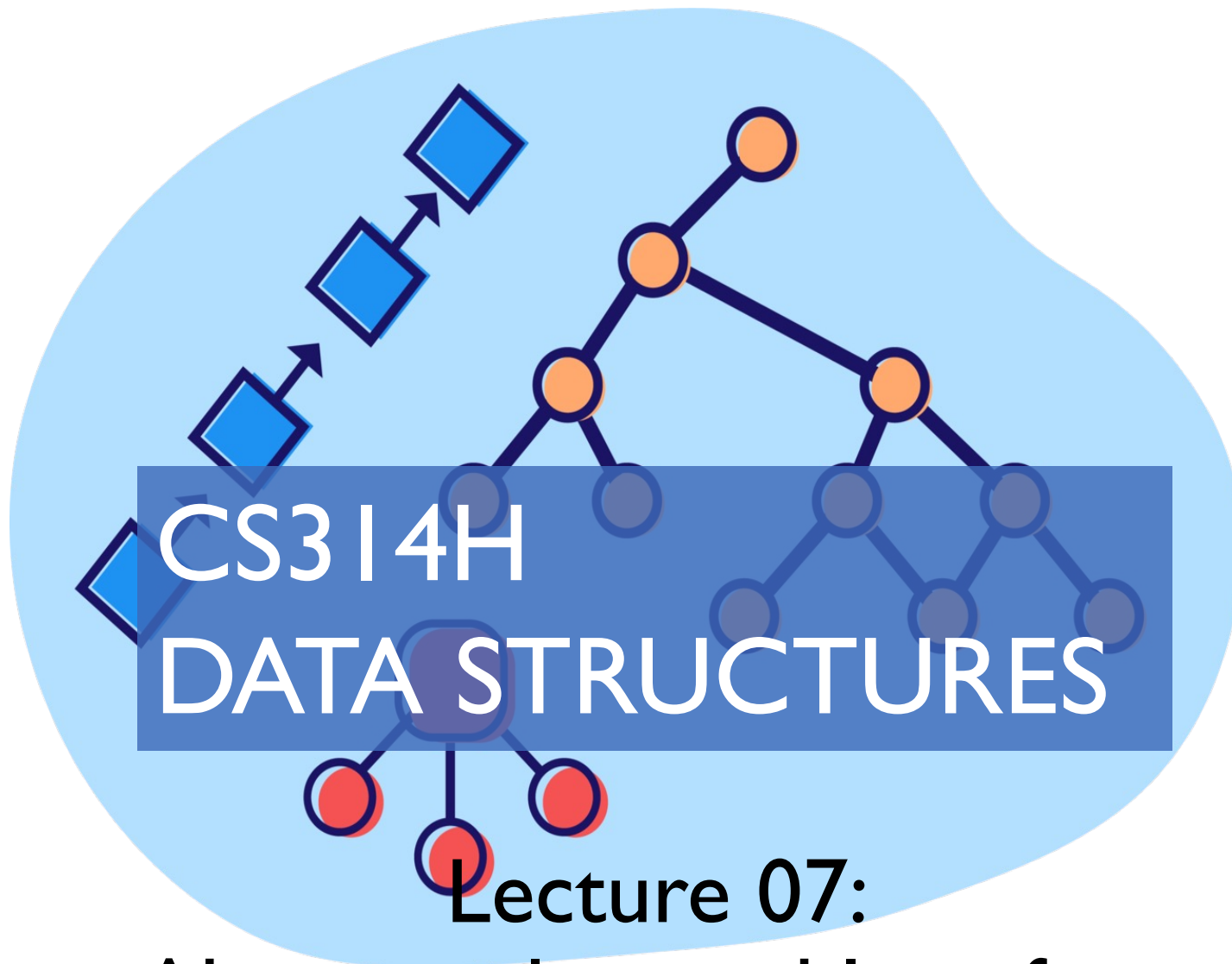




Abstract class and Interface

Mikyung Han



Please, interrupt and ask questions **AT ANY TIME !**

How to solve complex problem?

