
Development of Component- and Service-Oriented Software Architectures with UML and UML-RT — Objects and Classes —

Ingolf H. Krueger
ikrueger@ucsd.edu

Department of Computer Science & Engineering
University of California, San Diego
La Jolla, CA 92093-0404, USA



California Institute for Telecommunications
and Information Technologies
La Jolla, CA 92093-0405, USA



Overview

- Objects
- Classes
- Class and Object Diagrams

Advantages of Object-Orientation

- Seamless methodological approach covering analysis to implementation
 - OO modeling techniques (classes/objects)
 - OO implementation languages (C++, Java, Smalltalk, ...)
- Support for understanding the application domain
 - Mapping of “real” entities to objects of the model
 - Abstraction mechanisms available
- Starting points for reuse of structure and behavior:
 - Classes,
 - Interfaces,
 - Objects/components

Advantages of Object-Orientation

- Adequate even for modeling of large systems
- Support for developing “safe” components:
 - Interfaces
 - Encapsulation of data structures and behavior
- Support for distribution/parallelism
 - Objects as independent and active entities executing in parallel

Objects

- Objects model (individual) entities of the application domain
- Example (telecommunication):
 - Caller
 - Connection
 - Account
 - ...
- Objects have unique identity
- Objects encapsulate
 - Data/state
 - Behavior

