Programmierung 2

DHBW Mannheim - WWI17SCB - May 2018

Bastian Heist

* Full-Stack Webentwickler @ Sandstorm Media GmbH
* Neos CMS Core Team - Projektmarketing
* SAP ERP Consultant @ Merck KGaA

@beheist bastian.heist@sandstorm.de

sandstorm

You?

* Namensschilder
* Firma
* Vorerfahrungen mit Programmierung

Spielregeln

Grundsätzliches

Laptops

Ablauf

Fragen

Übungen

Prüfungsleistungen

Programmierprojekt = 2/3

Klausur = 1/3

Programmierprojekt

* Programmiert ein **"Vier Gewinnt"-Spiel**
* Gruppen à 5-6 Personen
* Schickt mir die Gruppeneinteilungen bis zur nächsten VL per Mail
* Abgabetermin: Tag der Klausur
* Abgabeform: Git-Repository auf github.com

Programmierprojekt - Anforderungen

* Grafische Oberfläche
* Zwei menschliche Spieler können gegeneinander spielen
* Ein menschlicher Spieler kann gegen den Computer spielen
  (Mindestanforderung: Zufallszüge des Computerspielers)
* Anlegen beliebig vieler Spielerprofile (Name)
* Auswahl von Spielerprofilen vor dem Start
* Speichern der Spielhistorie (Siege, Niederlagen, ø Züge, ...)
* Anzeige der Spielhistorie aller Spieler in einem Highscore-Menü

Programmierprojekt - Bewertungskriterien

* **50% Funktionalität**
  * Vorhandensein aller angeforderten Funktionen
  * Fehlerfreies Funktionieren (keine Abstürze, Hänger etc.)
  * Stärke des Computerspielers
* **20% Code-Qualität**
  * Verständlicher und lesbarer Code, sinnvolle Kommentare
  * Sinnvolle Verwendung von Stoff aus der VL (Interfaces, Exceptions, Collections...)

Programmierprojekt - Bewertungskriterien

* **20% Tests**
  * Testabdeckung des bestehenden Codes
  * Sinnvolle Tests

* **10% Dokumentation**
  * Mindestens: Installationsanweisung (README.md)
  * Zusätzlich: ER-Diagramme, ...
  * Verständliche Git-History

Programmierprojekt - Bewertungskriterien

* **20% Kreative Zusatzfunktionen**
  * Tobt euch aus!

Klausur

Prüfungsrelevantes Thema

Nicht prüfungsrelevantes (aber spannendes!) Thema

Vorlesungsinhalte

Vorlesungsinhalte

1. Interfaces
2. Exception Handling
3. Collection Framework
4. GUIs mit Swing
5. Streams

(6. Threads - wenn Zeit)

Zusatzthemen für Interessierte

1. Versionsverwaltung / Git
2. Testing / TDD
3. Frameworks / Dependency Management
4. Richtig Dokumentieren

Je nach Bedarf / Zeit in der Vorlesung

Slides

https://goo.gl/U45wY3

1. Interfaces

Interfaces - Lernziele

* Wesentliche Eigenschaften von Interfaces beschreiben
* Interfaces deklarieren und implementieren
* Interfaces vererben
* Interfaces in der UML darstellen
* Bedeutung von Interfaces in Bezug auf Mehrfachvererbung
* Polymorphie mit Hilfe von Interfaces realisieren

Interfaces - Eigenschaften

* Besondere Form der Klassen
* Definieren lediglich den Aufbau einer Schnittstelle
  * Namen von Methoden
  * Signaturen der Methoden
  * Attribute
* Beinhalten keine eigene Funktionalität
* Funktionalität wird in Klassen implementiert, die das Interface nutzen wollen

Interfaces - Eigenschaften

* Ausschließlich abstrakte öffentliche Methoden
* Alle Attribute sind Konstanten
* Deklaration mit dem Schlüsselwort `interface`
* Vererbung von Interfaces mit `extends`

Interfaces - Definition und Verwendung

```
1 interface Animal {
2
3     // Implizit: public static final int SPEED = 42;
4     int SPEED = 42;
5
6     // Implizit: public void eat(String food);
7     void eat(String food);
8
9     // Implizit: public void travel();
10    void travel();
11 }
```

Interfaces - Definition und Verwendung

```
1 public class Dog implements Animal {
2
3     @Override
4     public void eat(String food) {
5         System.out.println("The dog eats a " + food);
6     }
7
8     @Override
9     public void travel() {
10        System.out.println("The dog runs at " + this.SPEED + " km/h.");
11    }
12 }
```

Überschreiben von Methoden

Die `@Override`-Annotation:

1. Compiler prüft, ob Methode wirklich überschrieben wird
2. Lesbarkeit / Verständlichkeit des Codes wird erhöht

```
1 @Override
2 public void eat2() {
3   ...
4 }
```

```
1 Error:(14, 5) java: method does not override or implement a method from a supertype
```

Interfaces - Definition und Verwendung

```
1 public class Zoo {
2
3     public static void main(String[] args) {
4         Dog myDog = new Dog();
5         myDog.eat("bone");
6         myDog.travel();
7     }
8 }
```

Interfaces - Definition und Verwendung

```
1 The dog eats a bone.
2 The dog runs at 42 km/h.
```

Interfaces - Eigenschaften

* Um zu einem Interface die Funktionalität zur Verfügung zu stellen, muss eine Klasse das Interface implementieren
* Klasse **muss alle** Methoden des Interfaces implementieren oder als abstrakte Klasse deklariert werden

Interfaces - Eigenschaften

* Definition eindeutiger Schnittstellen

```
1  public class Logger {
2
3      static void log(Describable thing) {
4          System.out.println("I logged a " + thing.getDescription());
5      }
6
7      public static void main(String[] args) {
8          Something myThing = new Something();
9          log(myThing);
10     }
11 }
```

Interfaces - Eigenschaften

* Definition eindeutiger Schnittstellen

```
1 interface Describable {
2     String getDescription();
3 }
4
5 public class Something implements Describable {
6     public String getDescription() {
7         return "Something!";
8     }
9 }
```

Interfaces - Polymorphie

* Verwendung als Datentyp für Referenzvariablen

`myDog` hat hier den Laufzeittyp `Animal` anstelle von `Dog`

```
1 public class Zoo {
2
3     public static void main(String[] args) {
4         Animal myDog = new Dog();
5         myDog.eat("Your Homework");
6         myDog.travel();
7     }
8 }
```

Interfaces - Mehrfachvererbung

* Klassen können im Gegensatz zur Vererbung mehrere Interfaces implementieren
* Durch die Implementierung mehrerer Interfaces wird eine Art von Mehrfachvererbung realisiert
* Objekte einer Klasse sind zu **allen** Interface-Datentypen kompatibel, sobald die Klasse das Interface implementiert

Interfaces - Mehrfachvererbung

```
1 public interface Foo {
2     void foo();
3 }
4
5 public interface Bar {
6     void bar();
7 }
```

Interfaces - Mehrfachvererbung

```
1  public class Baz implements Foo, Bar {
2      public void foo() {
3          System.out.println("foo!");
4      }
5      public void bar() {
6          System.out.println("bar!");
7      }
8  }
9
10 ...
11
12 Foo foo = new Baz();
13 Bar bar = new Baz();
```

