Subtyping and Inheritance

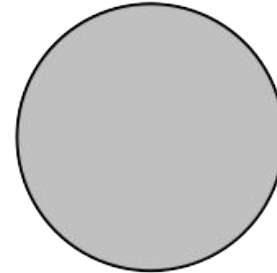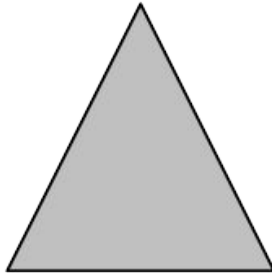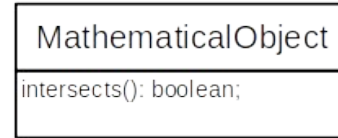Dirk Riehle, FAU Erlangen
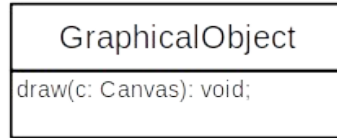
ADAP B03

Agenda

1. What is subtyping?
2. Liskov substitutability principle
3. Applied to class hierarchies
4. Co- and contravariance
5. Multiple inheritance
6. Abstract superclass rule
7. Cascading class hierarchies

Homework

2

1. What is Subtyping?

Subtyping Example 1 / 3

GraphicalObject
draw(c: Canvas): void;

MathematicalObject
intersects(): boolean;

4

Subtyping Example 2 / 3

GraphicalObject
draw(c: Canvas): void;

MathematicalObject
intersects(): boolean;

DR

5

Subtyping Example 3 / 3

GraphicalObject
draw(c: Canvas): void;
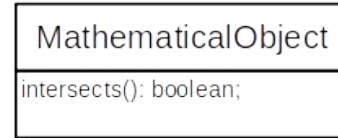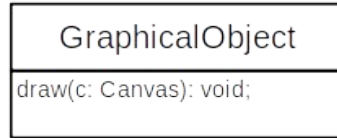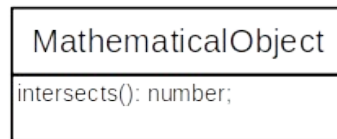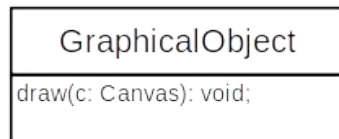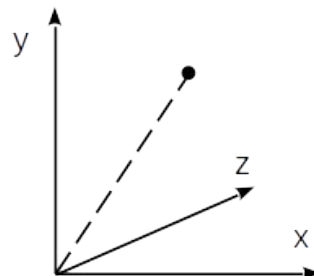
MathematicalObject
intersects(): number;

2. Liskov Substitutability Principle

The Subtype Requirement [1]

Let φ(x) be a property provable about objects x of type T. Then φ(y) should be provable for objects y of type S, where S is a subtype of T.

[1] A.k.a. the Liskov Substitutability Principle (LSP) [LW94]

In Simpler Words

All properties that hold for instances of a supertype should also hold for instances of a subtype [DR]

9

Even Simpler

Don't surprise use-clients

Quiz: What's the Surprise?

If you make Rectangle a subtype of Square?

If you make Square a subtype of Rectangle?

If you make 2DLine a subtype of Point?

If you make Point a subtype of 2DLine?