- **Decomposition: divide and conquer**
 - Divide into smaller scope problem
- **Abstraction: simplify**

Abstract Classes and Interfaces



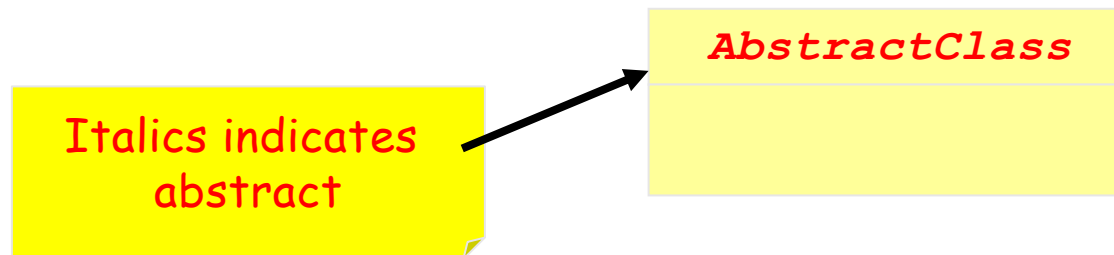
Outline

I. Administrative

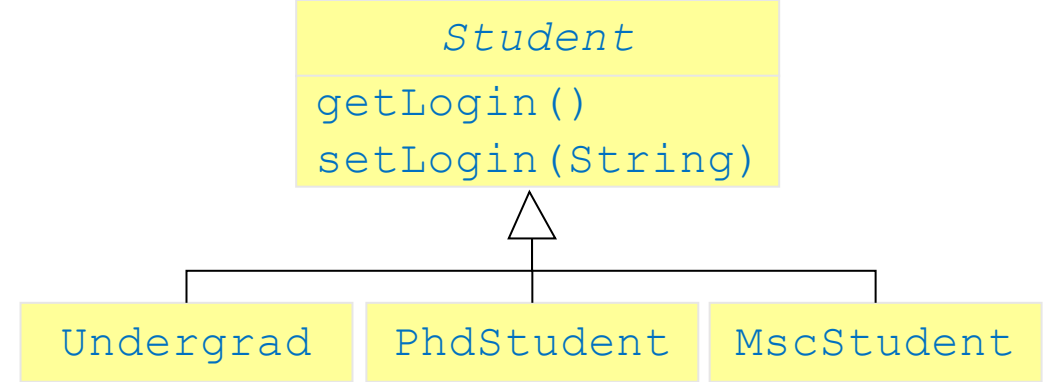
 2. Abstract Class

Abstract Classes

- Unlike classes, these cannot be instantiated.
- Like classes, they introduce types.
 - but no objects can have as actual type the type of an abstract class.
- Why use them?
 - Because common features/implementation exists
 - To prevent users from handling objects that are too generic (Example 1)
 - Cannot give a full implementation for the class (Example 2)



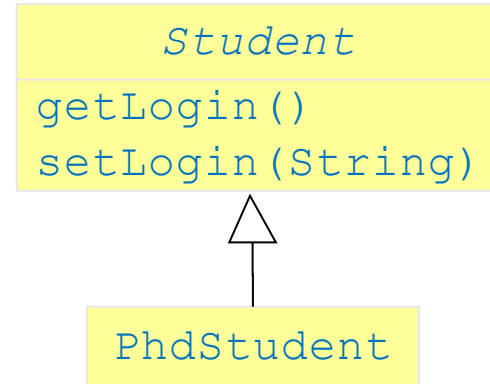
Example 1



- **The problem:**
 - Students are either undergraduate, PhD or MsC.
 - Need to guarantee that **nobody** creates a Student object.
 - Always need to create a specific kind of Student.
- **The solution:**
 - Declare Student as abstract.
- **Why have the Student class in the first place?**
 - A common implementation of common aspects of all students.
(e.g. *setLogin()* and *getLogin()*)
 - A place holder in my hierarchy that corresponds to some significant concept
 - Enable code reuse via subtype polymorphism