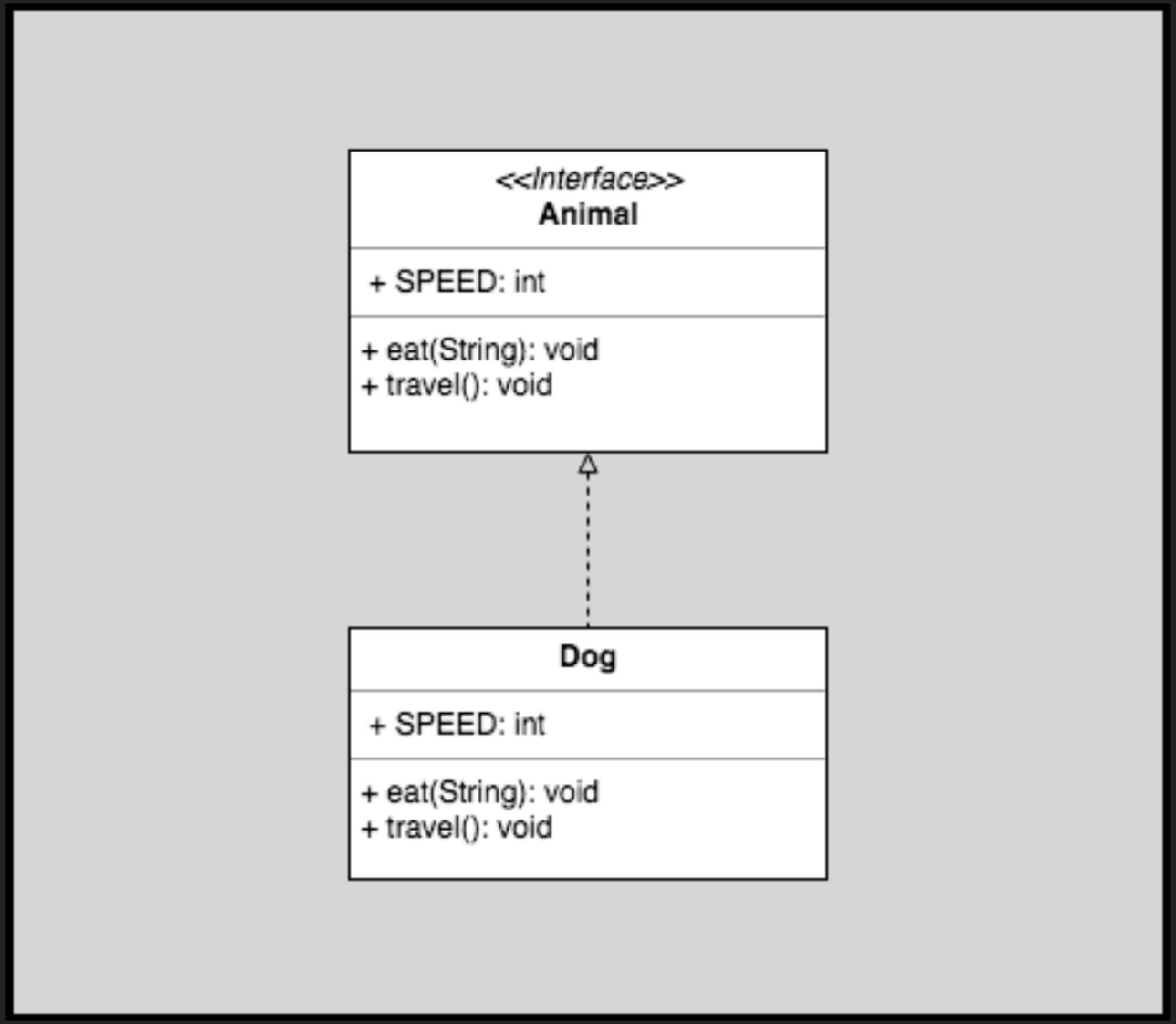
Interfaces - Mehrfachvererbung

* Feinheiten bei der Polymorphie mit Interfaces:

* Polymorphie durch Vererbung: Objekte einer **Subklasse** werden in Referenzvariablen vom **Typ der Superklasse** gespeichert

* Polymorphie durch Interfaces: Objekte der **implementierenden Klasse** werden in Referenzvariablen vom **Typ des implementierten Interfaces** gespeichert

Interfaces - UML



Interfaces - Default-Methoden

* ~~Beinhalten keine eigene Funktionalität~~
* Doch! Seit Java 8 gibt es **`default`**-Methoden

```
1  interface Animal {
2
3      // Implizit: public static final int CONSTANT = 42;
4      int SPEED = 42;
5
6      // Implizit: public void eat();
7      default void eat(String food) {
8          System.out.println("This " + this.getClass().getSimpleName() + " eats a " + food);
9      }
10
11     // Implizit: public void travel();
12     void travel();
13 }
```

Interfaces - Default-Methoden

```
1 public class Snake implements Animal {
2     @Override
3     public void travel() {
4         System.out.println("The snake slithers at " + this.SPEED + " km/h.");
5     }
6 }
7
8 public class Zoo {
9     public static void main(String[] args) {
10        Animal mySnake = new Snake();
11        mySnake.eat("frog");
12        mySnake.travel();
13    }
14 }
```

Interfaces - Default-Methoden

```
1 This Snake eats a frog
2 The snake slithers at 42 km/h.
```

* In anderen Sprachen (zB Scala, PHP) heißen Interfaces mit Default-Funktionalität "Traits".

Interfaces - Tag- oder Marker-Interfaces

* Interface ohne Methoden oder Attribute

```
1 public interface EventListener {}
```

* Verwendung als Sammel-Parentklasse für viele Unterklassen
* Betont wichtige Aspekte die sonst nirgends explizit codiert sind
* Erlaubt schnelles identifizieren aller "gemarkerten" Klassen
* Die gemeinsamen Eigenschaften bzw. der Zweck des Markers wird sinnvollerweise in dessen Javadocs beschrieben

Pause

Coding Time!

Interfaces - Codingaufgabe

30m

* Baut eine mehrstufige Interface-Hierarchie auf (evtl. ein Marker auf höchster Stufe)
* Legt einige abstrakte Methoden und einige Methoden mit default-Funktionalität an
* Baut mehrere Klassen, die diese Interfaces implementieren
* Baut eine main-Methode, die über ein Array verschiedener Objekte iteriert und die Interfacemethoden auf den Objekten aufruft
* Verwendet als Typ des Arrays einen Interfacetyp

2. Exception Handling

Exception Handling - Lernziele

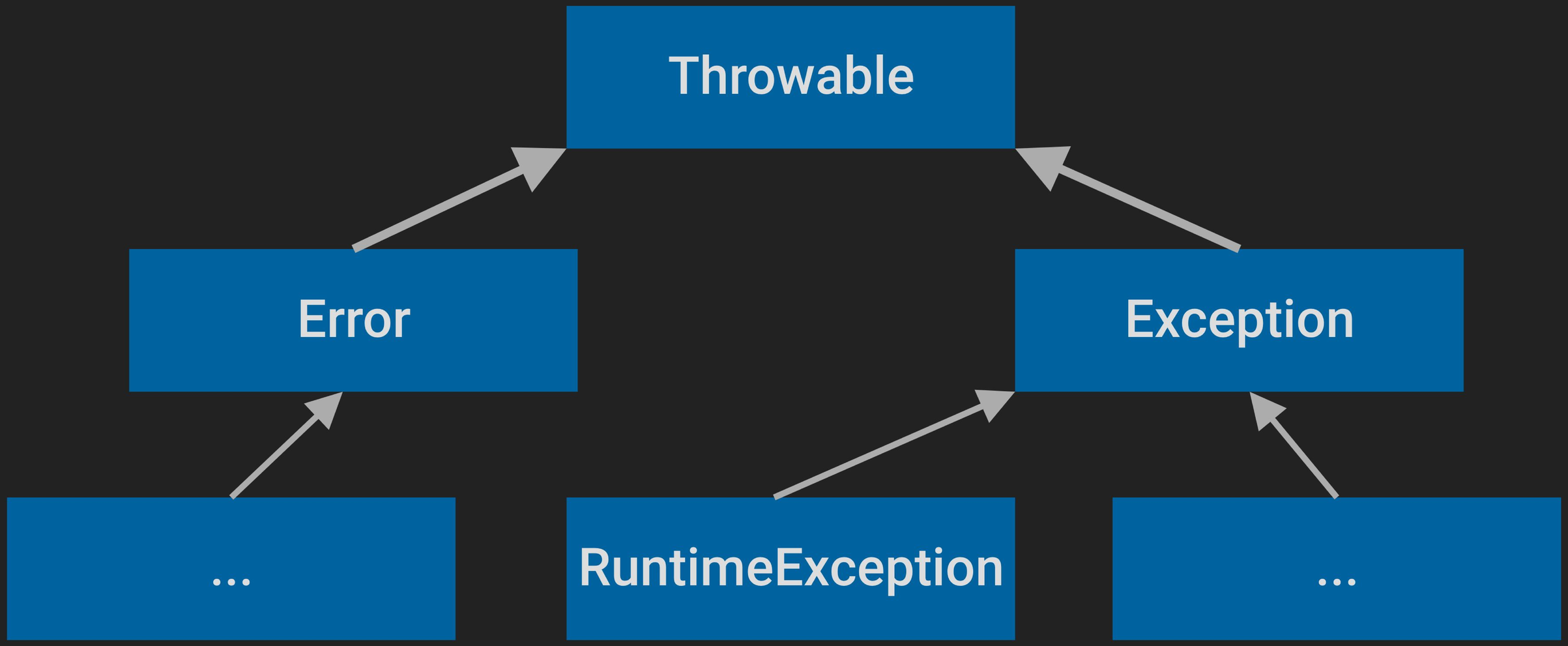
* Unterschiedliche Exceptions in Java kennen
* Eigene Exceptionklassen definieren
* Exceptions auslösen und weitergeben
* Exceptions behandeln
* Exceptionskonzept in Java erläutern
* Unterschied zwischen checked und unchecked Exceptions kennen

Exception Handling - Basics

* Compiler-Fehler
  * Entstehen beim Kompilieren des Programms
  * Können nicht behandelt werden, Programm startet nicht

* Laufzeitfehler
  * Error --> ernstes Problem, **sollte nicht** behandelt werden
  * Exceptions --> Ausnahmezustand, **können** (unchecked) bzw. **müssen** (checked) behandelt werden

Exception Handling - Basics



Exception Handling - Basics

!!! Terminologie !!!

* Exception (Konzept, bezieht sich auf **alle** Ausnahmeklassen - auch auf Errors)
* `java.lang.Exception` (Unterklasse von Throwable)

Java hat hier leider Konzept und Klassenname gleichgesetzt.

Exception Handling - Funktionsprinzip

* Auslösen einer Exception durch:
  * Explizites "werfen" einer E mittlels `throw`
  * Laufzeitfehler (z.B. Teilen durch 0, NullPointerException, ...)

