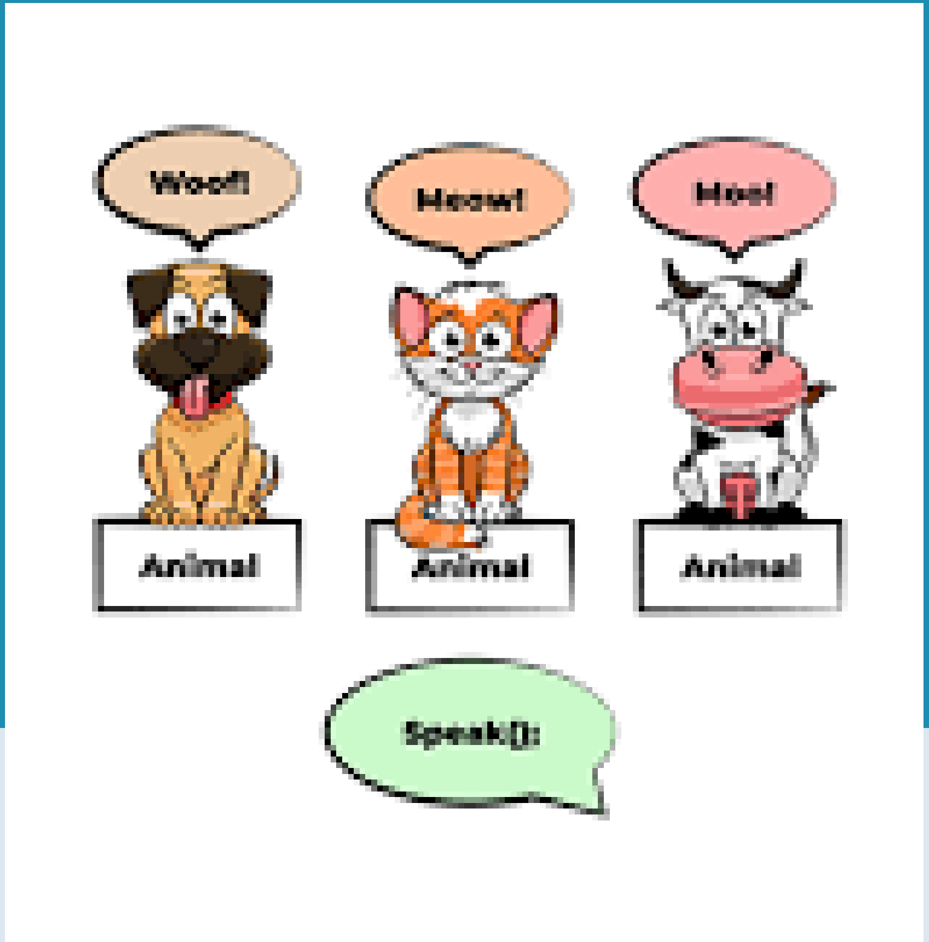
Programmeertechnieken/Programming Techniques

Part 1
Polymorphism



Koen Pelsmaekers

Campus Groep T, 2022-2023

Content of this part

- Polymorphism
    - Inheritance (the sequel)
        - Liskov’s substitution principle (LSP)
        - Abstract classes
    - Interface

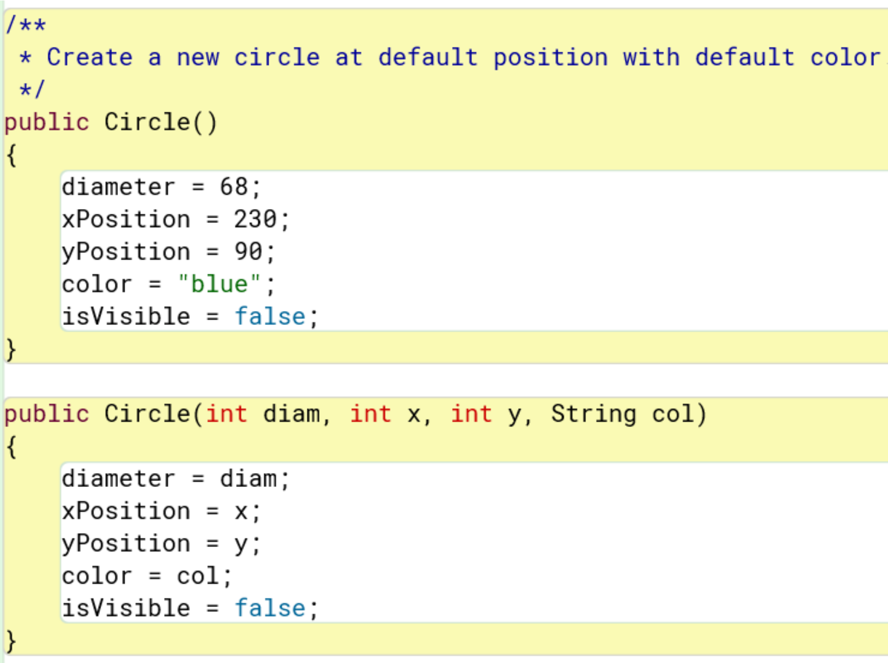
Polymorphism

- Many definitions of polymorphism in software engineering
  - Greek: poly (= many) + morphs (= forms) (biology)
- In object-oriented programming languages: “an object/variable can have many forms”
- Examples:
  - Polymorphic assignment
    - Liskov’s substitution principle (LSP) or substitutability
  - Polymorphic binding (also known as late binding or run-time binding)
- Programming constructs to support polymporphism
  - Method overloading (?)
  - Inheritance
  - Interface

