Berner Fachhochschule - Technik und Informatik

Object-Oriented Programming 2

Topic 1: Generic Programming

Prof. Rolf Haenni & Prof. Annett Laube

FS 2016

Berner Fachhochschule
Haute école spécialisée bernoise
Bern University of Applied Sciences

Outline

Outline

Motivation

- The goal of generic programming is to help implementing . . .
  - → algorithms (e.g. `sort`)
  - → data types (e.g. `ArrayList`)
  - → libraries (e.g. `Collection`)

  independently of concrete data types

- Advantages of generic programming:
  - → Increased code reusability, reduced code redundancy
  - → Stronger type checks at compile time
  - → Less runtime errors
  - → Elimination of casts
  - → Improved code readability

Non-Generic Array Lists

- Java support generic types since Java 5 (2004)
- Before Java 5, using an array list looked as follows:

```
ArrayList list = new ArrayList();
list.add("HellowWorld");

String s = (String) list.get(0);

Integer i = (Integer) list.get(0); // Runtime error
```

- The main problem was a lack of type safety: a non-generic array list stores elements of type `Object`

Generic Array Lists

- Since Java 5, using an array list looks as follows:

```
ArrayList<String> list = new ArrayList<String>();
list.add("HellowWorld");

String s = list.get(0);

Integer i = list.get(0); // Compilation error
```

- Java 7 introduced the diamond operator to simplify object creation

```
ArrayList<String> list = new ArrayList<>();
```

Type Parameters

- Generic classes and generic interfaces have one or multiple type parameters

```
public class ArrayList<E> implements List<E> {
  public ArrayList() { ... }
  public E get(int index) { ... }
  public void add(E element) { ... }
    :
}

public interface List<E> extends Collection<E> {
  public E get(int index);
  public void add(E element);
    :
}
```

Naming Conventions

- The convention for type parameters is to use single capital letters
  - → `T` general generic type
  - → `N` for numbers
  - → `E` for elements in a collection
  - → `K` for key in a map
  - → `V` for values in a map
  - → `S`, `U`, `W` for additional generic types

Outline

Non-Generic Example 1: StringBox I

```
public class StringBox {

  private String content;

  public StringBox(String content) {
    this.content = content;
  }

  public void setContent(String content) {
    this.content = content;
  }

  public String getContent() {
    return this.content;
  }
```

Non-Generic Example 1: StringBox II

```
  public static void main(String[] args) {

    StringBox stringBox = new StringBox("HelloWorld");
    String str = stringBox.getContent();

  }
}
```

Example 1: StringBox.java

Non-Generic Example: IntegerBox I

```
public class IntegerBox {

  private Integer content;

  public IntegerBox(Integer content) {
    this.content = content;
  }

  public void setContent(Integer content) {
    this.content = content;
  }

  public Integer getContent() {
    return this.content;
  }
```

Non-Generic Example: IntegerBox II

```
  public static void main(String[] args) {

    IntegerBox integerBox = new IntegerBox(100);
    int i = integerBox.getContent();

  }
}
```

Example 2: IntegerBox.java

Example: One Generic Type Parameter I

```
public class Box<T> {

  private T content;

  public Box(T content) {
    this.content = content;
  }

  public void setContent(T content) {
    this.content = content;
  }

  public T getContent() {
    return this.content;
  }
```

Example: One Generic Type Parameter II

```
  public static void main(String[] args) {

    Box<String> stringBox = new Box<>("HelloWorld");
    String s = stringBox.getContent();

    Box<Integer> integerBox = new Box<>(100);
    int i = integerBox.getContent();

    Box<Box<Integer>> doubleIntegerBox = new Box<>(
        integerBox);
    int j = doubleIntegerBox.getContent().getContent();
  }
}
```

Example 3: Box.java

Example: Two Generic Type Parameters I

```
public class Pair<S, T> {

  private S first;
  private T second;

  public Pair(S first, T second) {
    this.first = first;
    this.second = second;
  }

  public S getFirst() {
    return this.first;
  }

  public T getSecond() {
```

Example: Two Generic Type Parameters II

```
    return this.second;
  }

  public void setFirst(S first) {
    this.first = first;
  }

  public void setSecond(T second) {
    this.second = second;
  }

  public static void main(String[] args) {

    Pair<String, Integer> pair1 = new Pair<>("HelloWorld
        ", 100);
    String s1 = pair1.getFirst();
    int i1 = pair1.getSecond();
```