3. Applied to Class Hierarchies
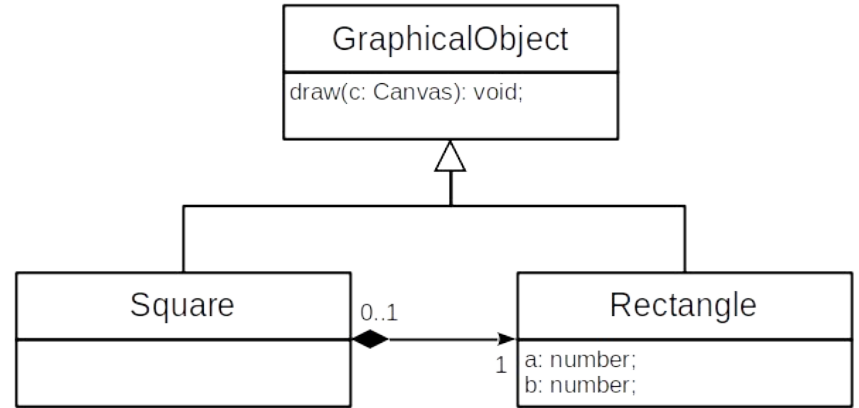
Subclasses as Extended Subtypes

Subclasses

- Add methods and state
- Do not constraint superclasses

DR

13

Subclasses as Constrained Subtypes

Subclasses

- Constrain behavior in defined space

In method signatures

- Using covariant redefinition

Leads to dual hierarchies

14
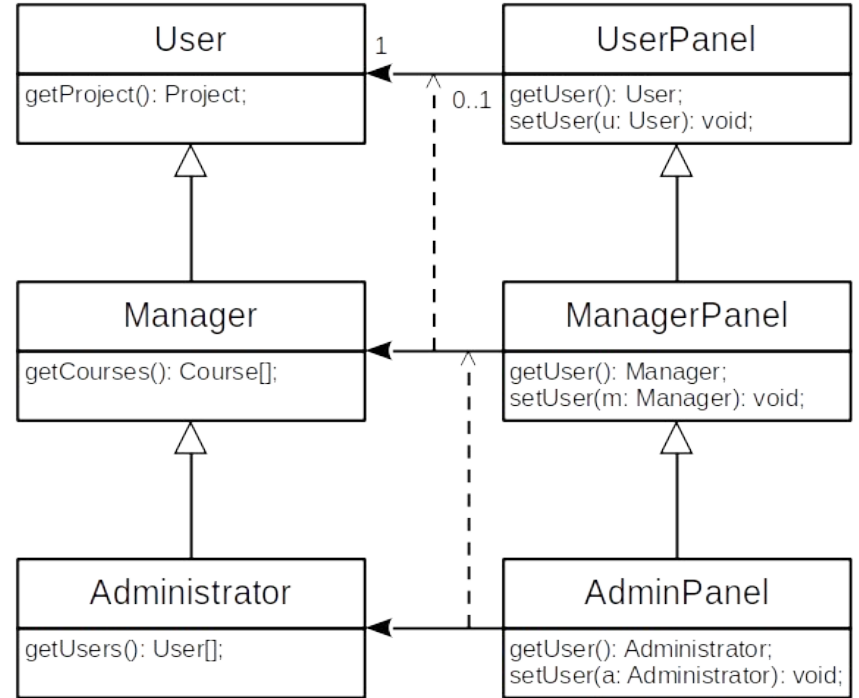
Extract Superclass Refactoring

A refactoring is a

- Behavior-preserving transformation of existing code

The goal is to improve readability, remove redundancy, etc.

The extraction of an abstract superclass is a common refactoring

Fowler's catalog [1] lists Extract Superclass (without "Abstract" though)

[1] Martin Fowler. Refactoring (2nd Edition). Addison Wesley, 2018.

4. Co- and Contravariance

Covariant Redefinition of Method Arguments

A method argument has been covariantly redefined, if

- The type of the argument in the subtype is a subtype of the argument in the supertype's method definition

Example of covariant redefinition of method argument

- User.setUser(u: User): void → Manager.setUser(m: Manager): void

The subtype's method "can only take less" than the supertype's method promises

- This violates the LSP and only makes sense if you think in relationships

17

Contravariant Redefinition of Method Arguments

The method of the subtype can take more than the supertype

- Satisfies the LSP but makes little sense in practice

Covariant Redefinition of Return Values

A return value has been covariantly redefined, if

- The type of the return value in the subtype is a subtype of the return value in the supertype's method definition

Example of covariant redefinition of return value

- UserPanel.getUser(): User → ManagerPanel.getUser(): Manager

The subtype's method "returns less" than the supertype's method promises

- Does not violate the LSP (is within expectations)