* Fehlerbehandlung:
  * Direkt in der Methode, wo die E entsteht **oder**
  * Weitergabe an die aufrufende Stelle

Exception Handling - Funktionsprinzip

* Aufrufende Stelle kann die Ausnahme behandeln
* Wird sie nicht behandelt, wird eine Ausnahme so lange weitergegeben, bis sie an die JVM gelangt
* Resultat: Fehlermeldung und Programmabbruch

Exception Handling - Explizites Auslösen

```
1 public class TestException {
2     public static void main(String[] args) {
3         doThing();
4     }
5
6     private static void doThing() {
7         throw new RuntimeException("You screwed up!");
8     }
9 }
```

Exception Handling - Explizites Auslösen

```
1 Exception in thread "main" java.lang.RuntimeException: You screwed up!
2     at exceptions.basics.TestException.doThing(TestException.java:9)
3     at exceptions.basics.TestException.main(TestException.java:5)
4
5 Process finished with exit code 1
```

Exception Handling - Laufzeitfehler

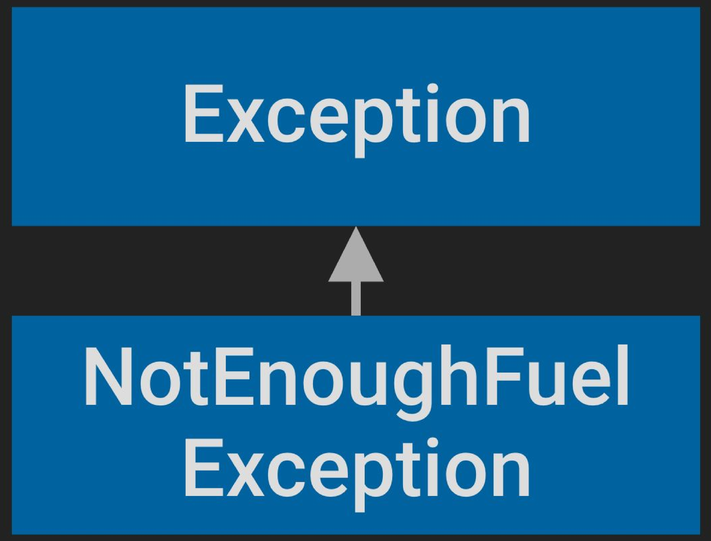
```
1 import java.util.Set;
2
3 public class TestException2 {
4     public static void main(String[] args) {
5         Set mySet = null;
6
7         int size = mySet.size();
8     }
9 }
```

Exception Handling - Laufzeitfehler

```
1 Exception in thread "main" java.lang.NullPointerException
2     at exceptions.basics.TestException2.main(TestException2.java:9)
3
4 Process finished with exit code 1
```

Exception Handling - Eigene Ausnahmeklassen

```
1 class NotEnoughFuelException extends Exception {
2     public NotEnoughFuelException(double remainingDistance, double requiredDistance) {
3         super("Remaining fuel is only sufficient for " + remainingDistance +
4                 " km; you wanted " + requiredDistance);
5     }
6 }
```



Exception Handling - Eigene Ausnahmeklassen

```
1  public class Car {
2      private double distanceTravelled = 0;
3      private double fuelInTank = 100;
4      private static final double LITERS_PER_100KM = 10;
5
6      public void drive(double distance) throws NotEnoughFuelException {
7          double fuelNeeded = distance * LITERS_PER_100KM / 100;
8
9          if (fuelNeeded <= fuelInTank) {
10             fuelInTank -= fuelNeeded;
11             distanceTravelled += distance;
12             System.out.println("Remaining fuel after " + distanceTravelled + "km: " + fuelInTank);
13         } else {
14             throw new NotEnoughFuelException(fuelInTank * LITERS_PER_100KM, distance);
15         }
16     }
17 }
18
```

Exception Handling - Eigene Ausnahmeklassen

```
1 public class RaceTrack {
2     public static void main(String[] args) throws NotEnoughFuelException {
3         Car myCar = new Car();
4         myCar.drive(900);
5         myCar.drive(120);
6     }
7 }
```

Exception Handling - Eigene Ausnahmeklassen

```
1 Exception in thread "main" exceptions.customExceptionClasses.NotEnoughFuelException:
2 Remaining fuel is only sufficient for 100.0 km; you wanted 120.0
3     at exceptions.customExceptionClasses.Car.drive(Car.java:16)
4     at exceptions.customExceptionClasses.Track.main(Track.java:9)
5
6 Process finished with exit code 1
```

Exception Handling - Abfangen und Behandeln

* Überwachen eines bestimmten Programmbereichs auf Fehler

  --> `try`

* Abfangen expliziter Fehlerklassen

  --> `catch (Exception e)`

* Behandlung mehrerer verschiedener Fehlerklassen möglich
* Ausführung von Cleanup-Code nach dem Überwachungsbereich

  --> `finally`

* Weitergabe an aufrufende Stelle

  --> `throws`

Exception Handling - Abfangen und Behandeln

```
1  public class RaceTrack {
2      public static void main(String[] args) {
3          Car myCar = new Car();
4          double distance = Math.random() * 2000;
5
6          try {
7              myCar.drive(distance);
8          } catch (NotEnoughFuelException e) {
9              System.out.println("Not enough fuel for " + distance + " km.");
10         } finally {
11             System.out.println("Remaining fuel: " + myCar.getFuelLeft());
12         }
13     }
14 }
```

Exception Handling - Abfangen und Behandeln

```
Not enough fuel for 1925.7513063822435 km.
Remaining fuel: 100.0

Process finished with exit code 0
```

```
Remaining fuel after 689.7871005210865km: 31.02128994789136
Remaining fuel: 31.02128994789136

Process finished with exit code 0
```

Exception Handling - Wozu finally?

```
1  public class RaceTrack {
2      public static void main(String[] args) {
3          Car myCar = new Car();
4          double distance = Math.random() * 2000;
5
6          try {
7              myCar.drive(distance);
8          } catch (NotEnoughFuelException e) {
9              System.out.println("Not enough fuel for " + distance + " km.");
10         }
11         System.out.println("Remaining fuel: " + myCar.getFuelLeft());
12     }
13 }
```

Exception Handling - Wozu finally?

* `finally` wird **immer** ausgeführt, auch wenn
  * Die Methode innerhalb von `try` mittels `return` verlassen wird
  * Ein `catch`-Block selbst eine Exception wirft
  * Eine Exception geworfen wird, die kein `catch`-Block fängt

* Was sollte in `finally`-Blöcke?
  * Aufräumcode, z.B. schließen von Resourcen-Handles

Exception Handling - Wozu finally?

```java
public class WeirdStuff {
    public static void main(String[] args) {
        System.out.println(testFinally());
    }

    private static int testFinally() {
        try {
            return 1;
        }
        finally {
            return 2;
        }
    }
}
```

```
2

Process finished with exit code 0
```

Exception Handling - Wichtige Methoden

* **`public String getMessage()`** --> Fehlertext

```
Remaining fuel is only sufficient for 100.0 km; you wanted 120.0
```

* **`public String toString()`** --> Objektbeschreibung und Fehlertext

```
exceptions.customExceptionClasses.NotEnoughFuelException:
Remaining fuel is only sufficient for 100.0 km; you wanted 120.0
```

* **`public void printStackTrace()`** --> Stacktrace

```
exceptions.customExceptionClasses.NotEnoughFuelException:
Remaining fuel is only sufficient for 100.0 km; you wanted 120.0
    at exceptions.customExceptionClasses.Car.drive(Car.java:16)
    at exceptions.customExceptionClasses.Track.main(Track.java:9)
```

Exception Handling - Unchecked vs Checked

* Checked Exceptions
  * Sind **alle Exceptions** außer `Error` und `RuntimeException` sowie deren Unterklassen
  * Müssen durch den Programmierer behandelt werden (durch `try/catch`) oder explizit an den Aufrufer weitergegeben werden (durch `throws`)

Exception Handling - Unchecked vs Checked

* Unchecked Exceptions
  * Sind vom Typ `Error` oder `RuntimeException`
  * Treten oftmals durch Programmierfehler auf (z.B. div/0)
  * Müssen nicht (sollten aber!) behandelt werden

Exception Handling - Unchecked vs Checked

* Wenn eine unchecked Exception nicht behandelt wird:

```
3  public class RaceTrack {
4      public static void main(String[] args) {
5          Car myCar = new Car();
6          myCar.drive( distance: 900);
7          myCar.drive( distance: 120);
10
```

Unhandled exception: exceptions.customExceptionClasses.NotEnoughFuelException

```
1 Error:(8, 20) java:
2 unreported exception exceptions.customExceptionClasses.NotEnoughFuelException;
3 must be caught or declared to be thrown
```

Exception Handlung - Checked vs Unchecked

* Warum die Unterscheidung in Checked / Unchecked?

"Any Exception that can be thrown by a method is part of the method's public programming interface. Those who call a method must know about the exceptions that a method can throw so that they can decide what to do about them. These exceptions are as much a part of that method's programming interface as its parameters and return value."

Exception Handlung - Checked vs Unchecked

* Da aber RuntimeExceptions per Definition für den Aufrufer nicht vorhersehbar sind, müssen sie nicht per **throws** markiert werden
* Wenn ein Aufrufer eine auftretende Exception sinnvoll behandeln kann, sollte sie checked sein. Wenn nicht, unchecked.

"Generally speaking, do not throw a RuntimeException or create a subclass of RuntimeException simply because you don't want to be bothered with specifying the exceptions your methods can throw."

Coding Time!