Example: Two Generic Type Parameters III

```
    Pair<Integer, String> pair2 = new Pair<>(100, "
        HelloWorld");
    int i2 = pair2.getFirst();
    String s2 = pair2.getSecond();
  }
}
```

Example 4: Pair.java

Example: Inheritance of Type Parameters I

```
public class Couple<T> extends Pair<T, T> {

  public Couple(T first, T second) {
    super(first, second);
  }

  public void swap() {
    T tmp = this.getFirst();
    this.setFirst(this.getSecond());
    this.setSecond(tmp);
  }

  public List<T> toList() {
    List<T> list = new ArrayList<>();
    list.add(this.getFirst());
```

Example: Inheritance of Type Parameters II

```
    list.add(this.getSecond());
    return list;
  }

  public static void main(String[] args) {

    Couple<Integer> couple = new Couple<>(100, 200);
    int i1 = couple.getFirst();
    int i2 = couple.getSecond();
    List<Integer> list = couple.toList();
  }
}
```

Example 5: Couple.java

Example: Nested Type Parameters I

```
public class PairOfBoxes<S, T> extends Pair<Box<S>, Box
    <T>> {

  public PairOfBoxes(Box<S> first, Box<T> second) {
    super(first, second);
  }

  public PairOfBoxes(S firstContent, T secondContent) {
    super(new Box<S>(firstContent), new Box<T>(
        secondContent));
  }

  public S getFirstContent() {
    return this.getFirst().getContent();
  }
```

Example: Nested Type Parameters II

```
public T getSecondContent() {
  return this.getSecond().getContent();
}

public void setFirstContent(S content) {
  this.getFirst().setContent(content);
}

public void setSecondContent(T content) {
  this.getSecond().setContent(content);
}

public static void main(String[] args) {

  PairOfBoxes<String, Integer> pairOfBoxes = new
      PairOfBoxes<>("HelloWorld", 200);
```

Example: Nested Type Parameters III

```
    Box<String> box1 = pairOfBoxes.getFirst();
    Box<Integer> box2 = pairOfBoxes.getSecond();

    String s = pairOfBoxes.getFirstContent();
    Integer i = pairOfBoxes.getSecondContent();
  }
}
```

Example 6: PairOfBoxes.java

Outline

Bounded Type Parameters

- In the examples before, the type parameters were unbounded
  - → No restrictions on the types used in instantiations
  - → No access to specific methods
- Note that the class declaration `class Box<T>` is a shortcut for `class Box<T extends Object>`
- Example of bounded type parameters:

```
public interface Country {
  public String getLanguage();
}
public class Germany implements Country { ... }
public class France implements Country { ... }
```

Example: Bounded Type Parameters I

```
public class Passport<T extends Country> {

  private T country;

  public Passport(T country) {
    this.country = country;
  }

  public T getCountry() {
    return this.country;
  }

  public String getLanguage() {
    return this.country.getLanguage();
  }
```

Example: Bounded Type Parameters II

```
  public static void main(String[] args) {
    France france = new France();
    Passport<France> passport = new Passport<>(france);
    France country = passport.getCountry();

    // the following line produces a compilation error
    Passport<String> pp = new Passport<>("France");
  }
}
```

Example 7: Passport.java

Bounded Type Parameters: Remarks

- When applied to type parameters, the keyword `extends` actually means "`extends` or `implements`", i.e., the bounds can be either classes or interfaces
- Occasionally, two or more type bounds are necessary, for example `<T extends Comparable & Cloneable>`
- If multiple type bounds are specified, and one of them is a class, then the class must be specified first

```
public class A { ... }
public interface B { ... }
public interface C { ... }

public class D <T extends A & B & C> { ... } // OK
public class E <T extends B & A & C> { ... } // Not OK
```

Subtype Rules

- When type parameters are instantiated, the following subtype rules apply:
  - → `List<String>` is a subtype of `Object`
  - → `List<String>` is a subtype of `List`
  - → `ArrayList<String>` is a subtype of `List<String>`
  - → `Couple<String>` is a subtype of `Pair<String, String>`
- Counter-examples:
  - → `List` is not a subtype of `List<String>`
  - → `List<Integer>` is not a subtype of `List<String>`
  - → `List<Number>` is not a subtype of `List<Integer>`
  - → `List<Integer>` is not a subtype of `List<Number>`
  - → `Passport<France>` is not a subtype of `Passport<Country>`
- The last two examples are surprising and often lead to misunderstanding