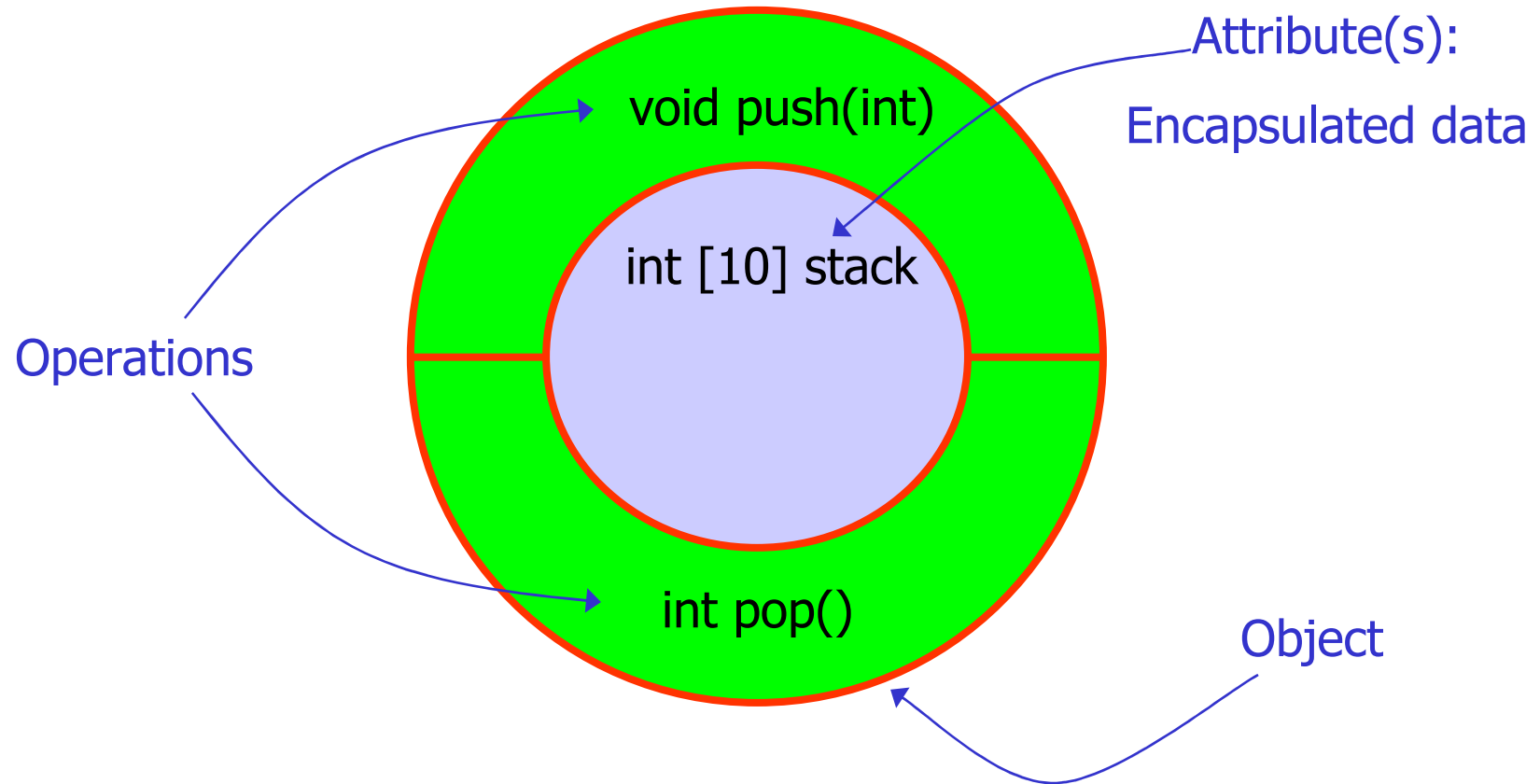
Objects

- Access only via explicit message exchange

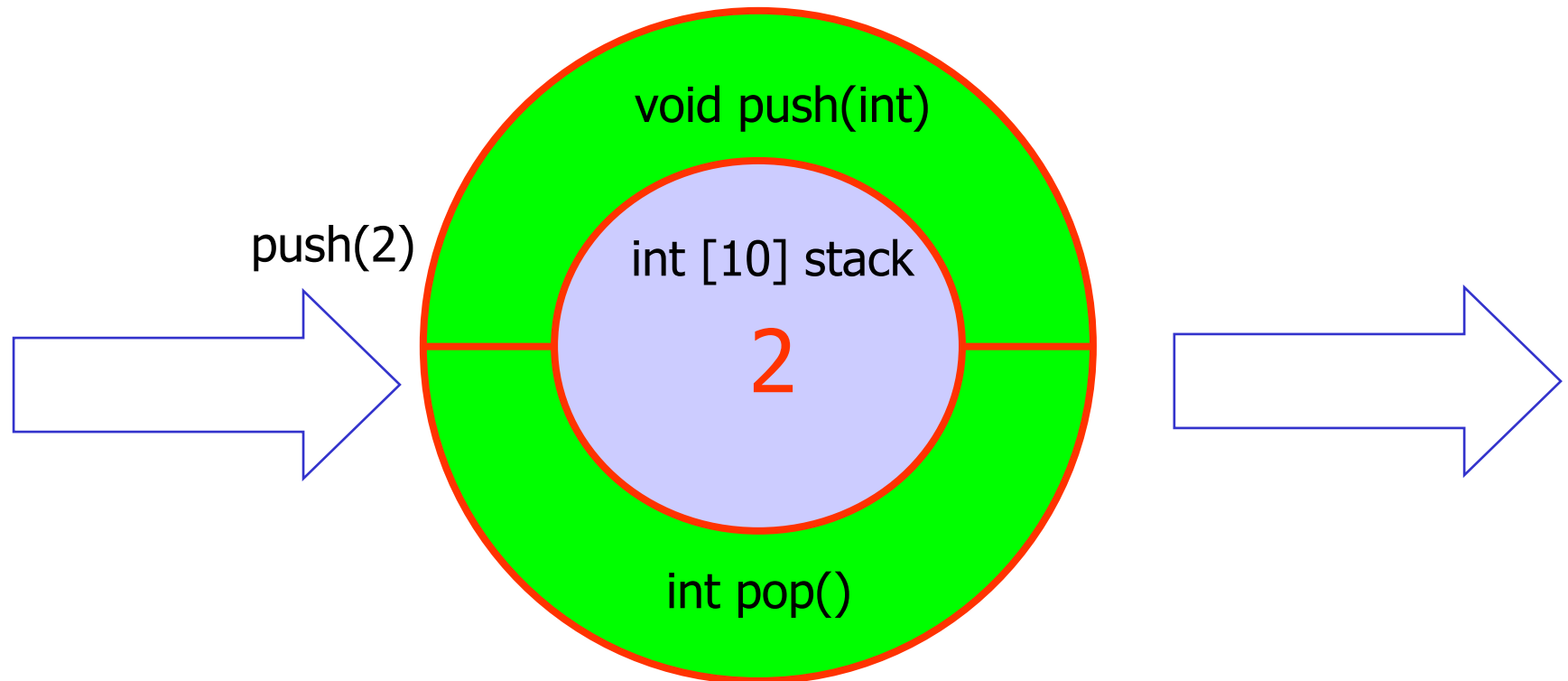
⇒ Integrity of data

- Operation/Method:
enable specification of pre-/postconditions

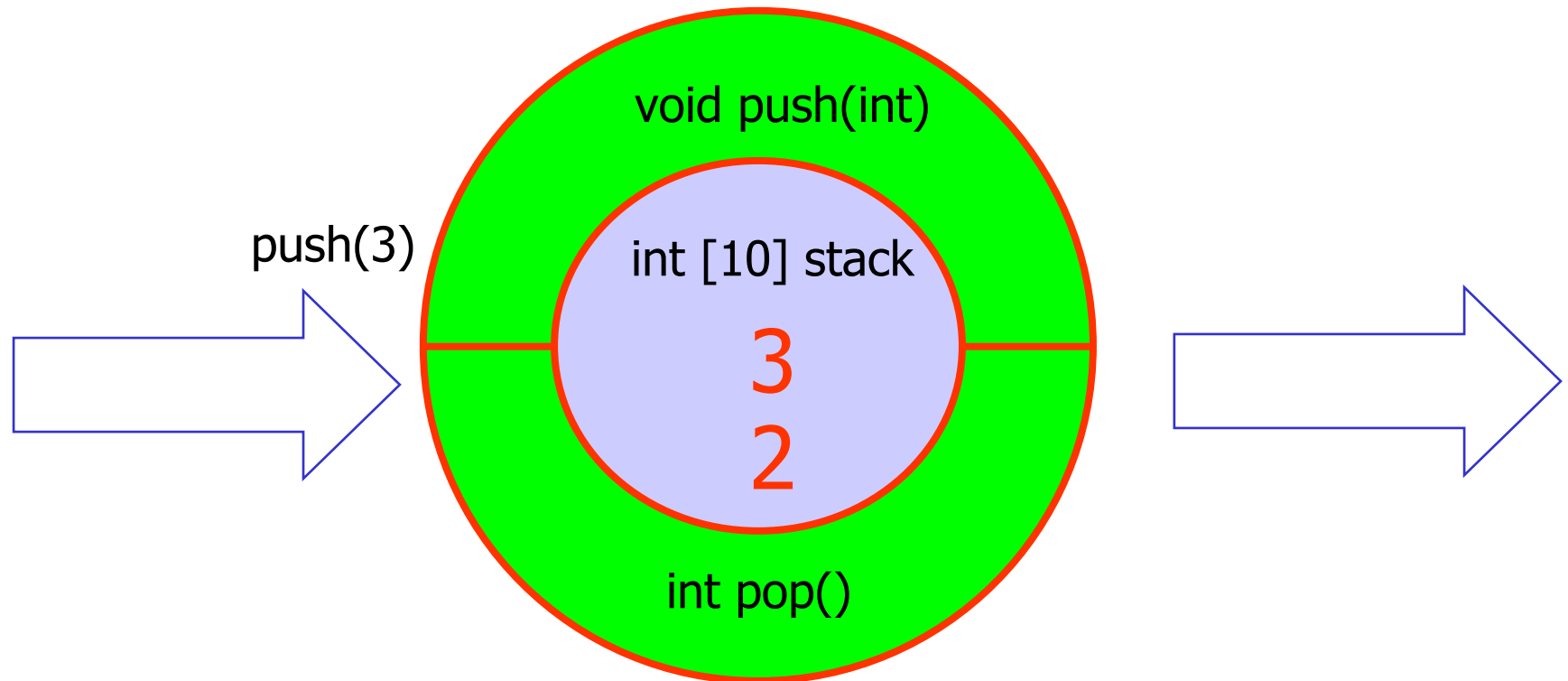
Example: Stack-Object



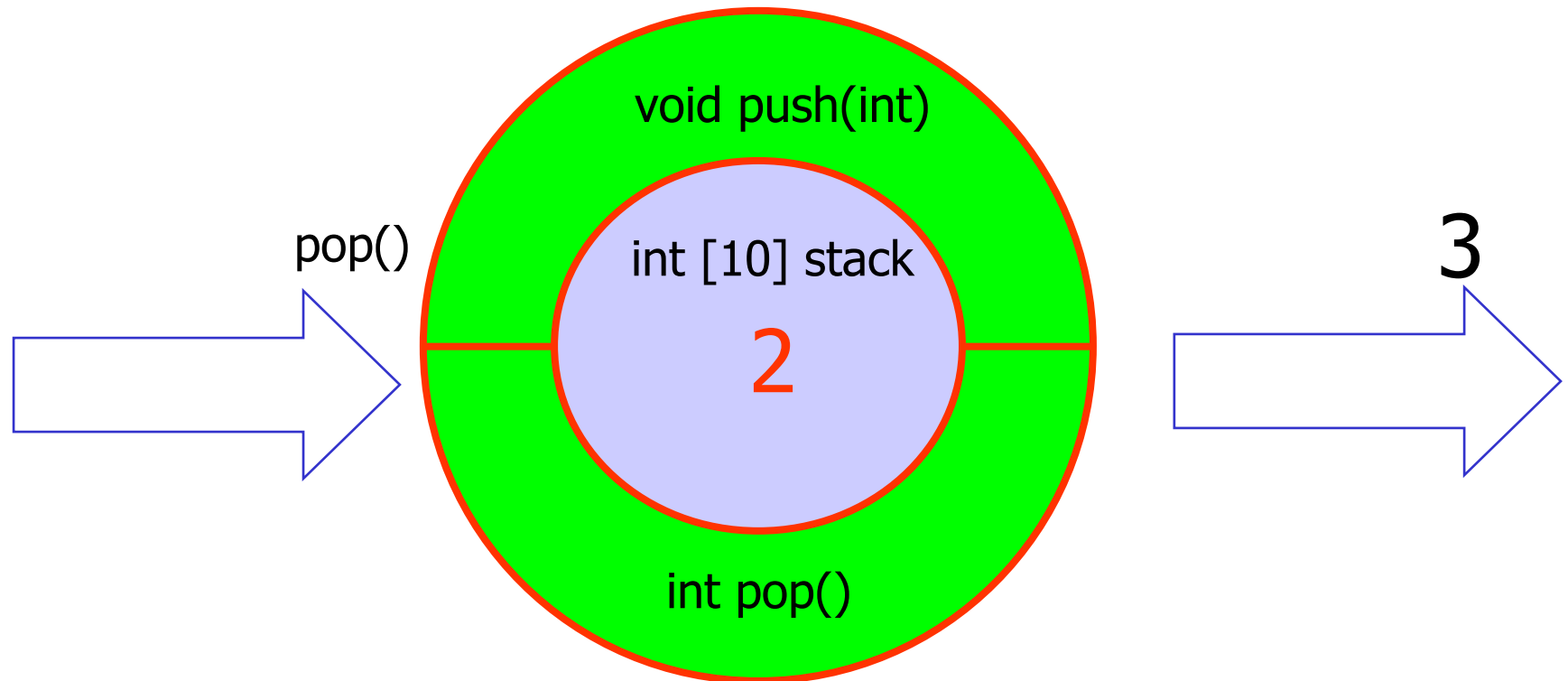
Example: Stack-Object



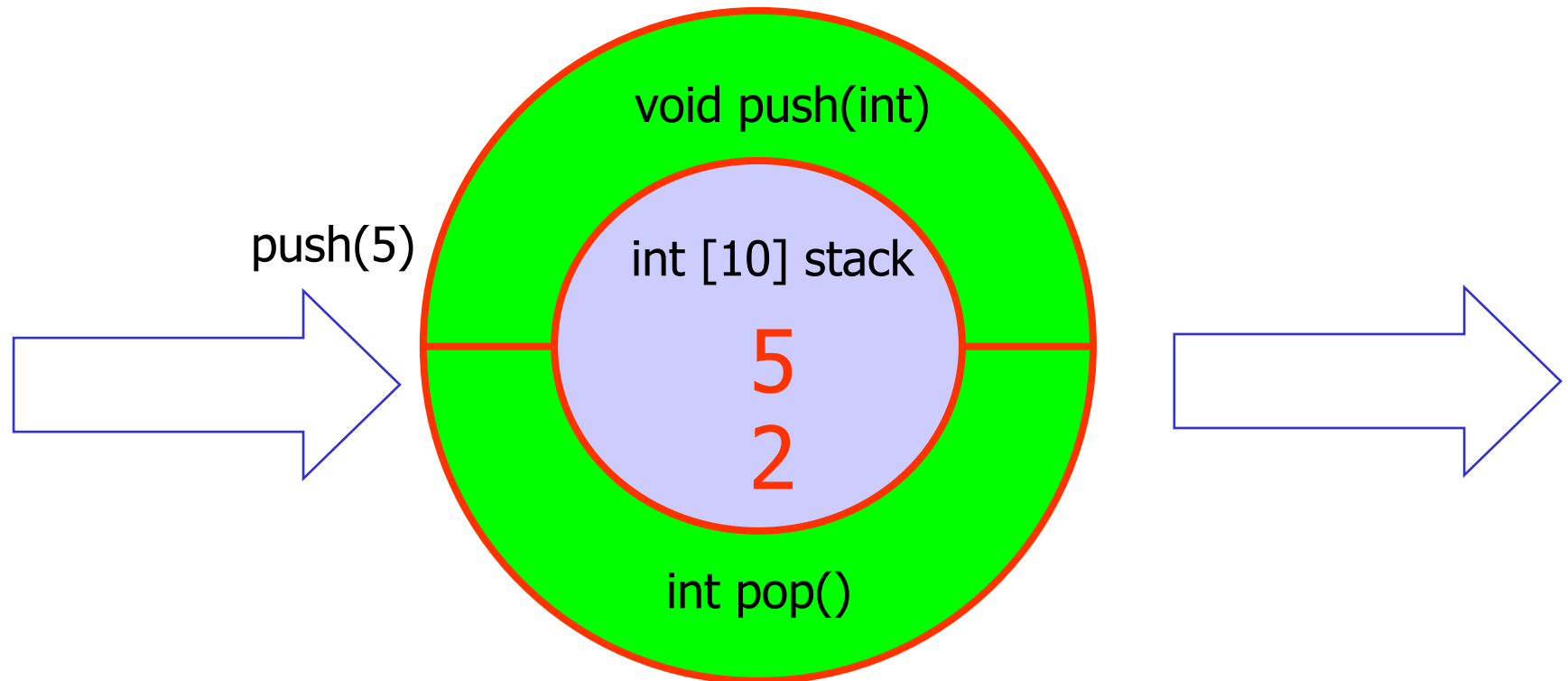
Example: Stack-Object



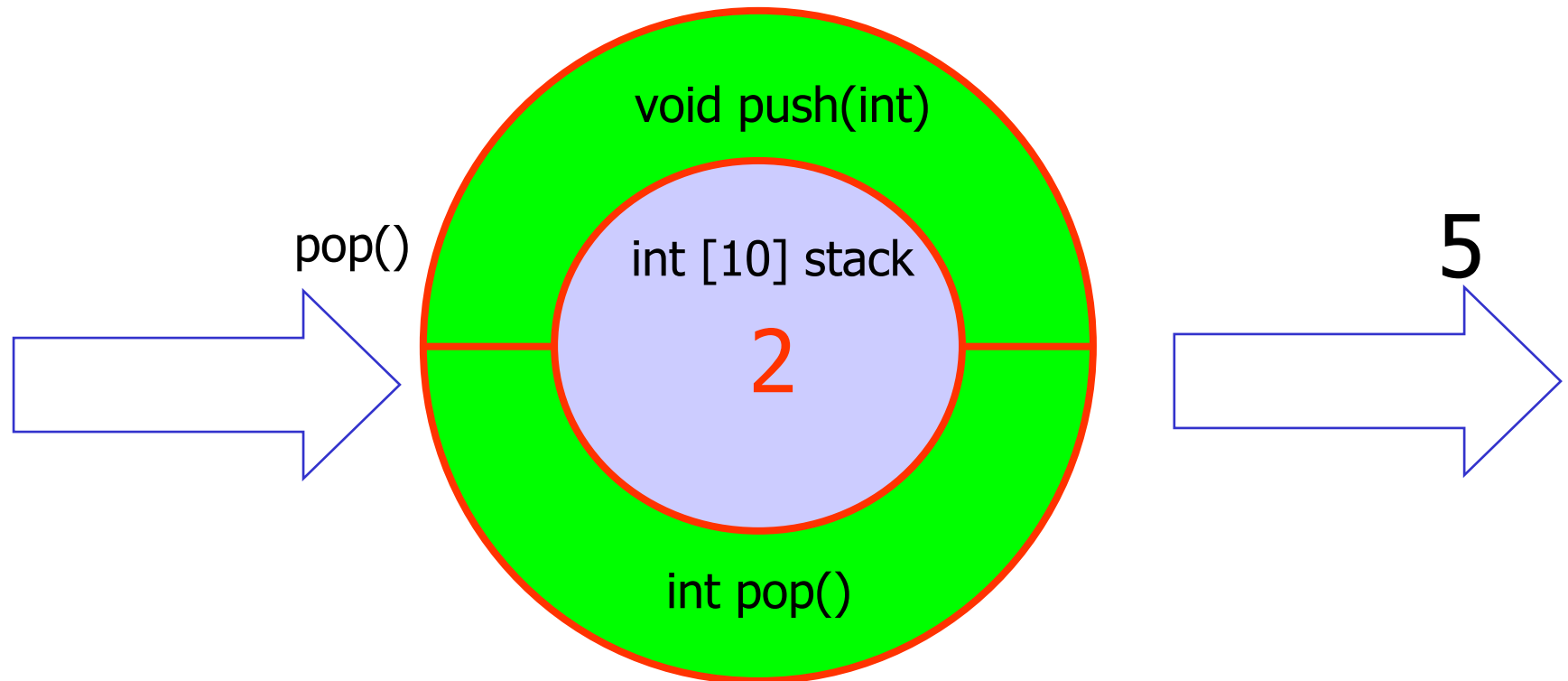
Example: Stack-Object



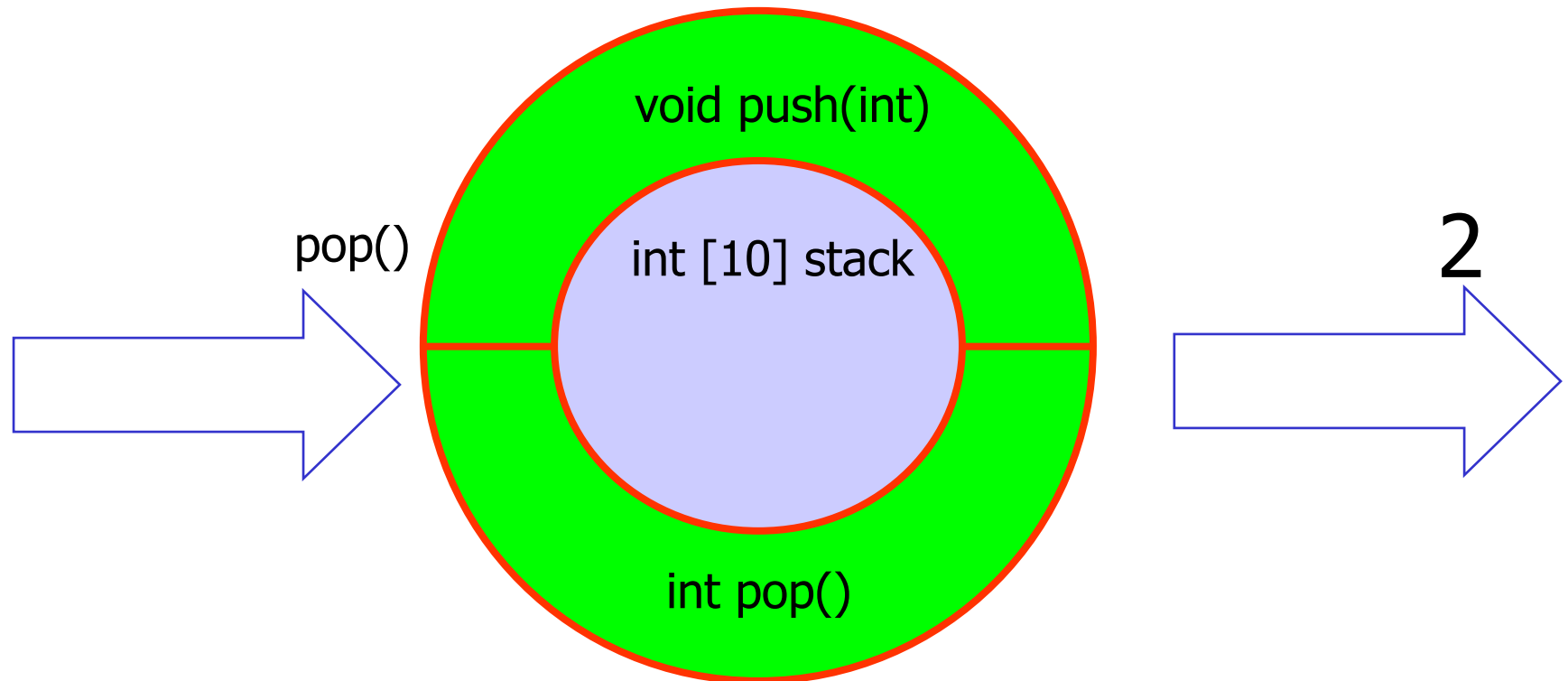