Abstract Classes in Java

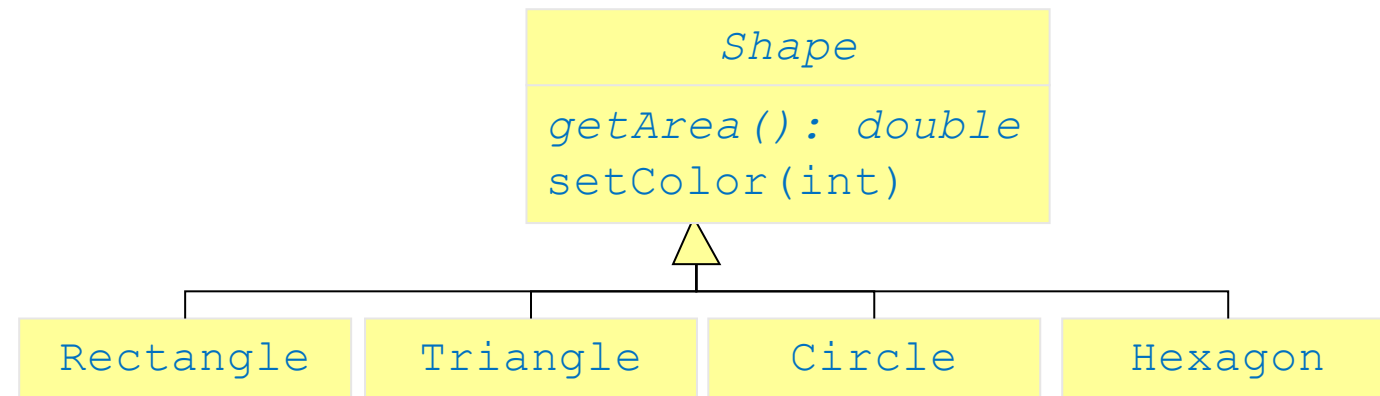
```
public abstract class Student {  
    protected String login, department, name;  
  
    public Student() {  
        login = ""; department = ""; name = "";  
    }  
  
    public void setLogin(String login) {  
        this.login = new String(login);  
    }  
  
    public String getLogin() {  
        return new String(login);  
    }  
}
```



PhdStudent is said
to be a concrete class

```
public class PhdStudent extends Student {  
    private String supervisor;  
  
    public void setSupervisor(String login) {  
        ...  
    }  
}
```


Example 2



- **The Problem**

- How to calculate the area of an arbitrary shape?
- We cannot allow *Shape* objects, because we cannot provide a reasonable implementation of *getArea()*;

- **The Solution**

- So we declare the *Shape* to be an **abstract** class.
- Furthermore, we declare *getArea()* as an **abstract method because** it has no implementation

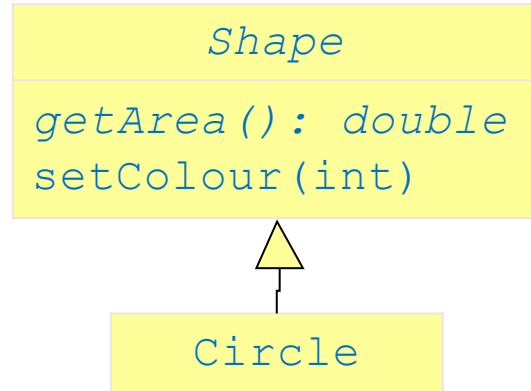
- **Why have the *Shape* class in the first place?**

- A placeholder for common features
- Plus that we want to force all shapes to provide an implementation for *getArea()*;
- Code reuse via subtype polymorphism

Abstract Methods in Java

```
public abstract class Shape {  
    private Color color;  
  
    public Shape() {  
        Color = Color.BLACK;  
    }  
  
    public void setColour(Color c) {  
        this.color = c;  
    }  
  
    public abstract double getArea();  
}
```

Abstract methods
have no body



```
public class Circle extends Shape {  
    final static double PI = 3.1419;  
    private int radius;  
  
    public Circle(int r) {  
        radius = r;  
    }  
  
    public double getArea() {  
        return (radius^2)*PI;  
    }  
}
```

If Circle did not implement getArea() then
it would have to be declared abstract too!