Outline

Raw Types

- A raw type is the name of a generic class or interface with an unspecified type parameter, for example `Box`
- Raw types are supported for compatibility reasons with Java 1.4 or less (using raw types is not recommended)
- Note that `Box` is not exactly the same as `Box<Object>`

```
Box rawBox;
rawBox = new Box("HelloWorld"); // OK
rawBox = new Box<String>("HelloWorld"); // OK
rawBox.setContent(10); // OK

Box<Object> objectBox;
objectBox = new Box<>("HelloWorld"); // OK
objectBox = new Box<String>("HelloWorld"); // Not OK
objectBox.setContent(10); // OK
```

Wildcard Types

- In generic programs, the wildcard ? represents an unknown type, for example Box<?>
- Note that Box<?> (unknown type) and Box (unspecified type) are not the same, because Box<?> guarantees type safety

```
Box rawBox;
rawBox = new Box("HelloWorld"); // OK
rawBox = new Box<String>("HelloWorld"); // OK
rawBox.setContent(10); // OK

Box<?> wildcardBox;
wildcardBox = new Box<>("HelloWorld"); // OK
wildcardBox = new Box<String>("HelloWorld"); // OK
wildcardBox.setContent(10); // Not OK
```

Wildcard Types

- Wildcards can be used in declaration of ...
  - → local variables
  - → instance variables
  - → static variables
  - → method parameters
  - → return values
- Using wildcards in return types is not recommended
- Wildcards are often used to bypass the restrictions of the strict subtype rules (while avoiding raw types)
- Note that wildcards cannot be used in the definition of generic classes or when invoking their constructors

Bounded Wildcard Types

- Note that `Box<?>` is a shortcut for `Box<? extends Object>`
- Wildcards can be bounded in two different ways:
  - → Upper-bounded: `Box<? extends Number>`
    $\Leftrightarrow$ {`Number`, `Byte`, `Double`, `Integer`, `Long`, `BigInteger`, ...}
  - → Lower-bound: `Box<? super Integer>`
    $\Leftrightarrow$ {`Integer`, `Number`, `Object`}
- The following subtype rules apply:
  - → `List<Integer>` is a subtype of `List<?>`
  - → `ArrayList<Integer>` is a subtype of `List<?>`
  - → `List<Integer>` is a subtype of `List<? extends Integer>`
  - → `List<Integer>` is a subtype of `List<? extends Number>`
  - → `List<Number>` is a subtype of `List<? super Integer>`
  - → `List<Number>` is a subtype of `List<? super Number>`

Using Unbounded Wildcards

- Unbounded wildcards are used when some code does not depend on the concrete type

```
public static void printList1(List<Object> list) {
  for (Object e : list)
    System.out.println(e + " ");
}

public static void printList2(List<?> list) {
  for (Object e : list)
    System.out.println(e + " ");
}

List<Integer> list = Arrays.asList(1, 2, 3);
printList1(list); // Not OK
printList2(list); // OK
```

Using Upper-Bounded Wildcards

- Upper-bounded wildcards are used when some code depends on the methods available for a certain type

```
public static double sum(List<? extends Number> list) {
  double sum = 0.0;
  for (Number number : list)
    sum += number.doubleValue();
  return sum;
}

List<Integer> list1 = Arrays.asList(1, 2, 3);
System.out.println("sum = " + sum(list1)); // OK

List<Double> list2 = Arrays.asList(1.2, 2.3, 3.5);
System.out.println("sum = " + sum(list2)); // OK
```

Using Lower-Bounded Wildcards

- Lower-bounded wildcards are occasionally used when a variable holds data for use elsewhere

```
public static void addInts(List<? super Integer> list) {
  for (int i = 1; i <= 10; i++)
    list.add(i);
}

List<Integer> list1 = new ArrayList<>();
addInts(list1); // OK

List<Object> list2 = new ArrayList<>();
addInts(list2); // OK
```

Outline

Generic Methods

- Generic methods are methods that specify their own type parameters (independently of any generic types of the class)
  - → They can be defined in generic or non-generic classes and interfaces
  - → Static and non-static generic methods are allowed, as well as generic constructors
- In a generic method, the type parameter's scope is limited to the method
- The type parameter of a generic method must appears before the method's return type
- The main purpose of generic methods is to ensure type safety between arguments and return values