Example: Stack-Object



Example: Stack-Object



Example: Stack-Object



Example: Stack-Operations

- Operation “push(arg)”:
 - only allowed, when stack not full (otherwise: exception handling)
 - arg becomes new top element of stack,
 - rest of stack remains unchanged

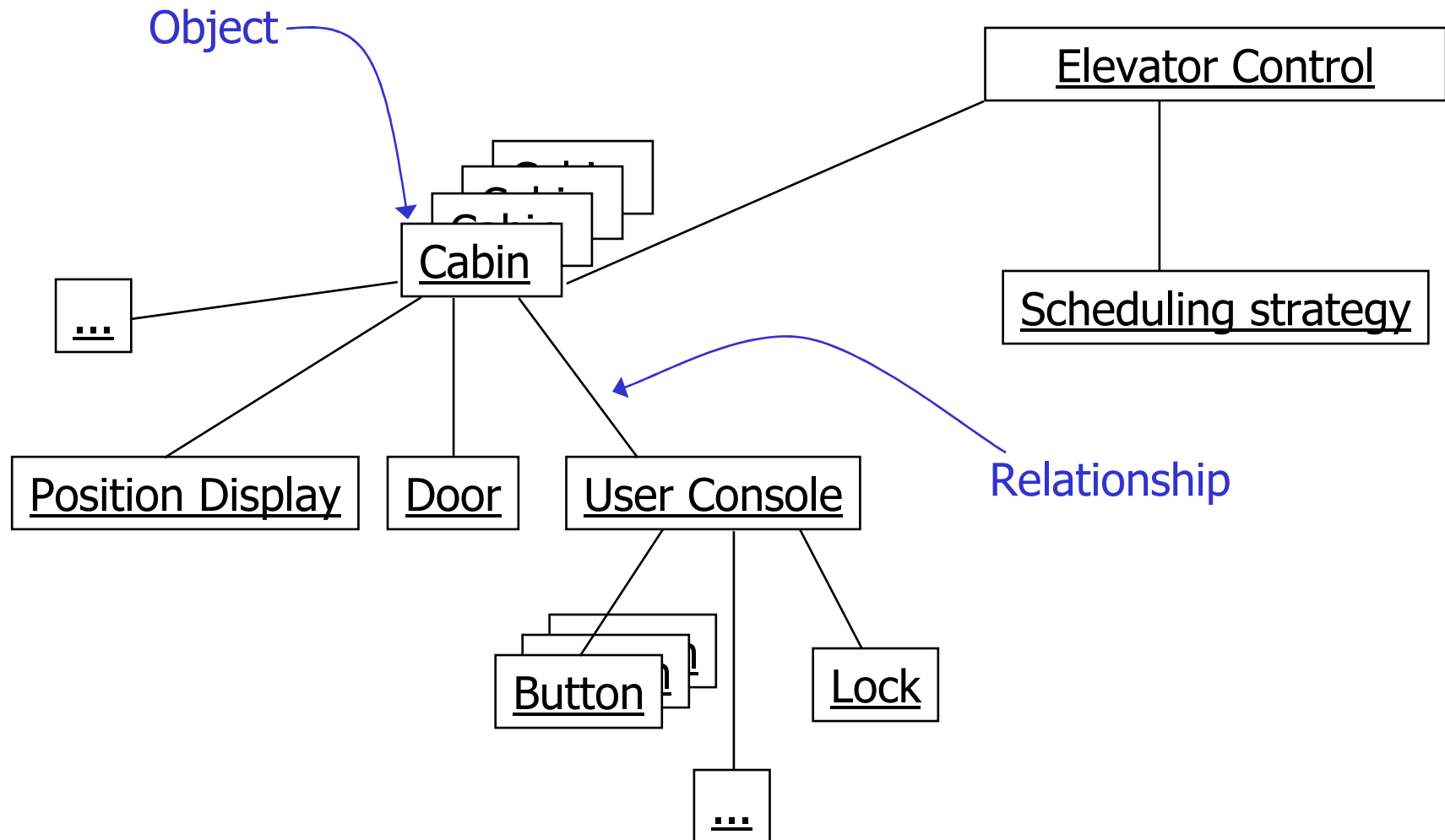
Precondition

Postcondition
- Operation “pop()”:
 - only allowed, when stack not empty (otherwise: exception handling)
 - top stack element is returned, and removed from stack,
 - rest of stack remains unchanged