19

Contravariant Redefinition of Return Types

The method of the subtype can return more than the supertype

- ● Violates the LSP and makes little sense in practice

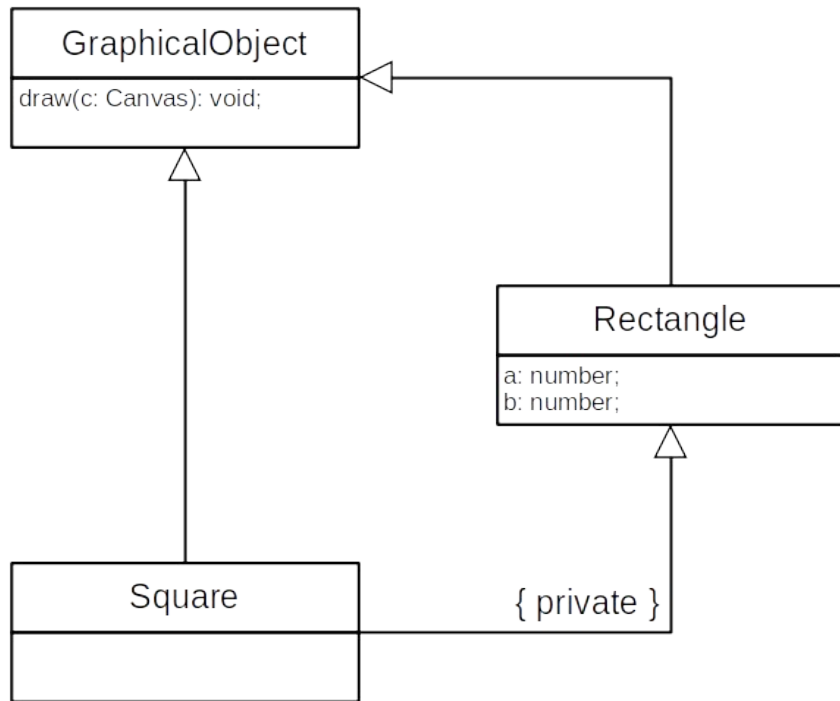# 5. Multiple Inheritance

Multiple Inheritance

Multiple inheritance is when

- A class has 2+ superclasses

Does not necessarily imply substitutability

- Cf. C++'s private inheritance

Not a Typescript feature
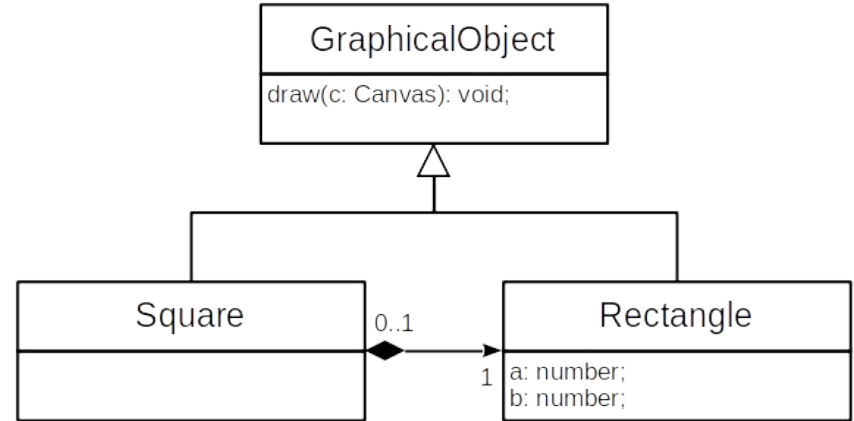

DR

22

Implementation Delegation

Implementation delegation is when

- A class delegates its implementation

Generally better than multiple inheritance

- Choose delegation over inheritance

Composition over Inheritance

The composition over inheritance principle states that

- You should favor object composition over class inheritance

A.k.a. delegation over inheritance (principle)

6. Abstract Superclass Rule

Inheritance vs. Abstractness

Inheritance is

- A relationship between two classes

Abstractness / concreteness

- A relationship between a class and its instances

Abstract Superclass Rule (ASR)

All superclasses must be abstract

Corollary: Never subclass a concrete class

ASR in Framework vs. Application

In a framework

- Leaf classes may be abstract (awaiting subclassing)
- Leaf classes may be concrete (if ready to use)

In an application (based on a framework)

- Framework leaf classes may be abstract if unused
- Application leaf classes must be concrete