Example: BoxUtil I

```
public class BoxUtil {

  // non-generic method using wildcards
  public static boolean compare1(Box<?> b1, Box<?> b2) {
    return b1.getContent().equals(b2.getContent());
  }

  // generic method
  public static <T> boolean compare2(Box<T> b1, Box<T> b2) {
    return b1.getContent().equals(b2.getContent());
  }

  // generic method
  public static <T> Couple<T> couple(Box<T> b1, Box<T> b2) {
    return new Couple<T>(b1.getContent(), b2.getContent());
  }
```

Example: BoxUtil II

```
  public static void main(String[] args) {
    Box<String> box1 = new Box<>("Hello");
    Box<String> box2 = new Box<>("World");
    Box<Integer> box3 = new Box<>(10);

    BoxUtil.compare1(box1, box2); // OK
    BoxUtil.compare1(box1, box3); // OK

    BoxUtil.compare2(box1, box2); // OK
    BoxUtil.compare2(box1, box3); // Not OK

    Couple<String> c1 = BoxUtil.couple(box1, box2); // OK
    Couple<String> c2 = BoxUtil.couple(box1, box3); // Not OK
  }
}
```

Example 8: BoxUtil.java

Outline

Type Erasure

- The Java compiler fully erases generic type parameters:
  - → Generic types are replaced with their type bounds
  - → Unbounded types are replace by `Object`
  - → Type casts are inserted if necessary
  - → Bridge methods are generated to preserve polymorphism (they may appear in a stack trace)
- The produced bytecode, therefore, contains only ordinary classes, interfaces, and methods
- As a consequence, generics does not create new classes and implies no computational overhead at runtime
- Knowing about type erasure types helps understanding the limitations of Java generics

Creating Instances of Type Parameters

- It is impossible to create instances of type parameters
- The problem in the following example is that `new T()` is replaced by `new Object()`

```
public static <T> void fill(T[] array) {
  for (int i = 0; i < array.length; i++)
    array[i] = new T(); // Not OK
}

public static <T> void fillWithDefaults(T[] array, T
    defaultValue) {
  for (int i = 0; i < array.length; i++)
    array[i] = defaultValue; // OK
}
```

Creating Arrays of Type Parameters

- It is impossible to create arrays of type parameters
- The problem is that, for compatibility reasons, creating arrays in Java requires knowledge about its type
- Read detailed explanation here

```
public class ArrayList<E> implements List<E> {

  private E[] elements; // OK

  public ArrayList() {
    this.elements = new E[10]; // Not OK
    this.elements = (E[]) new Object[10]; // OK
  }
}
```

Declaring Static Variables of a Generic Type

- Static variables of type parameters are not allowed, because they are shared by all instances of that class
- What would be the type of the variable `defaultContent` in the following example?

```
public class Box<T> {

  private static T defaultContent; // Not OK
  private T content;

  public Box() {
    this.content = defaultContent;
  }
}
```

Method Overloading

- A class cannot have two overloaded methods that will have the same signature after type erasure

```
public interface ListPrinter {
  public void print(List<String> strList);
  public void print(List<Integer> intList); // Not OK
  public void print(List<Double> intList); // Not OK
}
```

