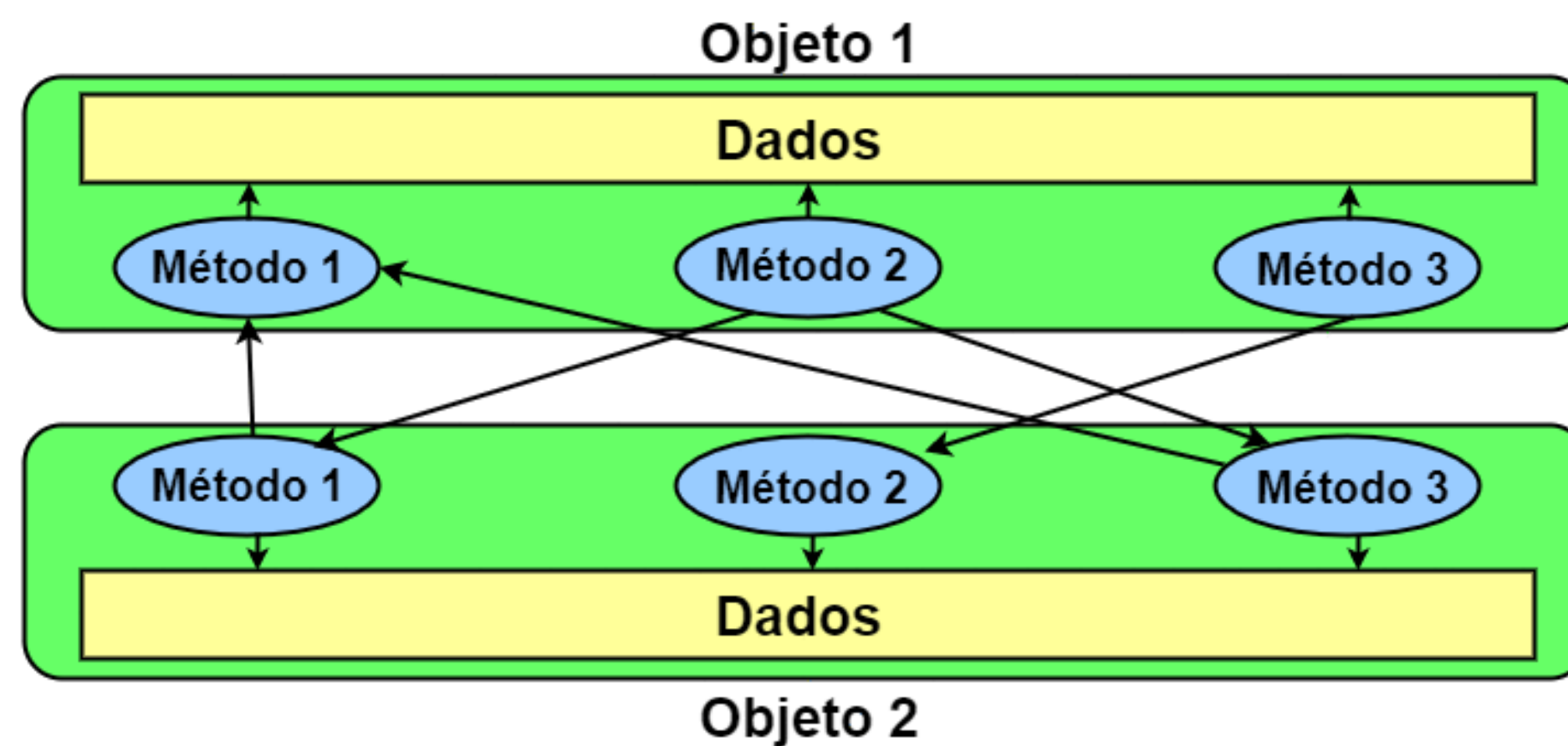
- Quanto mais complexo o programa se torna, mais difícil fica a manutenção de uma sequência organizada de código;
- Com esse paradigma é muito fácil deixar o código extenso e com muitas duplicações;
- Mesmo modularizações providas pelas linguagens podem deixar o código muito complexo.

Em resumo, a simplificação da representação das reais necessidades dos problemas a serem automatizados leva a uma facilidade de entendimento e representação. Porém, isso pode levar a uma complexidade de programação caso o nicho de negócio do sistema-alvo seja complexo.

