CS 520

Theory and Practice of Software Engineering
Fall 2019

Object Oriented (OO) Design Principles

September 17, 2019

Today

- Code review and (re)design of an MVC application
- OO design principles
    - Information hiding (and encapsulation)
    - Polymorphism
    - Open/closed principle
    - Inheritance in Java
    - The diamond of death
    - Liskov substitution principle
    - Composition/aggregation over inheritance

Let's review the code of the following application

Source code available on the course web site

OO design principles

- **Information hiding (and encapsulation)**
- Polymorphism
- Open/closed principle
- Inheritance in Java
- The diamond of death
- Liskov substitution principle
- Composition/aggregation over inheritance

Information hiding

```
MyClass
+ nElem : int
+ capacity : int
+ top : int
+ elems : int[]
+ canResize : bool
+ resize(s:int):void
+ push(e:int):void
+ capacityLeft():int
+ getNumElem():int
+ pop():int
+ getElems():int[]
```

```
public class MyClass {
  public int nElem;
  public int capacity;
  public int top;
  public int[] elems;
  public boolean canResize;
  ...
  public void resize(int s){...}
  public void push(int e){...}
  public int capacityLeft(){...}
  public int getNumElem(){...}
  public int pop(){...}
  public int[] getElems(){...}
}
```

Information hiding

```
MyClass
+ nElem : int
+ capacity : int
+ top : int
+ elems : int[]
+ canResize : bool
+ resize(s:int):void
+ push(e:int):void
+ capacityLeft():int
+ getNumElem():int
+ pop():int
+ getElems():int[]
```

```
public class MyClass {
  public int nElem;
  public int capacity;
  public int top;
  public int[] elems;
  public boolean canResize;
  ...
  public void resize(int s){...}
  public void push(int e){...}
  public int capacityLeft(){...}
  public int getNumElem(){...}
  public int pop(){...}
  public int[] getElems(){...}
}
```

What does MyClass do?

Information hiding

```
Stack
+ nElem : int
+ capacity : int
+ top : int
+ elems : int[]
+ canResize : bool
+ resize(s:int):void
+ push(e:int):void
+ capacityLeft():int
+ getNumElem():int
+ pop():int
+ getElems():int[]
```

```
public class Stack {
   public int nElem;
   public int capacity;
   public int top;
   public int[] elems;
   public boolean canResize;
   ...
   public void resize(int s){...}
   public void push(int e){...}
   public int capacityLeft(){...}
   public int getNumElem(){...}
   public int pop(){...}
   public int[] getElems(){...}
}
```

Anything that could be improved in this implementation?

Information hiding

```
Stack
+ nElem : int
+ capacity : int
+ top : int
+ elems : int[]
+ canResize : bool
+ resize(s:int):void
+ push(e:int):void
+ capacityLeft():int
+ getNumElem():int
+ pop():int
+ getElems():int[]
```

```
Stack
- elems : int[]
...
+ push(e:int):void
+ pop():int
...
```

Information hiding:

- Reveal as little information about internals as possible.
- Separate public interface from implementation details.
- Reduce complexity.

Information hiding vs. visibility

Public
???
Private

Information hiding vs. visibility

Public
???
Private

- Protected, package-private, or friend-accessible (C++).
- Not part of the public API.
- Implementation detail that a subclass/friend may rely on.

OO design principles

- Information hiding (and encapsulation)
- **Polymorphism**
- Open/closed principle
- Inheritance in Java
- The diamond of death
- Liskov substitution principle
- Composition/aggregation over inheritance

A little refresher: what is Polymorphism?

A little refresher: what is Polymorphism?

An object's ability to provide different behaviors.

Types of polymorphism

- Ad-hoc polymorphism (e.g., operator overloading)
  - `a + b` ⇒ String vs. int, double, etc.
- Subtype polymorphism (e.g., method overriding)
  - `Object obj = ...;` `obj.toString();` ⇒ toString() can be overridden in subclasses and therefore provide a different behavior.
- Parametric polymorphism (e.g., Java generics)
  - ⇒ A LinkedList can store elements regardless of their type but still provide full type safety.

```
class LinkedList<E> {
   void add(E) {...}
   E get(int index) {...}
```

A little refresher: what is Polymorphism?

An object's ability to provide different behaviors.