Method overloading

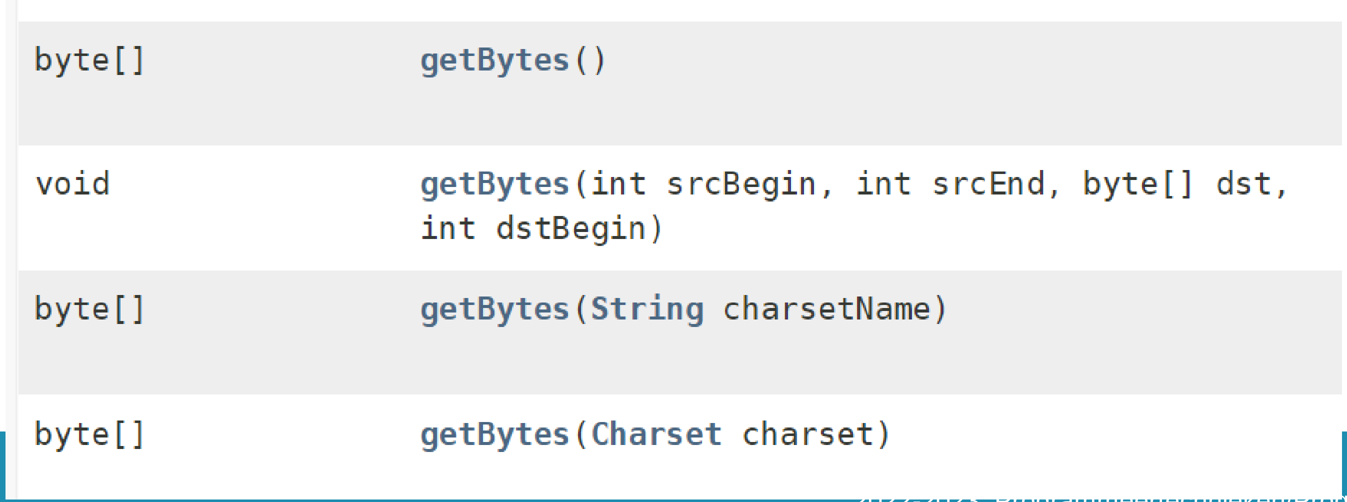
Method overloading

- Methods with same name, but different signature
  - “method polymorphism”
  - see 1st semester course examples
    - Circle: overloaded constructors
  - Java API
    - String: overloaded method “getBytes”



```
/**
 * Create a new circle at default position with default color
 */
public Circle()
{
    diameter = 68;
    xPosition = 230;
    yPosition = 90;
    color = "blue";
    isVisible = false;
}

public Circle(int diam, int x, int y, String col)
{
    diameter = diam;
    xPosition = x;
    yPosition = y;
    color = col;
    isVisible = false;
}
```



byte[]	getBytes()
void	getBytes(int srcBegin, int srcEnd, byte[] dst, int dstBegin)
byte[]	getBytes(String charsetName)
byte[]	getBytes(Charset charset)

Inheritance

Object-oriented programming

- Pre-requisites for an OOP language:
  - Classes and objects
  - Association/Aggregation/Composition
  - **Inheritance and polymorphism**

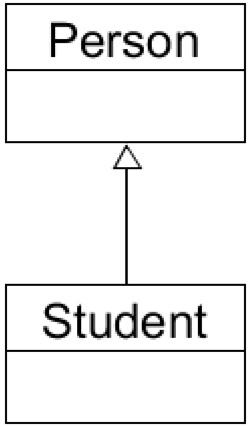
Inheritance

“is-a”

“is a more specialized/specific thing”

Inheritance

- Generalization/Specialization
  - superclass (more generic) vs. subclass (more specific)
  - shared properties and behaviour
  - substitutability (Barbara Liskov)



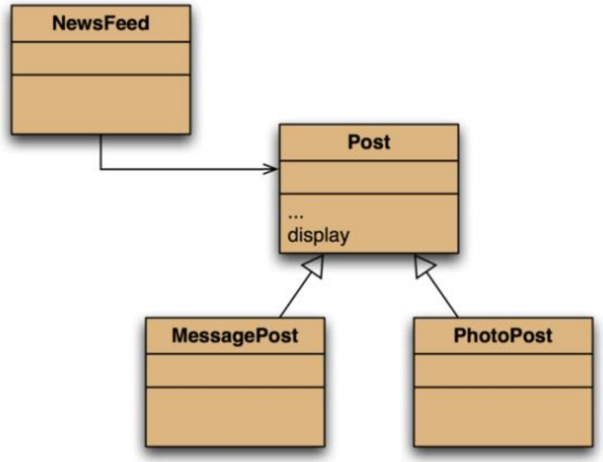
```
public class Person {
   //...
}
public class Student extends Person {
   //...
}

Person p = new Person();
p = new Student(); // substitutability
```

Examples

- Base class/Super class/Parent class vs. Derived class/Sub class/Child class

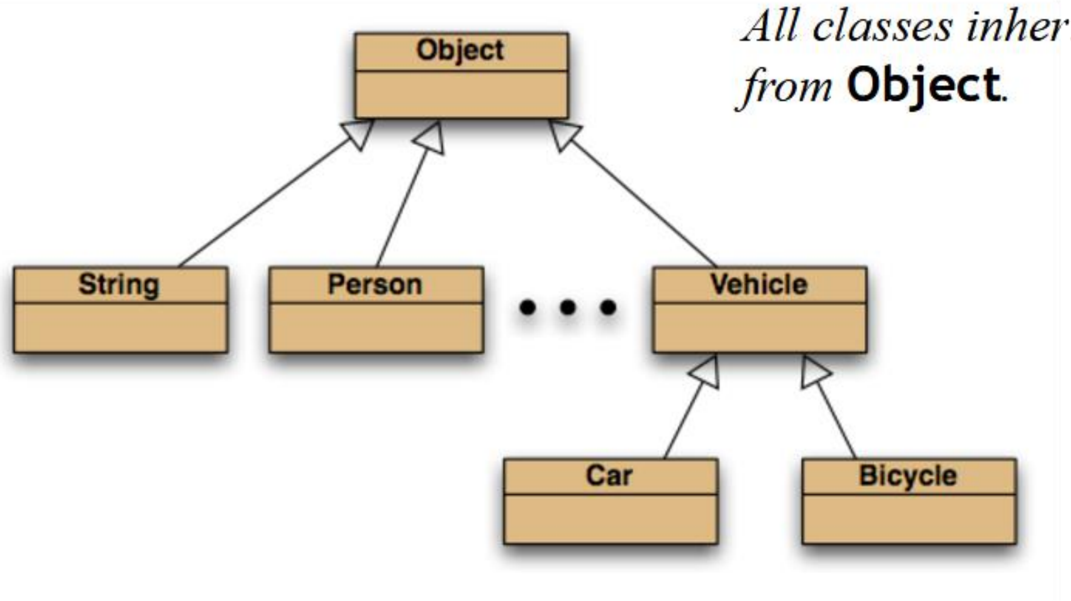
Base class (more general)	**Derived class (more specific)**
Post	MessagePost, PhotoPost, …
Shape	Circle, Triangle, Rectangle, …
Bike	MTB, Racebike, City bike, E-bike, …
…	…



Diagrams taken from: Objects First with Java, 6th Edition

Inheritance in Java: superclass Object

- Single rooted hierarchy: Object is (implicit) superclass of all classes
- Single inheritance (<-> C++ multiple inheritance): an object can inherit from only one superclass



All classes inherit from **Object**.

Diagram taken from: Objects First with Java, 6th Edition

Post, MessagePost & PhotoPost

```
public class Post
{
   private String username;
   private long timestamp;
   private int likes;
   private ArrayList<String> comments;


  …
}
```

final class

no extend

```
public class MessagePost extends Post
{
   private String message;
  …
}
```

```
public class PhotoPost extends Post
{
   private String filename;
   private String caption;
  …
}
```