# Por que usar OO?

Ao contrário do paradigma procedural, a OO preconiza que os dados relativos a uma representação de uma entidade do mundo real devem somente estar juntos de suas operações, quais são os responsáveis por manipular - exclusivamente - tais dados.

Assim, há uma separação de dados e operações que não dizem respeito a uma mesma entidade. Todavia, se tais entidades necessitarem trocar informações, farão isto através da chamada de seus métodos, e não de acessos diretos a informações da outra.



# Fundamentos da OO

## Abstração:

- Processo pelo qual se isolam características de um objeto, considerando os que tenham em comum certos grupos de objetos.
- Não devemos nos preocupar com características menos importantes, ou seja, acidentais. Devemos, neste caso, nos concentrar apenas nos aspectos essenciais. Por natureza, as abstrações devem ser incompletas e imprecisas.





# Fundamentos da OO

## Abstração:

- Com a abstração dos conceitos, conseguimos reaproveitar, de forma mais eficiente, o nosso “molde” inicial, que pode ser detalhado conforme for necessário para o nosso caso específico.
- Os processos de inicialmente se pensar no mais abstrato e, posteriormente, acrescentar ou se adaptar são também conhecidos como **generalização** e **especialização**, respectivamente.



# Fundamentos da OO

Reuso:

- Não existe pior prática em programação do que a repetição de código. Isto leva a um código frágil, propício a resultados inesperados. Quanto mais códigos são repetidos pela aplicação, mais difícil vai se tornando sua manutenção.
- O fato de simplesmente utilizarmos uma linguagem OO não é suficiente para se atingir a reusabilidade, temos de trabalhar de forma eficiente para aplicar os conceitos de **herança** e **associação**, por exemplo.
- Na herança, é possível criar classes a partir de outras classes. A classe filha, além do que já foi reaproveitada, pode acrescentar o que for necessário para si.
- Já na associação, o reaproveitamento é diferente. Uma classe pede ajuda a outra para poder fazer o que ela não consegue fazer por si só. Em vez de simplesmente repetir, em si, o código que está em outra classe, a associação permite que uma classe forneça uma porção de código a outra. Assim, esta troca mútua culmina por evitar a repetição de código.

# Fundamentos da OO

## Encapsulamento:

- Quando alguém se consulta com um médico, por estar com um resfriado, seria desesperados se ao final da consulta o médico entregasse a seguinte receita:

Receituário (Complexo)

- 400mg de ácido acetilsalicílico
- 1mg de maleato de dexclorfeniramina
- 10mg de cloridrato de fenilefrina
- 30mg de cafeína

Misturar bem e ingerir com água. Repetir em momentos de crise.

- A primeira coisa que viria em mente seria: onde achar essas substâncias? Será que é vendido tão pouco? Como misturá-las? Existe alguma sequência? Seria uma tarefa difícil - até complexa - de ser realizada. Mais simples do que isso é o que os médicos realmente fazer: passam uma cápsula onde todas estas substâncias já estão prontas. Ou seja, elas já vêm encapsuladas.

# Fundamentos da OO

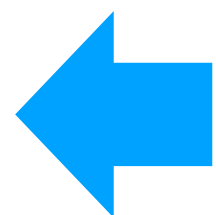
## Encapsulamento:

- Com isso, não será preciso se preocupar em saber quanto e como as substâncias devem ser manipuladas para no final termos o comprimido que resolverá o problema. O que interessa é o resultado final, no caso, a cura do resfriado. A complexidade de chegar a essas medidas e como misturá-las não interessa. É um processo que não precisa ser do conhecimento do paciente.

### Receituário (Encapsulado)

1 comprimido de Resfriol. Ingerir com água. Repetir em momentos de crise.

- Essa mesma ideia se aplica na OO. No caso, a complexidade que desejamos esconder é a de implementação de alguma necessidade. Com o encapsulamento, podemos esconder a forma como algo foi feito, dando a quem precisa apenas o resultado gerado.
- Uma vantagem deste princípio é que as mudanças se tornam transparentes, ou seja, quem usa algum processamento não será afetado quando seu comportamento interno mudar.