Abstract Classes

- What are the differences between both examples?
- In Example 1
 - I **choose** to declare Student abstract because I think it is convenient to prevent the existence of plain Students
- In Example 2
 - I **must** declare Shape abstract because it lacks an implementation for `getArea()`;

Prog1 with abstract class: Is this first case or second?

```
/**
 * Implements any color masking operation.
 */
abstract class InPlaceImageEffect extends ImageEffect {

    public int[][] apply(int[][] pixels, ArrayList<ImageEffectParam> params) {

        // modify pixels in-place
        for (int r = 0; r < pixels.length; r++) {
            for (int c = 0; c < pixels[r].length; c++) {
                pixels[r][c] = modifyPixel(pixels[r][c]);
            }
        }

        return pixels;
    }

    abstract int modifyPixel(int pixel);
}
```

```
/**
 * Filters out the red component of every pixel in an image.
 */
class NoRed extends InPlaceImageEffect {
    int modifyPixel(int pixel) {
        return makePixel(0, getGreen(pixel), getBlue(pixel));
    }
}
```

When you are NOT allowed to modify the original class introducing intermediate abstract class is particularly useful

Using abstract classes

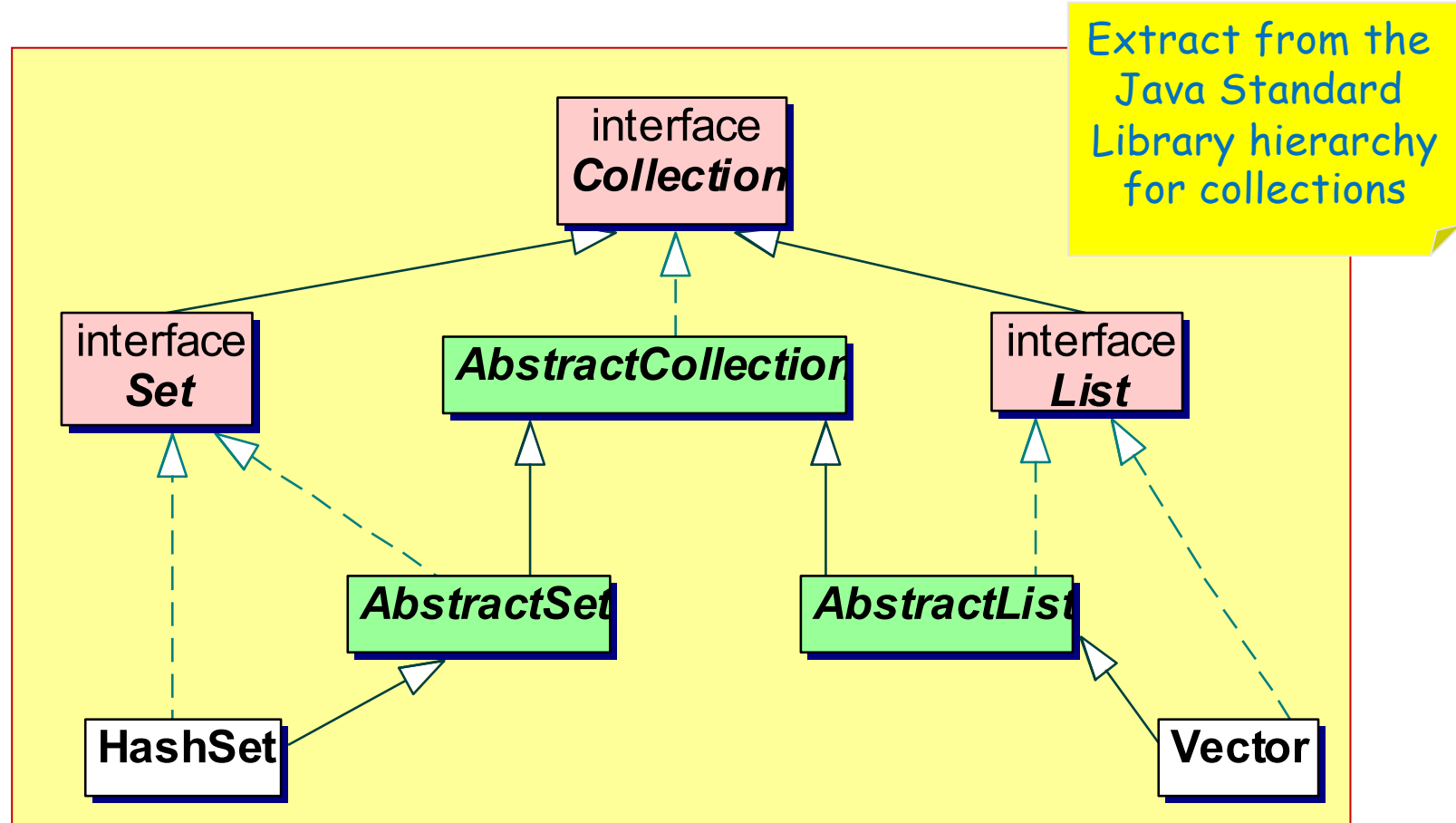
```
// Shape s = new Shape(); // ERROR  
Shape s = new Circle(4); // Ok  
double area = s.getArea(); // Ok - Remember polymorphism?  
Circle c = new Circle(3); // Ok  
c.setColour(Color.GREEN); // Ok  
area = c.getArea(); // Ok
```

