Software Engineering

Object-Oriented Programming

Structure of the OOSE Lectures

Revisit and deepen basics of programming.

Revisit and deepen basics of object-oriented programming.

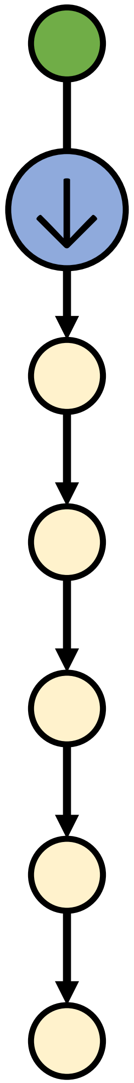
Cover advanced object-oriented principles.

How to model OO systems (UML) and map models to code.

Object-oriented modeling techniques.

Design patterns as means to realize OO concepts (I).

Design patterns as means to realize OO concepts (II).



Last Lecture

- General programming paradigms and languages.
- Imperative programming concepts: Variables, operations, conditions, loops, procedures, modularization.
- Memory management: Memory spaces and program execution.
- Basics of class-based OOP.
- Compiler overview.

Aims of this Lecture

- Understanding object identity.
- Understanding the meaning and the principles of types.
- Understanding interfaces and interface design.
- Understanding inheritance, dynamic binding and polymorphism.

Object Identity

Object Identity (OID)

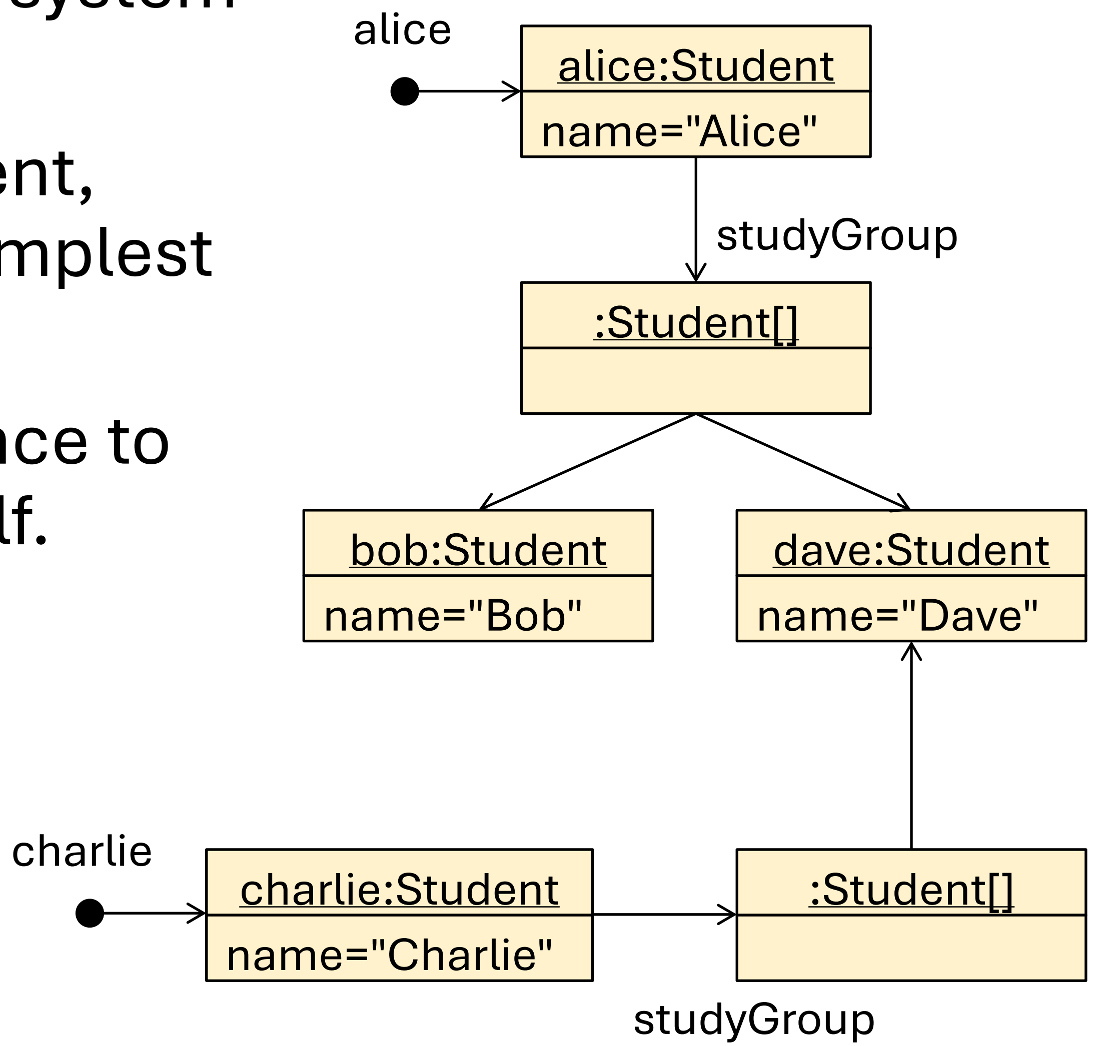
An object's identity is a way for the system to refer to an object.

The exact implementation is different, depending on the context. In the simplest case: a pointer.

Variables store OIDs, e.g., a reference to an object instead of the object itself.

```
public class Student {
   ...
   Student[] studyGroup;
   ...
}
```

```
Student alice = new Student()
```



Sharing / Aliasing

Benefit: Sharing. Two variables *intentionally* point to the same object.

Danger: Aliasing. Risk that two variables *unintentionally* point to the same object.

Changes applied to the object via one variable are also visible "on the other side".

Sharing and Aliasing describe the same phenomena, but with different intentions.

Shallow Clone / Flat Copy

Cloning an object by creating a new object whose attributes have the same values as the original (primitive as well as reference types) is called a shallow copy.

```
public Student clone() {
   Student clone = new Student();
   clone.firstname = this.firstname;
   clone.lastname = this.lastname;
   clone.studyGroup = this.studyGroup;
   ...
   return clone;
}
```

Deep Cloning

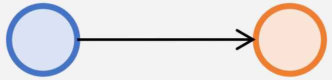
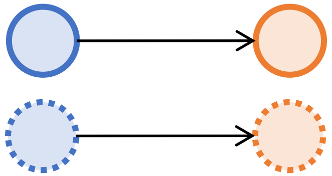
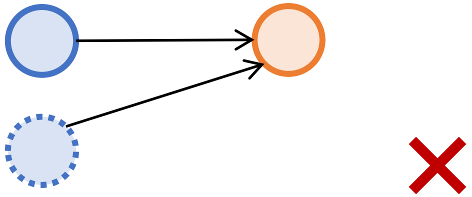
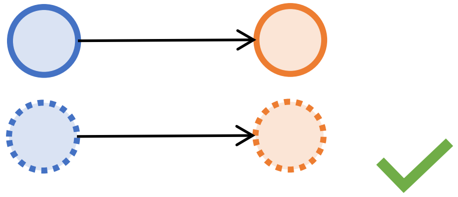
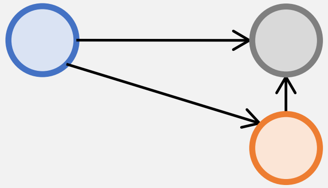
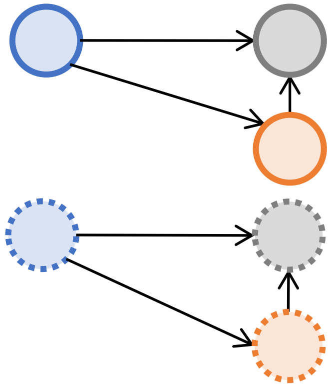
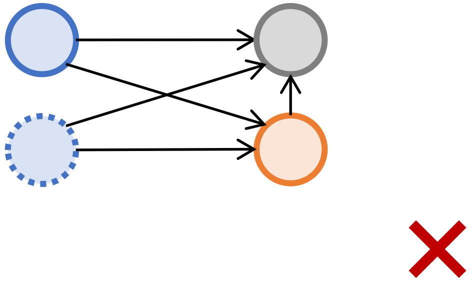
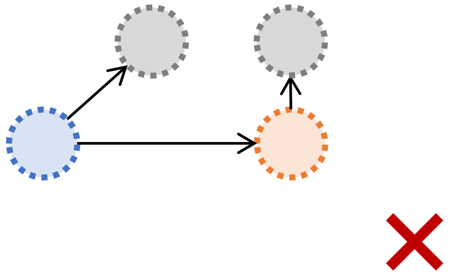
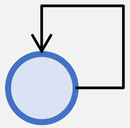
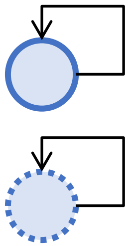
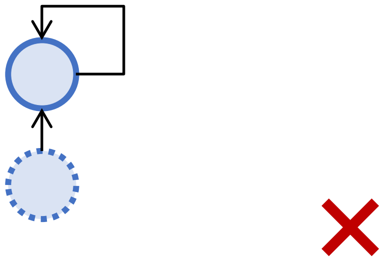
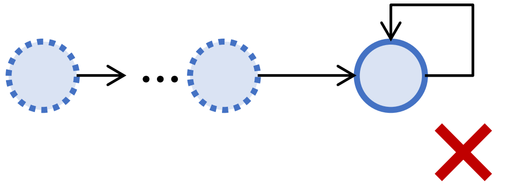
Shallow cloning bears the danger of aliasing. With deep cloning, we also clone the objects referenced by reference-type attributes.