# Introdução a Java



# O que é Java?

A linguagem Java começou a ser concebida no início da década de 1990, com o objetivo de resolver alguns dos problemas comuns em programação na época, tais como:

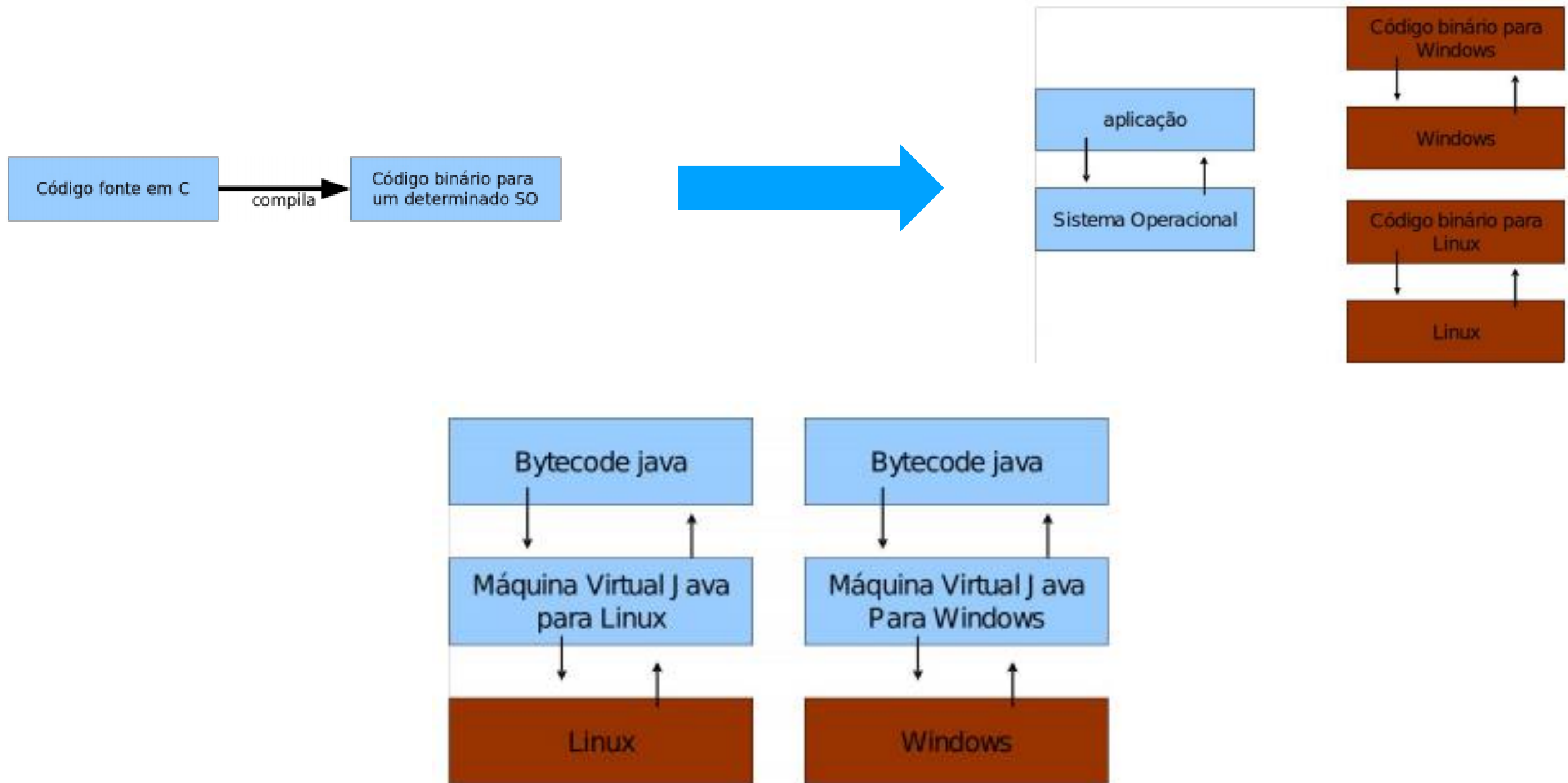
- Uso de ponteiros;
- Gerenciamento de memória;
- Organização;
- Falta de bibliotecas;
- Necessidade de reescrever parte do código ao mudar de sistema operacional;
- Custo financeiro de usar a tecnologia.