Types of polymorphism

- Subtype polymorphism (e.g., method overriding)
  - `Object obj = ...;` `obj.toString();` ⇒ toString() can be overridden in subclasses and therefore provide a different behavior.

Subtype polymorphism is essential to many OO design principles.

OO design principles

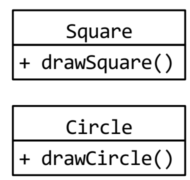
- Information hiding (and encapsulation)
- Polymorphism
- **Open/closed principle**
- Inheritance in Java
- The diamond of death
- Liskov substitution principle
- Composition/aggregation over inheritance

Open/closed principle

Software entities (classes, components, etc.) should be:

- **open** for extensions
- **closed** for modifications

```
public static void draw(Object o) {
  if (o instanceof Square) {
    drawSquare((Square) o)
  } else if (o instanceof Circle) {
    drawCircle((Circle) o);
  } else {
    ...
  }
}
```



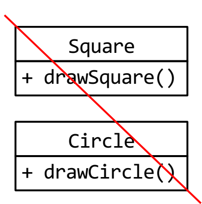
Good or bad design?

Open/closed principle

Software entities (classes, components, etc.) should be:

- **open** for extensions
- **closed** for modifications

```
public static void draw(Object o) {
  if (o instanceof Square) {
    drawSquare((Square) o)
  } else if (o instanceof Circle) {
    drawCircle((Circle) o);
  } else {
    ...
  }
}
```



Violates the open/closed principle!

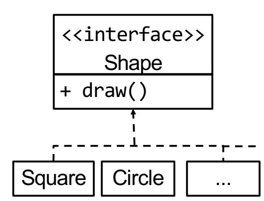
Open/closed principle

Software entities (classes, components, etc.) should be:

- **open** for extensions
- **closed** for modifications

```
public static void draw(Object s) {
  if (s instanceof Shape) {
    s.draw();
  } else {
    …
  }
}
```

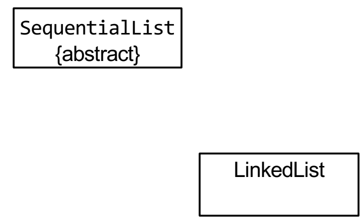
```
public static void draw(Shape s) {
  s.draw();
}
```



OO design principles

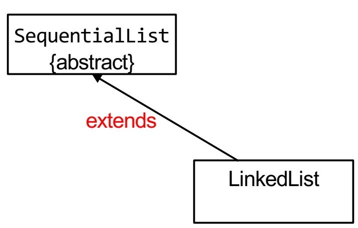
- Information hiding (and encapsulation)
- Polymorphism
- Open/closed principle
- **Inheritance in Java**
- The diamond of death
- Liskov substitution principle
- Composition/aggregation over inheritance

Inheritance: (abstract) classes and interfaces



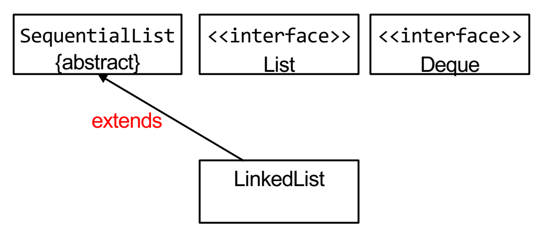
Inheritance: (abstract) classes and interfaces

LinkedList extends SequentialList



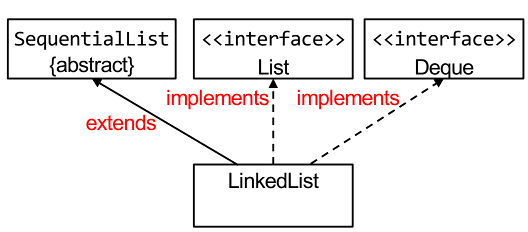
Inheritance: (abstract) classes and interfaces

LinkedList extends SequentialList

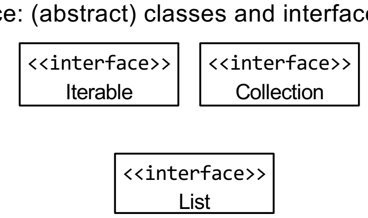


Inheritance: (abstract) classes and interfaces

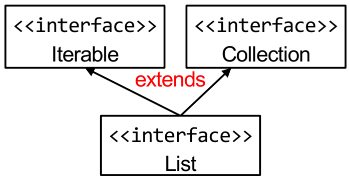
LinkedList extends SequentialList implements List, Deque