Interfaces - Codingaufgabe

15m

* Überlegt euch eine kreative Einsatzmöglichkeit für eigene Exception-Klassen, und implementiert sie
* Experimentiert mit try/catch/finally
* Löst eine RuntimeException in eurem Programm aus und beobachtet, was passiert.

3. Collection Framework

4. GUIs mit Swing

5. Streams

6. Threads

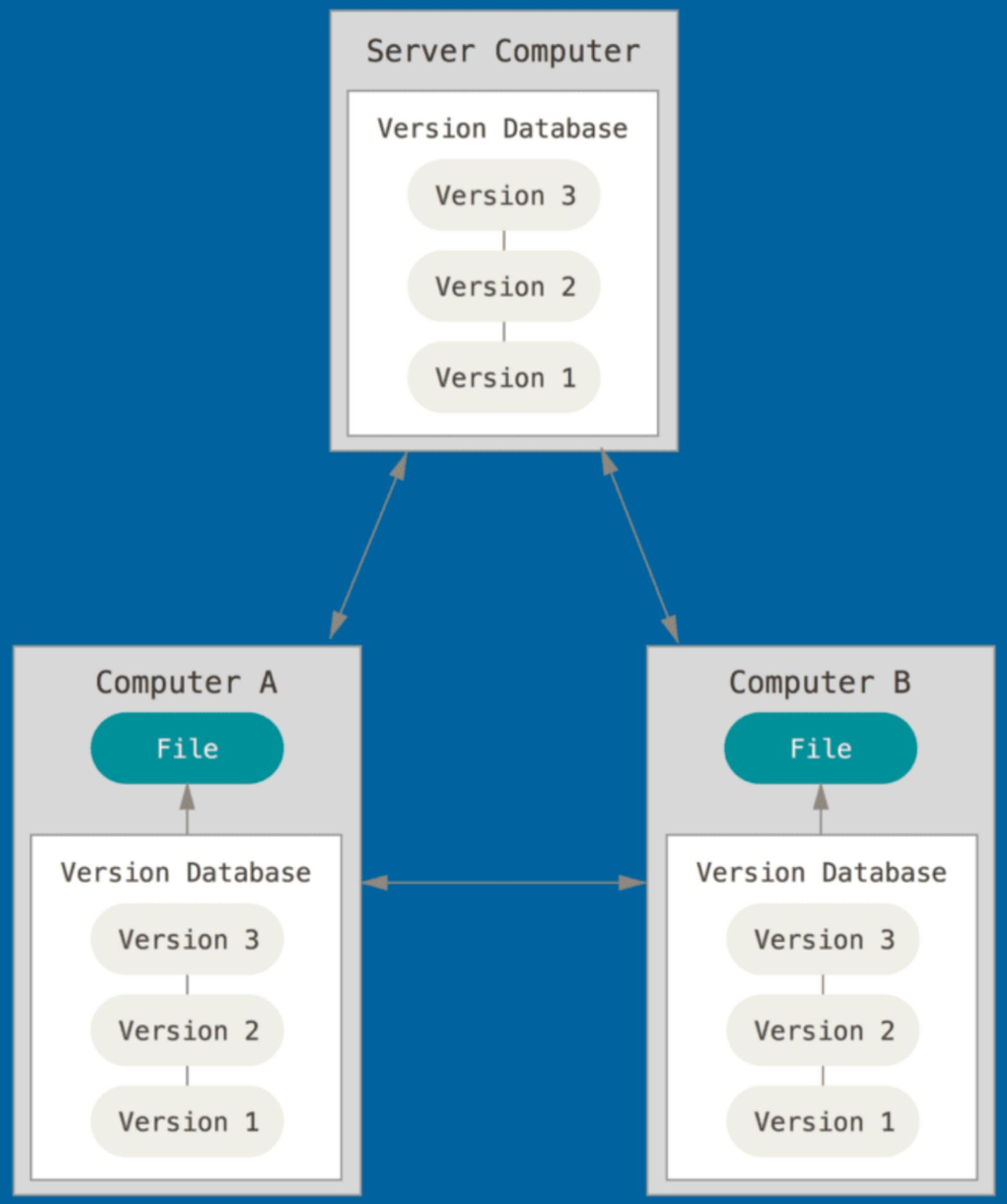
1. Versionsverwaltung

Was ist Versionsverwaltung?

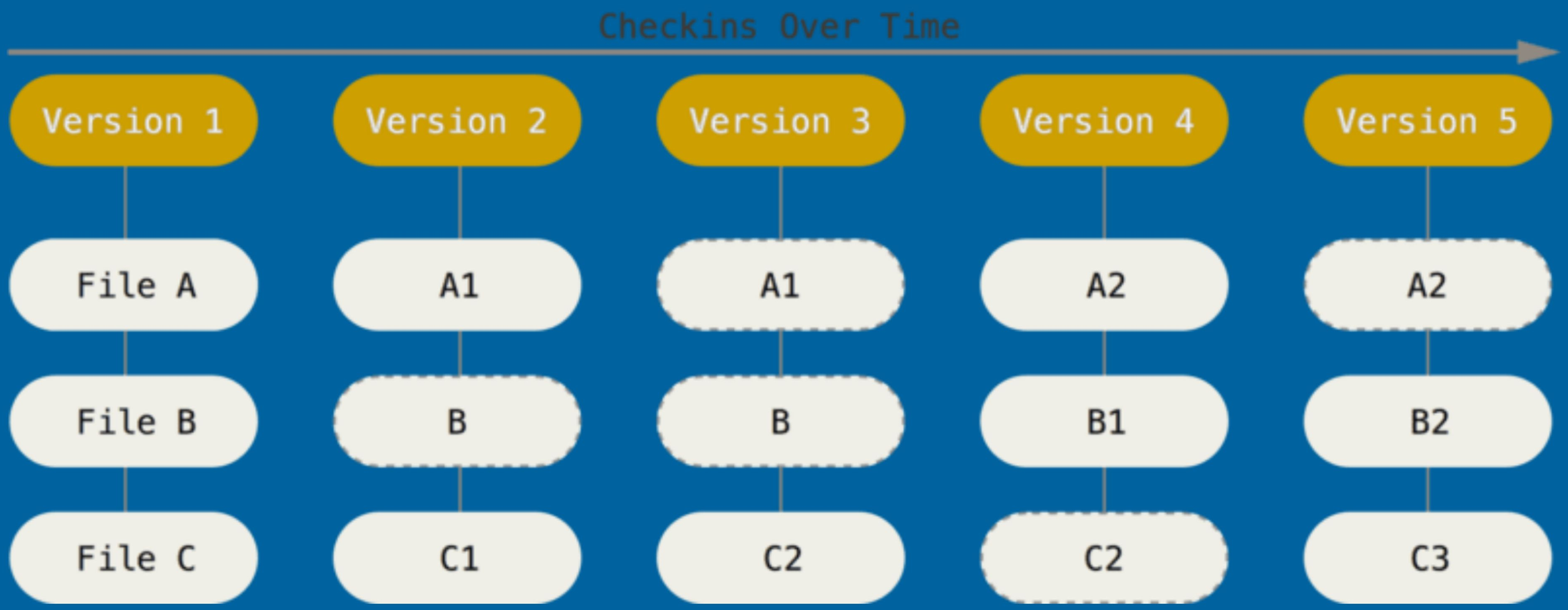
* Protokollieren von Änderungen an Dateien
* Verfolgung der Änderungshistorie über die Lebensdauer
* Herausfinden, wer Änderungen durchgeführt hat
* Änderungen finden, die Probleme verursachen
* Dateien auf einen früheren Zustand zurücksetzen
* Änderungen mit anderen teilen und deren Änderungen erhalten
* Grundlage für Continuous Integration / Delivery

Was ist Versionsverwaltung?

* Das mit Abstand meistverwendete VCS: Git (https://git-scm.com)
* Quelle: https://git-scm.com/book/en/v2



Source: https://git-scm.com/book/de/v2/Los-geht%E2%80%99s-Was-ist-Versionsverwaltung%3F



Source: https://git-scm.com/book/de/v2/Los-geht%E2%80%99s-Git-Grundlagen

Git Basics

* Pro Version (=Commit) wird eine SHA1-Checksum aller Dateien erstellt
* Sobald ein Commit erstellt ist, können die Änderungen praktisch nicht mehr verloren gehen
* Es ist möglich Commits zu löschen, dies sollte jedoch **nie** mit bereits veröffentlichten (mit anderen geteilten) Commits geschehen