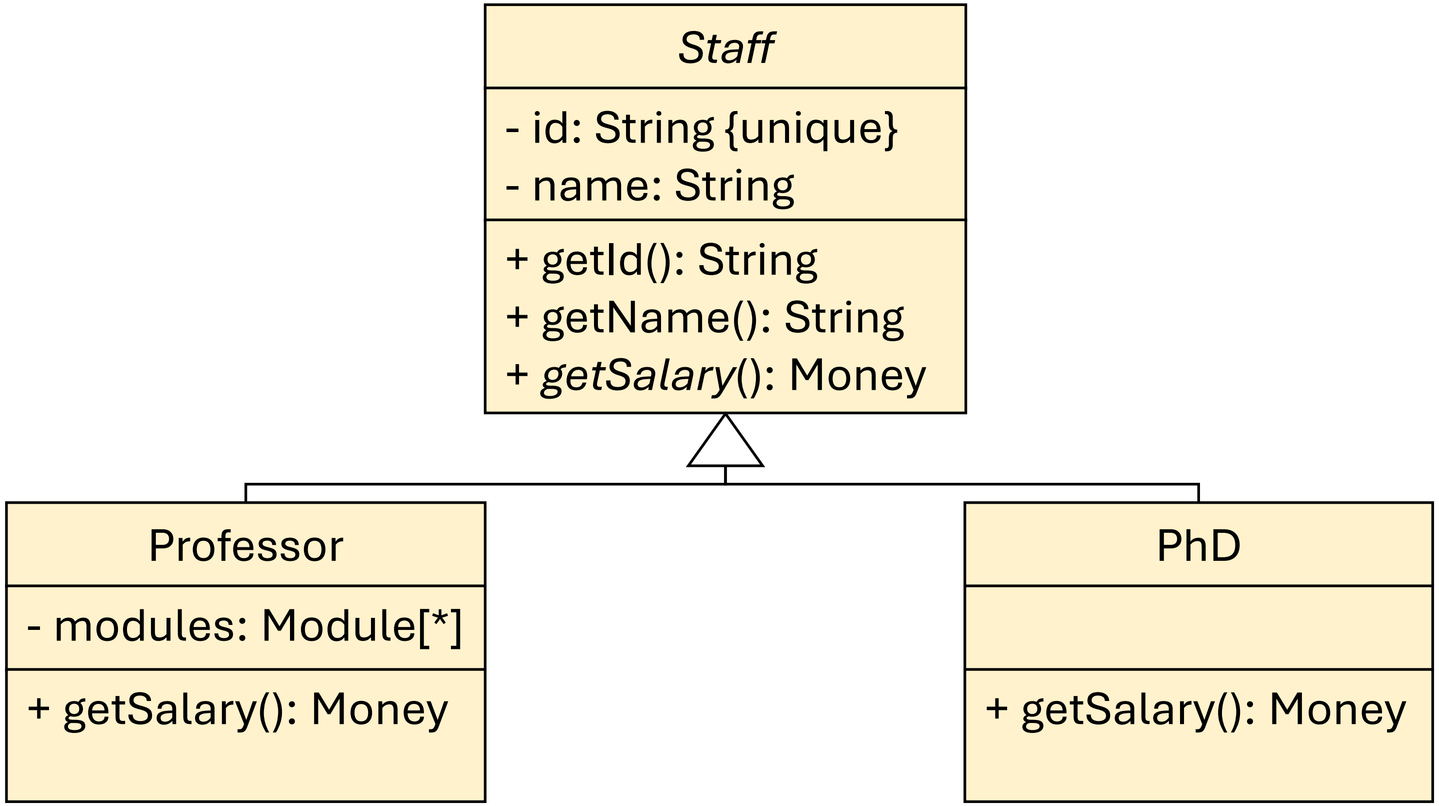
Approach: Recursively call `clone()` on non-primitive, cloneable (implements `Cloneable` interface) attributes.

```
public Student clone() {
   Student clone = new Student();
   clone.firstname = this.firstname;
   clone.lastname = this.lastname;
   clone.studyGroup = this.studyGroup.clone();
   ...
   return clone;
}
```

Cloning: Pitfalls

These cases can be solved with a `Map<Reference of Original, Reference of Clone>` of all copied references: If the original reference is in the map, do not clone again and instead use the associated new reference.

	Expected Result	Shallow Clone	Deep Clone
Simple Case 			
Sharing/Aliasing 			
Cyclic Reference 			

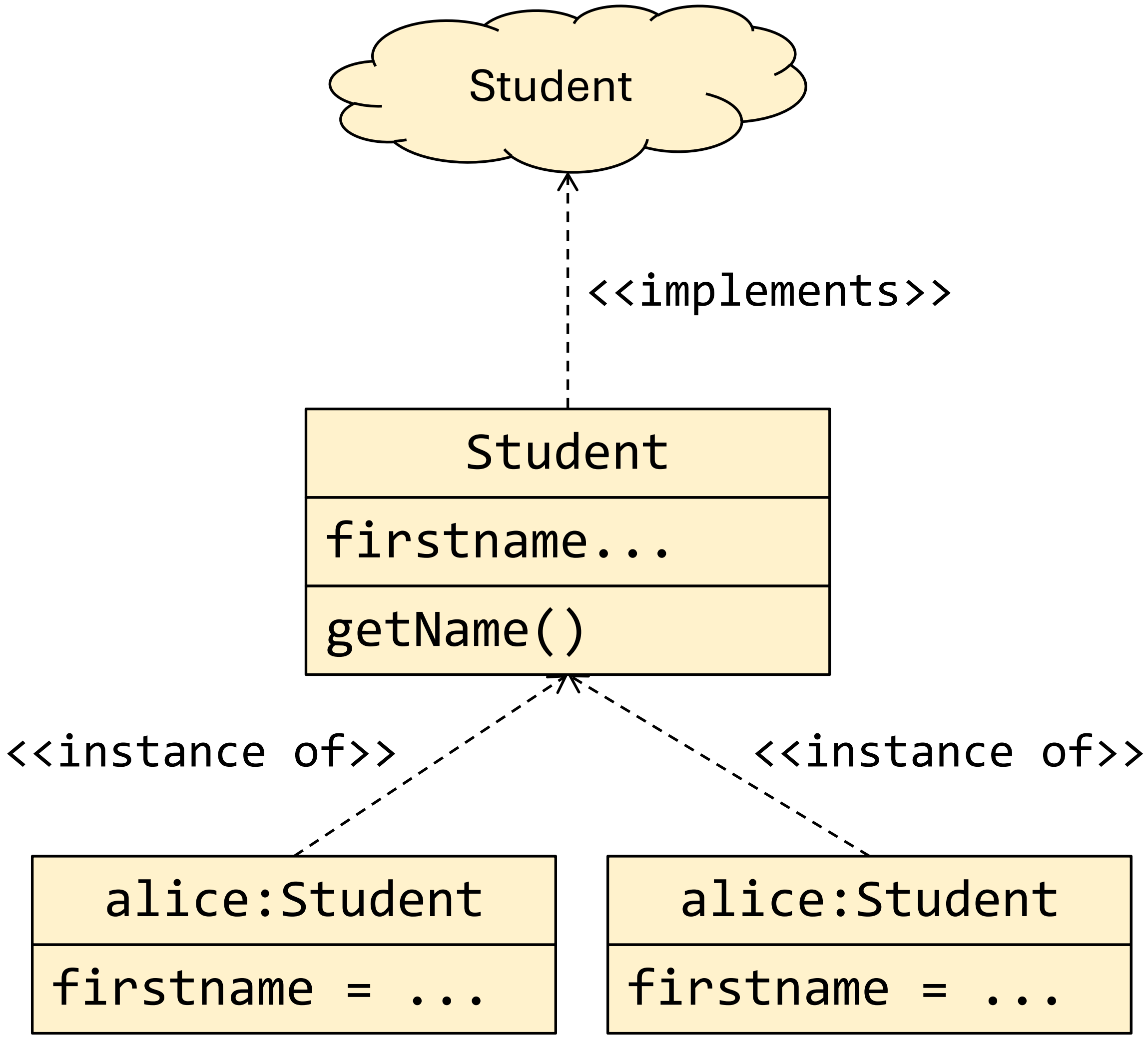


Types and Subtypes

Types, Classes and Instances

In literature, types and classes are not the same thing. A type is a concept that defines an interface and certain characteristics.

A class is a concrete, technical realization of a type.



Static vs. Dynamic Type

Static type: the type used in a declaration. Also called declared type or compile-time type.

Dynamic type: type of the expression at runtime. Also called runtime-time type.

```
Student adrian = new Student("Adrian", "Bajraktari", 26, 2969443);
```

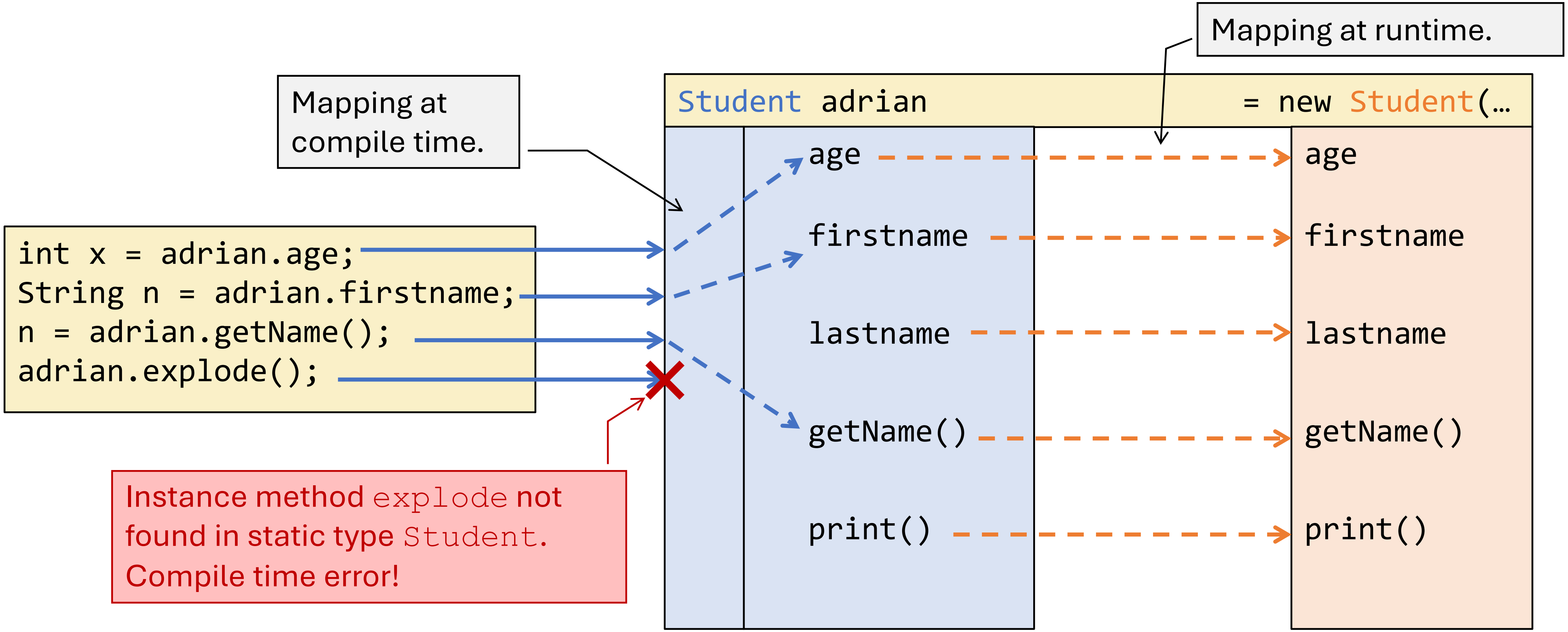
Static type

Dynamic type

- In this example easy: static type == dynamic type.