Uma das grandes motivações para a criação da plataforma Java era de que essa linguagem fosse usada em pequenos dispositivos, como TVs. Apesar disso a linguagem teve seu lançamento focado no uso em clientes web (browsers) para rodar pequenas aplicações (applets). Hoje em dia esse não é o grande mercado do Java: apesar de ter sido idealizado com um propósito e lançado com outro, o Java ganhou destaque no lado do servidor.

# Uma breve história

- Time criado em 1992 na Sun, liderado por James Gosling, considerado o pai do Java.
- Ideia inicial de criar um interpretador para pequenos dispositivos, facilitando a reescrita de software para aparelhos eletrônicos.
- A ideia não deu certo, devido ao conflito de interesses e custos.
- Hoje, sabemos que o Java domina o mercado de aplicações para celulares com mais de 2.5 bilhões de dispositivos compatíveis, porém em 1994 ainda era muito cedo para isso.
- Com o advento da web, a ideia pode ser reaproveitada, já que na internet havia uma grande quantidade de sistemas operacionais e browsers, e com isso seria grande vantagem poder programar numa única linguagem, independente da plataforma.
- O Java 1.0 foi lançado focado em transformar o browser de apenas um terminal “burro” em uma aplicação que possa também realizar operações avançadas, e não apenas renderizar HTML.
- Em 2009 a Oracle comprou a Sun e fortaleceu a marca e a plataforma Java.

# Máquina virtual?



# Máquina virtual?

O conceito de máquina virtual é bem mais amplo que o de um interpretador:

- Uma VM tem tudo que um computador tem: é responsável por gerenciar memória, threads, a pilha de execução, etc.
- Sua aplicação roda sem nenhum envolvimento com o sistema operacional, então a JVM pode tirar métricas, decidir onde é melhor alocar a memória, entre outros.
- Se uma JVM termina abruptamente, só as aplicações que estavam rodando nela irão terminar.



# Onde usar?

É preciso ficar claro que a premissa do Java não é a de criar sistemas pequenos, onde temos um ou dois desenvolvedores, mais rapidamente que linguagens como PHP, Perl, e outras.

O foco da plataforma é outro: aplicações de médio a grande porte, onde o time de desenvolvedores tem várias pessoas e sempre pode vir a mudar e crescer.

Não tenha dúvidas que criar a primeira versão de uma aplicação usando Java, mesmo utilizando IDEs e ferramentas poderosas, será mais trabalhoso que muitas linguagens script ou de alta produtividade. Porém, com uma linguagem orientada a objetos e madura como o Java, será extremamente mais fácil e rápido fazer alterações no sistema, desde que você siga as boas práticas e recomendações sobre design orientado a objetos.

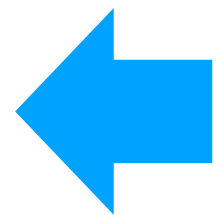
Além disso, a quantidade enorme de bibliotecas gratuitas para realizar os mais diversos trabalhos é um ponto fortíssimo para adoção do Java: você pode criar uma aplicação sofisticada, usando diversos recursos, sem precisar comprar um componente específico, que costuma ser caro. O ecossistema do Java é enorme.

# Onde usar?

Cada linguagem tem seu espaço e seu melhor uso. O uso do Java é interessante em aplicações que virão a crescer, em que a legibilidade do código é importante, onde temos muita conectividade e se há muitas plataformas (ambientes e sistemas operacionais) heterogêneas (Linux, Unix, OSX e Windows misturados).

Você pode ver isso pela quantidade enorme de ofertas de emprego procurando desenvolvedores Java para trabalhar com sistemas web e aplicações de integração no servidor.





# Conceitos estruturais

# Introdução

Embora a OO tenha vantagens em relação aos paradigmas que a precederam, existe uma desvantagem inicial: ser um modo mais complexo e difícil de se pensar. Isso pode ser atribuído à grande quantidade de conceitos que devem ser assimilados para podermos trabalhar orientado a objetos.