--> *"Never change public history!"*

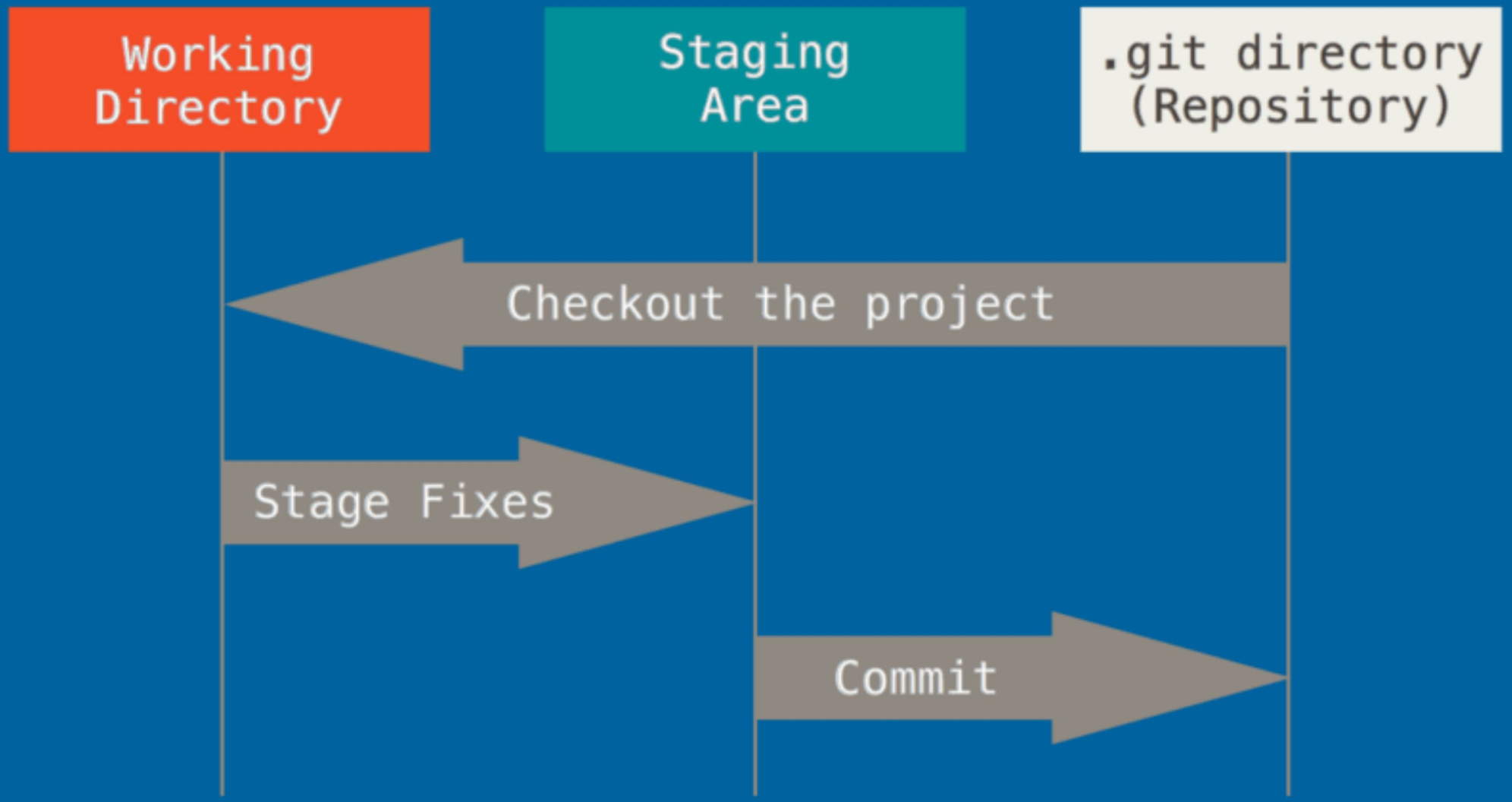
Git Basics - "Three Trees"

Arbeitsablauf mit Git:

1. Änderungen im Dateisystem ("Working Directory") machen
2. Bestimmte Änderungen für den nächsten Commit markieren ("Staging Area")
3. Änderungen committen, d.h. in die Git-History aufnehmen. ("Git Repository")

Ein Commit sollte funktional und logisch zusammengehörige Änderungen enthalten.

Git Basics - "Three Trees"



Source: https://git-scm.com/book/en/v2/Getting-Started-Git-Basics

Git Basics - Verwendung

* Nutzung von git - normalerweise über das CLI
* Einfache Installation: https://desktop.github.com/
* Mac-only: z.B. `brew install git`
* Der erste Schritt nach der Installation: eigene Identität festlegen

```
git config --global user.name "John Doe"
git config --global user.email johndoe@example.com
```

Git Basics - Repository anlegen

```
cd /your/project/directory
git init
```

Dann beliebige Dateien in das Verzeichnis legen.

Es ist natürlich auch möglich, in einem bereits gefüllten Verzeichnis ein git-Repository anzulegen.

```
$ git init
Initialized empty Git repository in /Users/bastian/src/vl/gitdemo/.git/
```

Git Basics - Dateien in die Staging Area aufnehmen

Alle Dateien im aktuellen Verzeichnis in die Staging Area schieben - damit werden sie in den nächsten Commit aufgenommen.

```
git add .
```

oder:

```
git add file1.txt
```

Git Basics - Status überprüfen

Zustand des Working Directory und der Staging Area anzeigen:

`git status`

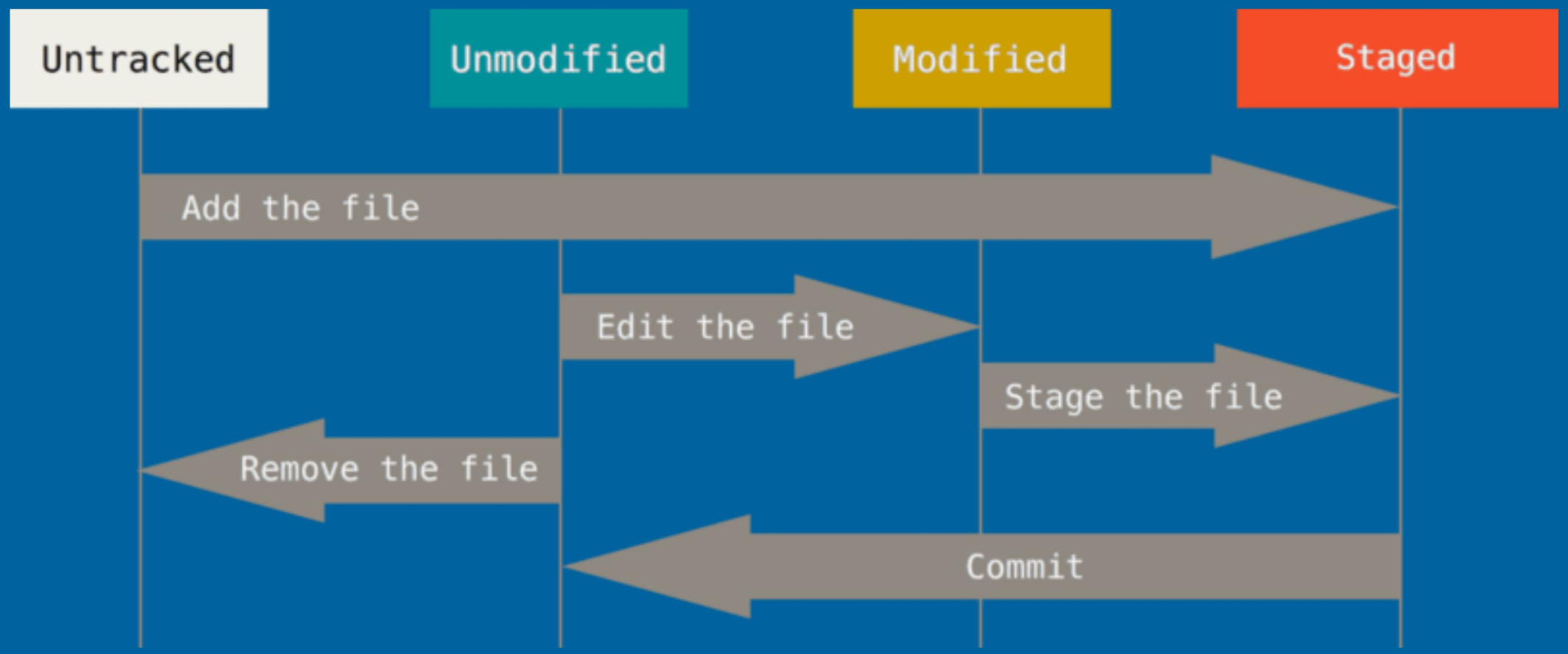
```
$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

    new file:   file1.txt
```

Git Basics - Status überprüfen



Source: https://git-scm.com/book/en/v2/Git-Basics-Recording-Changes-to-the-Repository

Git Basics - Commit erstellen

Alle Dateien in der Staging Area committen:

`git commit -m "Initial commit"`

```
$ git commit -m "Initial commit"
[master (root-commit) 54470f0] Initial commit
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 file1.txt
```

Git Basics - Commits anzeigen

`git log`

```
$ git log
commit 54470f019fbd79f85e150fbedaa0320c5528b17a (HEAD -> master)
Author: Bastian Heist <bastian.heist@sandstorm.de>
Date:   Fri May 25 01:34:51 2018 +0200
    Initial commit
```

Mit Anzeige der Änderungen:

`git log -p`

Besser hierfür: Git-GUI wie gitk oder GitHub Desktop

Git Basics - Repository clonen

"Clonen" eines bereits bestehenden Repositories.