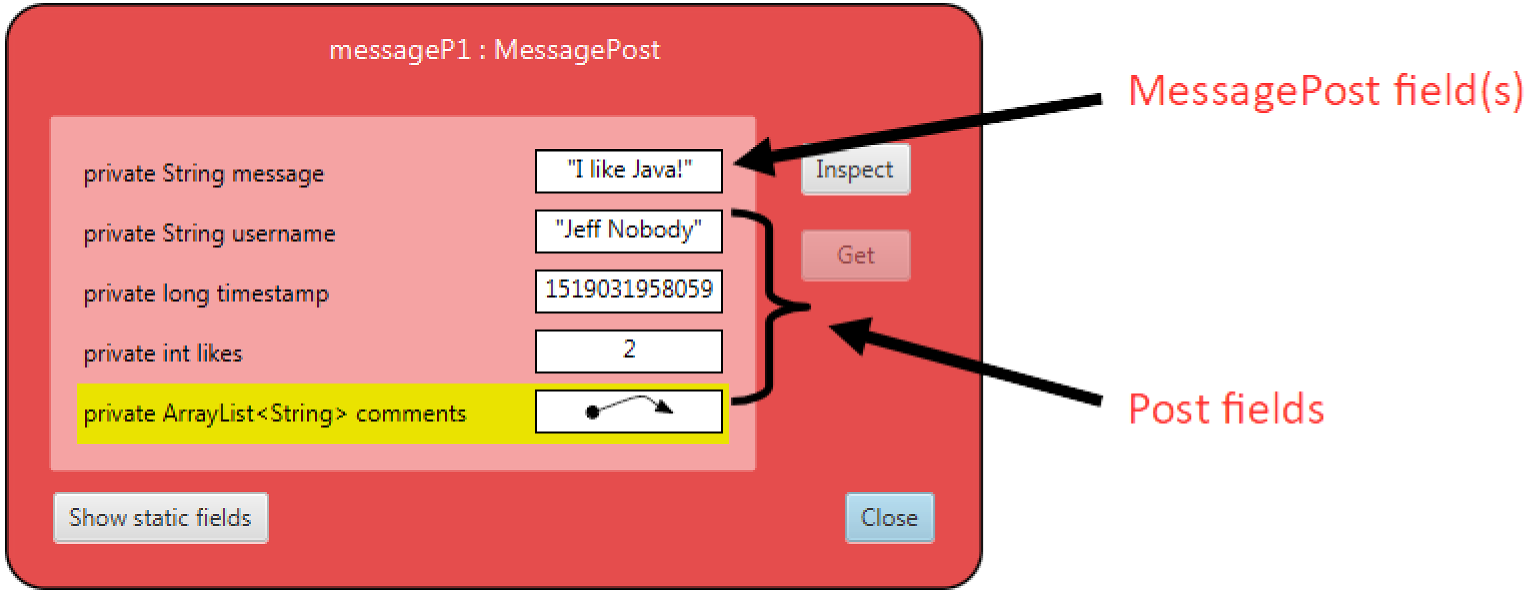
Taken from: Objects First with Java, 6th Edition

Subclass object has superclass fields/methods

the worst thing

copy paste

- MessagePost has Post object (Post fields)
  - subclass constructor calls (implicitly?) the superclass constructor



messageP1 : MessagePost

private String message	"I like Java!"
private String username	"Jeff Nobody"
private long timestamp	1519031958059
private int likes	2
private ArrayList<String> comments	

MessagePost field(s)

Post fields

Diagram taken from: Objects First with Java, 6th Edition

Inheritance and constructors

```
public class Post
{
   private String username;
   private long timestamp;
   private int likes;
   private ArrayList<String> comments;

   public Post(String author)
   {
      username = author;
      timestamp = System.currentTimeMillis();
      likes = 0;
      comments = new ArrayList<>();
   }

  …
}
```

```
public class MessagePost extends Post
{
   private String message;

   public MessagePost(String author, String text)
   {
      super(author);
      message = text;
   }

  …
}
```

Superclass constructor call must be first statement in subclass constructor (Java)

Constructors & inheritance in Java

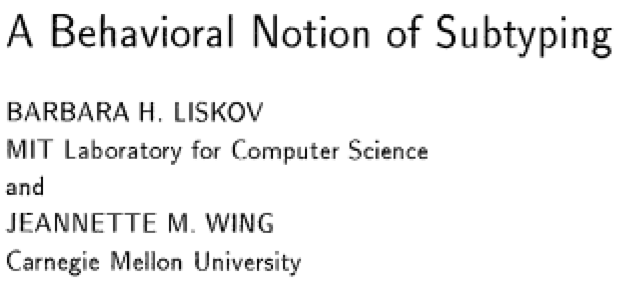
- Implicit constructor (= no-args constructor)
  - in superclass
  - in subclass

    + implicit super() constructor call
- User-defined constructor exists => no implicit constructor
  - subclass constructor needed?
- Call to superclass constructor from subclass constructor must be first statement in this subclass constructor

Inheritance: advantages

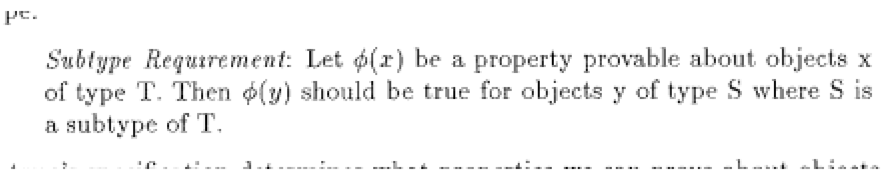
- Polymorphism: substitutability and dynamic method binding
  - See next slide
- Avoid code duplication
  - pull common fields and code up to the superclass
- Code reuse
  - reuse super class code when new subclass is added
- Easier maintenance
- Extendibility
  - easy to add new subclasses, for instance other post types

Barbara Liskov’s Subtitution Principle (LSP)



A Behavioral Notion of Subtyping

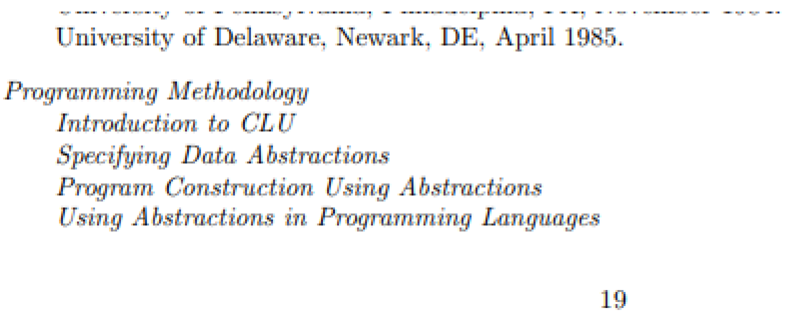
BARBARA H. LISKOV
MIT Laboratory for Computer Science
and
JEANNETTE M. WING
Carnegie Mellon University



pe.

Subtype Requirement: Let $\phi(x)$ be a property provable about objects x of type T. Then $\phi(y)$ should be true for objects y of type S where S is a subtype of T.

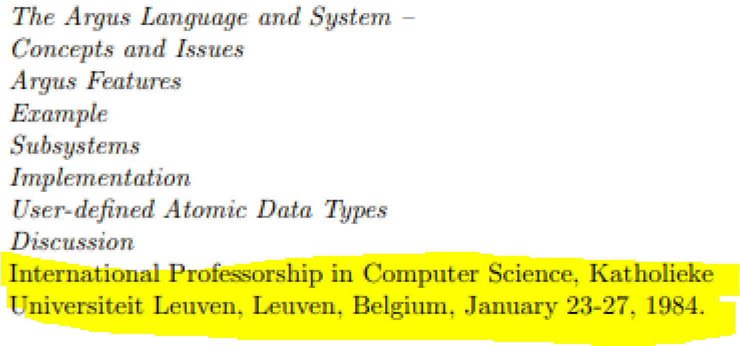
A type's specification determines what properties we can prove about objects



University of Delaware, Newark, DE, April 1985.