- Class *Shape* cannot be instantiated
- Abstract methods can be called on concrete type object

Java abstract class examples

- <http://docs.oracle.com/javase/tutorial/java/landl/abstract.html>
- Which case is this?
 - Choose to be abstract
 - Has to be abstract
- Java Collection Hierarchy
 - [AbstractCollection](#) -> [AbstractList](#) -> [ArrayList](#)
 - [AbstractCollection](#) -> [AbstractSet](#) -> [HashSet](#)

Collections



`class HashSet extends AbstractSet implements Set ... { ... }`

Outline

1. Administrative

2. Abstract Class

 3. Interface

Interfaces (Java)

- An interface is a set of methods and constants that is identified with a name.

- They are similar to abstract classes

- You cannot instantiate interfaces
- An interface introduces types
- But, they are completely abstract (no implementation)

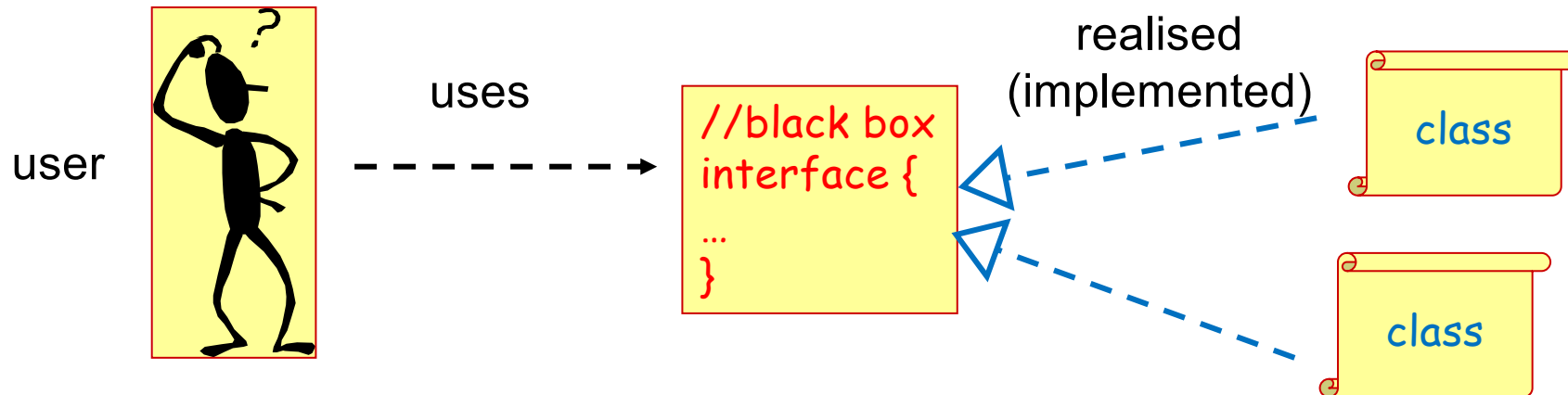
- Classes and abstract classes realize or implement interfaces.