Inheritance: (abstract) classes and interfaces

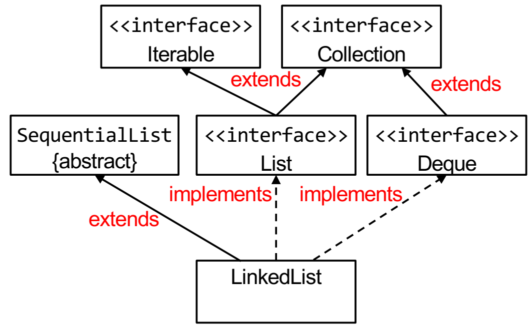


Inheritance: (abstract) classes and interfaces



List extends Iterable, Collection

Inheritance: (abstract) classes and interfaces

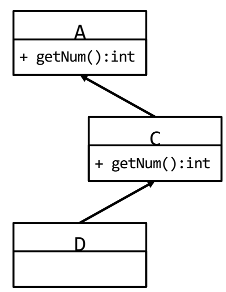


OO design principles

- Information hiding (and encapsulation)
- Polymorphism
- Open/closed principle
- Inheritance in Java
- **The diamond of death**
- Liskov substitution principle
- Composition/aggregation over inheritance

The “diamond of death”: the problem

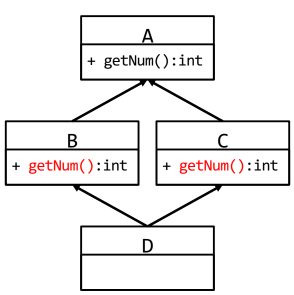
```
...
A a = new D();
int num = a.getNum();
...
```



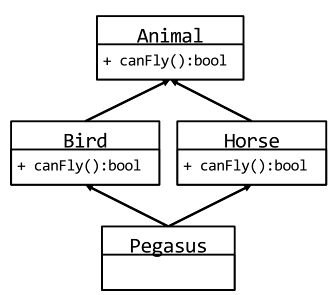
The “diamond of death”: the problem

```
...
A a = new D();
int num = a.getNum();
...
```

Which getNum() method should be called?



The “diamond of death”: concrete example



Can this happen in Java? Yes, with default methods in Java 8.

OO design principles

- Information hiding (and encapsulation)
- Polymorphism
- Open/closed principle
- Inheritance in Java
- The diamond of death
- **Liskov substitution principle**
- Composition/aggregation over inheritance

Design principles: Liskov substitution principle

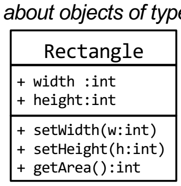
Motivating example

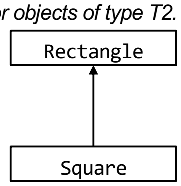
We know that a square is a special kind of a rectangle. So, which of the following OO designs makes sense?

Design principles: Liskov substitution principle

Subtype requirement

Let object x be of type T1 and object y be of type T2. Further, let T2 be a subtype of T1 (T2 <: T1). Any provable property about objects of type T1 should be true for objects of type T2.



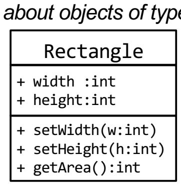


Is the subtype requirement fulfilled?

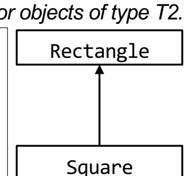
Design principles: Liskov substitution principle

Subtype requirement

Let object x be of type T1 and object y be of type T2. Further, let T2 be a subtype of T1 (T2 <: T1). Any provable property about objects of type T1 should be true for objects of type T2.



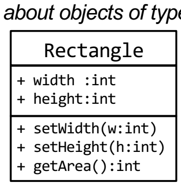

```
Rectangle r =
  new Rectangle(2,2);

int A = r.getArea();
int w = r.getWidth();
r.setWidth(w * 2);

assertEquals(A * 2,
        r.getArea());
```



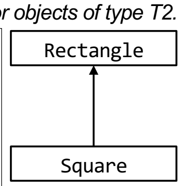
Design principles: Liskov substitution principle

Subtype requirement

Let object x be of type T1 and object y be of type T2. Further, let T2 be a subtype of T1 (T2 <: T1). Any provable property about objects of type T1 should be true for objects of type T2.