Metaphor: Static Type as Guard

Static types restrict the set of legal member accesses.



Static vs. Dynamic Type

Static type: the type used in a declaration. Also called declared type or compile-time type.

Dynamic type: type of the expression at runtime. Also called runtime-time type.

```
Person adrian = new Student("Adrian", "Bajraktari", 26, 2969443);
```

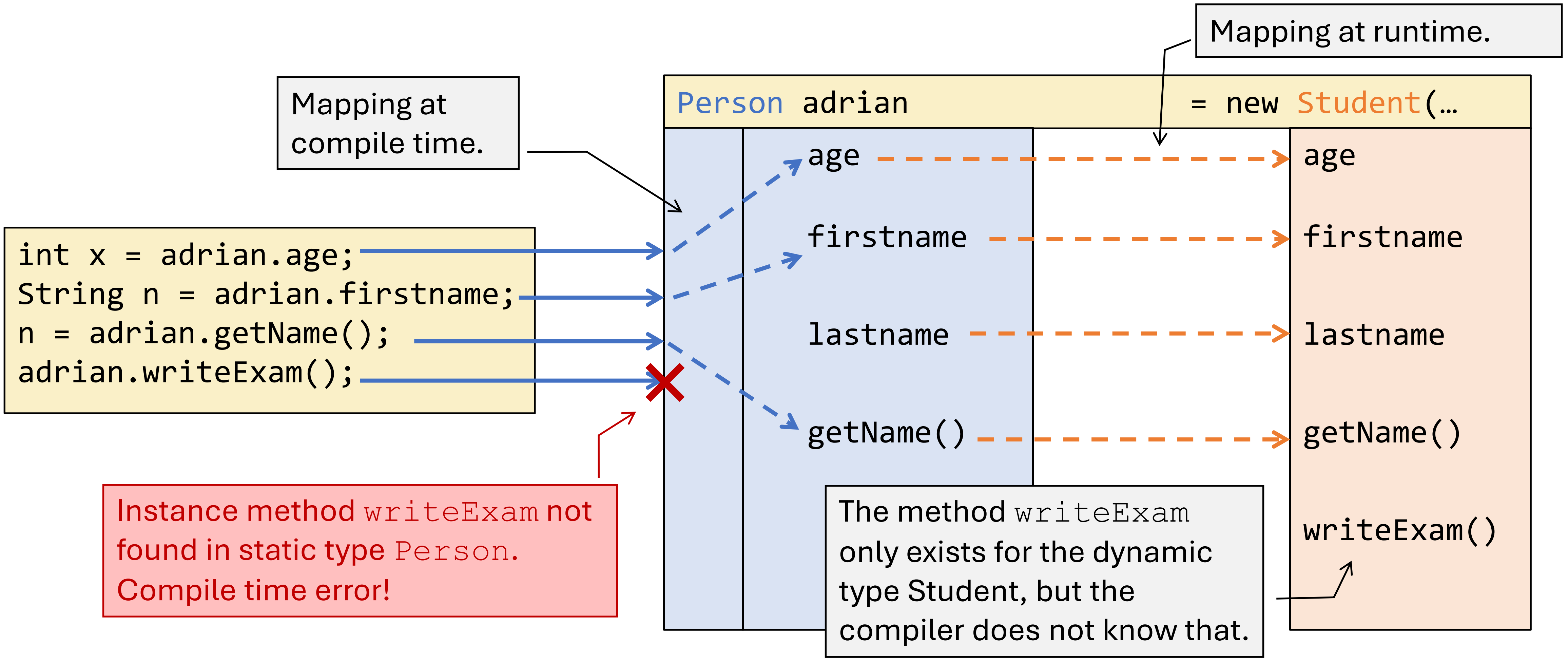
Static type

Dynamic type

- The dynamic type must be compatible to the static type.

Metaphor: Static Type as Guard with different Dynamic Type

Static types restrict the set of legal member accesses.



Subtype Relation

A dynamic type `B` must be compatible to a static type `A`

- They must be related via subtyping.

Structural subtyping: `B` contains at least the same elements as `A`.

- This is used in C and Go.

Nominal subtyping: We explicitly define `B` as subtype of `A`.

- Usually this also implies structural subtyping, since `B` not having all members of `A` would violate the *Liskov substitution principle*.

Static and Dynamic Type

The static type let's the compiler know...

- which members the dynamic type must at least provide.
- to which members we can send a message.

"I expect a value that is of type `Person`"

"I provide a value of type `Student`"

```
Person adrian = new Student(...
```

"I assure that all `Student` values are also `Person` values.

```
class Student extends Person
```

We can assign any value with arbitrary dynamic type to a variable, as long as the dynamic type is a subtype of the static type.

Example: Evolution of the dynamic Type at Runtime

The compiler can guarantee that all these accesses are legal, since...

- it, via the static type, knows that the dynamic type has to allow all these messages.
- ...the dynamic type, at each point in time, must provide the member (since it is a subtype of the static type).

```
Person adrian = new Student(...);
int x = adrian.age;
String n = adrian.firstname;
...
adrian = new PhD(...);
int y = adrian.age;
String s = adrian.firstname;
...
adrian = new Professor(...);
...
```

Class Inheritance

In Java, we can define that a class inherits from another class with the `extends` keyword. The subclasses inherit all instance members from the superclass.

```
public class Person {
   int age;
   String firstname;
   String lastname;
}
```

```
public class Student
extends Person {

   public void writeExam() {
      ...
   }
}
```

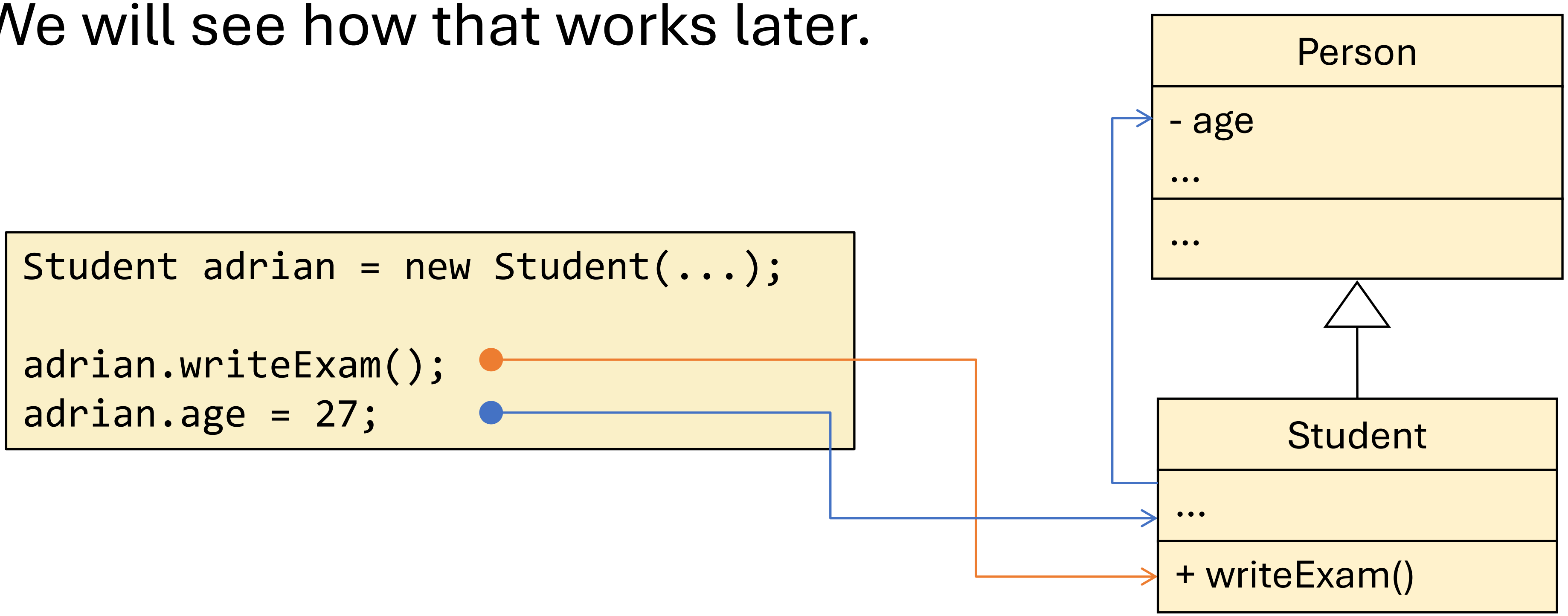
```
public class PhD
extends Person {

   public void writeDiss() {
      ...
   }
}
```

```
public class Professor
extends Person {

   public void
      putApplication() {
      ...
}}
```

Code Reuse via Inheritance

Members that are not implemented in the class itself are looked up in the superclasses.