Programming Methodology
Introduction to CLU
Specifying Data Abstractions
Program Construction Using Abstractions
Using Abstractions in Programming Languages

19



The Argus Language and System –
Concepts and Issues
Argus Features
Example
Subsystems
Implementation
User-defined Atomic Data Types
Discussion
International Professorship in Computer Science, Katholieke Universiteit Leuven, Leuven, Belgium, January 23-27, 1984.

Barbara Liskov's Subtitution Principle (LSP)

- Polymorphic assignment: supertype variable can hold subtype object
  - every MessagePost object is-a Post object
- Compile-time type or Static type vs. Run-time type or Dynamic type
- Polymorphic or Dynamic or Run-time binding (see "method overriding")
  - uses dynamic type to bind method implementation with method call

because it is pointer work

so no slice:or ..

static type

dynamic type

```
Post post = new MessagePost("Jeff", "I like Java!");
post.display();
```

o<->□

```
public void show() {
   // display all posts
   for(Post post : posts) {
      post.display();
      System.out.println();
   }
}
```

compile time → Post

Run time' display → MessagePost . display

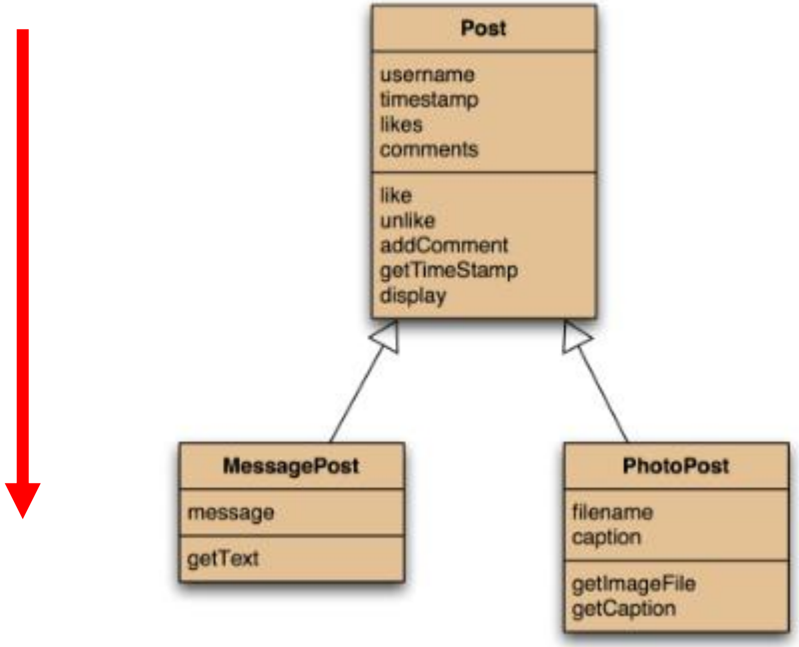
polymorphic or dynamic method binding

maybe slow

Java have garbage collector

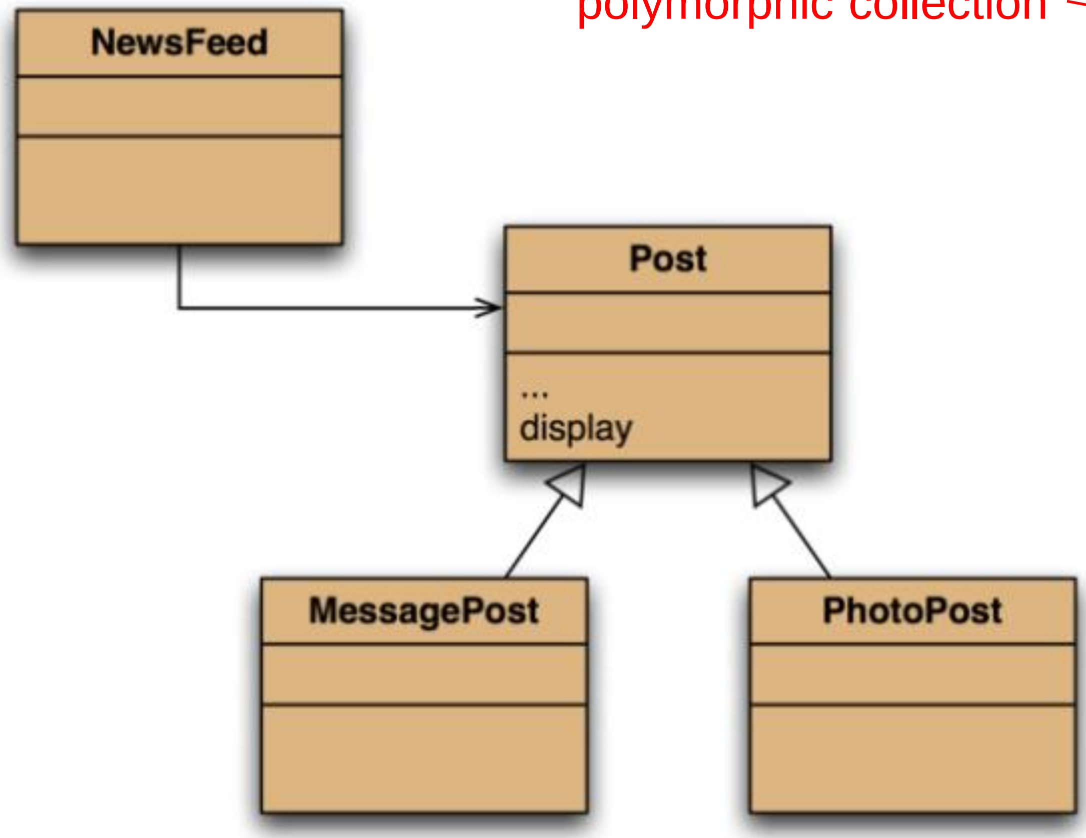
Type casting

- Only necessary in rare cases
  - “down” cast
  - subclass specific methods can be used
  - introduce inheritance?
- ClassCastException?
- instanceof operator



```
public void handlePost(Post p) {
   if (p instanceof MessagePost) {
      ((MessagePost) p).handleMessage();
   } else if (p instanceof PhotoPost) {
      ((PhotoPost) p).handlePost();
   }
}
```