Vamos falar de cinco conceitos estruturais principais:

- A classe
- O atributo
- O método
- O objeto
- A mensagem

# Classes

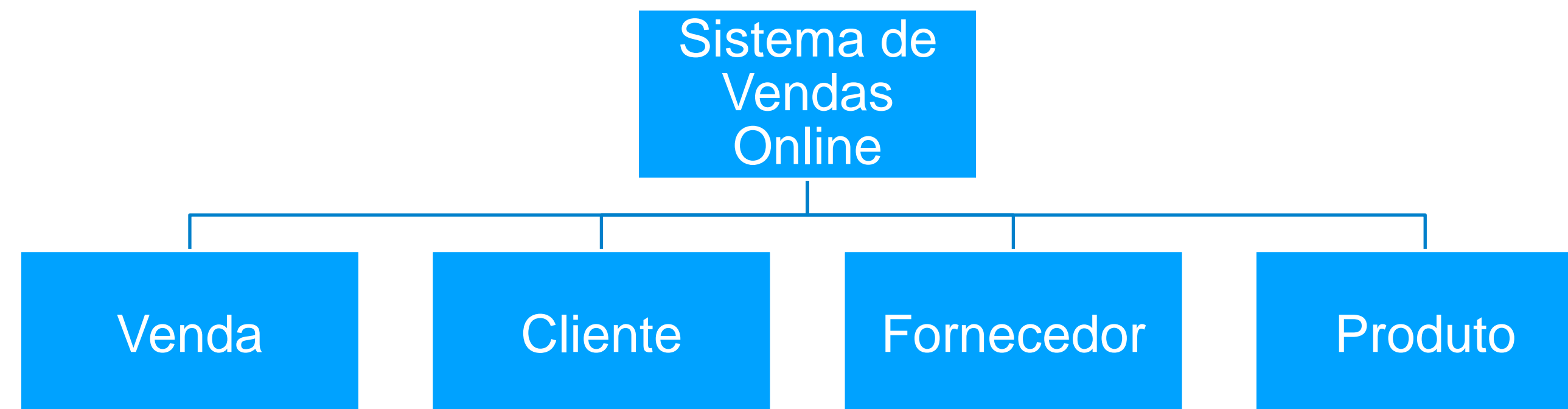
Apesar do paradigma ser nomeado como Orientado a Objetos, tudo começa com a definição de uma classe.

Antes mesmo de ser possível manipular objetos, é preciso definir uma classe, pois esta é a unidade inicial e mínima de código na OO. É a partir de classes que futuramente será possível criar objetos.

O objetivo de uma classe é definir, servir de base, para o que futuramente será o objeto.

- A classe é um “molde” que deverá ser seguido pelos objetos.

Uma classe pode ser definida como uma abstração de uma entidade, seja ela física (bola, pessoa, carro, etc.) ou conceitual (viagem, venda, estoque, etc.) do mundo real.



# Classes

O nome de uma classe deve representar bem sua finalidade dentro do contexto ao qual ela foi necessária.

- Em um sistema de controle hospitalar, podemos ter uma classe chamada **Pessoa**;
- Em um sistema de ponto de vendas (PDV), também temos o conceito de **Pessoa**.

Entretanto, nota-se que o termo *pessoa* pode gerar uma ambiguidade, embora esteja correto.

Assim, recomenda-se ser mais específico na nomeação das classes, como **Médico**, **Paciente**, **Cliente**, **Vendedor**.

Embora possa parecer preciosismo, classes com nomes pobremente definidos podem dificultar o entendimento do código e até levar a erros de utilização. Pense bem antes de nomear uma classe.

# Atributos

Após o processo inicial de identificar as entidades (classes) que devem ser manipuladas, começa a surgir a necessidade de detalhá-las.

- Quais informações devem ser manipuladas através desta classe?

Devemos, então, caracterizar as classes definidas. Essas características é que vão definir quais informações as classes poderão armazenar e manipular. Na OO, estas características e informações são denominadas de **atributo**.

Essa definição deixa bem claro que os atributos devem ser definidos dentro da classe. É a partir do uso de atributos que será possível caracterizar (detalhar) as classes.

Assim como nas classes, os atributos podem ser representados a partir de substantivos. Além destes, podemos também usar adjetivos. Pensar em ambos pode facilitar o processo de identificação dos atributos.

# Atributos

Classe **Paciente** em um sistema hospitalar → quais seriam os possíveis atributos?

- Nome
- CPF
- Data de nascimento
- Histórico médico

Todos estes são substantivos, mas alguns de seus valores poderiam ser adjetivos.

Quanto mais for realizado o processo de caracterização, mais detalhada será a classe e, com isso, ela terá mais atributos.