- Eventually over multiple layers (superclasses of superclasses of...)
- We will see how that works later.



Hints for Modeling

Java's inheritance combines subtyping and code reuse.

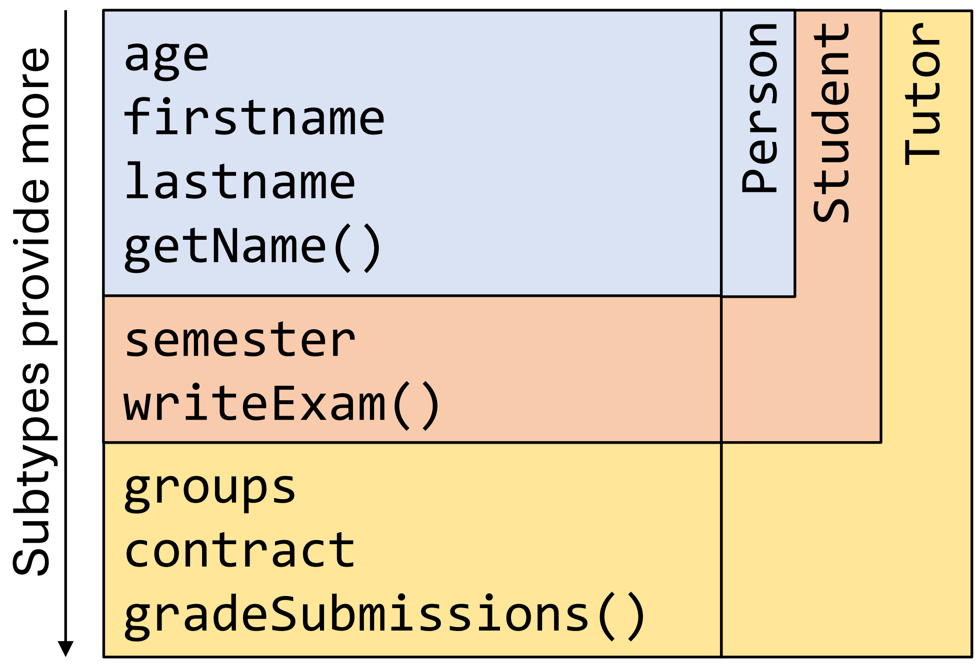
However, you should only use inheritance if a conceptual specialization exists.

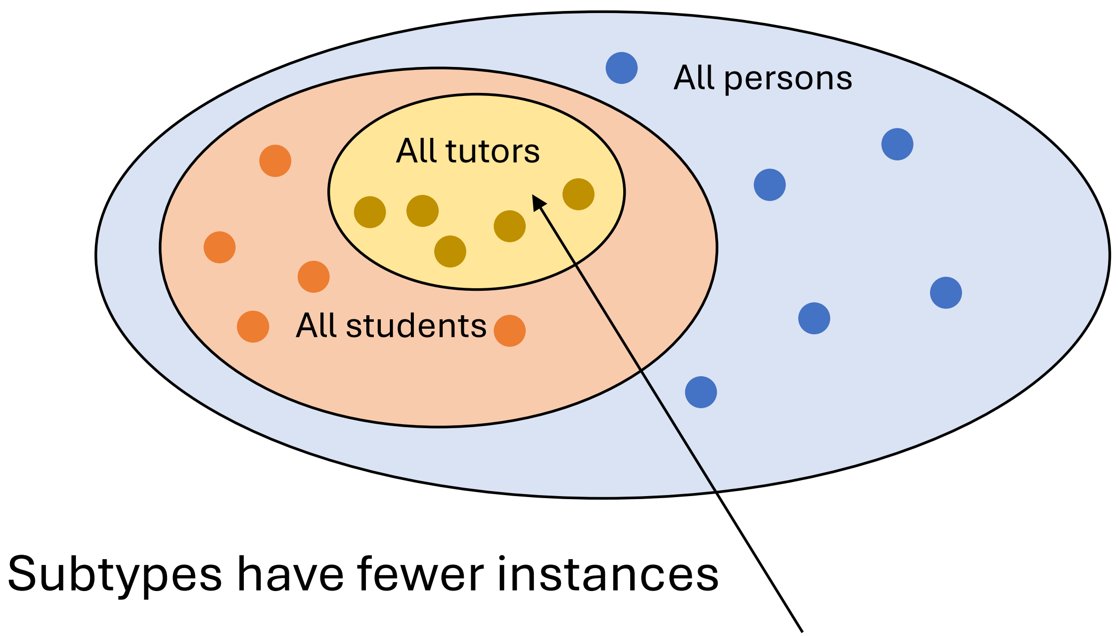
- Only if the question "Is `B` also an `A`?" can be answered with yes.
- **Do not** use inheritance only to reuse functionalities of other classes. Instead: *Forwarding* or *delegation*! (see later)

Subtypes: Subset View (single inheritance!)

Subtypes provide more details (fields, methods).

Because subtypes define more details, there are fewer instances that provide all of them.





Abstract Types and Interfaces

Abstract Classes

Abstract classes are defined using the `abstract` keyword.

These classes can not be instantiated directly*.

They can only be used as static type in declarations, for which a concrete subtype must be substituted.

```
Person adrian = new Person(...);
```

```
Person adrian = new Student(...);
```

```
public abstract class Person {
    ...
}
```

```
public class Student
extends Person {
    ...
}
```

```
public class PhD
extends Person {
    ...
}
```

Abstract Methods

Abstract methods are method headers with no body. They can only be declared in abstract classes. Abstract methods must be overridden in concrete subclasses.

```java
public abstract class Person {
   public abstract String getInfo();
}
```

```java
public class Student
extends Person {
   public String getInfo() {
      return this.firstname + " " +
            this.lastname + ", " +
            this.matriculationNumber;
   }
}
```

```java
public class PhD
extends Person {
   public String getInfo() {
      return this.firstname + " " +
            this.lastname + ", " +
            this.id + ", " + this.chair;
   }
}
```

Even more abstract Classes

Abstract classes:

- Force us to define subtypes.
- Allow us to define abstract methods that enforce overriding.
- Allow us to define a common data structure[1].

Problem: abstract classes still allow to define data structures.

- This takes the freedom to apply the same functionalities to different data structures.
- Idea: abstract from data!

[1] *Data structure is not meant as "trees", "stacks", etc., but as the set of all instance variable values.*

Interfaces

Interfaces are maximally abstract types.

- They do not contain any instance variable declarations.
- Class variables may only be defined as constants.
- All operations are either abstract, class methods, `default`-methods or `private` instance methods.
- In interfaces,...
  - all variables are implicitly `public`, `static` and `final` (→ public constants).
  - all non-class methods, non-`default`-methods and non-`private` methods are `public` and `abstract`.

Interfaces

Classes implement interfaces. This means that the class implements all abstract methods of the interface.

Essential differences between classes and interfaces:

- A class may implement one or more interfaces (multiple classification).
- An interface can be seen as the definition of a part of an objects interface, rather than a template.
- Syntactic: `extends` for classes, `implements` for interfaces.

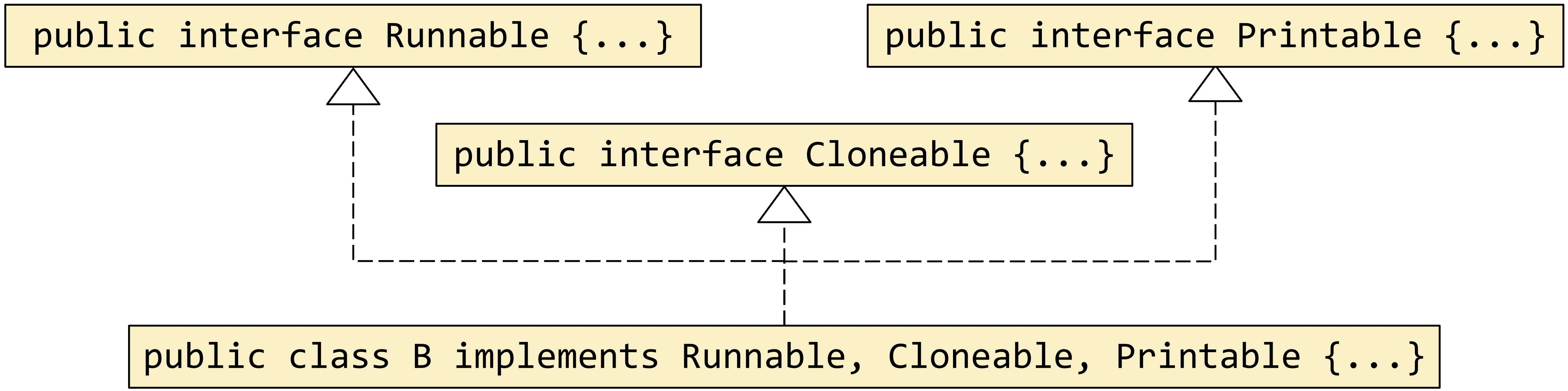
```
public interface Entity {
   String getInfo();
}
```

```
public abstract class
Person implements Entity {
   ...
   public String getInfo() {
      ...
   }
}
```

```
public class Student
extends Person {
   ...
}
```

Interface

Example for a class implementing multiple interfaces.