Polymorphic collection/polymorphic parameter



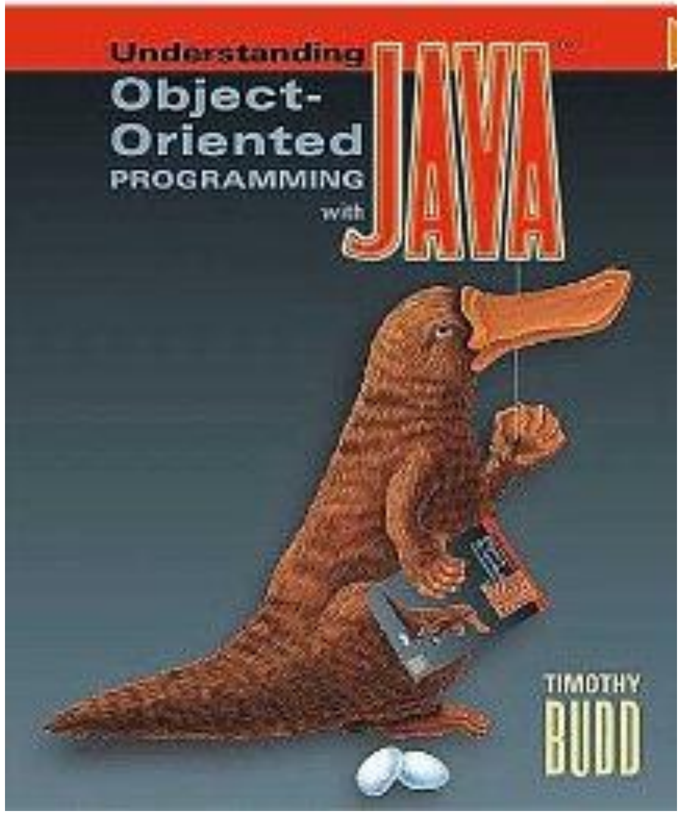
polymorphic collection

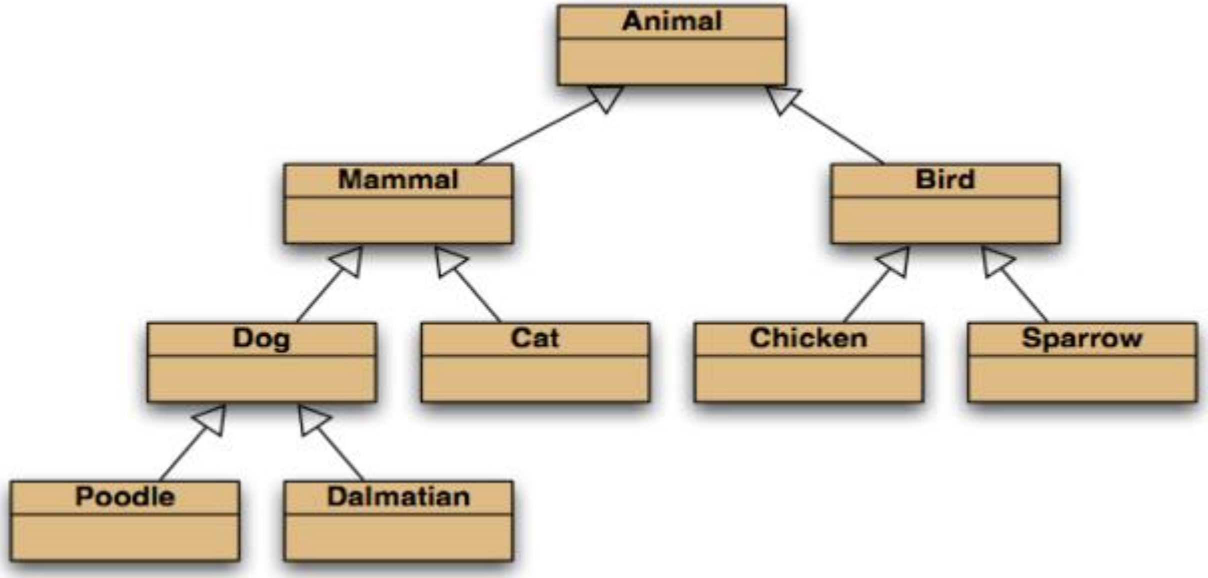
polymorphic parameter

```java
public class NewsFeed
{
    private ArrayList<Post> posts;

     /**
      * Add a post to the news feed.
      *
      * @param post  The post to be added.
      */
    public void addPost(Post post)
    {
        posts.add(post);
    }
    ...
}
```

Method overriding





Method overriding

- Superclass and subclass define methods with the same signature
- Each has access to the fields of its class and the ~~public~~/protected fields of superclasses
- Superclass satisfies static type check
- Subclass method is called at runtime – it overrides the superclass version
- @Override annotation
  - compiler directive to inform the compiler about your intent to override a method
  - not obligatory
  - goal: compile-time check and improved readability of your code

Call to super class method

Type change

Do I really need it?

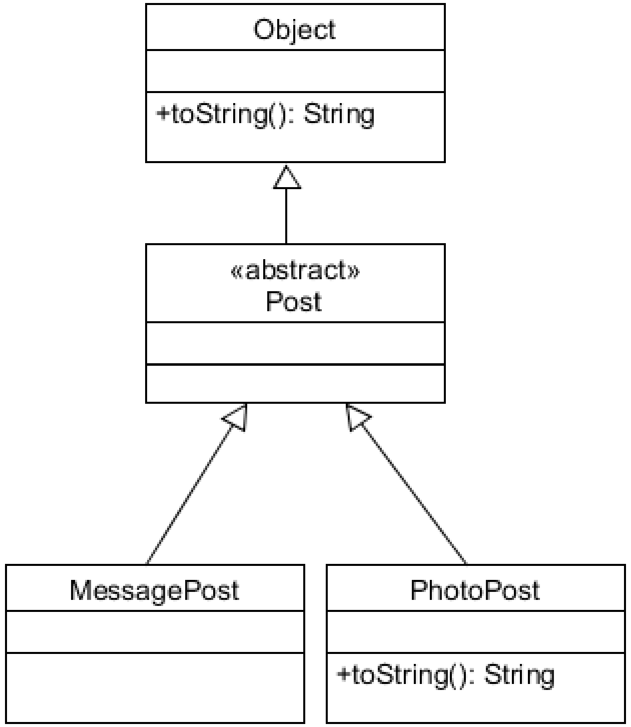
- Overriding hides super class method
  - use “super” to call the super class method
  - for instance in PhotoPost class:

```
public void display()
{
   super.display();
   System.out.println("  [" + filename + "]");
   System.out.println("  " + caption);
}
```

Override Object class methods

- If not overridden, Object class implementation is used (inheritance behaviour)
- Useful methods in class Object
  - toString()
    - commonly overridden to return a String representation of an object
    - implicitly called when a String object is needed
    - the default implementation (“classname@hashCode()”) is not particular useful
  - equals() & hashCode()
    - useful in collection implementations, check for existence, hashtables, …
  - clone()
    - see later: create a deep or shallow copy of an object
    - default implementation: identity copy (= pointer copy; two pointers to same object)

Exercise 1: compile-time vs. run-time



```
Post post = new MessagePost("Nobody", "Java rules!");
System.out.println(post.toString()); //1
post = new PhotoPost("An", "world.jpg", "Hello world!");
System.out.println(post.toString()); //2
```

- Will this code compile?
- What is the output?
  - toString() is implemented in
    - Object (= the java.lang.Object)
    - PhotoPost
  - toString() is <u>not</u> implemented in
    - Post (= an abstract class)
    - MessagePost

Abstract class/abstract methods

- abstract: useful for superclass
  - cannot be instantiated
  - can have abstract methods
- abstract method
  - no body/implementation
  - concrete subclass has to complete the implementation of all abstract classes from superclass, otherwise the subclass has to be defined abstract too
- used for keeping common fields and methods in class hierarchy and to please the compiler when there is no usefull implementation of a method in a superclass

Abstract class: example

- Post class defines a generic Post object, that does not exist or makes no sense
- Only instances of concrete subclasses do exist