Porém, é preciso ter parcimônia no processo de identificação dos atributos!

- Hobby?
- Possui carro?



# Atributos

Nem sempre uma informação, mesmo sendo importante, deve ser transformada em um atributo. Por exemplo, **idade**. Ter que recalcular a idade toda vez que a pessoa fizer aniversário é custoso e propenso a erros.

Neste caso, seria melhor usar o que é conhecido como **atributo calculado** ou **atributo derivado**. A idade não se torna um atributo em si, mas tem seu valor obtido a partir de um método.

Não diferentemente de linguagens estruturadas, um atributo possui um tipo. Como sua finalidade é armazenar um valor que será usado para caracterizar a classe, ele precisará identificar qual o tipo do valor armazenado em si. Linguagens orientadas a objetos proveem os mesmos tipos de dados básicos - com pequenas variações - que suas antecessoras.

# Atributos

A nomeação de atributos deve seguir a mesma preocupação das classes: deve ser o mais representativo possível. Nomes como **qtd**, **vlr** devem ser evitados.

Esses nomes eram válidos na época em que as linguagens de programação e os computadores que as executavam eram limitados, portanto deveríamos sempre abreviar os nomes das variáveis.

Entretanto, atualmente não temos essa limitação, e os editores modernos possuem recursos para reaproveitar nomes de atributos e variáveis sem precisar digitar tudo novamente. Portanto, passe a usar **quantidade** e **valor**.

Utilize nomes claros. Evite, por exemplo, o atributo **data**. Uma data pode significar várias coisas: nascimento, morte, envio de um produto, cancelamento de venda. Escreva exatamente o que se deseja armazenar nesse atributo: **dataNascimento**, **dataObito**, etc.

# Métodos

Tendo identificado a classe com seus atributos, as seguintes perguntas podem surgir:

- Mas o que fazer com eles?
- Como utilizar a classe e manipular os atributos?

É nessa hora que o método entra em cena. Este é responsável por identificar e executar as operações que a classe fornecerá. Essas operações, via de regra, têm como finalidade manipular os atributos.

Para facilitar o processo de identificação dos métodos de uma classe, podemos pensar em verbos. Isso ocorre devido à sua própria definição: **ações**. Ou seja, quando se pensa nas ações que uma classe venha a oferecer, estas identificam seus métodos.

# Métodos

No processo de definição de um método, a sua assinatura deve ser identificada. Esta nada mais é do que o nome do método e sua lista de parâmetros. Mas como nomear os métodos? Novamente, uma expressividade ao nome do método deve ser fornecida, assim como foi feito com o atributo.

Por exemplo, no contexto do hospital, imagine termos uma classe **Procedimento**, logo, um péssimo nome de método seria **calcular**. Calcular o quê? O valor total do procedimento, o quanto cada médico deve receber por ele, o lucro do plano? Neste caso, seria mais interessante **calcularTotal**, **calcularGanhosMedico**, **calcularLucro**.

Veja que, ao lermos esses nomes, logo de cara já sabemos o que cada método se propõe a fazer. Já a lista de parâmetros são informações auxiliares que podem ser passadas aos métodos para que estes executem suas ações. Cada método terá sua lista específica, caso haja necessidade. Esta é bem livre e, em determinados momentos, podemos não ter parâmetros, como em outros podemos ter uma classe passada como parâmetro, ou também tipos primitivos e classes ao mesmo tempo.



# Métodos

Há também a possibilidade de passarmos somente tipos primitivos, entretanto, isto remete à programação procedural e deve ser desencorajado. Via de regra, se você passa muitos parâmetros separados, talvez eles pudessem representar algum conceito em conjunto. Neste caso, valeria a pena avaliarmos se não seria melhor criar uma classe para aglutiná-los.

Por fim, embora não faça parte de sua assinatura, os métodos devem possuir um retorno. Se uma ação é disparada, é de se esperar uma reação. O retorno de um método pode ser qualquer um dos tipos primitivos vistos na seção sobre atributos.

Além destes, o método pode também retornar qualquer um dos conceitos (classes) que foram definidos para satisfazer as necessidades do sistema em desenvolvimento, ou também qualquer outra classe - não criada pelo programador - que pertença à linguagem de programação escolhida.