Static and Dynamic Type

Class types as static type:

"I expect a value that is of type `Person`"

"I provide a value of type `Student`"

"I assure that all `Student` values are also `Person` values.

```
Person adrian = new Student(...
```

```
class Student extends Person
```

Interface types as static type:

"I expect a value that provides the `Person` interface"

"I provide a value of type `Student`"

"I assure that all `Student` values provide the `Person` interface.

```
Person adrian = new Student(...
```

```
class Student implements Person
```

Interface Design

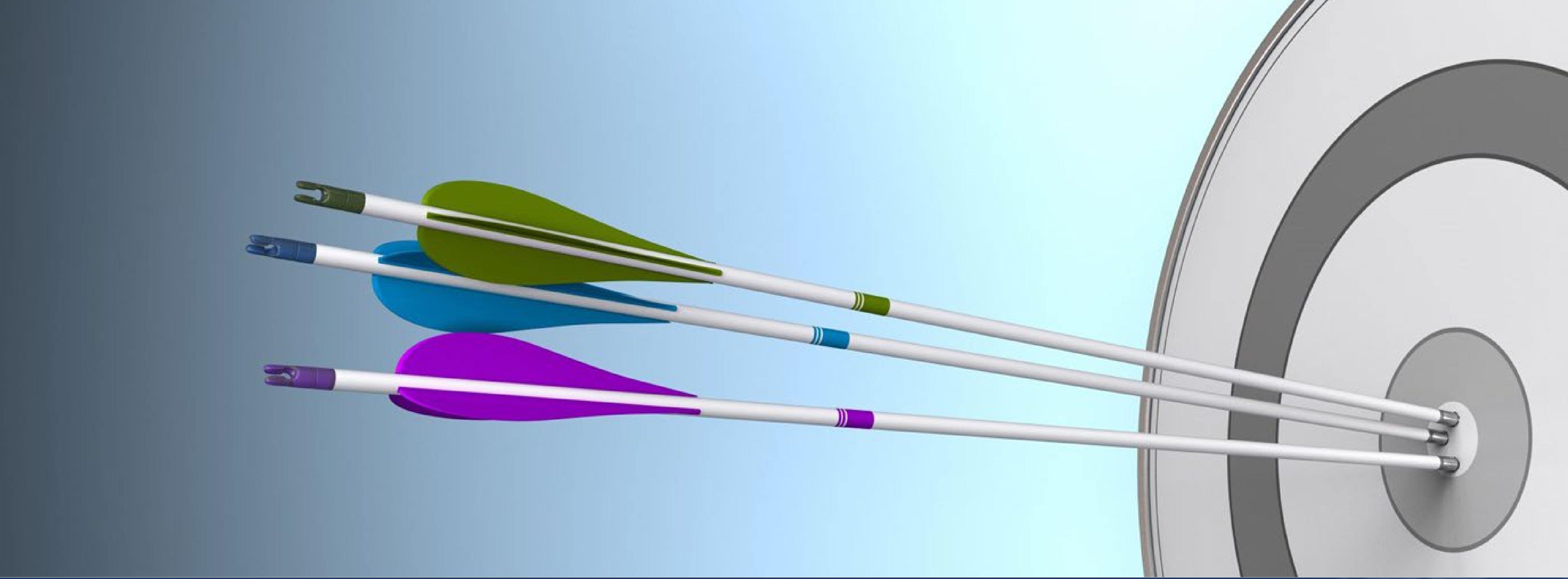
For variables with static type of an interface, all implementing classes can be used as dynamic type.

This leads to maximal reusable code.

This idea manifests in the Dependency Inversion Principle:

- Use interfaces as static type as often as possible.
- Use concrete classes only for object creation.

We will discuss this principle later.



Overriding

Hiding

In Java, we can hide class variables, class methods and instance variables of the super class using the same name in the subclass.

Here, the class `Student` now has its own instance variable `firstname`, additionally to the one from `Person`.

- We can access `Person::firstname` from `Student` via `super`.
- `firstname` or `this.firstname` accesses `Student::firstname`.

```
public abstract class Person {
   String firstname;
      ...
}
```

```
public class Student
extends Person {
   String firstname;
   ...
}
```

Hiding is really bad coding practice! Do not use it without very good reasons.

Dynamic Binding

```
Person e;
Student adrian;
...
e = adrian;        //Allowed, Student :> Entity
...
e.getInfo();       //Allowed, getInfo is in e
```

What shall happen here?

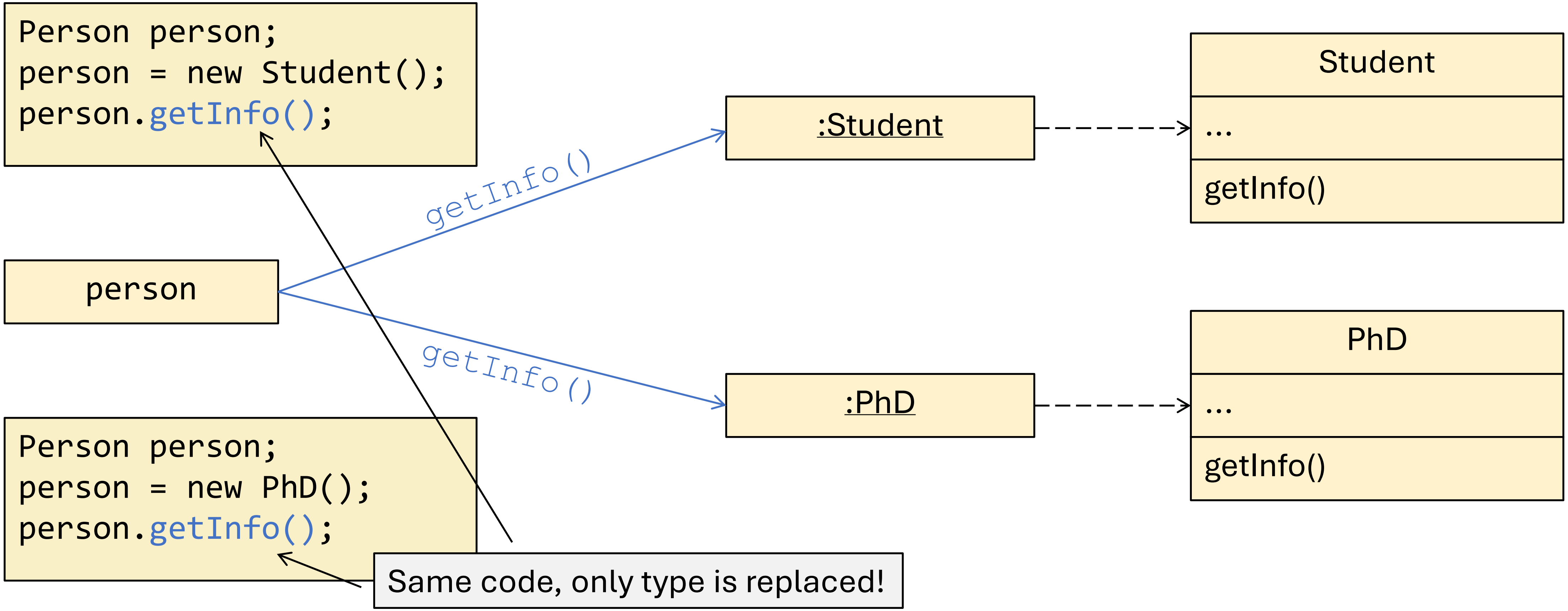
The method of the currently referenced object's class, `Student`, is executed!

```
public interface Person {
   String getInfo();
}
```

```
public class Student
implements Person {
   ...

   public String getInfo() {...}
}
```

Dynamic binding allows to determine the receiver of a message and thus the implementation to be executed at runtime.

Example of Dynamic Binding



Polymorphism

Polymorphic expressions are expressions that can take on values of different types.

Subtype polymorphism: Expressions can take on values of the static type or of subtypes.

With polymorphism and dynamic binding, the same code can apply to arbitrary many different receiver objects.

Class-based OOP Implementation

"Imperative Style"

In imperative, non-OO languages, all memory addresses are statically known. The same is true for class attributes and class methods.

```
public class Student {
   ...
   static int getNumberOfStudents() {
      ...
   }
   ...
}
```

Compiles to

```
0x4F: ...//code of
      //getNumberOfStudents
```

```
Student.getNumberOfStudents();
```

Compiles to

```
...//code for passing
   //arguments
call 0x4F;
```

"Object-Oriented Style"

Instance attributes are assigned indices in the object's memory layout.

- The indices are the same across all objects of a type.
- Inherited attributes have the same indices as in the superclass.
- Then follow the own instance attributes.

Before the attributes is an object header with metadata. The only relevant metadata for us is the class reference.

"Object-Oriented Style"

Instance methods are not stored per object, as they are the same for all instances of a type. Instead, they are stored in the instances' classes' vTable.