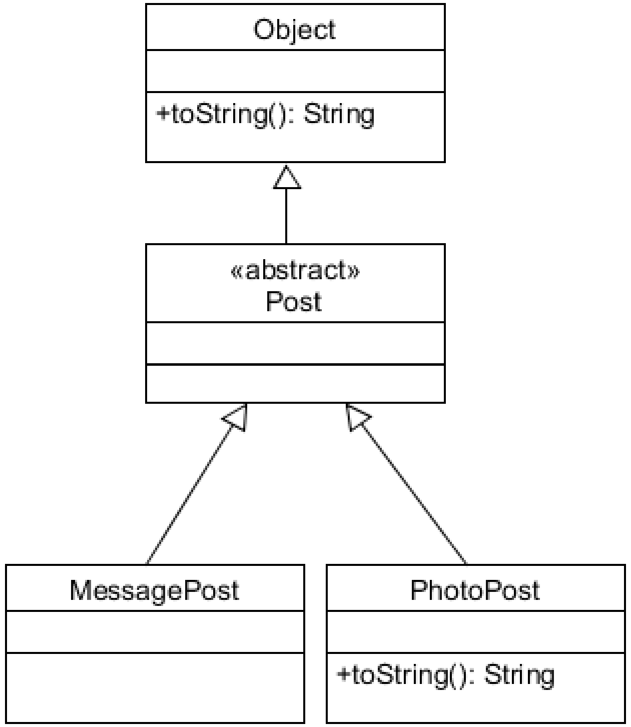
=> make superclass Post abstract

```
public abstract class Post
{
   private String username;
   private long timestamp;
   private int likes;
   private ArrayList<String> comments;

  …

}
```

Exercise 2



- How to add a method “handlePost()” that acts different for both kind of concrete posts?
  - This method will print:
    - “I am handling a MessagePost” or
    - “I am handling a PhotoPost”
  - There is no “handlePost()” method in class Post nor in class Object
  - Class Post has no useful implementation for the method “handlePost()”
- Use dynamic binding to avoid switch/case statements with type-checking

Interface

Interface

“acts-as”

Interface: introduction

compared to

① there is no super() in constructor
② can implement more than One interface

- Sensu lato: the public interface (= all public methods) of a class
- Sensu stricto: Java language feature
  = a list of methods specifications/declarations:
  “acts as a”, “has an implementation for”, “contract”
  - …able: Runnable, Cloneable, Iterable, Observable, …
  - …Listener: MouseListener, ActionListener, … (callback) [*]

all in Interface are abstract

- Ultimate separation between declaration (= the interface) and implementation (= the class) => powerful language construct

```
static A() {
    xxy
}
    B() {
    }
```

```
A();

xy.B();
```

[*] “listens” until some events happens

Interface: example

```
public interface BePolite {
   public abstract String sayThankYou();
}
```

```
public class Person implements BePolite {
   @Override
   public String sayThankYou() {
      return "Thanks!";
   }
}
```

```
BePolite politePerson = new Person();

BePolite politeNLPerson = new BePolite() {
   @Override
   public String sayThankYou() {
      return "Dank u!";
   }
};

System.out.println(politePerson.sayThankYou());
System.out.println(politeNLPerson.sayThankYou());
```

all methods in an interface are “public” and “abstract” by default

Java interface

It can extends

also implement but which?

superclass im..

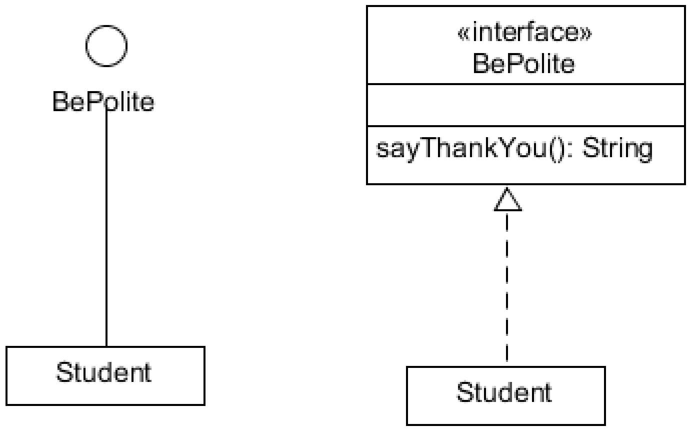
or subclass im...

- Subtype definition
- Can inherit from other interfaces (f.i. List -> Collection)
- Alternative for multiple inheritance: inherit from one class, implement multiple interfaces
- No instances, no constructors, no instance fields
- All methods are abstract (= no implementation)
    - except for default method implementations (since Java 8)

    => a way to extend existing interfaces without breaking implementations
- Can have static fields and static methods
- Multiple implementations possible
    - a class can implement more than one interface
    - an interface can be implemented by more than one class

Interface: UML notation

- Two notations
  - little circle
  - «interface» stereotype notation
- Realization
  - A class that implements (“realizes”) an interface

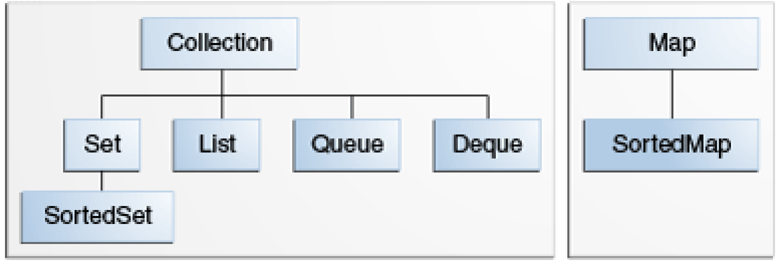


Interface: implementation in Java

- Interface can be implemented by
  - a class
  - an anonymous inner class
  - a lambda expression (in case of a @FunctionalInterface)
- Java alternative for multiple inheritance
  - Object “is-a” (only one!) thing and “acts like” other things
    - “is-a”: inheritance (extends)
    - “acts like”: interface (implements)
- Subtype definition (“Substitutability”)
  - Static type vs. Dynamic type (see: inheritance)

Interface: examples

- Class demo: StudentBehavior and BePolite interface
- Java API
  - Icon interface
  - Sorting elements (sorting countries)
    - Comparable interface
    - Comparator interface
  - Collection hierarchy (see part 2)
- Lambda expressions (see part 3)
- Design patterns based on interface (see part 4)



Icon interface

```java
public interface Icon
{
   /**
    * Draw the icon at the specified location.  Icon implementations
    * may use the Component argument to get properties useful for
    * painting, e.g. the foreground or background color.
    *
    * @param c  a {@code Component} to get properties useful for painting
    * @param g  the graphics context
    * @param x  the X coordinate of the icon's top-left corner
    * @param y  the Y coordinate of the icon's top-left corner
    */
   void paintIcon(Component c, Graphics g, int x, int y);

   /**
    * Returns the icon's width.
    *
    * @return an int specifying the fixed width of the icon.
    */
   int getIconWidth();

   /**
    * Returns the icon's height.
    *
    * @return an int specifying the fixed height of the icon.
    */
   int getIconHeight();
}
```

Comparable & Comparator interface

when there's only 1 abstract function . use lambda

maybe easier

- Comparable => natural ordering (compare argument with “this”)

```
@FunctionalInterface
public interface Comparable<T> {
    int compareTo(T o);
}
```

if the return is int add the '-' make an inverse sort

- Comparator => external ordering (compare two arguments of same type T)

```
@FunctionalInterface
public interface Comparator<T> {
    int compare(T o1, T o2);
}
```

anonymous inner class

unnamed instance

Type casting?

