Recap: types of Polymorphism

- Ad-hoc polymorphism (e.g., operator overloading)
  - \(a + b\) \(\Rightarrow\) String vs. int, double, etc.

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

- Parametric polymorphism (e.g., Java generics)
  - class LinkedList\(\langle E \rangle\) {
    - void add\(\langle E \rangle\) (...) \(\Rightarrow\) A LinkedList can store elements regardless of their type but still
    - \(E\) get\(\langle int index\rangle\) (...) provide full type safety.

Recap: inheritance of classes and interfaces

SequentialList
{abstract}

List

Deque

LinkedList
Recap: inheritance of classes and interfaces

**LinkedList** extends **SequentialList** implements **List**, **Deque**

Today

**Software design principles**
- The diamond of death
- Composition/aggregation over inheritance
- Information hiding (and encapsulation)
- Open/closed principle
- Liskov substitution principle
The “diamond of death”

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

### What version of `getNum()` to call?

The “diamond of death”

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

### Can you think of a particular method in Java for which this problem could arise (if Java would allow multiple inheritance)?

Classes, abstract classes, and interfaces

```
public class MyClass {
  public void op() {
    ...
  }

  public int op2() {
    ...
  }
}
```

Recall how default methods (Java 8) fit into this spectrum?

```
public class MyClass {
  public void op() {
    ...
  }

  public int op2() {
    ...
  }
}
```

```
public abstract class MyAbstractClass {
  public abstract void op();
  public int op2() {
    ...
  }
}
```

```
public interface MyInterface {
  public void op();
  public int op2();
}
```

```
public class MyClass {
  public void op() {
    ...
  }

  public int op2() {
    ...
  }
}
```

```
public abstract class MyAbstractClass {
  public abstract void op();
  public int op2() {
    ...
  }
}
```

```
public interface MyInterface {
  public void op();
  public int op2();
}
```
So, how does an interface with default methods differ from an abstract class?

Why does Java 8 allow default methods?
- Compile all files in cs320 → what do you observe?
- Fix the compilation issue.

Pretend 10 years have passed...
- Add a new method to the List interface (List.java):
  - public int getNumElems()
- Compile all files in cs320 → what do you observe?

Make getNumElems a default method in List.java
- Compile all files in cs320 → what do you observe?
Default methods in Java can cause the diamond of death!

- Remove the `canFly()` method in `Pegasus.java`
- Compile the code → what do you observe?

```
public class Pegasus implements Horse, Bird {
    public boolean canFly() {
        return true;
    }
}
```

How can you resolve the conflict without hard-coding the return value in the `canFly` method?
Today

Software design principles

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

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.

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

Composition/aggregation over inheritance allows more flexibility.
Today

Software design principles
● The diamond of death
● Composition/aggregation over inheritance
● Information hiding (and encapsulation)
● Open/closed principle
● Liskov substitution principle

Information hiding

<table>
<thead>
<tr>
<th></th>
<th>MyClass</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ nElem : int</td>
<td></td>
</tr>
<tr>
<td>+ capacity : int</td>
<td></td>
</tr>
<tr>
<td>+ top : int</td>
<td></td>
</tr>
<tr>
<td>+ elems : int[]</td>
<td></td>
</tr>
<tr>
<td>+ canResize : bool</td>
<td></td>
</tr>
</tbody>
</table>

+ 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[]

Anything that could be improved in this implementation?

Information hiding vs. visibility

Public
- ???

Private

Information hiding vs. visibility

Public
- ???

Private

Information hiding:
- Reveal as little information about internals as possible.
- Segregate public interface and implementation details.
- Reduces complexity.

Information hiding vs. visibility

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

**Software design principles**
- The diamond of death
- Composition/aggregation over inheritance
- Information hiding (and encapsulation)
- **Open/closed principle**
- Liskov substitution principle

Design principles: open/closed principle

**Software entities** (classes, components, etc.) should be:
- **open** for extensions
- **closed** for modifications

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

Good or bad design?

Square  
+ `drawSquare()`

Circle  
+ `drawCircle()`

Violates the open/closed principle!

Design principles: open/closed principle

**Software entities** (classes, components, etc.) should be:
- **open** for extensions
- **closed** for modifications

```java
public static void draw(Figure f) {
  f.draw();
}
```

```
public static void draw(Object f) {
  if (f instanceof Figure) {
    f.draw();
  } else {
    ...
  }
}
```

<<interface>>
Shape  
+ `draw()`

Violation of open/closed principle!
Today

Software design principles

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

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?*

- Rectangle
- Square

Subtype requirement

Let object $x$ be of type $T_1$ and object $y$ be of type $T_2$. Further, let $T_2$ be a subtype of $T_1$ ($T_2 <: T_1$). Any provable property about objects of type $T_1$ should be true for objects of type $T_2$.

```java
Rectangle r = new Rectangle(2,2);
int A = r.getArea();
int w = r.getWidth();
r.setWidth(w * 2);
assertEquals(A * 2, r.getArea());
```

Is the subtype requirement fulfilled?

```java
Rectangle r = new Rectangle(2,2);
int A = r.getArea();
int w = r.getWidth();
rsetWidth(w * 2);
assertEquals(A * 2, r.getArea());
```
Design principles: Liskov substitution principle

Subtype requirement

Let object $x$ be of type $T_1$ and object $y$ be of type $T_2$. Further, let $T_2$ be a subtype of $T_1$ ($T_2 <: T_1$). Any provable property about objects of type $T_1$ should be true for objects of type $T_2$.

Rectangle
+ width :int
+ height:int
+ setWidth(w:int)
+ setHeight(h:int)
+ getArea():int

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!

Rectangle
+ width :int
+ height:int
+ setWidth(w:int)
+ setHeight(h:int)
+ getArea():int

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());

Summary

Software design principles
- The diamond of death
- Composition/aggregation over inheritance
- Information hiding (and encapsulation)
- Open/closed principle
- Liskov substitution principle