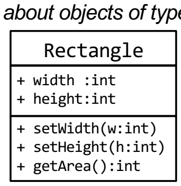

```
Rectangle r =
  new Rectangle(2,2);
  new Square(2);

int A = r.getArea();
int w = r.getWidth();
r.setWidth(w * 2);

assertEquals(A * 2,
        r.getArea());
```



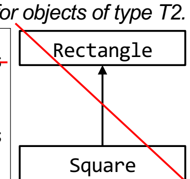
Design principles: Liskov substitution principle

Subtype requirement

Let object x be of type T1 and object y be of type T2. Further, let T2 be a subtype of T1 (T2 <: T1). Any provable property about objects of type T1 should be true for objects of type T2.




```
Rectangle r =
  new Rectangle(2,2);
  new Square(2);

int A = r.getArea();
int w = r.getWidth();
r.setWidth(w * 2);

assertEquals(A * 2,
        r.getArea());
```

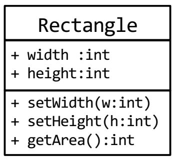


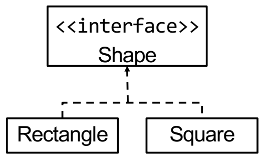
Violates the Liskov substitution principle!

Design principles: Liskov substitution principle

Subtype requirement

Let object x be of type T1 and object y be of type T2. Further, let T2 be a subtype of T1 (T2 <: T1). Any provable property about objects of type T1 should be true for objects of type T2.

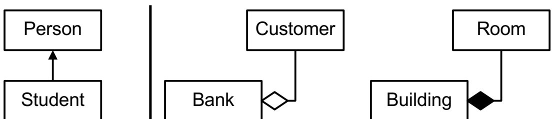




OO design principles

- Information hiding (and encapsulation)
- Polymorphism
- Open/closed principle
- Inheritance in Java
- The diamond of death
- Liskov substitution principle
- **Composition/aggregation over inheritance**

Inheritance vs. (Aggregation vs. *Composition*)




```
public class Student
    extends Person{
 public Student(){
 }

 ...
}
```

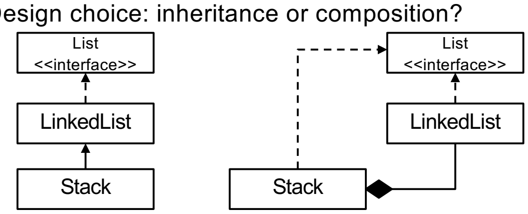
```
public class Bank {
 Customer c;
 public Bank(Customer c){
  this.c = c;
 }
 ...
}
```

```
public class Building {
 Room r;
 public Building(){
  this.r = new Room();
 }
 ...
}
```

is-a relationship

has-a relationship

Design choice: inheritance or composition?

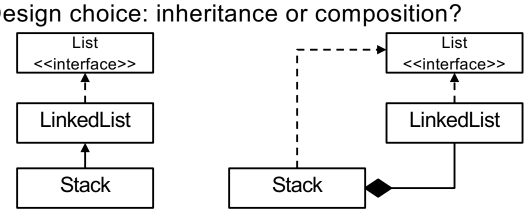



```
public class Stack<E>
     extends LinkedList<E> {
  ...
}
```

```
public class Stack<E> implements List<E> {
  private List<E> l = new LinkedList<>();
  ...
}
```

Hmm, both designs seem valid -- what are pros and cons?

Design choice: inheritance or composition?



Pros
- No delegation methods required.
- Reuse of common state and behavior.

Cons
- Exposure of all inherited methods (a client might rely on this particular superclass -> can't change it later).
- Changes in superclass are likely to break subclasses.

Pros
- Highly flexible and configurable: no additional subclasses required for different compositions.

Cons
- All interface methods need to be implemented -> delegation methods required, even for code reuse.

Composition/aggregation over inheritance allows more flexibility.

OO design principles: summary

- Information hiding (and encapsulation)
- Open/closed principle
- Liskov substitution principle
- Composition/aggregation over inheritance