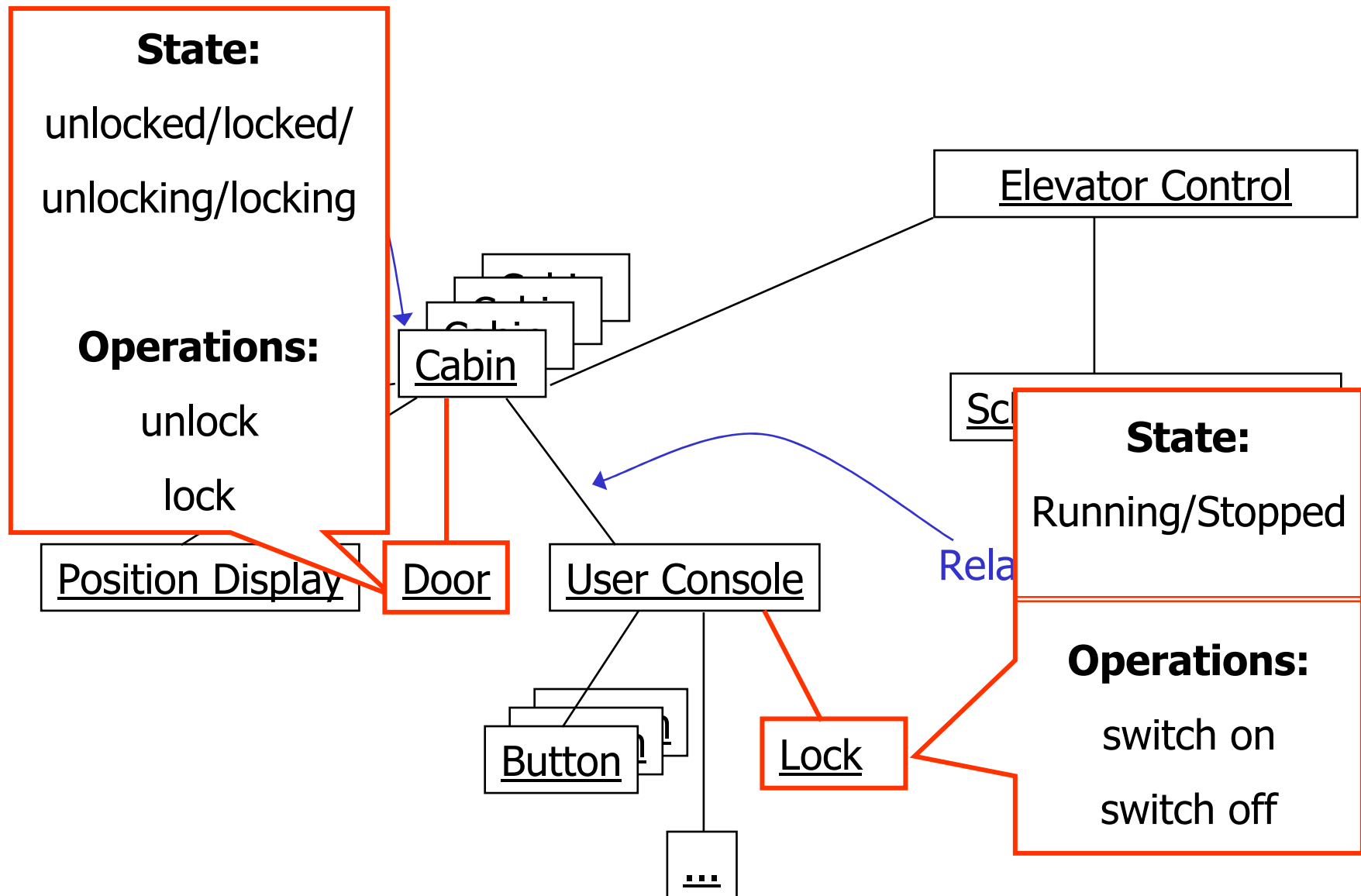
Precondition

Postcondition

Example: Elevator Controller



Example: Elevator Controller



Overview

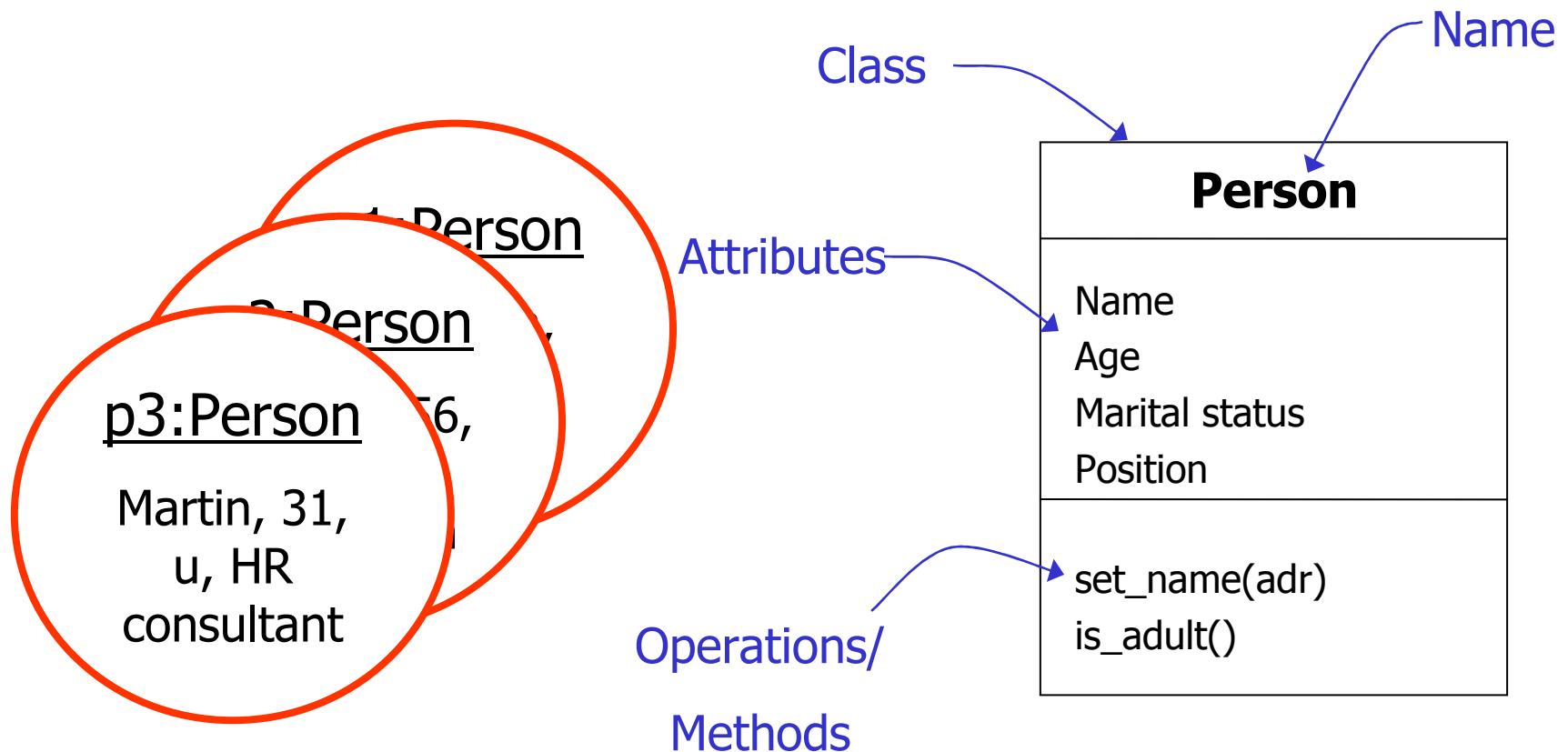
- Objects
- Classes
- Class and Object Diagrams

Classes

- Objects model (individual) entities of the application domain
- Many objects follow the same “blue-print”
Example (elevator):
 - Cabins
 - Locks
 - Buttons
 - Scheduling strategies
- “blue-print” consists of
 - data-/state-structures
 - Patterns of behavior

Classes

- **Class** in OO approach:
 - Description of the similarities (attributes and operations) of a set of objects
 - Template for instantiating individual objects



Classes

- Classes are one of the key means for structuring and abstraction in OO
- A class models a **concept** of the application domain / implementation
- Each object is an **instance** of a class
synonyms: object, instance, class instance, ...
- Classes define templates for individual objects
- Class structure, relations among classes:
class diagram
- Object structure, relations among objects:
object diagram

Overview

- Objects
- Classes
- Class and Object Diagrams

Class and Object Diagrams

- **Class diagrams:**
 - Representation of classes and their relationships
 - Examples for relationships among classes:
 - Existence of a communication path
 - Inclusion
 - Generalization/Specialization
- **Object Diagrams:**
 - Syntax as in class diagrams
 - But: representation of objects instead of classes

Class Diagrams: Modeling Elements

- Classes
- Attributes
- Operations



Description of “object template”

- Association
- Aggregation/Composition
- Generalization/Specialization



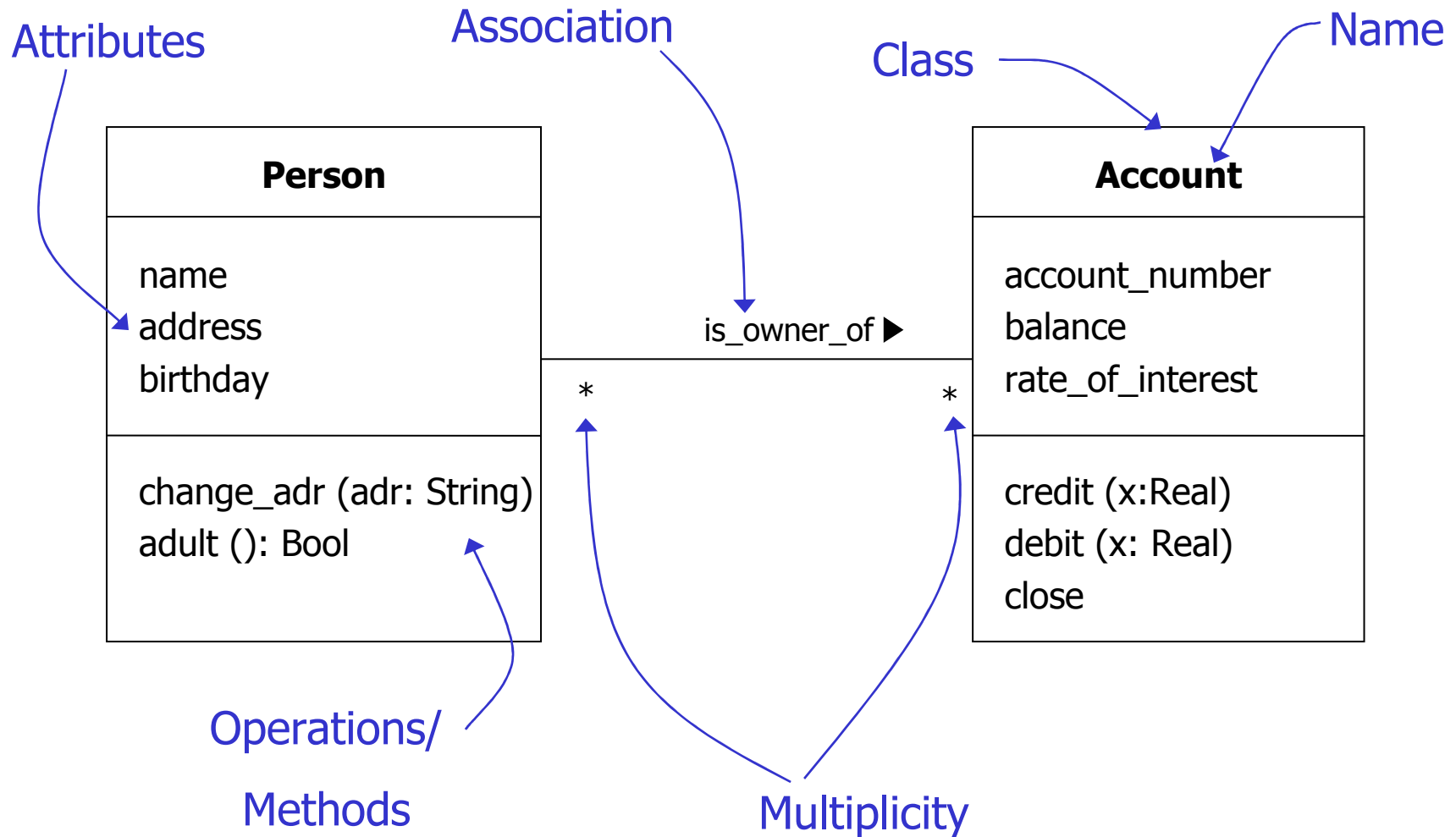
Description of relationships
among multiple classes

- Conditions



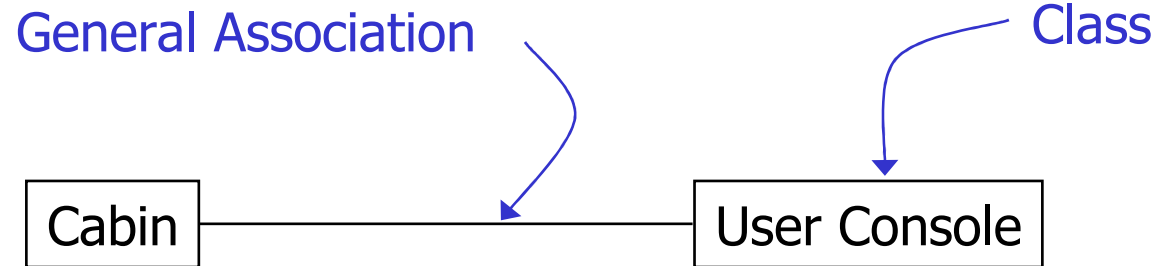
Restrictions of the
corresponding object model

Class Diagrams



Associations

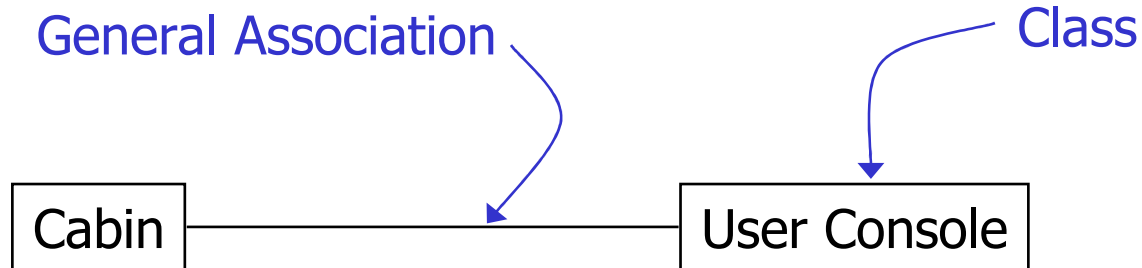
Associations describe **navigation paths**
through the class/object model



Meaning: There is a **link** (e.g. communication path) between instances of classes "Cabin" und "User Console".