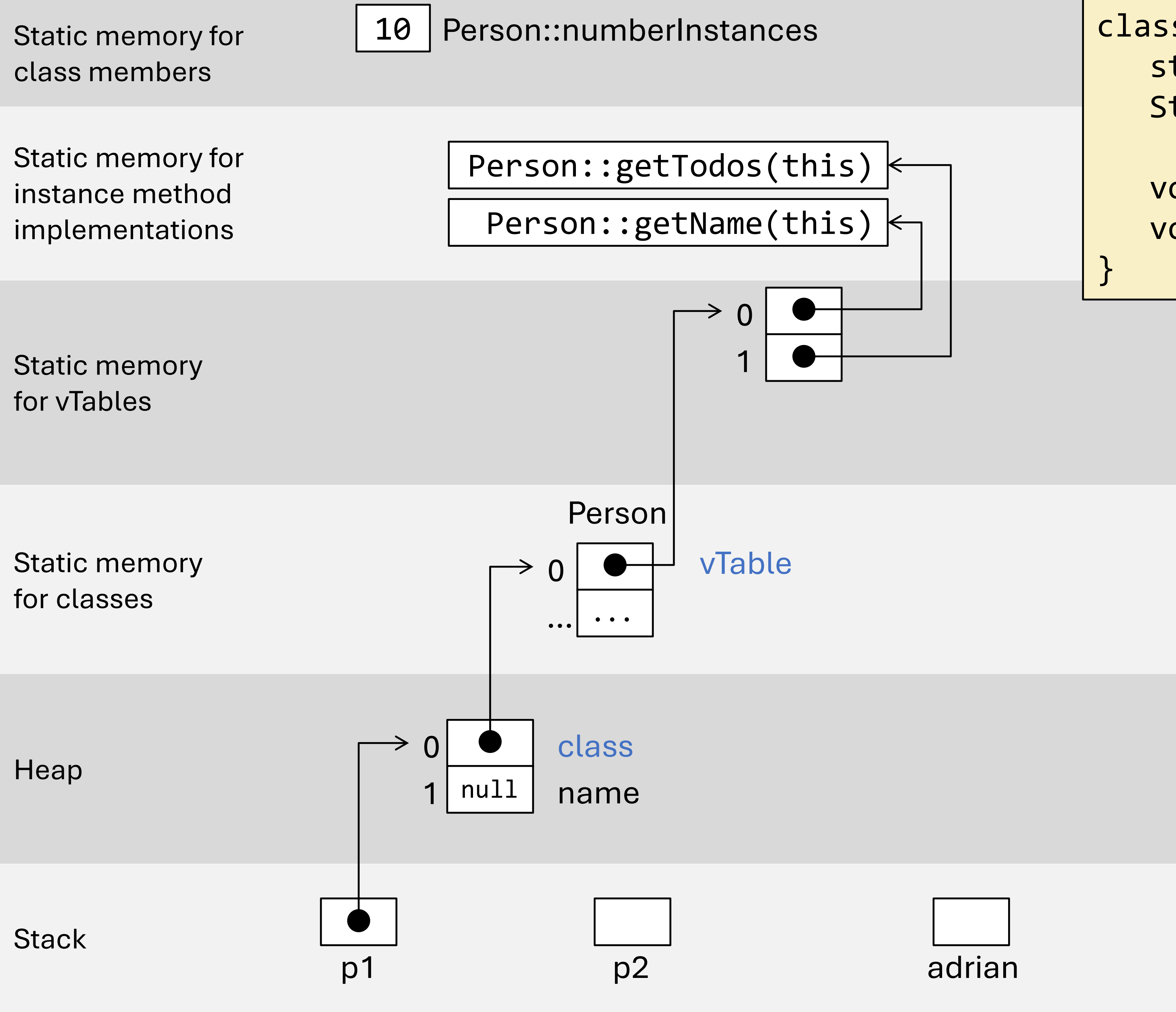
- Each instance method is assigned an index in the vTable.
- Inherited and overridden instance methods have the same index for the current type as for the super types.
- Then follow the own methods.

vTable = virtual function table (term coined by Bjarne Stroustrup for C++ that has been adapted)

```
Person p1, p2;
Student adrian;
p1 = new Person();                  -
adrian = new Student("Adrian",      -
13);
p2 = adrian;
p1.name;                            1
p1.getName();                       0
p1.getTodos();                      1
p1.getTodos(4);//not in Person      X

p2.getName();                       0
p2.getTodos();                      1
p2.getTodos(4);                     2

adrian.getName();                   0
adrian.getTodos();                  1
adrian.getTodos(4);                 2
adrian.name;                        3
```

```
class Person {
   static int numberInstances = 10;                           -
   String name;                                               1

   void getName() {...}                                       0
   void getTodos() {...}                                      1
}
```

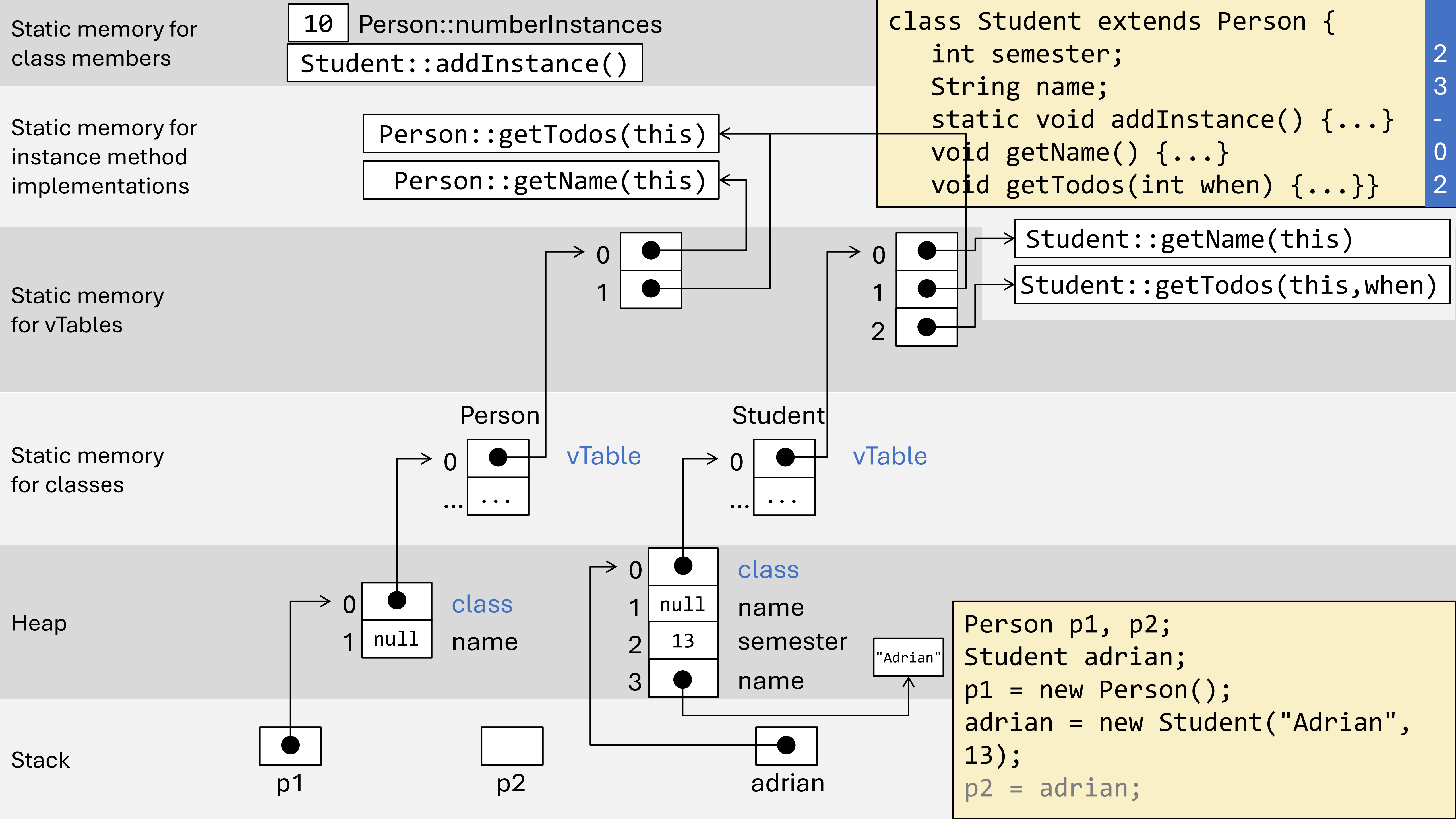
```
class Student extends Person {
   int semester;                          //Addition                2
   String name;                           //Hiding                  3

   static void addInstance() {...}                                  -
   void getName() {...}                   //Overriding              0
   void getTodos(int when) {...}   //Addition + Overloading         2
   Student(String name, int semester) {                             -
      this.name = name;
      this.semester = semester;
   }
}
```




```
class Person {
    static int numberInstances = 10;    -
    String name;                        1

    void getName() {...}                0
    void getTodos() {...}               1
}
```

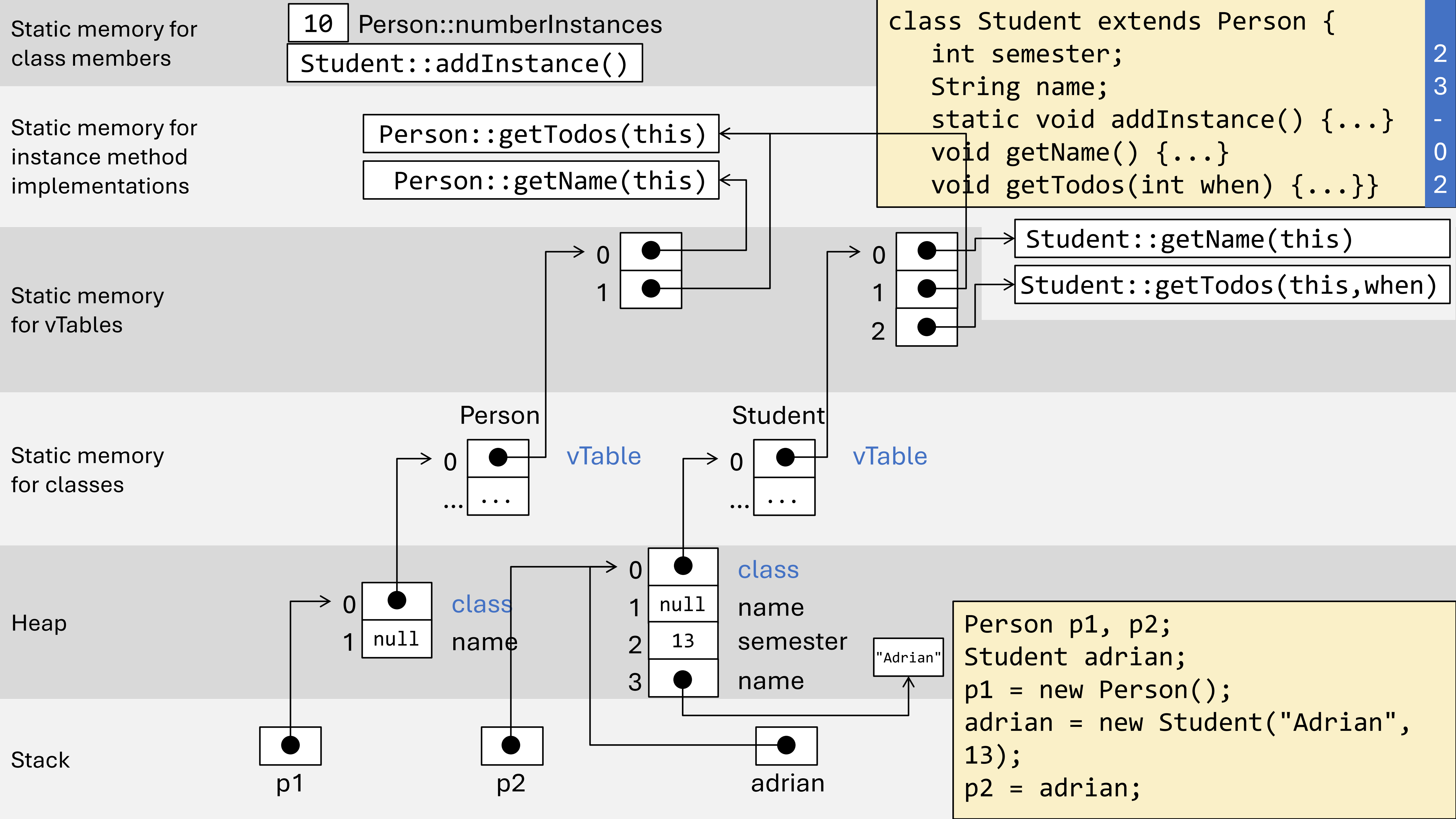
```
Person p1, p2;
Student adrian;
p1 = new Person();
adrian = new Student("Adrian",
13);
p2 = adrian;
```

```
class Student extends Person {
   int semester;
   String name;
   static void addInstance() {...}
   void getName() {...}
   void getTodos(int when) {...}}
```




```
Person p1, p2;
Student adrian;
p1 = new Person();
adrian = new Student("Adrian",
13);
p2 = adrian;
```

```
class Student extends Person {
    int semester;
    String name;
    static void addInstance() {...}
    void getName() {...}
    void getTodos(int when) {...}}
```




```
Person p1, p2;
Student adrian;
p1 = new Person();
adrian = new Student("Adrian",
13);
p2 = adrian;
```