Beispiel: Neos-Projekt

```
git clone git@github.com:neos/neos-base-distribution.git
```

Git Basics - Remotes

Durch clonen wird automatisch die URL, von der gecloned wurde, als remote "origin" angelegt.

```
git remote -v
```

Git Basics - Remotes

Remotes können auch manuell hinzugefügt werden:

```
git remote add remotename https://github.com/some/repo
```

Git Basics - Remotes

Änderungen aus einem Remote-Repository namens `origin` holen:

```
git fetch origin
```

Achtung: Änderungen sind damit noch nicht mit dem eigenen Arbeitsstand zusammengeführt ("gemerged"). Es werden nur die remote existierenden Commits abgeholt und lokal bereitgestellt.

Git Basics - Remotes

Änderungen holen und direkt in den eigenen Stand mergen:

```
git pull
```

Hierdurch wird ein merge der Remote-Änderungen in den lokalen Branch (siehe nächste Folien) ausgeführt.

Für eine sauberere Commit-History ohne Merge-Commits:

```
git pull --rebase
```

Git Basics - Remotes

Eigene Änderungen in den Branch "master" der Remote "pushen":

```
git push origin master
```

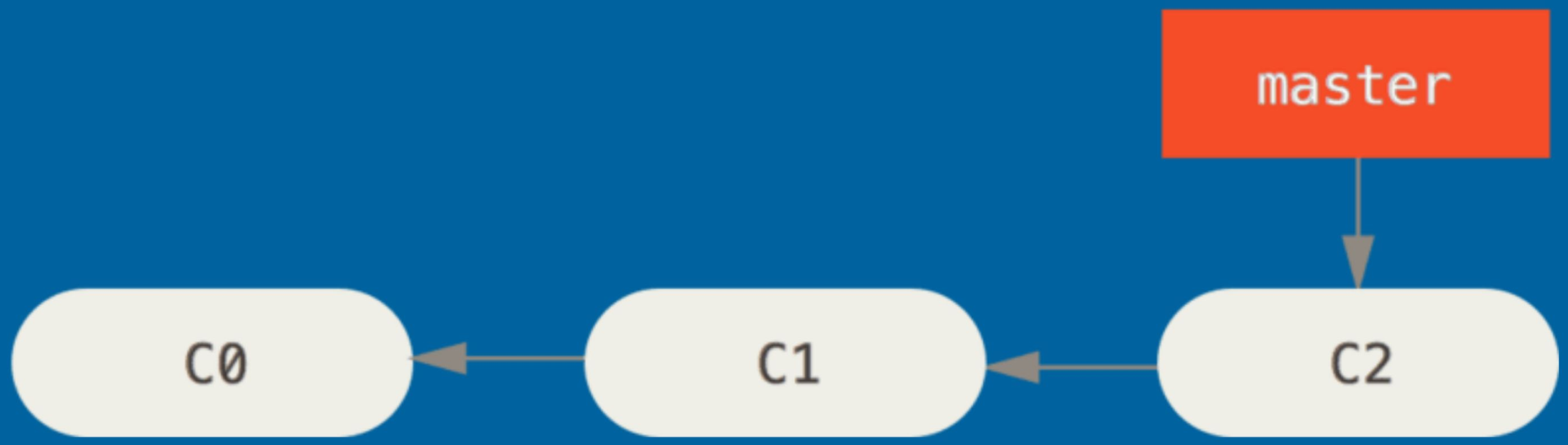
Falls der aktuelle Branch noch keinen "Upstream"-Branch hat, kann dieser zuvor festgelegt werden:

```
git push --set-upstream origin master
```

Danach reicht dann:

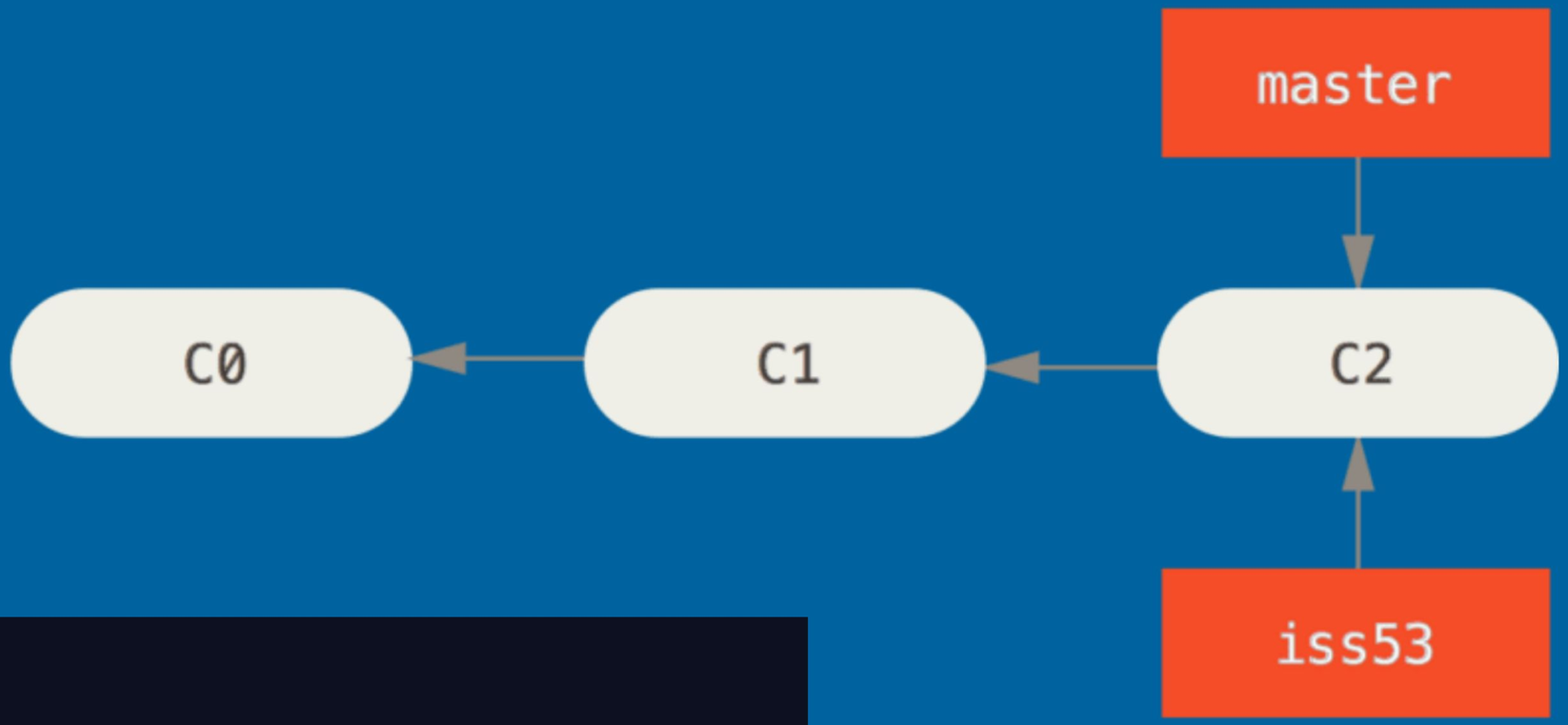
```
git push
```

Git Basics - Branching



Source: https://git-scm.com/book/en/v2/Git-Branching-Basic-Branching-and-Merging

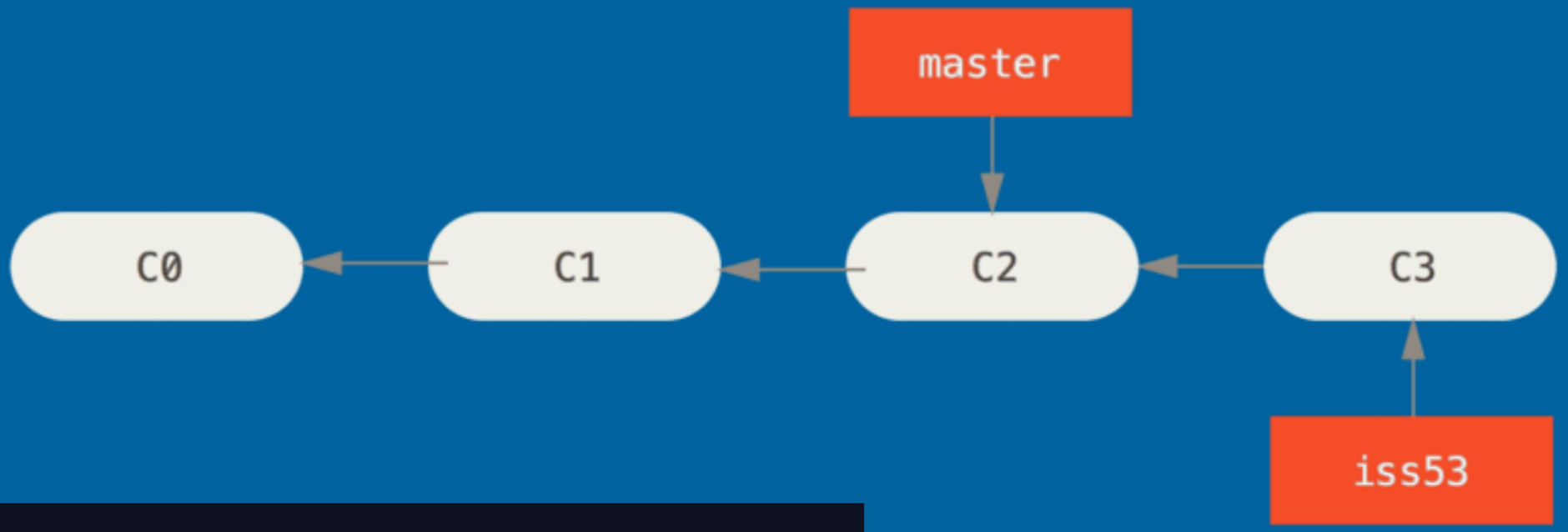
Git Basics - Branching




```
$ git branch iss53
$ git checkout iss53
```