- They must have (at least) all the methods and constants of the interface with public visibility

interface <i>Clock</i>
<i>MIDNIGHT:Time</i>
<i>setTime(Time):void</i>

Why use Interfaces?

- To *separate (decouple)* the specification available to the user from implementation
 - Subtype polymorphism: I can use any class that implements the interface through the interface type



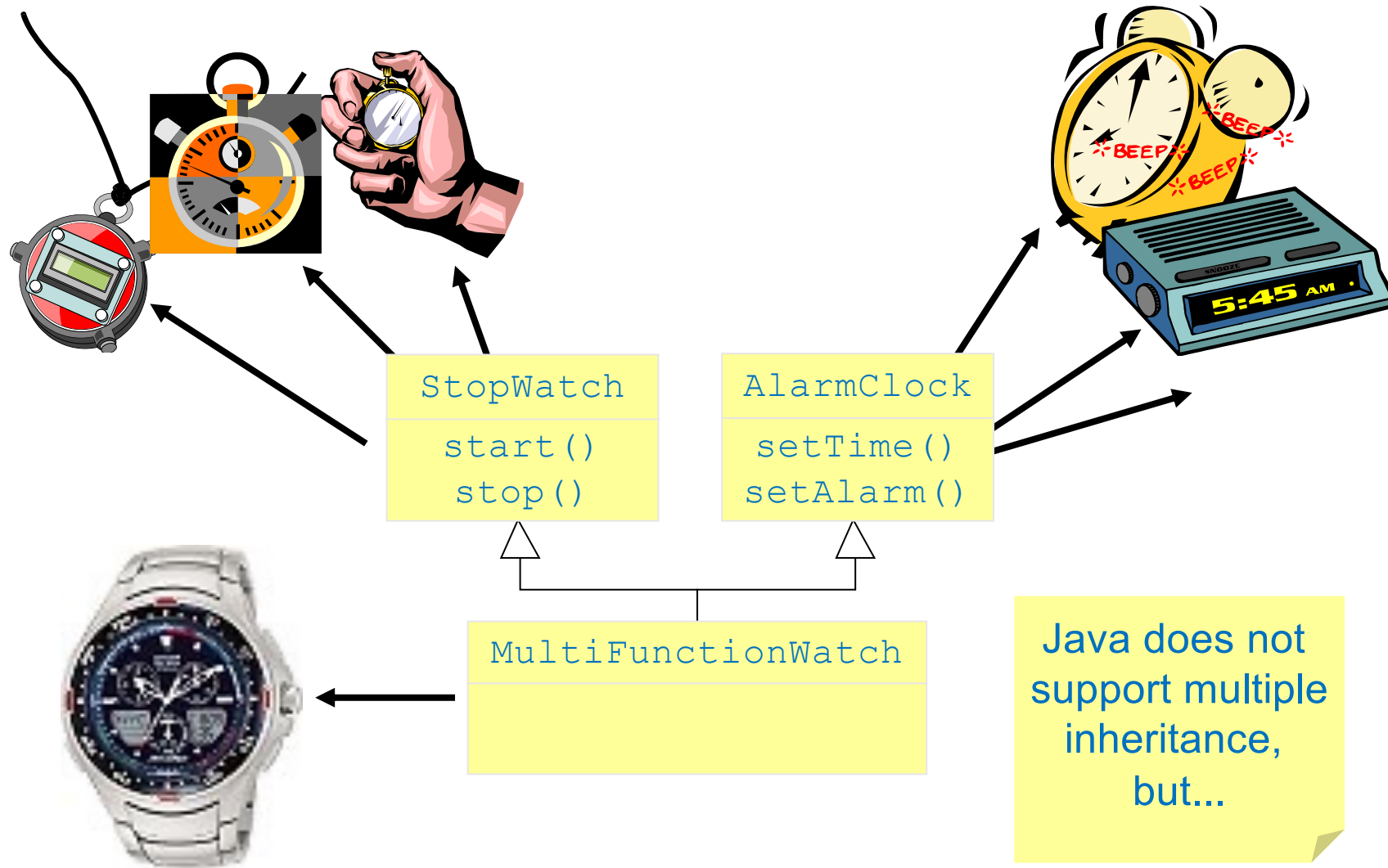
- As a partial solution to Java's lack of multiple inheritance