Here: **navigation in both directions possible**

Associations: Implementation



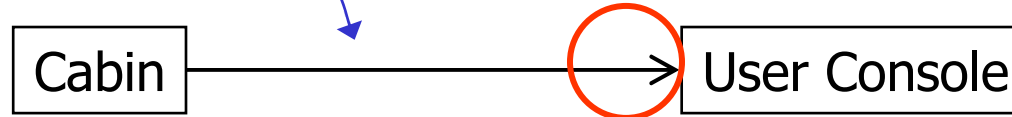
```
class Cabin {
    User_Console bc = null;
    void setCo(User_Console _bc) {
        bc = _bc;
    } ... }
```

```
class User_Console {
    Cabin c = null;
    void setCa(Cabin _c) {
        c = _c;
    } ... }
```

```
User_Console console = new User_Console();
Cabin cabin = new Cabin();
console.setCa(cabin); cabin.setCo(console);
```

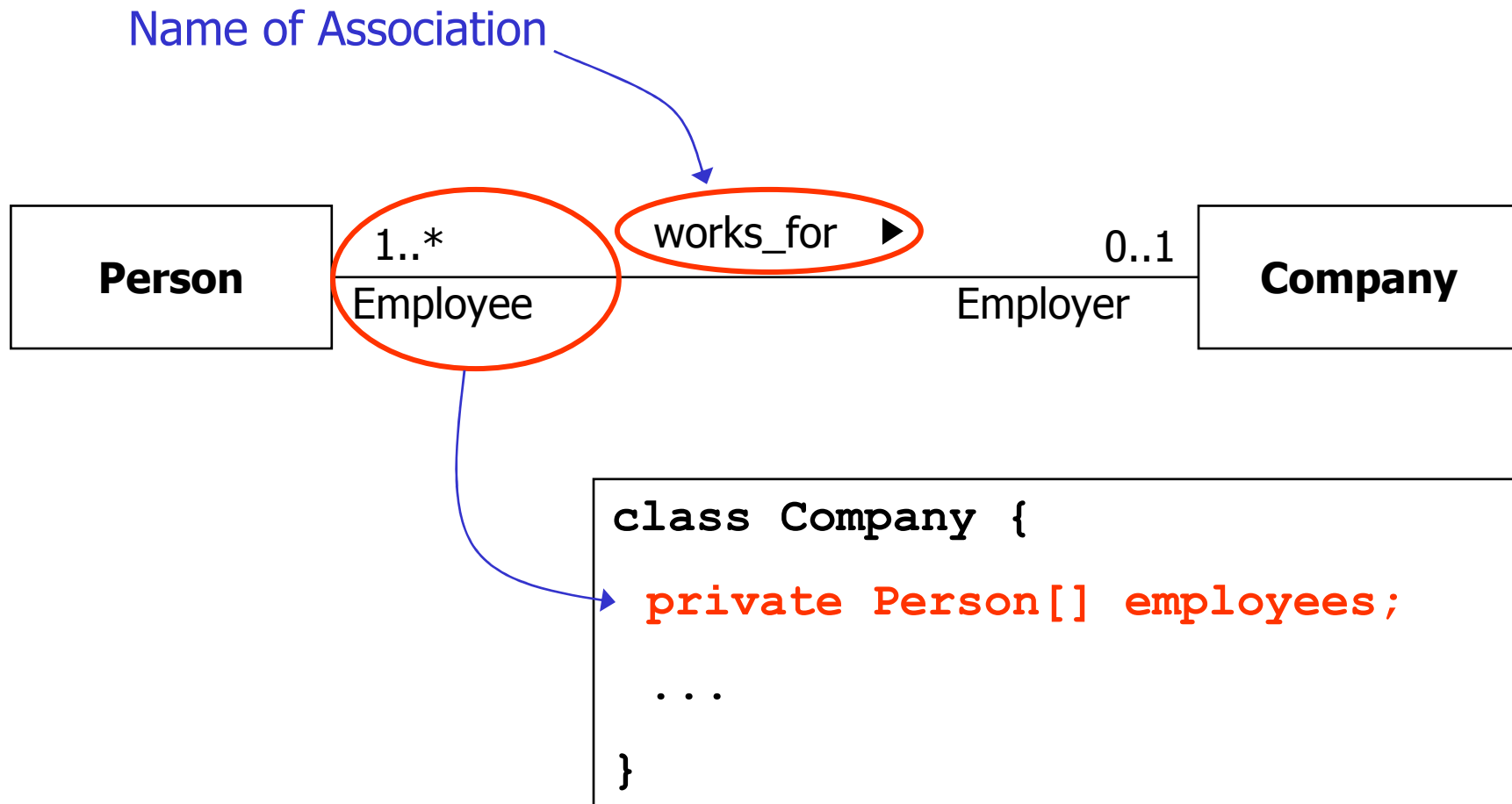
Other Association Types

directed association

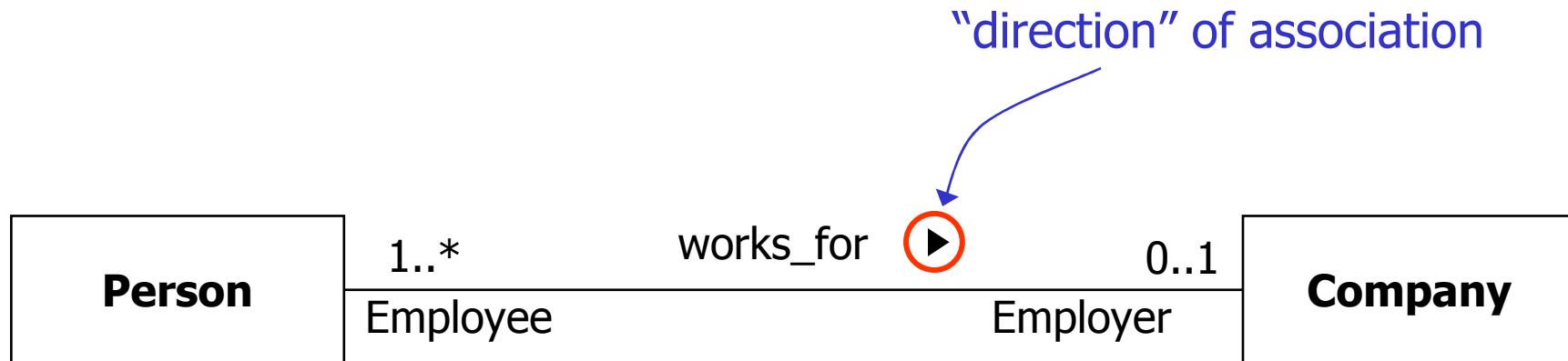


“Cabin” instances can access “User Console” instances,
but not the other way around!

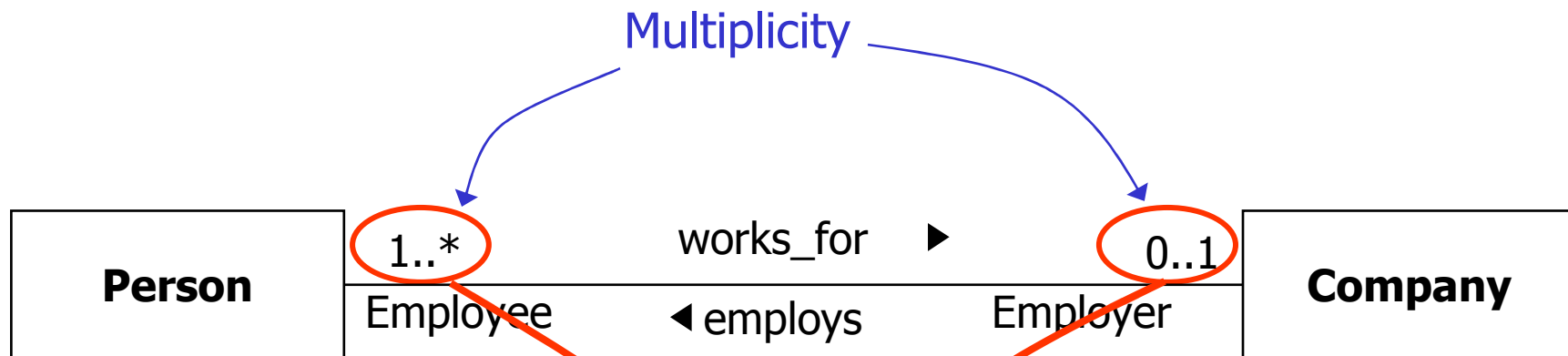
Other Association Types



Other Association Types



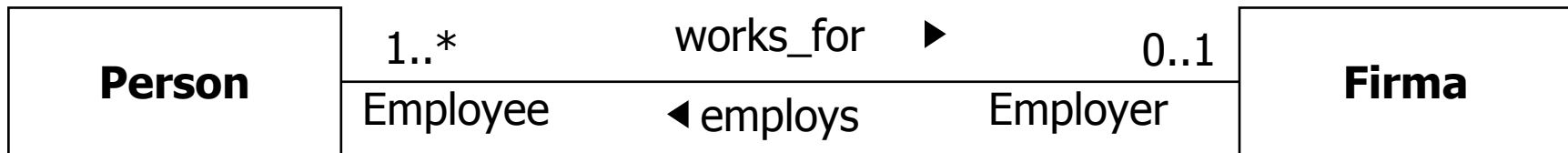
Other Association Types



A **Person** works for at most one (i.e. "0..1") **Company**

A **Company** employs at least one (i.e. "1..*") **Person**

Other Association Types



General Format for Multiplicities:

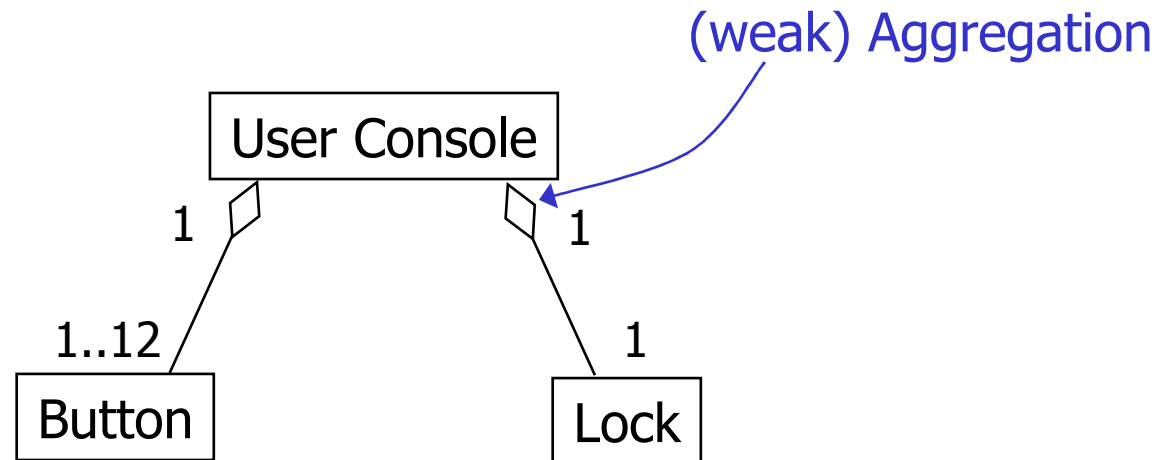
$a..b$ (a, b: natural numbers or "*")

$*$ = $*..*$ = $0..*$

1 = $1..1$ (exactly one)

n = $n..n$ (n : natural number)

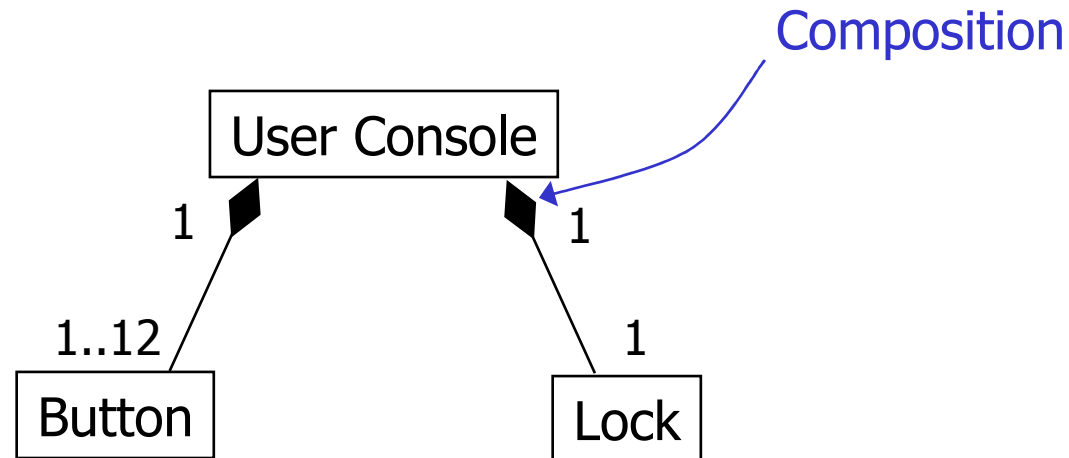
Other Association Types: Aggregation



Every user console "has" 1-12 buttons and exactly one lock

Every lock/button belongs to exactly one user console

Other Association Types: Composition



Every user console *contains* 1-12 buttons and exactly one lock

The user console is responsible for **construction** and **destruction** of locks and buttons; it controls the “life cycle” of its locks/buttons

Every button/lock belongs to exactly one user console

General Forms of Aggregation

- Constant Aggregation

Every sub-object is part of at most one “container” over the system’s runtime

- Dependent Aggregation

No sub-object can exist without a “container”; it must be part of at least one “container”

- Encapsulated Aggregation

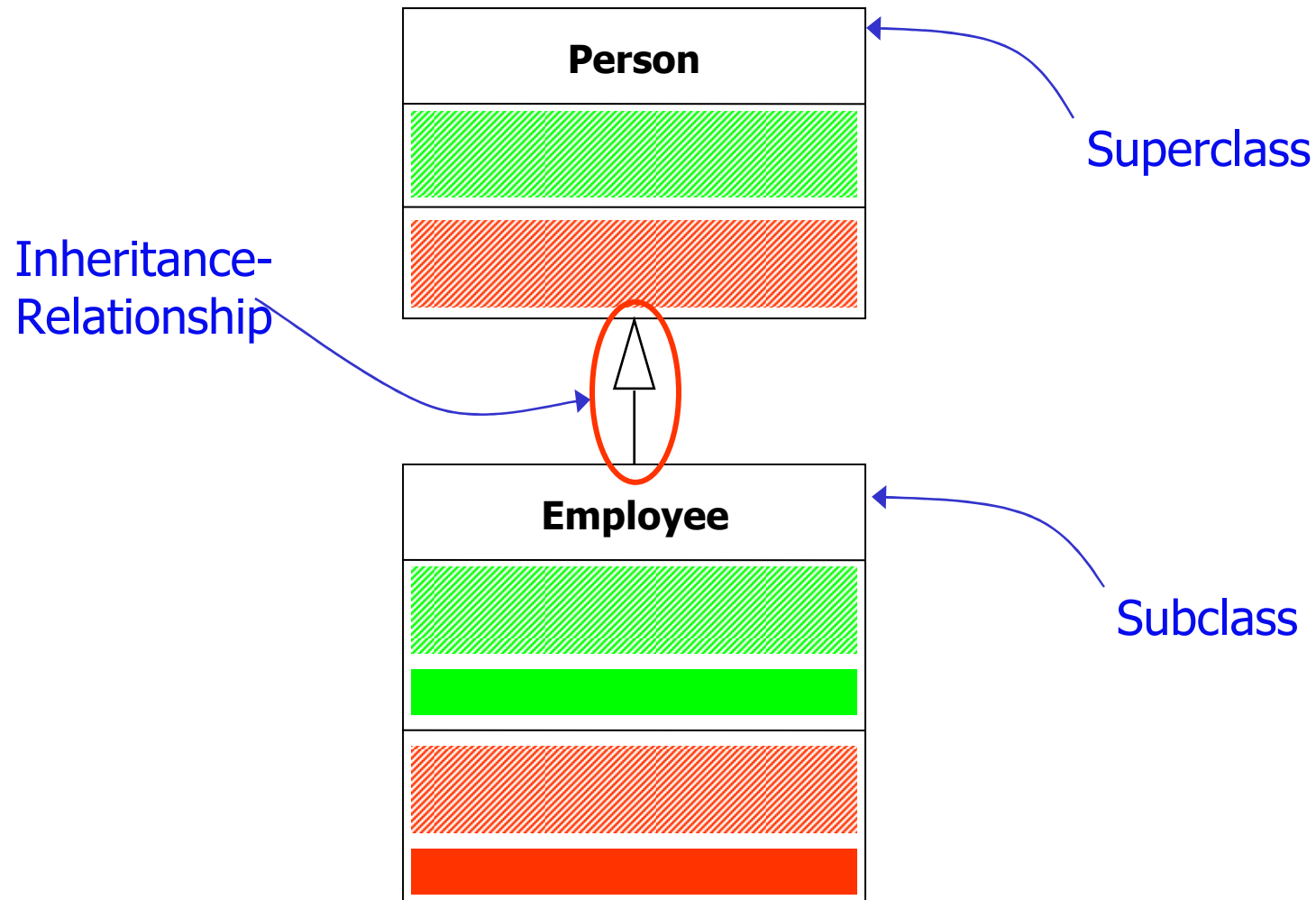
Sub-objects receive messages only from within their container; there is no communication beyond container boundaries

- UML:

- Composition = Constant Aggregation + Dependent Aggregation
- Weak Aggregation: No restrictions