Source: https://git-scm.com/book/en/v2/Git-Branching-Basic-Branching-and-Merging

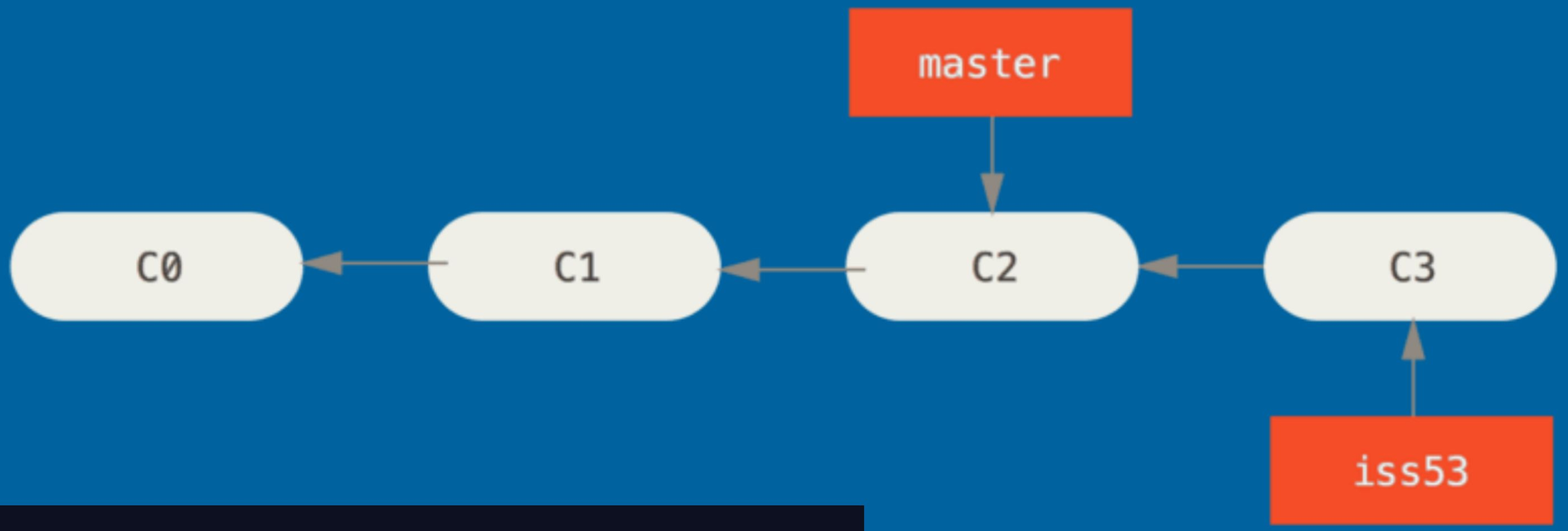
Git Basics - Branching




```
$ vim index.html
$ git commit -a -m 'added a new footer [issue 53]'
```

Source: https://git-scm.com/book/en/v2/Git-Branching-Basic-Branching-and-Merging

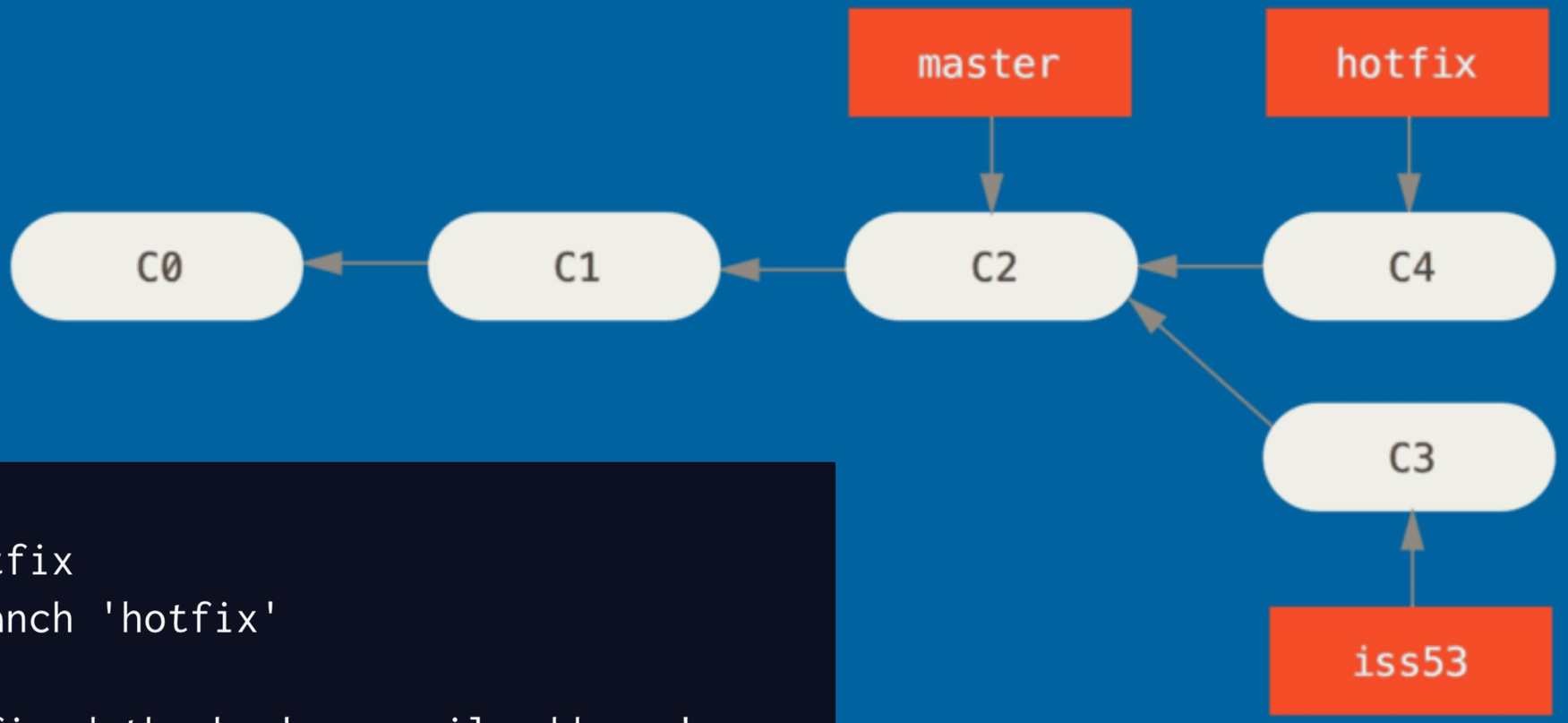
Git Basics - Branching




```
$ git checkout master
Switched to branch 'master'
```

Source: https://git-scm.com/book/en/v2/Git-Branching-Basic-Branching-and-Merging

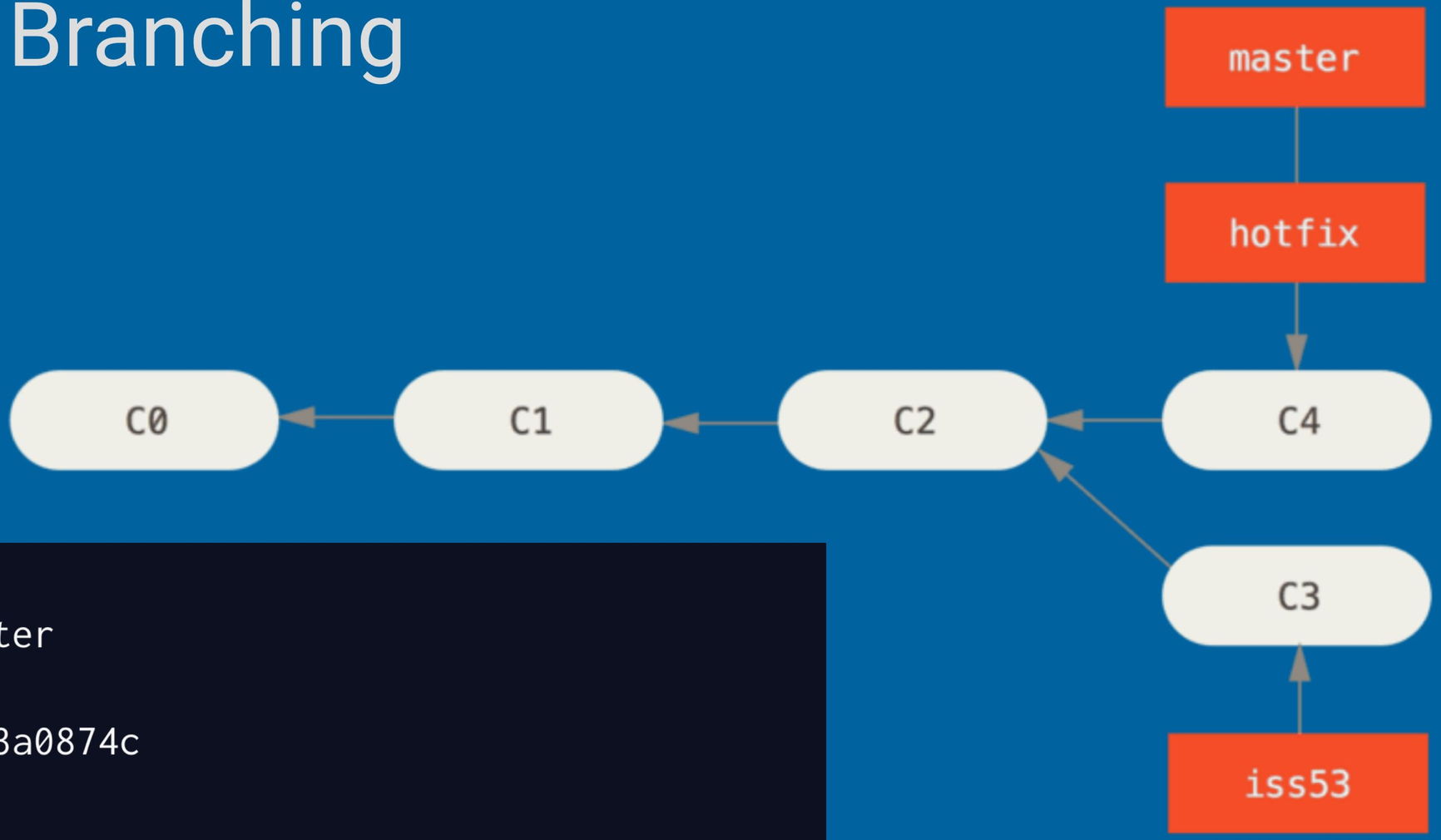
Git Basics - Branching




```
$ git checkout -b hotfix
Switched to a new branch 'hotfix'
$ vim index.html
$ git commit -a -m 'fixed the broken email address'
[hotfix 1fb7853] fixed the broken email address
 1 file changed, 2 insertions(+)
```