Outline

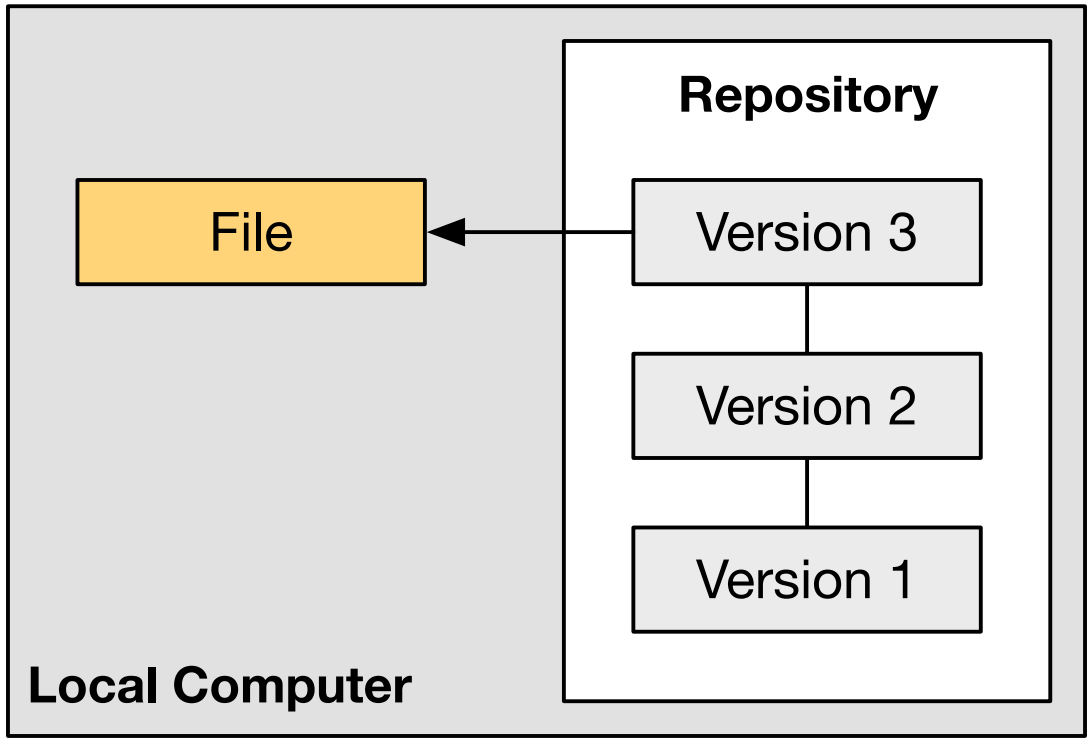
Version Control Systems

- Version control systems (VCS) are widely used in today's software engineering practice
- Benefits of using a VCS in software engineering:
  - → Software development in teams
  - → Remote collaboration
  - → Keep control over changes to source code
  - → Management of versions and updates
  - → Automatic backup
- Most popular VCS
  - → RCS (since 1982 almost not used anymore)
  - → CVS (since 1986, almost not used anymore)
  - → Subversion (since 2000, still in use today)
  - → Git (since 2005, the most widely used VCS today)