Static Code Generation for Dynamic Access

Access to instance variables are mapped to accesses to the statically known indexed positions in the object's layout.

Messages are accesses to the statically known indexed positions of the vTable the object's class is pointing to.

`p1.name;`	1
`p1.getName();`	0
`p1.getTodos();`	1
~~`p1.getTodos(4);`~~`//not in Person`	X
`p2.getName();`	0
`p2.getTodos();`	1
`p2.getTodos(4);`	2
`adrian.getName();`	0
`adrian.getTodos();`	1
`adrian.getTodos(4);`	2
`adrian.name;`	3

Compiled to

```
p1[1];
p1.class.vTable[0](p1);
p1.class.vTable[1](p1);


p2.class.vTable[0](p2);
p2.class.vTable[1](p2);
p2.class.vTable[2](p2, 4);

adrian.class.vTable[0](adrian);
adrian.class.vTable[1](adrian);
adrian.class.vTable[2](adrian, 4);
adrian[3];
```

Dynamic Binding: Efficiency

The access to `obj.class.vtable[index](...)` reduces the cost of dynamic binding to 3 reading memory accesses and 1 jump to the address of the method.

This jump is the most expensive part, as the processor loads subsequent instructions into a pipeline based on speculation.

Details to that in a machine-level / technical computer science lecture near you.

```
Exception in thread "main"
sse.exceptions.TitleImageException: No title image found.
	at Lecture.advancedProgramming(Lecture.java:9)
```

Exception Handling

What does the program do?

```
> java ExceptionTest 3
```

```
i = 3
```

```
> java ExceptionTest
```

```
Exception in thread "main"
java.lang.ArrayIndexOutOfBoundsException: Index 0
out of bounds for length 0
        at ExceptionTest.main(ExceptionTest.java:5)
```

```
class ExceptionTest {
    public static void main (String[] args) {
      int i = Integer.parseInt(args[0]);
      System.out.println("i = " + i);
    }
}
```

```
> java ExceptionTest a1
```

```
Exception in thread "main" java.lang.NumberFormatException: For input string: "a1"
        at java.base/java.lang.NumberFormatException.forInputString(NumberFormatException.java:67)
        at java.base/java.lang.Integer.parseInt(Integer.java:660)
        at java.base/java.lang.Integer.parseInt(Integer.java:778)
        at Main2.main(Main2.java:5)
```

Structure of an Exception Message

Affected thread.

Qualified name of the exception.

Message.

```
Exception in thread "threadname" package.ExceptionName: The message
        at package.ExceptionName.method(ExceptionName.java:42)
        at ...
        ...
        at Main.main(Main.java:42)
```

Stack trace: chain of method invocations from exception source to `main()`.

Direct link to code line (filename:line).

Handling Exceptions

On the client side, code that potentially might cause exceptions must be wrapped in a try-catch-(finally) block.

Potential exception causing code.

```
try {...}
catch (Exception1 e) {...}
catch (Exception2 e) {...}
catch (Exception3 e) {...}
...
finally {...}
```

Code that should be executed if Exception3 or any subtype of it occurs.

Code that is executed always, after try as well as after catches. Usually for "clean-up".

Causing Exceptions

A method that encounters an exceptional situation creates and throws an exception object.

```
throw new ExceptionClass(...);
```

The runtime system cancels the execution of any method that is not capable of handling the exception. The control flow continues in the first matching `catch` block.

Be aware that all `finally` blocks will still be executed.

Example: Reading from URLs

```
try {
   URL infoKöln = new URL("https://www.cs.uni-koeln.de");
   InputStream in = infoKöln.openStream();
   ...
   in.read();
   ...;
   in.close();
}
catch (MalformedURLException me) {
   System.err.println("MalformedURLException: " + me);
}
catch (IOException ioe) {
   System.err.println("IOException: " + ioe);
}
```

Order important! Put more specific exceptions first! Remember subtyping!

Throws Declaration

A method may delegate the exceptions handling to its clients:

```
public int count(Stream s) throws java.io.IOException {
   int count = 0;
   while (s.read()!= Stream.END)
     count++;
   return count;
}
```

`Stream::read()` throws `IOException`

This is useful if the exception handling process depends on the context of the method invocation.

- Usually in libraries.

Throws Declaration

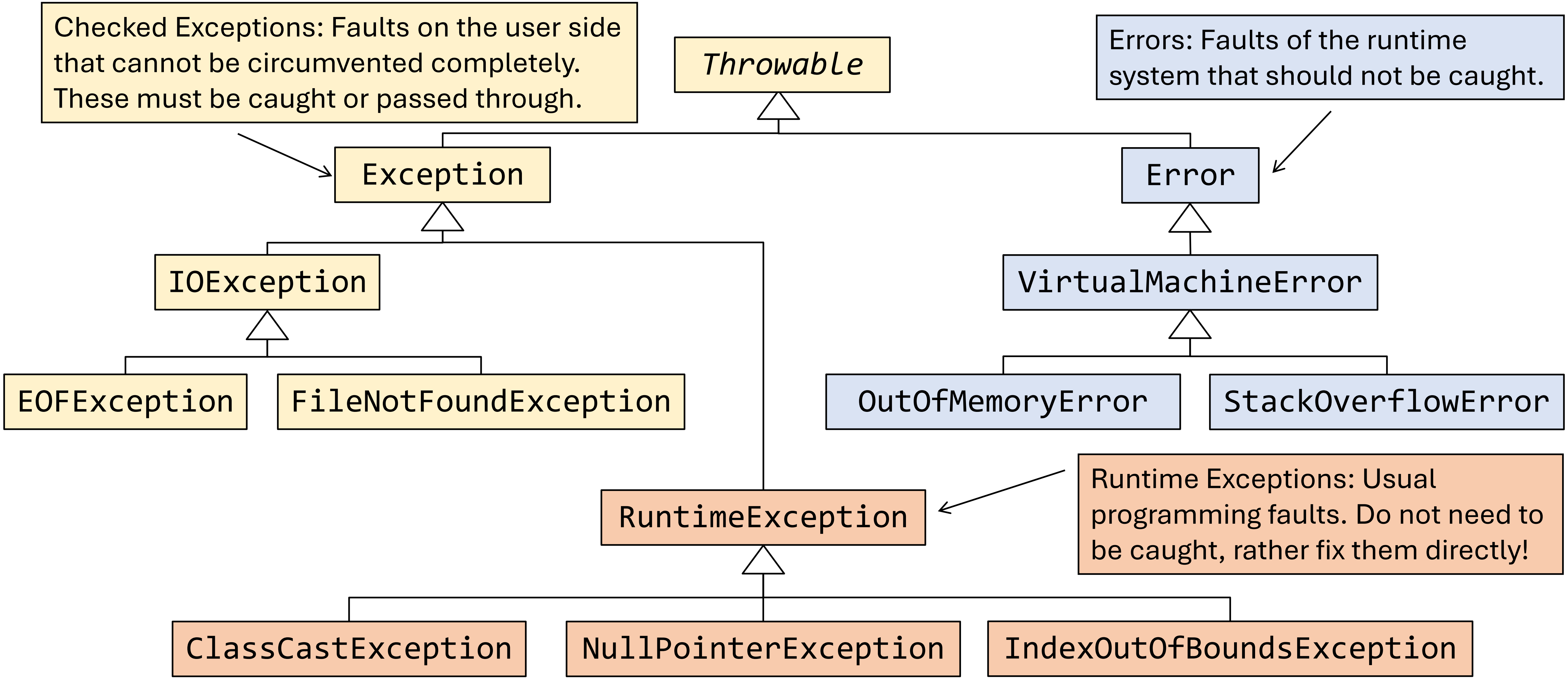
Standard return type

```
ReturnType f(Param1 p1,..., ParamN pN) throws Except1, ..., ExceptN {
    ...
}
```

Exceptions as "alternate return types"

All checked exceptions (either thrown in method or received by called methods) must be declared.

Types of Exceptions and Most Common Exceptions



Questions?

Threads

Why Threads?

Programs are sequential. The world is concurrent, however. Thus, we need a way to let programs run in parallel.

Processes: A process is an independent execution of a program.

Thread: A thread is a parallel execution of parts of a program within the same process.

Threads are easier to handle than processes, for both developing as well as execution. In Java, threads are the only way to develop concurrently.