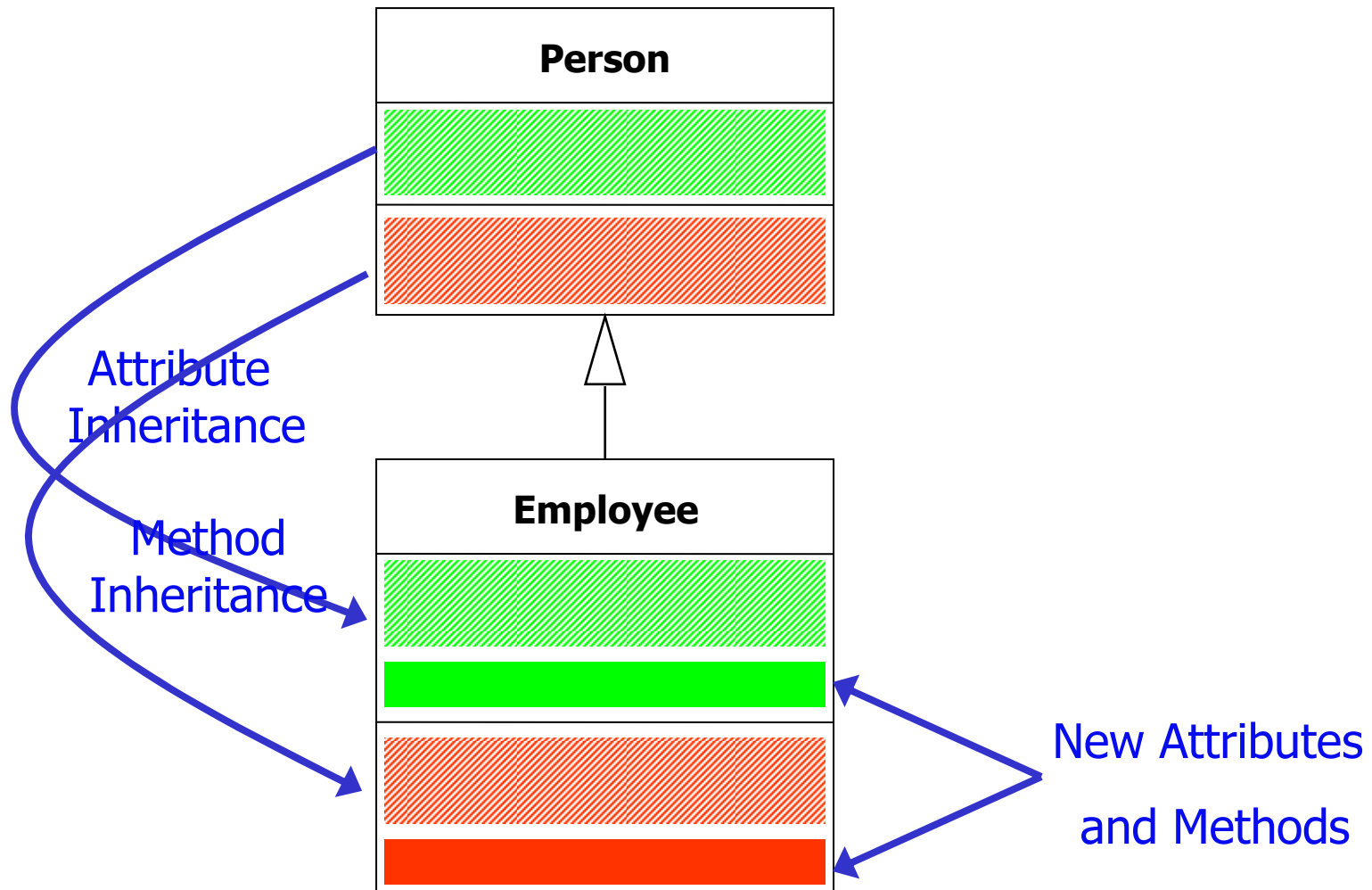
Inheritance

Inheritance: Generalization/Specialization on class level

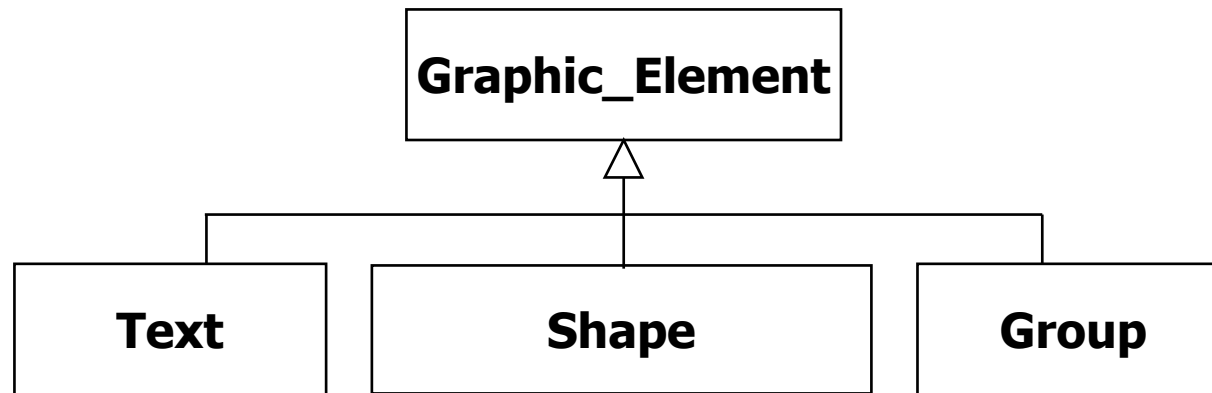


Inheritance

Inheritance: Generalization/Specialization on class level

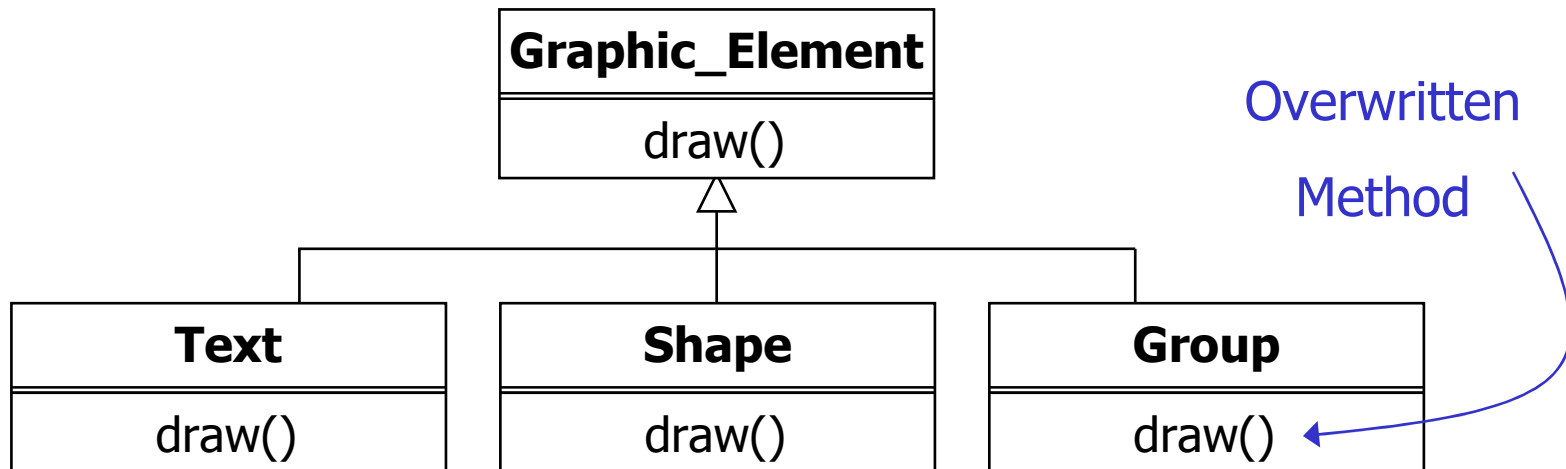


Inheritance



- Subclasses inherit attributes, associations and operations from their superclasses
- Subclasses can overwrite inherited operations
- Each instance of a subclass is also an instance of its superclass (Polymorphism)

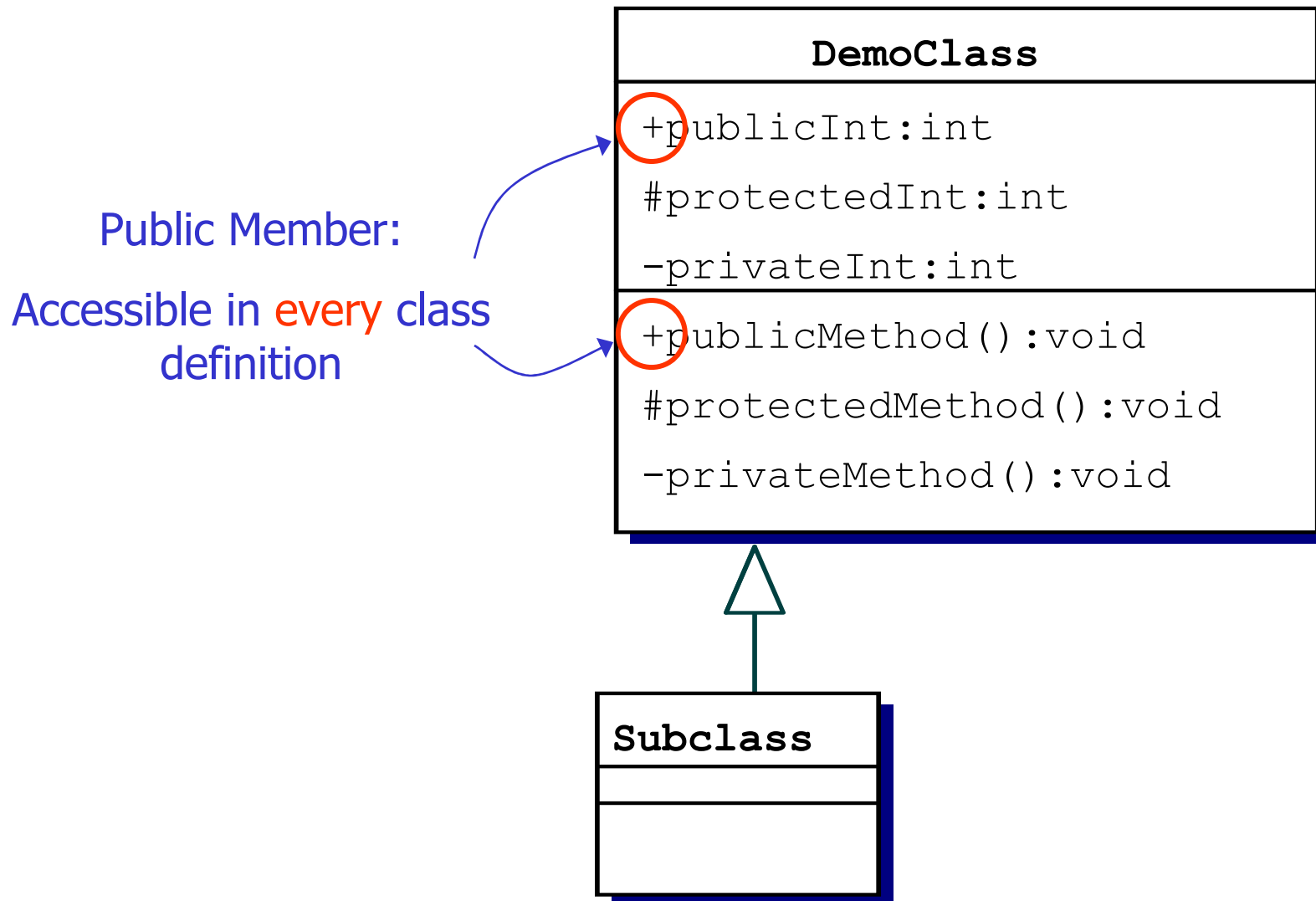
Polymorphism



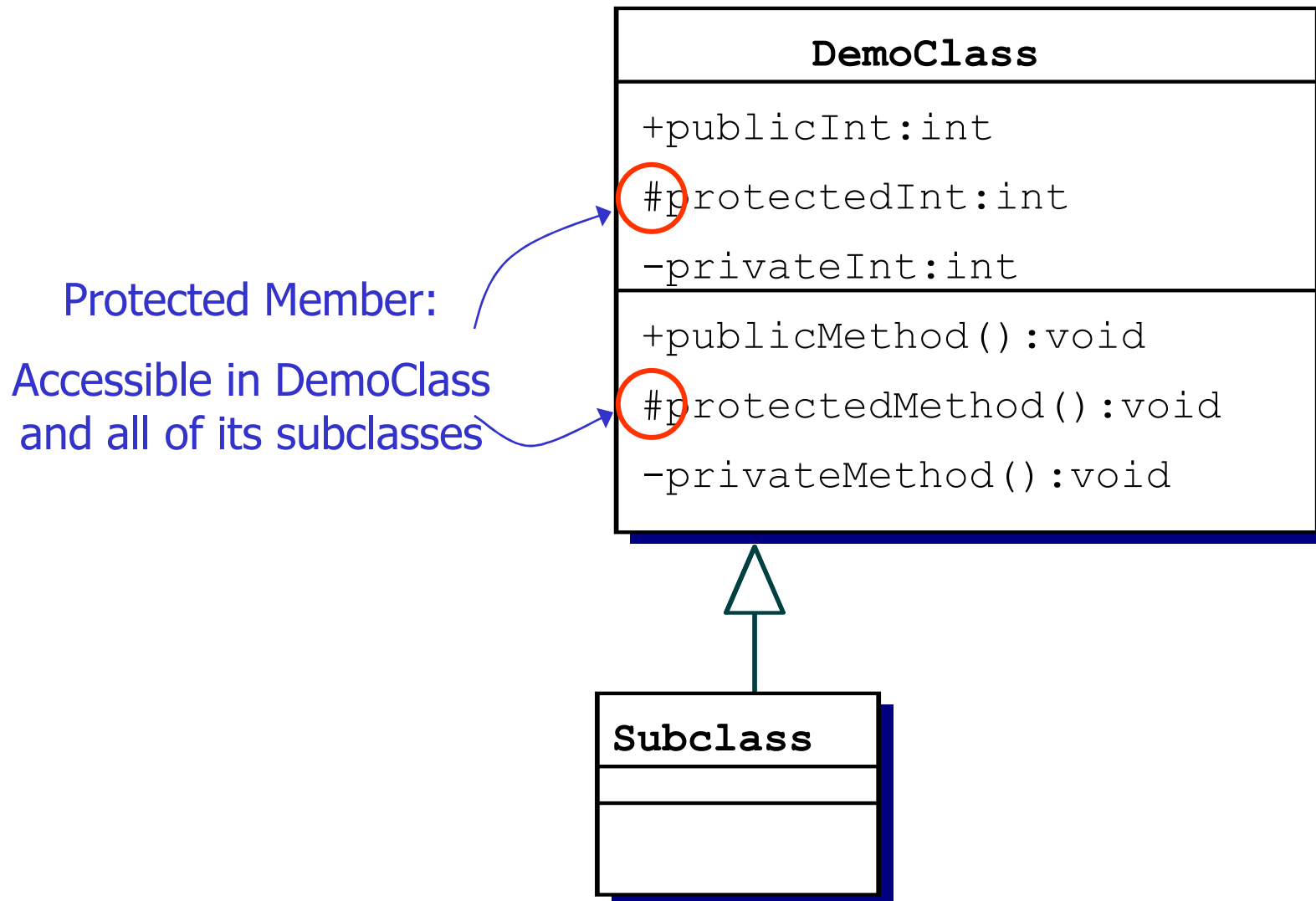
```
void draw_content(Graphic_Element ge) {  
    ge.draw();  
}
```

```
draw_content( new Text() );  
draw_content( new Shape() );  
draw_content( new Group() );
```