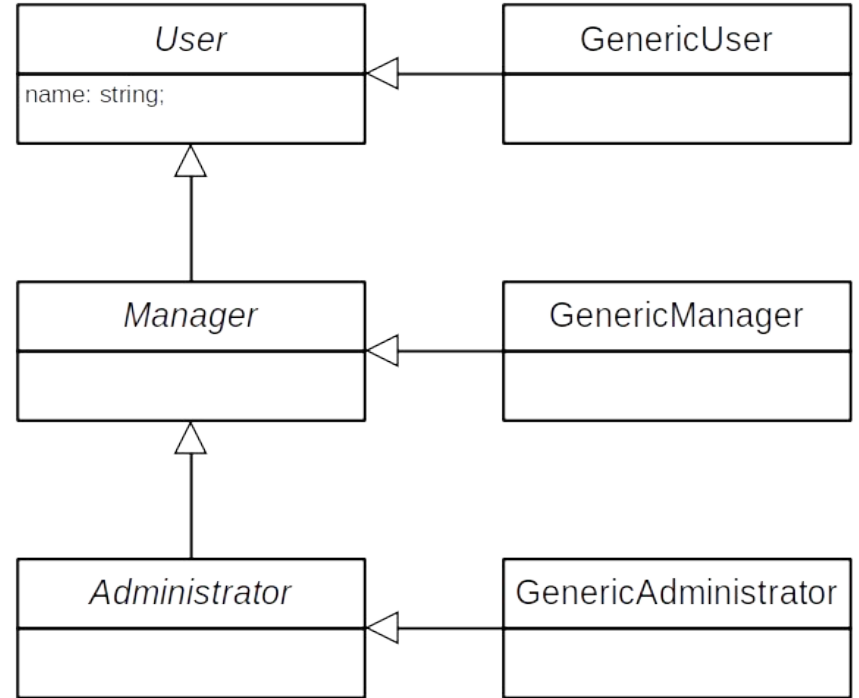# ASR and LSP

The ASR helps to comply with the LSP

- The ASR automatically casts subclasses as constrained subtypes
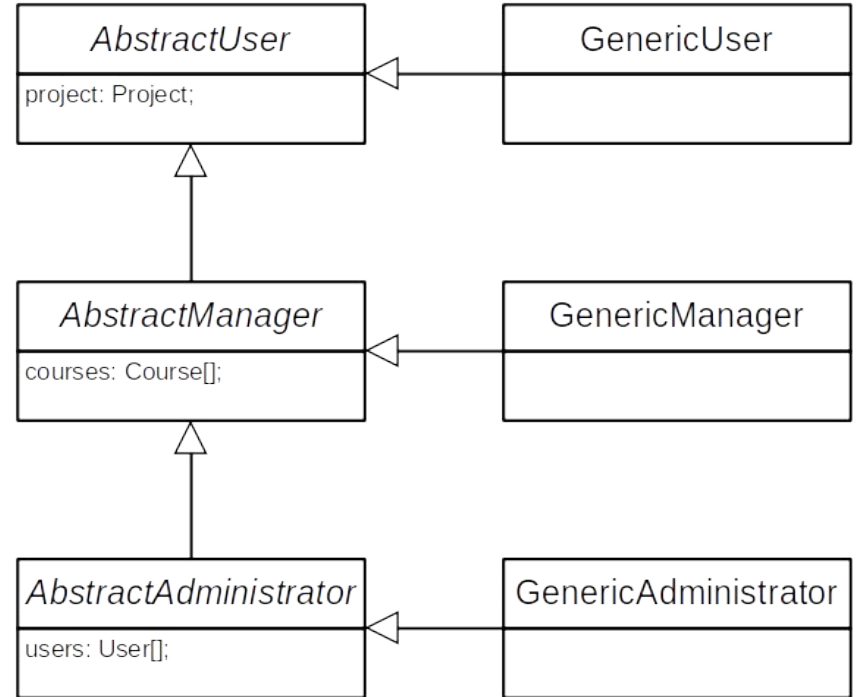
Pragmatics of ASR

Logically separate abstract class from generic implementation subclass

Pragmatically, merge implementation class into abstract class

Make abstract class concrete but maintain inheritance interface

Class Hierarchy Evolution
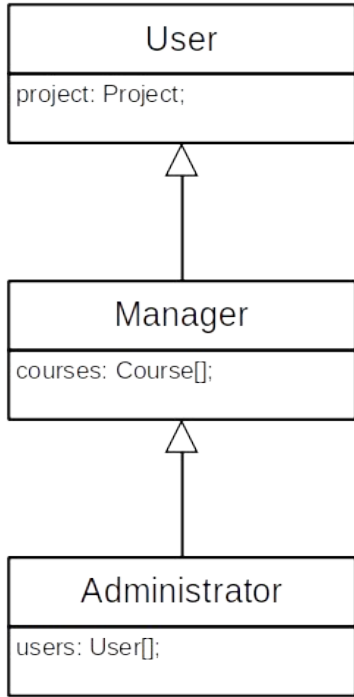
7. Cascading Class Hierarchies