If you feel the need to do a type-cast…..

think twice!

Good design principle: “Program towards an interface”

- Decouple declaration from implementation: “what” vs. “how”
- Information hiding or encapsulation: do not expose the internals of you implementation
- Defer choice of actual class
- Criteria for designing a good interface (see later):
  - Cohesion: describes a single abstraction
  - Completeness: provides all operations necessary
  - Convenience: makes common tasks simple
  - Clarity: do not confuse the programmers
  - Consistency: keep the level of abstraction

Conclusion

Abstract class vs. Interface

- Abstract class
  - fields
  - concrete and abstract methods
  - constructors

- Interface
  - no fields
  - abstract methods
  - default methods (good idea?)
  - no constructors

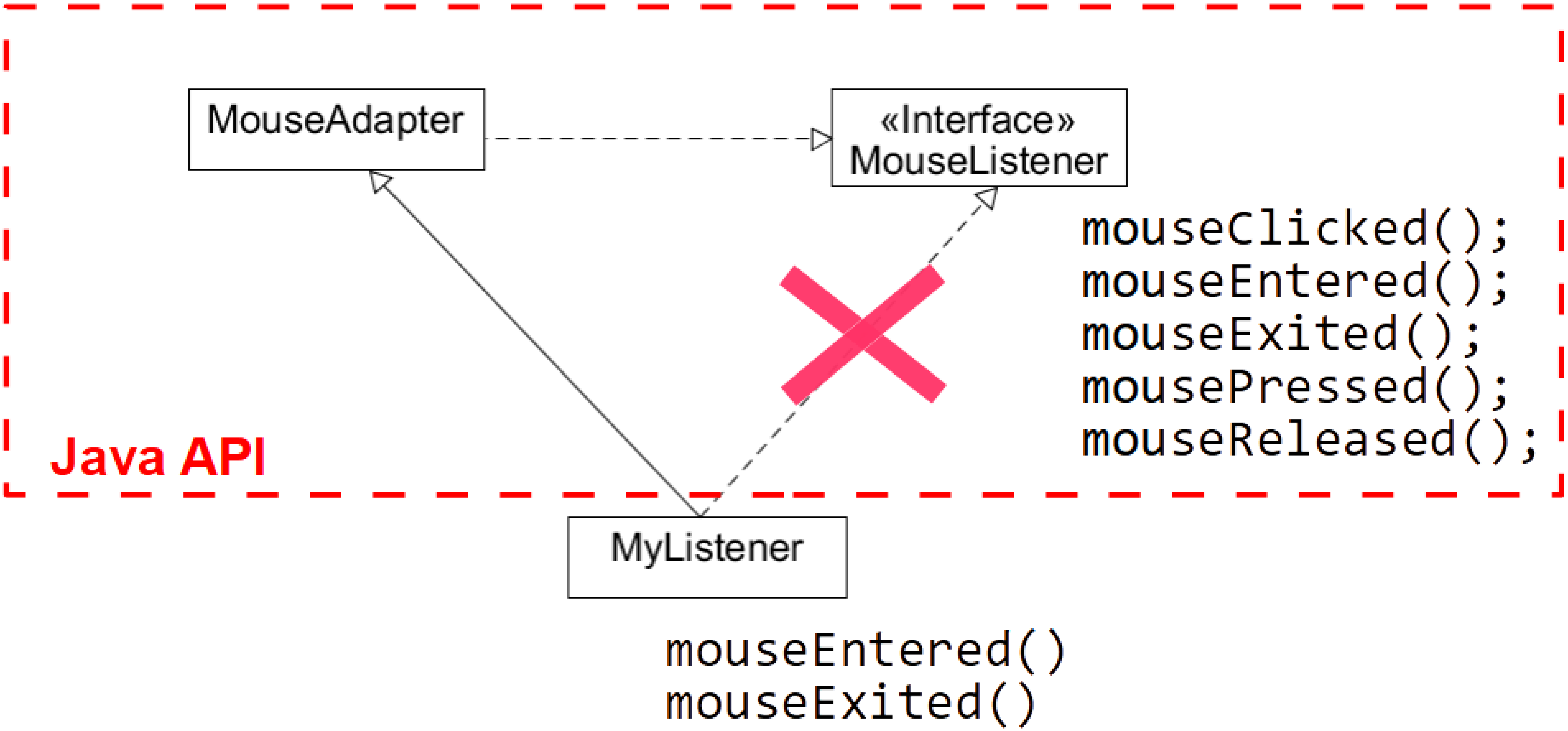
Prefer interface above abstract class

- more losely coupled: specification vs. implementation

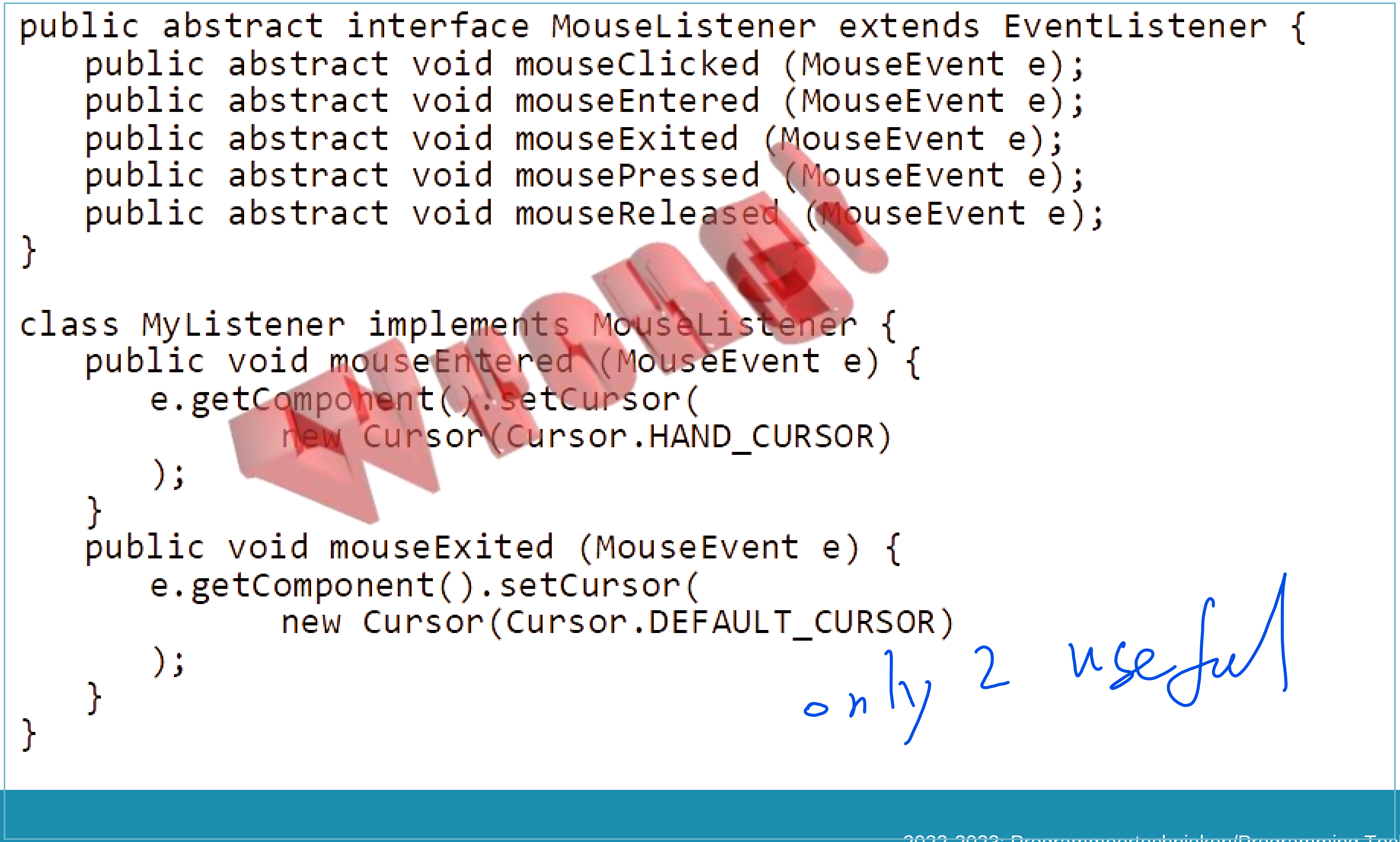
  but: default methods?
- more lightweight type
- the implementing class can still inherit from another class