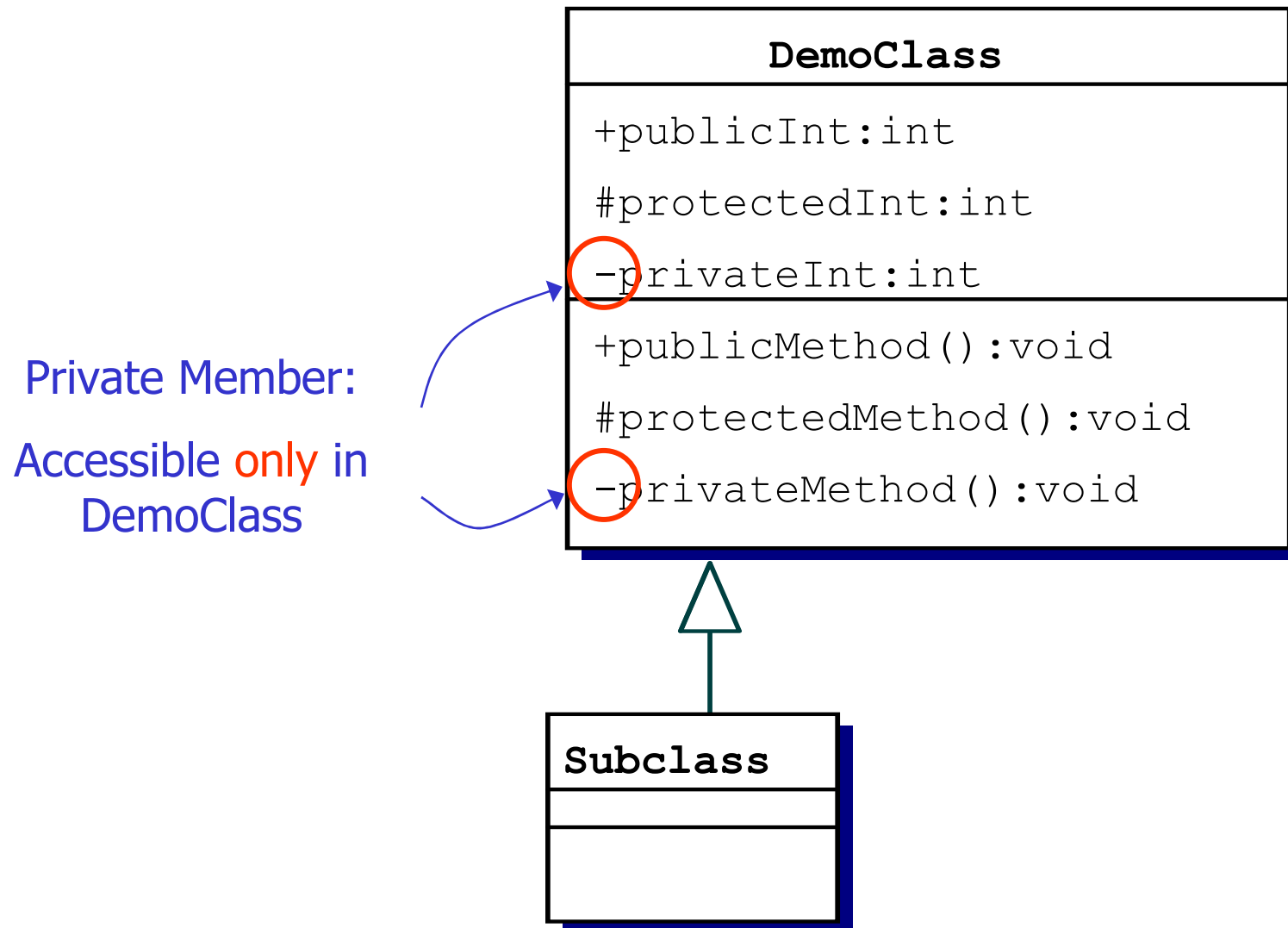
Limited Inheritance



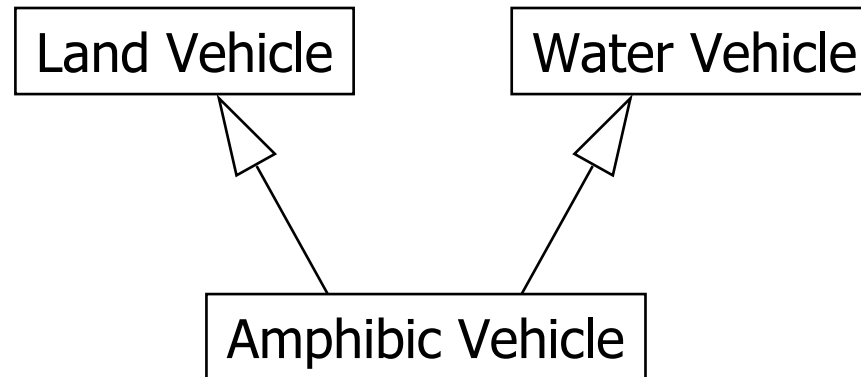
Limited Inheritance



Limited Inheritance



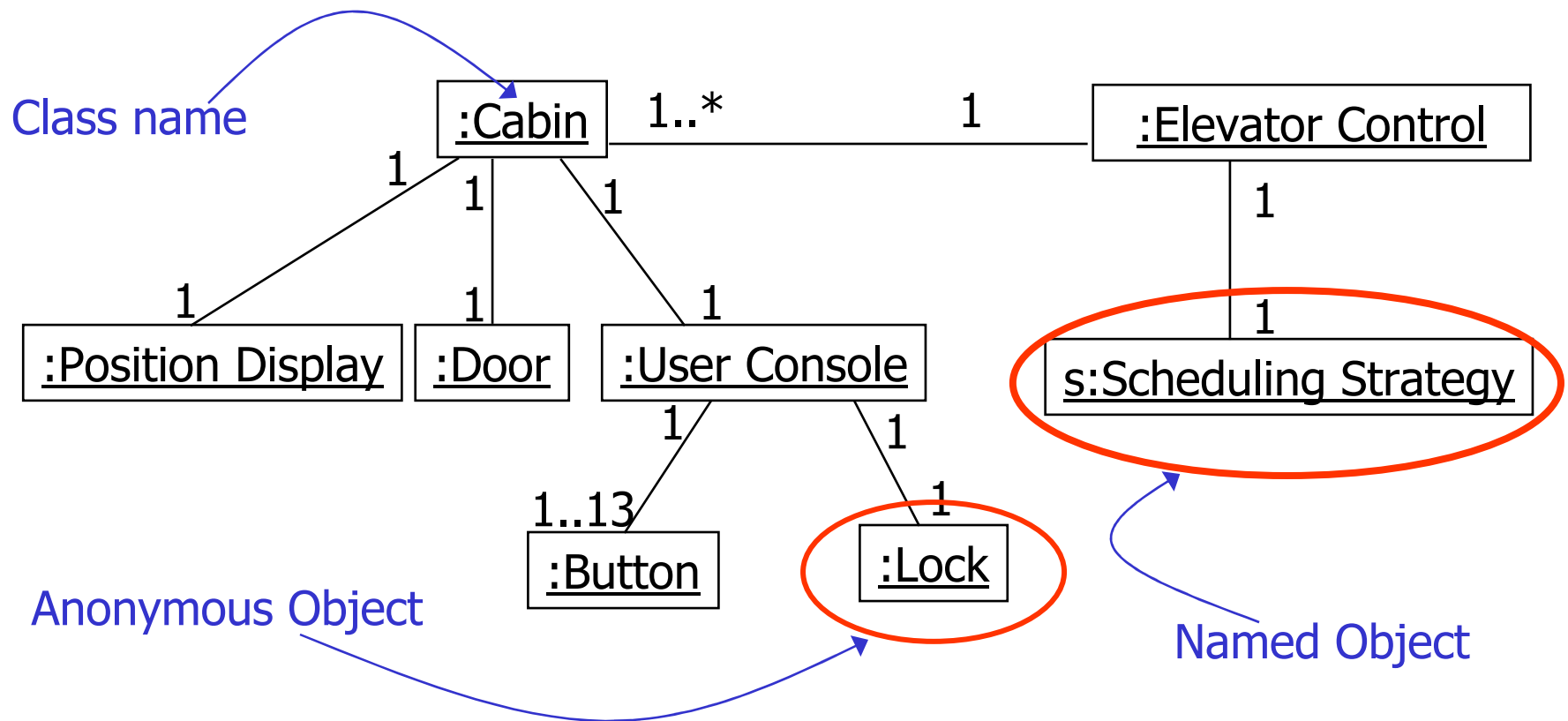
Multiple Inheritance



Attention: reserve for abstract classes/interfaces

Object Diagrams

- Syntax same as for class diagrams
- Typically no attributes and operations shown
- Object name and class name is underscored



Summary Class and Object Diagrams

- Class and object diagrams describe classes and objects, respectively, and their relationships:
 - intuitively: objects of class A “know” objects of class B (have references to objects of class B)
 - Multiplicities, Association names, Role names
 - Aggregation/Composition
 - Generalization/Specialization
 - Polymorphism
- Object diagrams: “snapshot” of running system
- View on system structure
- Direct transformation into implementation