Before and After Methods

Before and after methods wrap a method's main body

They typically come in pairs and are about a meta issue

- The before method sets something up
- The after method tears it down

App with Service Example
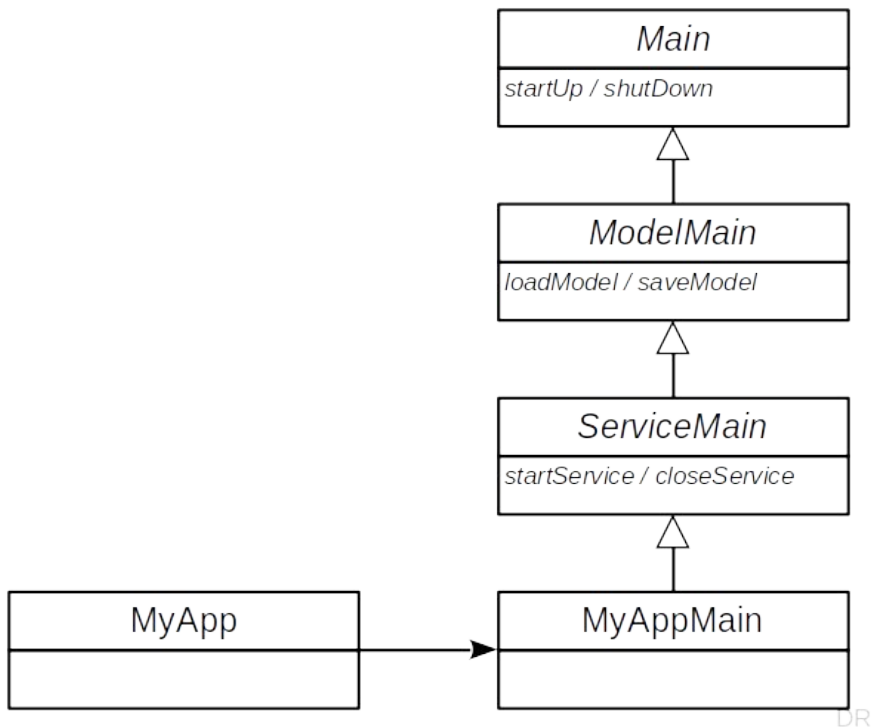

```typescript
import { MyAppMain } from "./MyAppMain";

function main(args: string[]) {
    let appMain: MyAppMain = new MyAppMain();
    appMain.run(args);
}

let args: string[] = process.argv;
args = args.slice(2);
main(args);
```

34

Cascading Inheritance Interfaces 1 / 2

```typescript
export abstract class Main {

    public run(args: string[]): void {
        this.parseArgs(args);
        this.startUp();
        this.execute();
        this.shutDown();
    };

    protected parseArgs(args: string[]): void {
        // do nothing (expect subclass to override)
    }

    protected startUp(): void {
        // do nothing (expect subclass to override)
    }

    protected abstract execute(): void;

    protected shutDown(): void {
        // do nothing (expect subclass to override)
    }

}
```

```typescript
import { Main } from "./Main";

export abstract class ModelMain extends Main {

    protected startUp(): void {
        super.startUp();
        this.loadModel();
    }

    protected loadModel(): void {
        // do nothing (expect subclass to override)
    }

    protected shutDown(): void {
        this.saveModel();
        super.shutDown();
    }

    protected saveModel(): void {
        // do nothing (expect subclass to override)
    }

}
```

35

Cascading Inheritance Interfaces 2 / 2

```typescript
import { ModelMain } from "./ModelMain";

export abstract class ServiceMain extends ModelMain {

    protected startUp(): void {
        super.startUp();
        this.startService();
    }

    protected startService(): void { /* ... */ }

    protected execute(): void {
        // start main event loop
    }

    protected shutDown(): void {
        this.closeService();
        super.shutDown();
    }

    protected closeService(): void { /* ... */ }

}
```

```typescript
import { ServiceMain } from "./ServiceMain";

export class MyAppMain extends ServiceMain {

    protected loadModel(): void {
        // do something
    }

    protected startService(): void {
        // do something
    }

    protected saveModel(): void {
        // do something
    }

    protected closeService(): void {
        // do something
    }

}
```

36

Homework

Homework Instructions

- Extract AbstractName superclass from StringName and StringArrayName
  - Identify and implement the narrow (minimal) inheritance interface
  - Move as much as you sensibly can into the AbstractName class

- Adapt your previous work to this homework as you see fit
- Commit homework by deadline to homework folder

Summary

1. What is subtyping?
2. Liskov substitutability principle
3. Applied to class hierarchies
4. Co- and contravariance
5. Multiple inheritance
6. Abstract superclass rule
7. Cascading class hierarchies

Thank you! Any questions?

dirk.riehle@fau.de – https://oss.cs.fau.de

dirk@riehle.org – https://dirkriehle.com – @dirkriehle

Legal Notices

License

- Licensed under the CC BY 4.0 International license

Copyright

- © 2012, 2018, 2024 Dirk Riehle, some rights reserved