Interfaces (Java)

- Can an interface extend (inherit from) another interface?
- Can an interface extend multiple interfaces?
- Can an interface implement another interface?

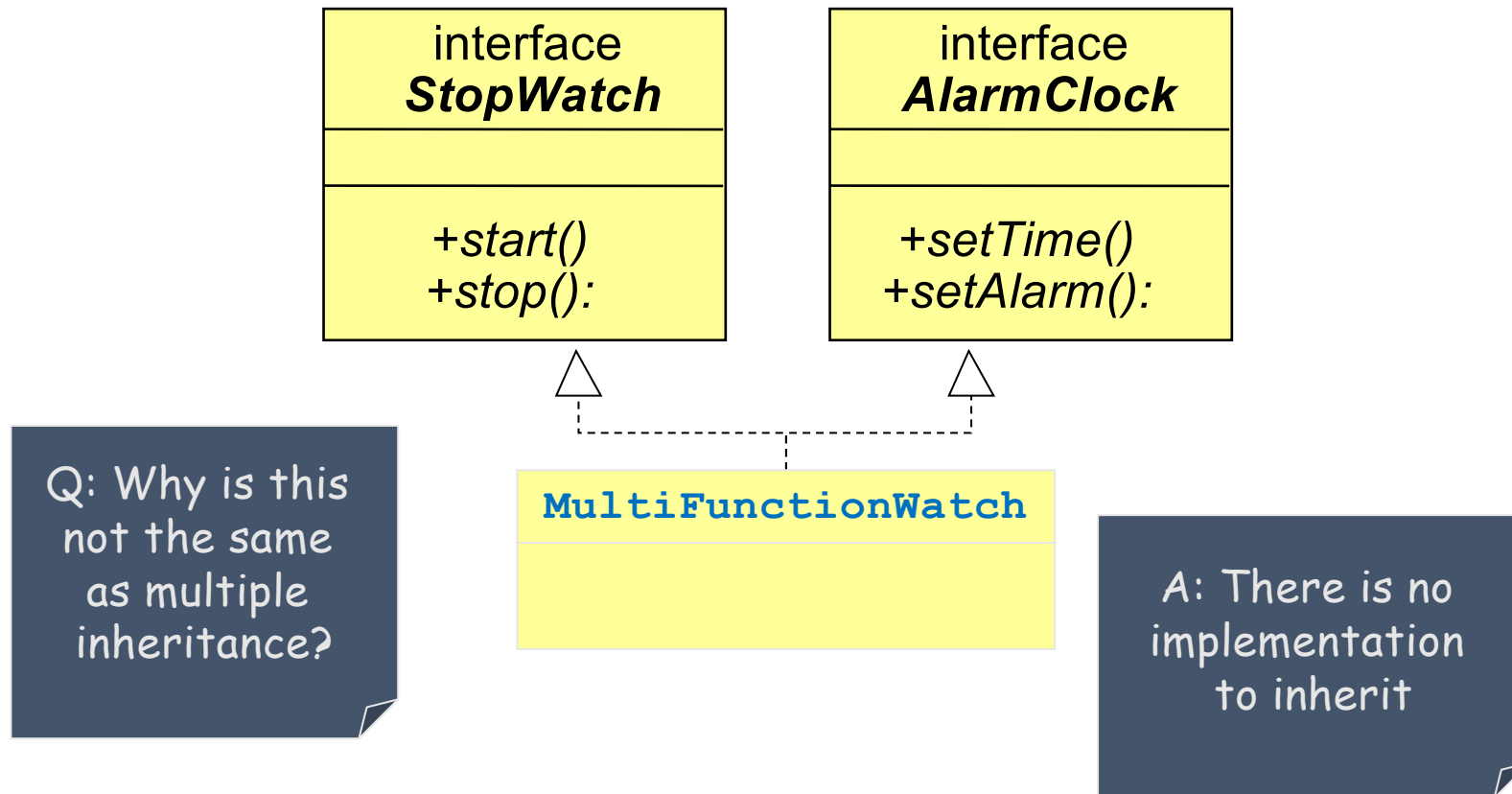
- All methods are implicitly **public**
- All fields are implicitly **public static final**
- Why?