Threads in Java: Subclassing

In Java, we can define a class as a thread in two ways.

```
class MyThread extends Thread {
   public void run() { ... }
}
```

We can define a class as subclass to `Thread`.

- Creation of thread via constructor.
- Starting via calling inherited `start` method.

```
Thread t1 = new MyThread();
Thread t2 = new MyThread();
```

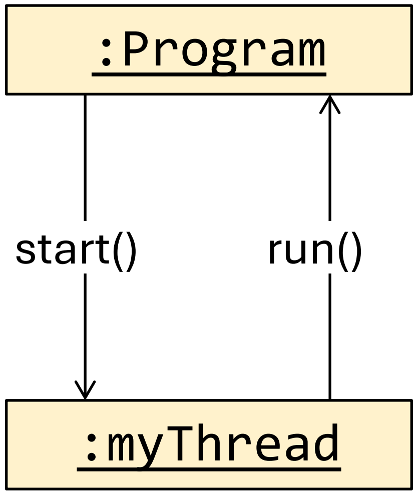
```
t1.start();
t2.start();
```

Threads in Java: Implementing Runnable

We let our class implement the interface `Runnable` and implement the `run` method.

- Creation of thread: passing instance of the class as argument to constructor of Thread.
- Starting via calling `Thread::start()`.

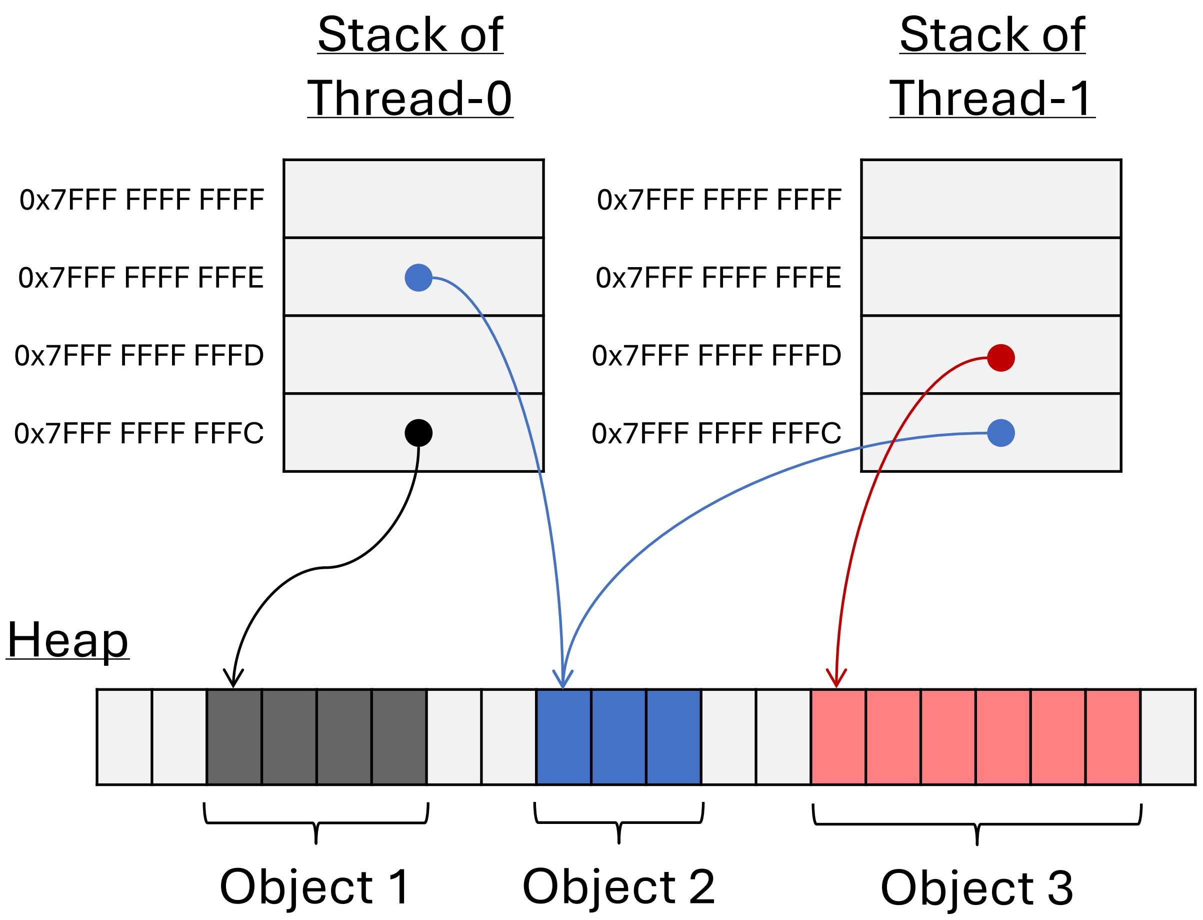
```
class Program extends Superclass implements Runnable {
   Thread myThread;

   public void run() {...}

   public void start() {
      myThread = new Thread(this);
      myThread.start();
   }
}
```



Threads need Synchronization

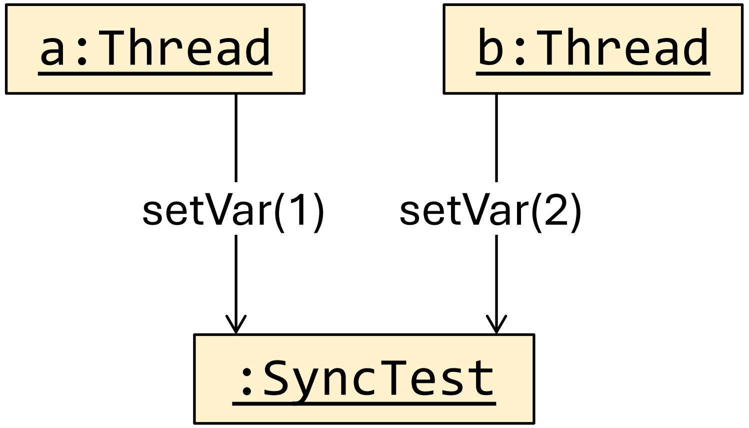
Each thread has its own stack.
All threads share the same heap space.

Therefore, access to non-primitive types need to be synchronized!



Thread Synchronization

```
class SyncTest {
   int var;
   public synchronized void setVar(int var) {
      this.var = var;
   }
}
```



We can synchronize a method via the keyword `synchronized`. This guarantees that the execution of the method is not interrupted by another synchronized method of the same class.

Any other methods, however, can run in parallel.

Controlling Threads

- `Thread.sleep(millis)`
  - Let active thread pause for *n* milliseconds.
  - The thread does not release its locks!

- `Thread.yield()`
  - Pause active thread and release its locks.

Controlling Threads

- `t.join(millis)`
  - Let active threads wait for *n* milliseconds for the end of thread *t* and continue afterwards.
  - Without parameter: Wait until *t* ends.

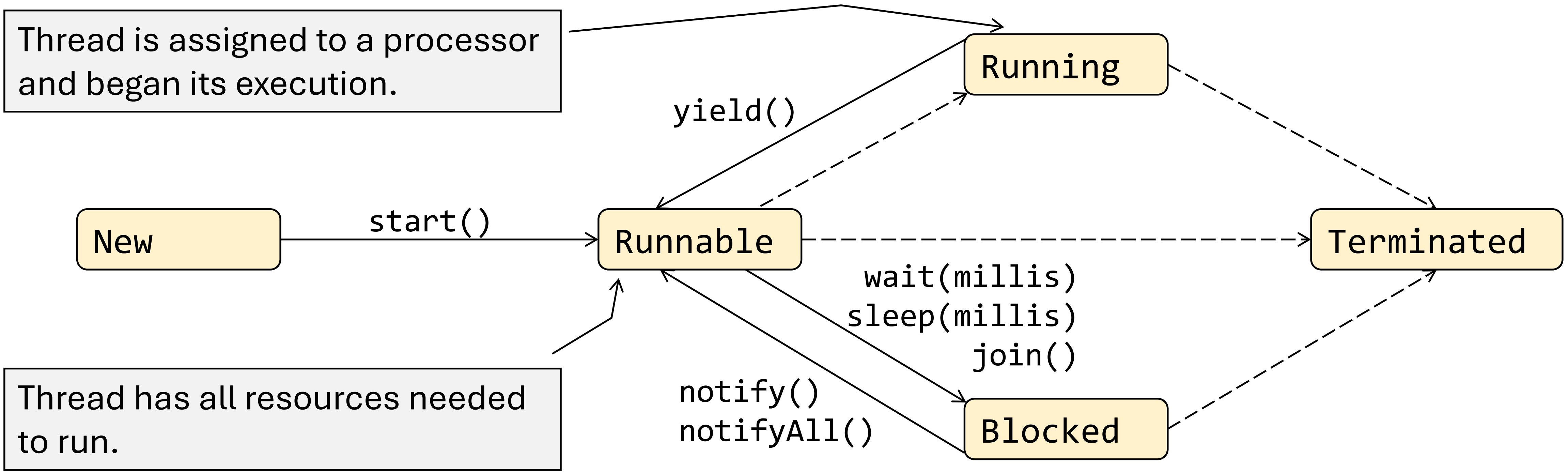
- `object.notify()`
  - The active threads releases its locks on `object`.
  - A threads that waits via `wait()` on `object` is activated.

Controlling Threads

- `object.wait(millis)`
  - The active thread releases its locks on `object`.
  - Then, the active thread wait n milliseconds on the release of locks on `object` of other threads.
  - When time is up or `object` is released, the current thread is reactived and added to the waitinglist of `object`.
  - Without parameter: Wait arbitrary long.

Thread States

State transitions without explicit method (drawn with dashed lines) happen automatically.



Questions?

Summary

In today's lecture:

- What is object identity?
- What are types (for)?
- What is inheritance? (and what is it not?)
- What are interfaces (for)?
- What is dynamic binding and how does it work?
- Exception handling.
- Concurrency via threads.

Use this knowledge when developing code, in your job as well as in the exercises.