# Dois métodos especiais

Em uma classe, independente de qual conceito ela queira representar, podemos ter quantos métodos forem necessários. Cada um será responsável por uma determinada operação que a classe deseja oferecer. Além disso, independente da quantidade e da finalidade dos métodos de uma classe, existem dois especiais que toda classe possui: o construtor e o destrutor.

Características do construtor:

- Responsável por criar objetos a partir da classe em questão;
- Prover valores iniciais para o objeto;
- Possui nome igual ao da classe;
- Não possui retorno, ao contrário dos outros métodos (omitimos inclusive o **void**);
- Está presente de forma implícita em linguagens como Java e C#;
- Construtores implícitos possuem como assinatura o mesmo nome da classe e não possuem parâmetros.



# Dois métodos especiais

Características do destrutor:

- Destrói o objeto criado a partir da classe;
- Seu nome padrão em Java é **finalize**, e retorna o tipo **void**;
- Não possui parâmetros;
- Usar um destrutor libera possíveis recursos do computador;
- O Java fornece um destrutor implícito para toda classe;
- Não devemos utilizar diretamente, para isso o Java tem o **Garbage Collector**.

# Garbage Collector

- É um programa usual em toda linguagem orientada a objetos, como Smalltalk 80, Java e C#;
- No caso do Java, é um programa que roda dentro da JVM;
- O Garbage Collector é responsável por automaticamente identificar objetos que não estão sendo mais usados e eliminá-los;
- No momento da eliminação, o Garbage Collector utiliza os destrutores definidos nas classes;
- Ele possui algoritmos de identificação de objetos ociosos e elimina os que não são mais usados, tirando a responsabilidade do desenvolvedor.

# Sobrecarga de métodos

Muitas vezes, é preciso que um mesmo método possua entradas (parâmetros) diferentes. Isso ocorre porque ele pode precisar realizar operações diferentes em determinado contexto. Este processo é chamado de **sobrecarga de método**.

Para realizá-la, devemos manter o nome do método intacto, mas alterar sua lista de parâmetros. Podem ser acrescentados ou retirados parâmetros para assim se prover um novo comportamento. Por exemplo, se uma determinada aplicação tivesse uma classe para representar um quadrilátero, ela deveria se chamar **Quadrilátero** e possuir o método **calcularArea**, seguindo as boas práticas já citadas. Mas sabemos que um quadrado, retângulo, losango e trapézio também são quadriláteros.

Mais interessante do que possuir um método para cada figura (**calcularAreaQuadrado**, **calcularAreaLosango**, etc.) é definir o mesmo método **calcularArea** com uma lista de parâmetros que se adeque a cada um desses quadriláteros.

# Sobrecarga de métodos

```
class Quadrilatero {  
    // área do quadrado  
    double calcularArea(double lado) {  
        return lado * lado;  
    }  
  
    // área do retângulo  
    double calcularArea(double baseMaior, double baseMenor) {  
        return baseMaior * baseMenor;  
    }  
  
    // área do trapézio  
    double calcularArea(double baseMaior, double baseMenor, double altura) {  
        return ((baseMaior * baseMenor) * altura) / 2;  
    }  
  
    // área do losango  
    double calcularArea(float diagonalMaior, float diagonalMenor) {  
        return diagonalMaior * diagonalMenor;  
    }  
}
```

# Sobrecarga de métodos

Caso haja necessidade, os tipos também podem ser mudados.

Sempre que a lista de parâmetros muda - seja acrescentando ou eliminando parâmetros, mudando seus tipos e até mesmo sua sequência -, estaremos criando sobrecargas de um método. Mas lembre-se de que o nome dele deve permanecer intacto.

A vantagem de usar a sobrecarga não se limita à "facilidade" de se manter o mesmo nome do método. Na verdade, existe uma questão conceitual, que é manter a abstração.