Multiple Inheritance



Multiple Interfaces

- Classes are allowed to implement multiple interfaces



Abstract classes vs. Interfaces

- Can have **data** fields
 - Methods **may** have an implementation
 - Classes and abstract classes **extend** abstract classes.
 - Class **cannot extend multiple** abstract classes
 - **Substitution** principle is assumed
- Can only have static (constants)
 - Methods have **no** implementation
 - Classes and abstract classes **implement** interfaces
 - Interfaces can **extend multiple** interfaces
 - A class can **implement multiple** interfaces
 - Substitution principle not assumed

Abstract Classes or Interfaces?

- If there is common concrete implementation -> Abstract Class
- If there is no common implementation -> Interface
 - Interfaces allow classes implementing multiple interfaces...
 - Abstract classes can be subsequently extended without breaking subclasses...
 - No clear cut decision...

I am still not clear.. when to use what?

- [Credit](#)

Outline

1. Administrative

2. Abstract Class

3. Interface



4. More discussions

Should subclass of abstract class provide implementation for all abstract methods?

- No. But you need to mark the subclass as “abstract”
- Only if you provide implementation of all abstract methods, you can be a concrete class.
- **The same holds for a class that implements an interface**
 - In order for this class to be “concrete” class then it must override all abstract methods and provide concrete implementation for these methods.

Is this legal?

```
public static void main(String[] args) {  
    Link l = new Sausage(); //Concrete class Sausage implements interface Link  
    foo(l); //legal?  
}  
  
public static void foo(Object o){  
    //...  
}
```

Does this mean Interface inherits from a concrete class Object?

Interfaces are part of “type” hierarchy

```
public static void main(String[] args) {  
    Link l = new Sausage(); //Concrete class Sausage implements interface Link  
    foo(l); //legal?  
}  
  
public static void foo(Object o){  
    //...  
}
```

Inheritance is NOT the requirement for subtype polymorphism

Why this is legal?

```
public static void main(String[] args) {  
    Link l = new Sausage(); //Concrete class Sausage implements interface Link  
    foo(l); //legal?  
}  
  
public static void foo(Object o){  
    //...  
}
```

l should have everything *o* has

Yes, indeed interface has all methods in Object implicitly defined