Recap: types of Polymorphism
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 and therefore provide a different behavior.

- **Parametric polymorphism (e.g., Java generics)**
  - class LinkedList$<E>$ {
    void add(E) {...}
    E get(int index) {...}
  } \Rightarrow A LinkedList can store elements regardless of their type but still 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
Recap: inheritance of classes and interfaces

SequentialList {abstract}

LinkedList

Iterable

Collection

List

Deque

implements

extends

implements

extends
Today

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

<table>
<thead>
<tr>
<th>MyClass</th>
<th>MyAbstractClass</th>
<th>MyInterface</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>public class</td>
<td>public class</td>
</tr>
<tr>
<td></td>
<td>MyClass</td>
<td>MyAbstractClass</td>
</tr>
<tr>
<td>meclss</td>
<td></td>
<td>{abstract}</td>
</tr>
<tr>
<td>public void</td>
<td>public abstract</td>
<td>public void</td>
</tr>
<tr>
<td>op() {</td>
<td>class void op();</td>
<td>void op();</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>public int</td>
<td>public int</td>
<td></td>
</tr>
<tr>
<td>op2() {</td>
<td>op2() {</td>
<td>}</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>}</td>
</tr>
</tbody>
</table>

Recall how default methods (Java 8) fit into this spectrum?
Classes, abstract classes, and interfaces

MyClass

public class MyClass {
    public void op() {
        ...
    }
    public int op2() {
        ...
    }
}

MyAbstractClass

{abstract}

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

MyInterface

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

So, how does an interface with default methods differ from an abstract class?
Coding example: cs320/ArrayList.java (v1,v2)

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

Source code is available on the course web site.
Coding example: cs320/ArrayList.java (v3)

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?

Source code is available on the course web site.
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
  - `public default int getNumElems() {...}`
- Compile all files in cs320 → what do you observe?

Source code is available on the course web site.
Coding example: inheritance/Pegasus.java

Default methods in Java can cause the diamond of death!

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

Source code is available on the course web site.
Coding example: inheritance/Pegasus.java

```
Animal
  + canFly():bool

Bird
  + canFly():bool

Horse
  + canFly():bool

Pegasus
```
Coding example: inheritance/Pegasus.java

Default methods in Java can cause the diamond of death!
- Remove the `canFly()` method in `Pegasus.java`
- Compile the code → what do you observe?

```java
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?
Coding example: inheritance/Pegasus.java

Default methods in Java can cause the diamond of death!

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

```java
public class Pegasus implements Horse, Bird {
    public boolean canFly() {
        return Bird.super.canFly();
    }
}
```
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?

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

```java
public class Stack<E> extends LinkedList<E> {
    private List<E> l = new LinkedList<>();
    ...
}
```
Design choice: inheritance or composition?

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 has-a relationships.

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

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>MyClass</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ nElem : int</td>
</tr>
<tr>
<td>+ capacity : int</td>
</tr>
<tr>
<td>+ top : int</td>
</tr>
<tr>
<td>+ elems : int[]</td>
</tr>
<tr>
<td>+ canResize : bool</td>
</tr>
<tr>
<td>+ resize(s:int):void</td>
</tr>
<tr>
<td>+ push(e:int):void</td>
</tr>
<tr>
<td>+ capacityLeft():int</td>
</tr>
<tr>
<td>+ getNumElem():int</td>
</tr>
<tr>
<td>+ pop():int</td>
</tr>
<tr>
<td>+ getElems():int[]</td>
</tr>
</tbody>
</table>
## Information hiding

<table>
<thead>
<tr>
<th>MyClass</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ nElem : int</td>
</tr>
<tr>
<td>+ capacity : int</td>
</tr>
<tr>
<td>+ top : int</td>
</tr>
<tr>
<td>+ elems : int[]</td>
</tr>
<tr>
<td>+ canResize : bool</td>
</tr>
<tr>
<td>+ resize(s:int):void</td>
</tr>
<tr>
<td>+ push(e:int):void</td>
</tr>
<tr>
<td>+ capacityLeft():int</td>
</tr>
<tr>
<td>+ getNumElem():int</td>
</tr>
<tr>
<td>+ pop():int</td>
</tr>
<tr>
<td>+ getElems():int[]</td>
</tr>
</tbody>
</table>