Source: https://git-scm.com/book/en/v2/Git-Branching-Basic-Branching-and-Merging

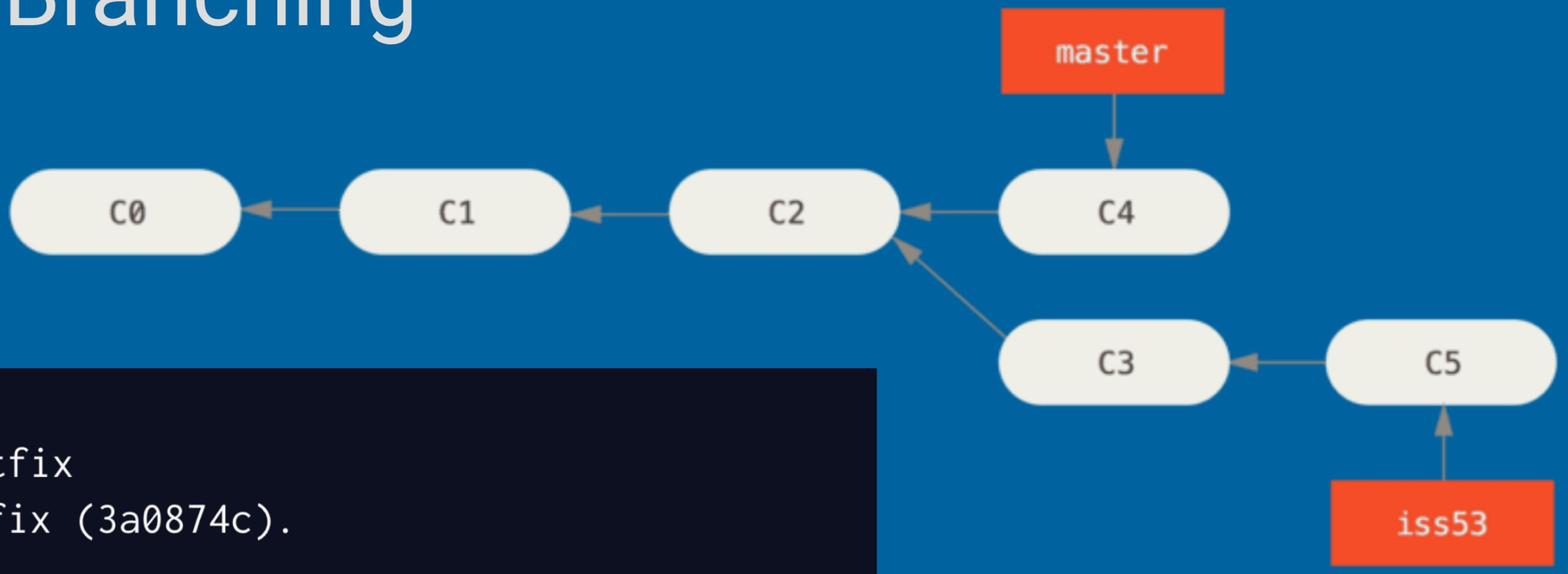
Git Basics - Branching




```
$ git checkout master
$ git merge hotfix
Updating f42c576..3a0874c
Fast-forward
 index.html | 2 ++
 1 file changed, 2 insertions(+)
```

Source: https://git-scm.com/book/en/v2/Git-Branching-Basic-Branching-and-Merging

Git Basics - Branching

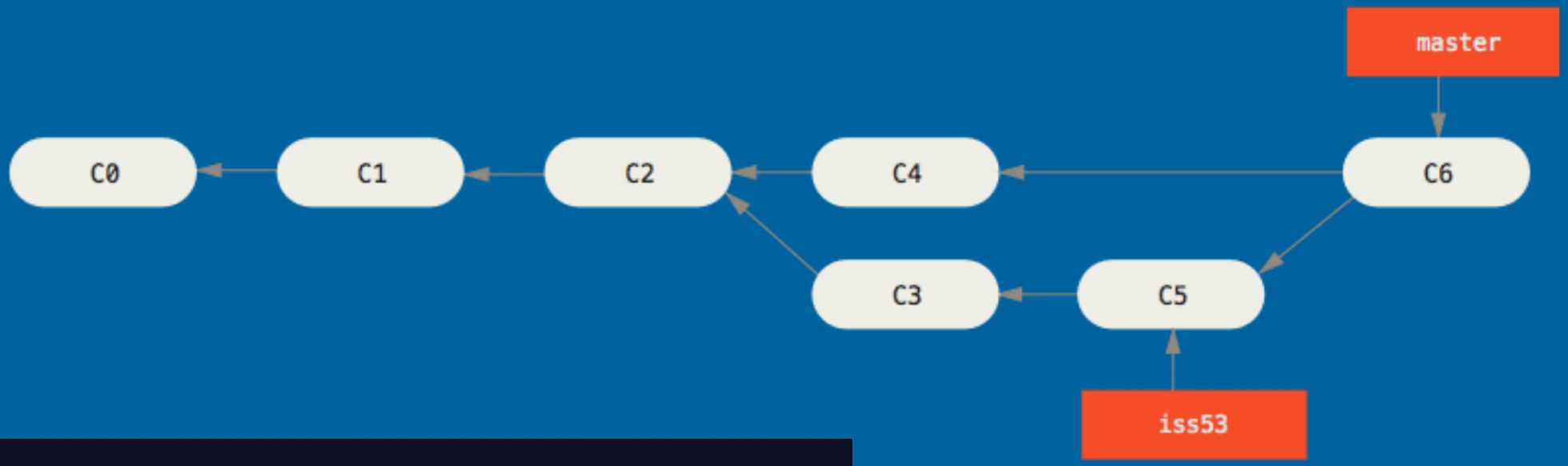



```
$ git branch -d hotfix
Deleted branch hotfix (3a0874c).

$ git checkout iss53
Switched to branch "iss53"
$ vim index.html
$ git commit -a -m 'finished the new footer [issue 53]'
[iss53 ad82d7a] finished the new footer [issue 53]
1 file changed, 1 insertion(+)
```

Source: https://git-scm.com/book/en/v2/Git-Branching-Basic-Branching-and-Merging

Git Basics - Branching




```
$ git checkout master
Switched to branch 'master'
$ git merge iss53
Merge made by the 'recursive' strategy.
index.html |    1 +
1 file changed, 1 insertion(+)
```

Source: https://git-scm.com/book/en/v2/Git-Branching-Basic-Branching-and-Merging

Git Basics - Summary

* Empfehlung: Lest (und versteht!) die ersten 3 Kapitel von https://git-scm.com/book/en/v2
* Git eignet sich nicht nur für Quellcode, sondern für Dateien jeglicher Art.
* Eine super Idee, um die eigenen Änderungen an Projekt- und Bachelorarbeiten nachzuvollziehen (und ggf. ein Backup zu haben, wenn Word mal wieder eine Datei zerschießt!)
* Kostenlose private Repos gibts auf https://bitbucket.com

Coding Time!

Git - Codingaufgabe

30m

* Installiert git
* Legt ein Repository an und experimentiert mit Commits, Branches, Merges und Remotes
* Pusht und pullt Änderungen (legt euch einen GitHub-Account an!)

2. Testing / TDD

3. Frameworks / Dependency Management

4. Richtig Dokumentieren