Epilogue

Listener & Adapter (1)



Listener & Adapter (2)



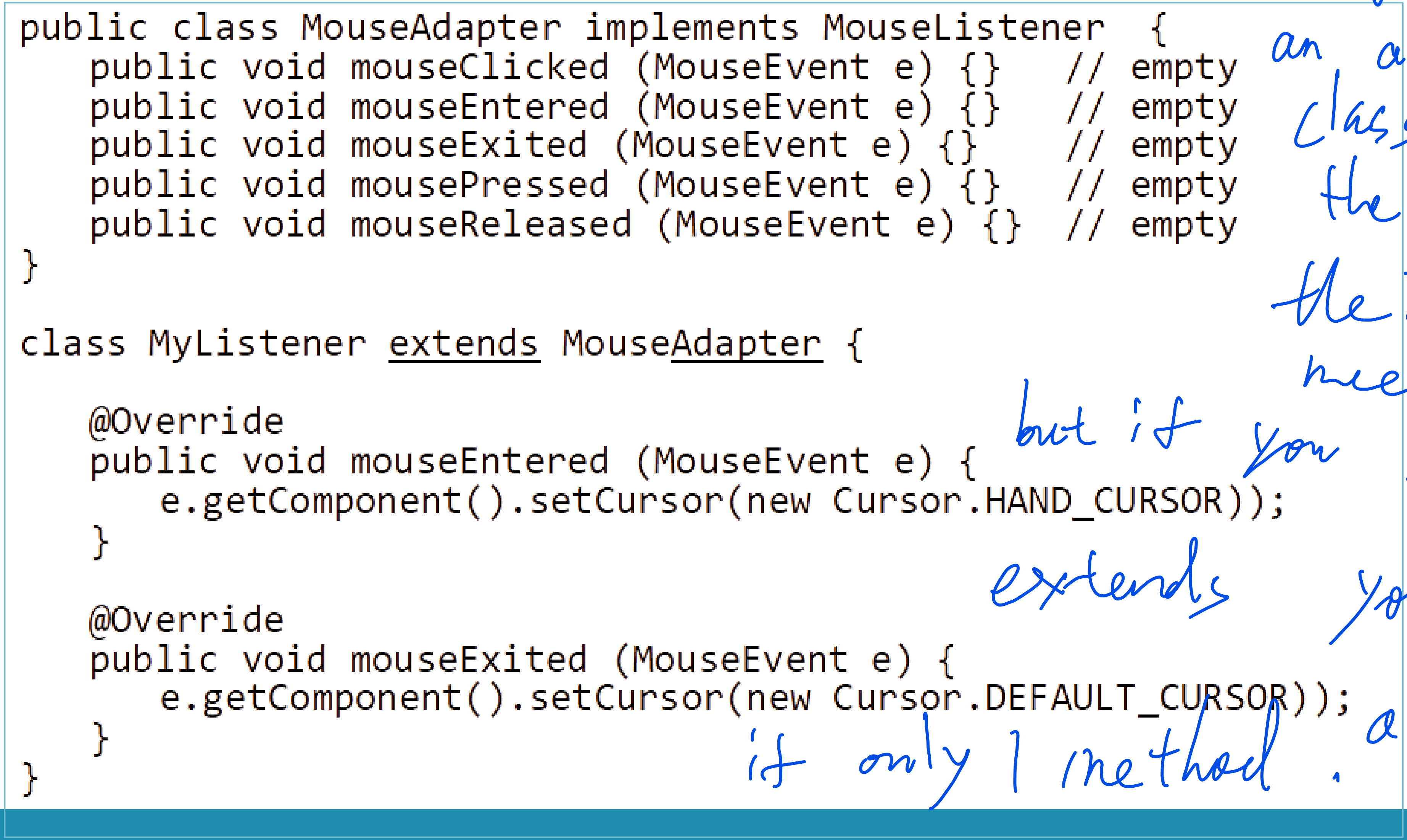
```
public abstract interface MouseListener extends EventListener {
   public abstract void mouseClicked (MouseEvent e);
   public abstract void mouseEntered (MouseEvent e);
   public abstract void mouseExited (MouseEvent e);
   public abstract void mousePressed (MouseEvent e);
   public abstract void mouseReleased (MouseEvent e);
}

class MyListener implements MouseListener {
   public void mouseEntered (MouseEvent e) {
      e.getComponent().setCursor(
            new Cursor(Cursor.HAND_CURSOR)
      );
   }
   public void mouseExited (MouseEvent e) {
      e.getComponent().setCursor(
            new Cursor(Cursor.DEFAULT_CURSOR)
      );
   }
}
```

only 2 useful

Listener & Adapter (3)

instead implement all we extend from an already implemen class the overrid the 2 we need



```
public class MouseAdapter implements MouseListener  {
   public void mouseClicked (MouseEvent e) {}   // empty
   public void mouseEntered (MouseEvent e) {}   // empty
   public void mouseExited (MouseEvent e) {}    // empty
   public void mousePressed (MouseEvent e) {}   // empty
   public void mouseReleased (MouseEvent e) {}  // empty
}

class MyListener extends MouseAdapter {

   @Override
   public void mouseEntered (MouseEvent e) {
      e.getComponent().setCursor(new Cursor.HAND_CURSOR));
   }

   @Override
   public void mouseExited (MouseEvent e) {
      e.getComponent().setCursor(new Cursor.DEFAULT_CURSOR));
   }
}
```

but if you already extends you can't

if only 1 method: adapter is not needed