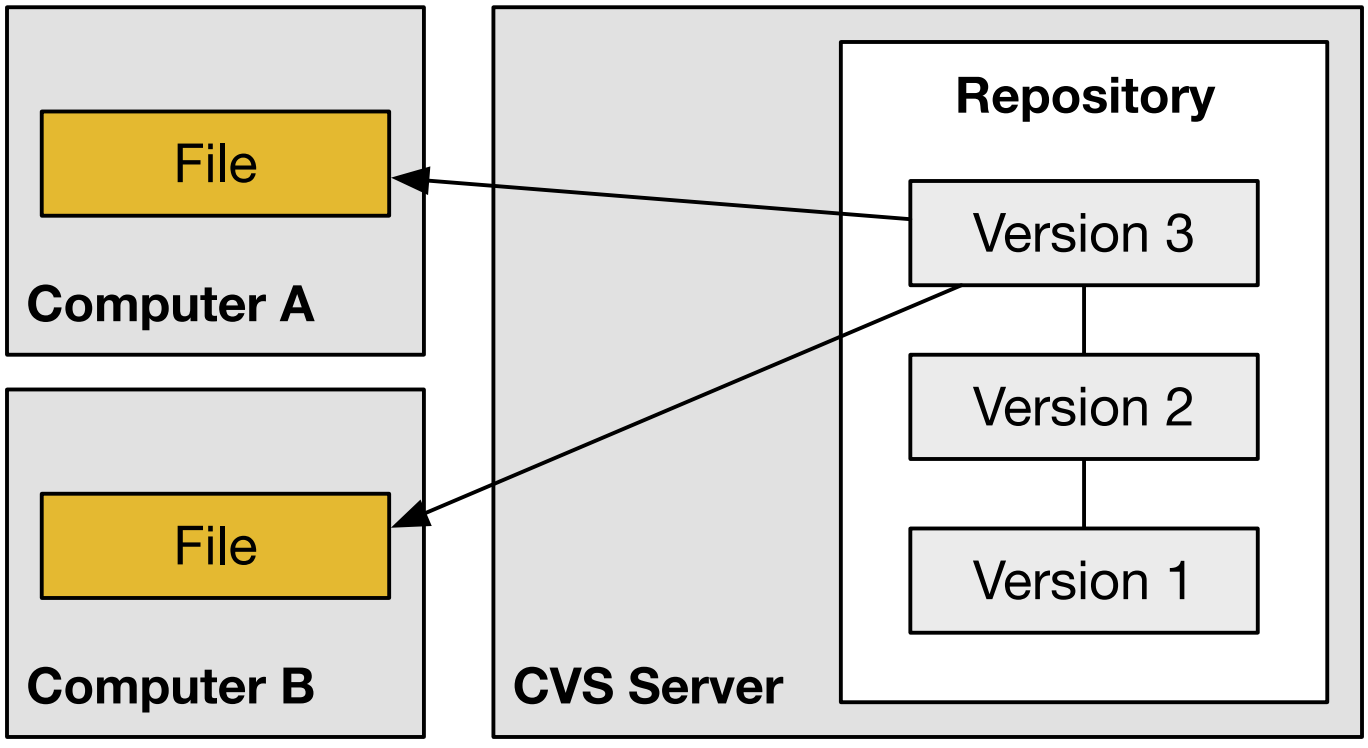
Git and Github

- Git is a free open-source VCS designed by Linus Torvalds
  - → Current version: 2.8.3 (April 2016)
  - → Web page: https://git-scm.com
  - → Tutorial: "Become a Git Guru"
- Git is a distributed VCS, which means that the whole version database (called repository) is stored everywhere
- A distributed VSC combines the advantages of a local (such as RVC) a centralized VCS (such as Subversion)
- Dedicated Git server software helps to add access control, manage multiple repositories, and display them on the web
- Two of the most popular Git servers are GitHub and GitLab