```java
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

<table>
<thead>
<tr>
<th>MyClass</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ nElem : int</td>
</tr>
<tr>
<td>+ capacity : int</td>
</tr>
<tr>
<td>+ top : int</td>
</tr>
<tr>
<td>+ elems : int[]</td>
</tr>
<tr>
<td>+ canResize : bool</td>
</tr>
<tr>
<td>+ resize(s:int):void</td>
</tr>
<tr>
<td>+ push(e:int):void</td>
</tr>
<tr>
<td>+ capacityLeft():int</td>
</tr>
<tr>
<td>+ getNumElem():int</td>
</tr>
<tr>
<td>+ pop():int</td>
</tr>
<tr>
<td>+ getElems():int[]</td>
</tr>
</tbody>
</table>

**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

<table>
<thead>
<tr>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ nElem : int</td>
</tr>
<tr>
<td>+ capacity : int</td>
</tr>
<tr>
<td>+ top : int</td>
</tr>
<tr>
<td>+ elems : int[]</td>
</tr>
<tr>
<td>+ canResize : bool</td>
</tr>
<tr>
<td>+ resize(s:int):void</td>
</tr>
<tr>
<td>+ push(e:int):void</td>
</tr>
<tr>
<td>+ capacityLeft():int</td>
</tr>
<tr>
<td>+ getNumElem():int</td>
</tr>
<tr>
<td>+ pop():int</td>
</tr>
<tr>
<td>+ getElems():int[]</td>
</tr>
</tbody>
</table>

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:
- Reveal as little information about internals as possible.
- Segregate public interface and implementation details.
- Reduces 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.
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 {
        ...
    }
}
```

Square
- + drawSquare()

Circle
- + drawCircle()

Good or bad design?
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 {
        ...
    }
}
```

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

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

![UML diagram showing inheritance and method draw]
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
  ▲
   Rectangle
```

```
Square
  ▲
   Square
```
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 $T_2$.

<table>
<thead>
<tr>
<th>Rectangle</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ width :int</td>
</tr>
<tr>
<td>+ height :int</td>
</tr>
<tr>
<td>+ setWidth(w:int)</td>
</tr>
<tr>
<td>+ setHeight(h:int)</td>
</tr>
<tr>
<td>+ getArea():int</td>
</tr>
</tbody>
</table>

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 $T2$.

```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

<table>
<thead>
<tr>
<th>Rectangle</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ width : int</td>
</tr>
<tr>
<td>+ height : int</td>
</tr>
<tr>
<td>+ setWidth(w:int)</td>
</tr>
<tr>
<td>+ setHeight(h:int)</td>
</tr>
<tr>
<td>+ getArea():int</td>
</tr>
</tbody>
</table>

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 $T_1$ 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 $T_2$.

---

<table>
<thead>
<tr>
<th>Rectangle</th>
<th>Rectangle $r =$</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ width : int</td>
<td></td>
</tr>
<tr>
<td>+ height : int</td>
<td></td>
</tr>
<tr>
<td>+ setWidth(w : int)</td>
<td></td>
</tr>
<tr>
<td>+ setHeight(h : int)</td>
<td></td>
</tr>
<tr>
<td>+ getArea() : int</td>
<td></td>
</tr>
</tbody>
</table>

```java
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 $T_1$ 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 $T_2$.

<table>
<thead>
<tr>
<th>Rectangle</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ width :int</td>
</tr>
<tr>
<td>+ height:int</td>
</tr>
<tr>
<td>+ setWidth(w:int)</td>
</tr>
<tr>
<td>+ setHeight(h:int)</td>
</tr>
<tr>
<td>+ getArea():int</td>
</tr>
</tbody>
</table>

Diagram:
- Shape
  - Rectangle
  - Square
Summary

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