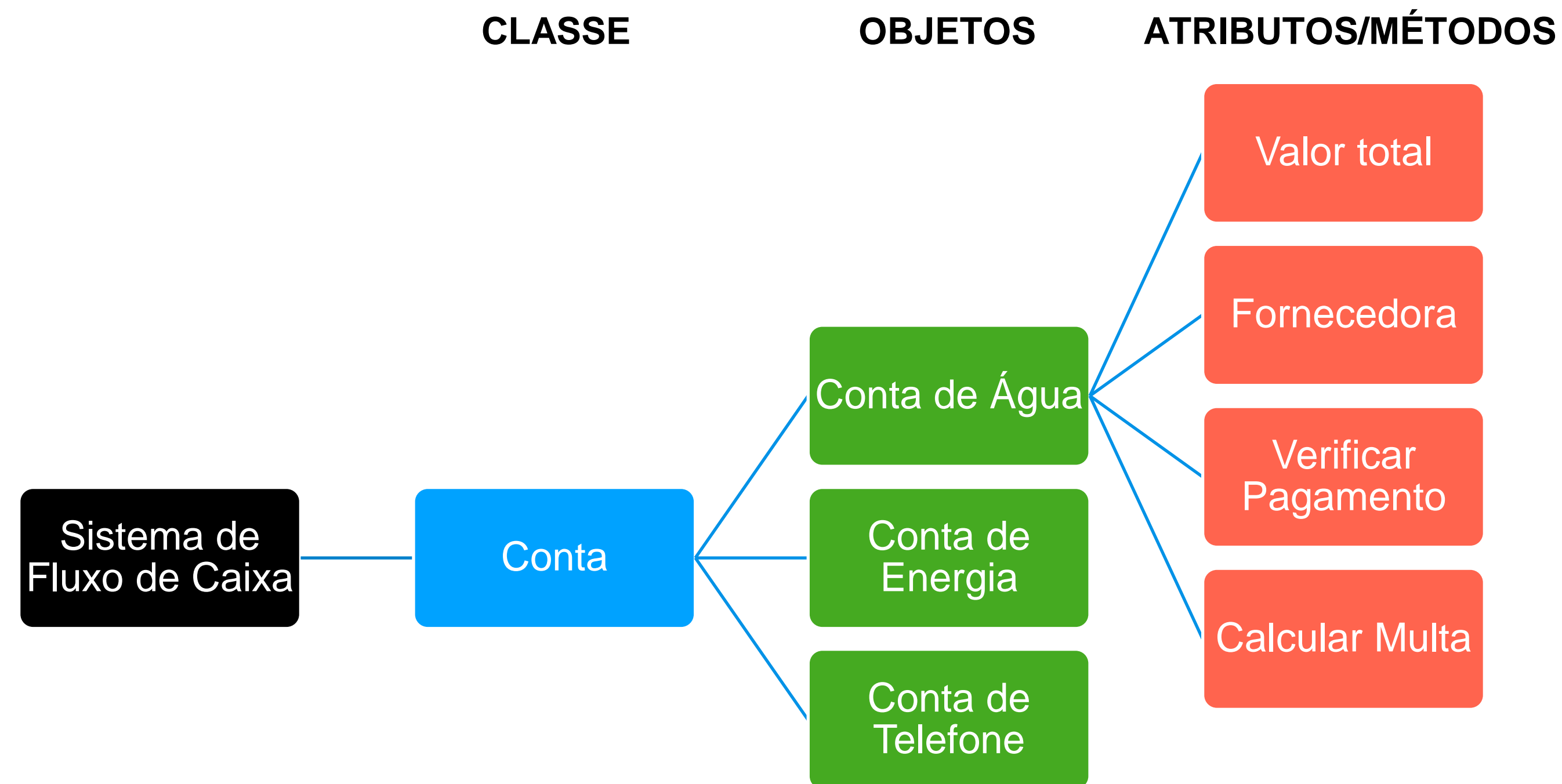
Assim, no caso de nosso exemplo do quadrilátero, a abstração é calcular sua área, independente de sua real forma. Deste modo, a sobrecarga possibilitou que tal cálculo pudesse receber os devidos parâmetros de acordo com a sua necessidade - no caso, a forma do quadrilátero - e, mesmo assim, manteve-se a abstração-alvo, que é calcular a área.



# Objetos

Objeto é a instanciação de uma classe.

Como já explicado, a classe é a abstração base a partir da qual os objetos serão criados. Quando se usa a OO para criar um software, primeiro pensamos nos objetos que ele vai manipular/representar. Tendo estes sido identificados, devemos então definir as classes que servirão de abstração base para que os objetos venham a ser criados (instanciados).





OBRIGADO!



[www.ibmec.br](http://www.ibmec.br)

 /ibmec

 ibmec

 @ibmec\_oficial

 ibmec