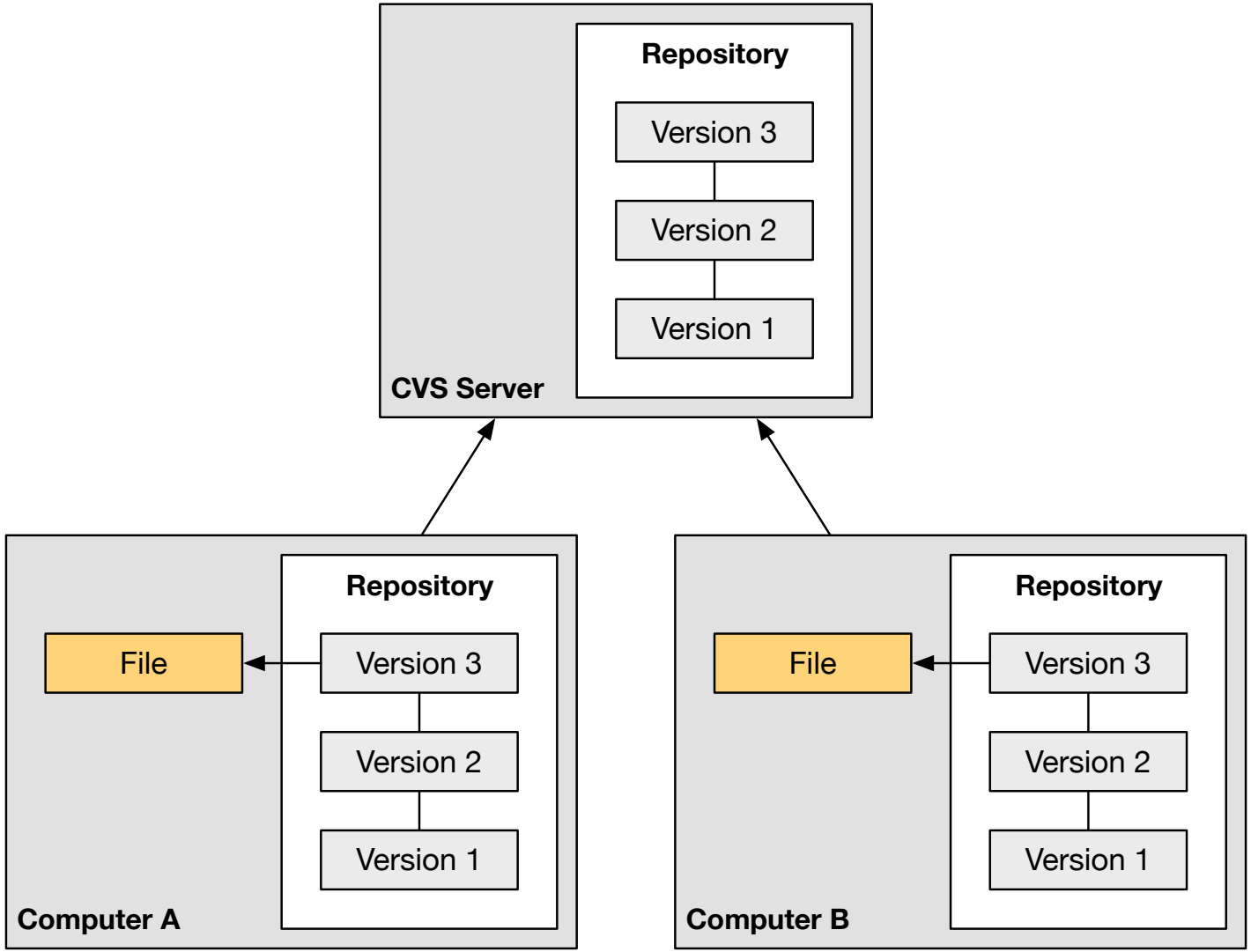
Local VCS


Repository
File
Version 3
Version 2
Version 1
Local Computer

Centralised VCS



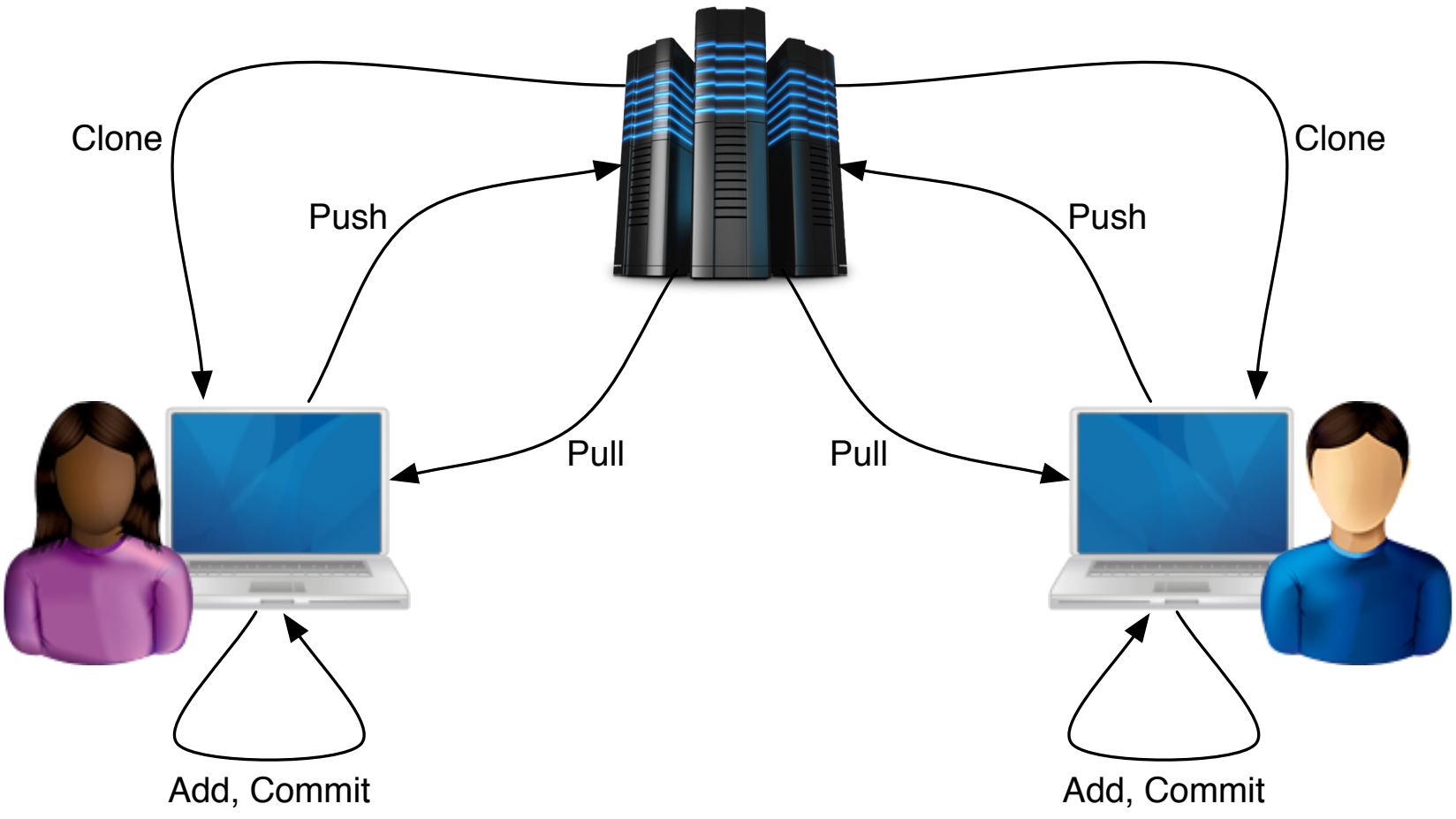
Distributed VCS



Git Commands

- `clone`: Makes a repository copy from a remote source
- `add`: Adds files changed locally to stage (ready to commit)
- `commit`: Includes staged files into local repository (the changes are described in the commit message)
- `push`: Sends the all changed files from local repository to remote repository
- `pull`: Fetches all changed files from remote repository and merges them with the local one (`fetch` followed by `merge`)
- Calling `pull` followed by `push` is called synchronization, but `sync` is not an official Git command

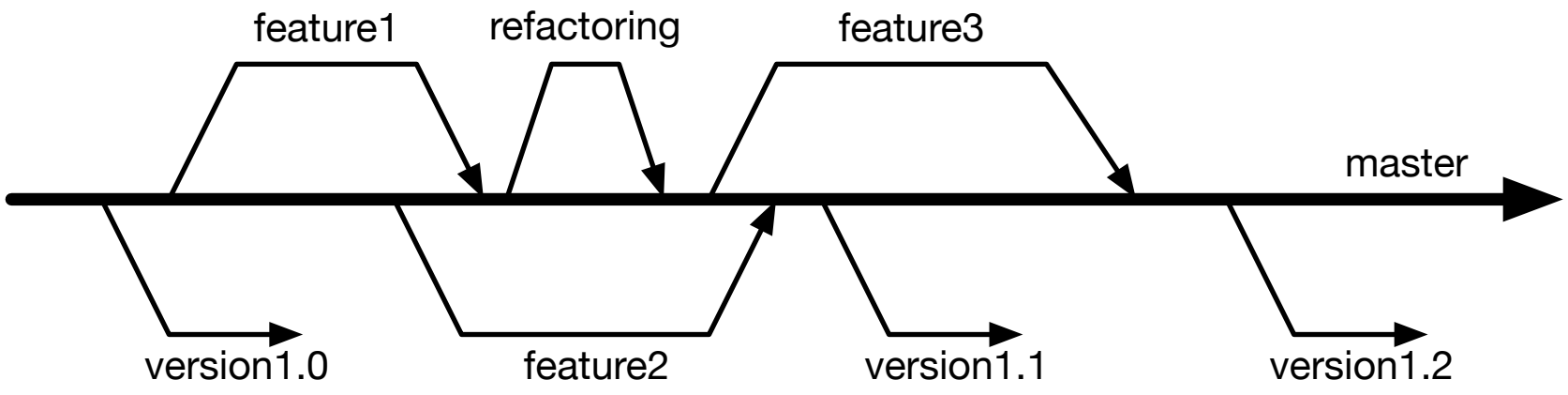
Git Workflow



Branching and Merging

- Git repositories can have multiple branches
- Initially, a Git repository contains a single master branch
- Creating branches keeps multiple streams of work independent from each other (e.g., new features)
- If independent work in a branch has stabilized, it can be merged into the master branch
- The release of a new software version (and its updates) can be defined as a separate branch
- Each team decides about the optimal workflow for its projects

Branching and Merging



Conflict Solving

- Collaboration in teams can lead to code conflicts
- Most of the potential conflicts are resolved automatically by Git during the merge operation
- A conflict appears when two developers edit the same line of code simultaneously
- In such a case, both lines of code appear in the merged document
- After solving the conflict (someone has to decide which code will be kept), `commit` and `push` operations are needed to update the merged files in the repository and the server

Conflict Solving

- Collaboration in teams can lead to code conflicts
- Most of the potential conflicts are resolved automatically by Git during the merge operation
- A conflict appears when two developers edit the same line of code simultaneously
- In such a case, both lines of code appear in the merged document
- After solving the conflict (someone has to decide which code will be kept), `commit` and `push` operations are needed to update the merged files in the repository and the server

Using Git in Practice

- Git is a command line tool, but there are various graphical Git tools to simplify its use
- For repositories hosted on GitHub, the tool GitHub Desktop is recommended
  - → History of commits
  - → Selection of current branch
  - → List of uncommitted changes
  - → Menus and buttons for basic Git commands (clone, commit, pull, push, etc.)
- Git is also directly accessible from IDEs such as Eclipse
  - → Install the EGit plug-in
  - → It is good practice to place any Git repositories outside the Eclipse workspace