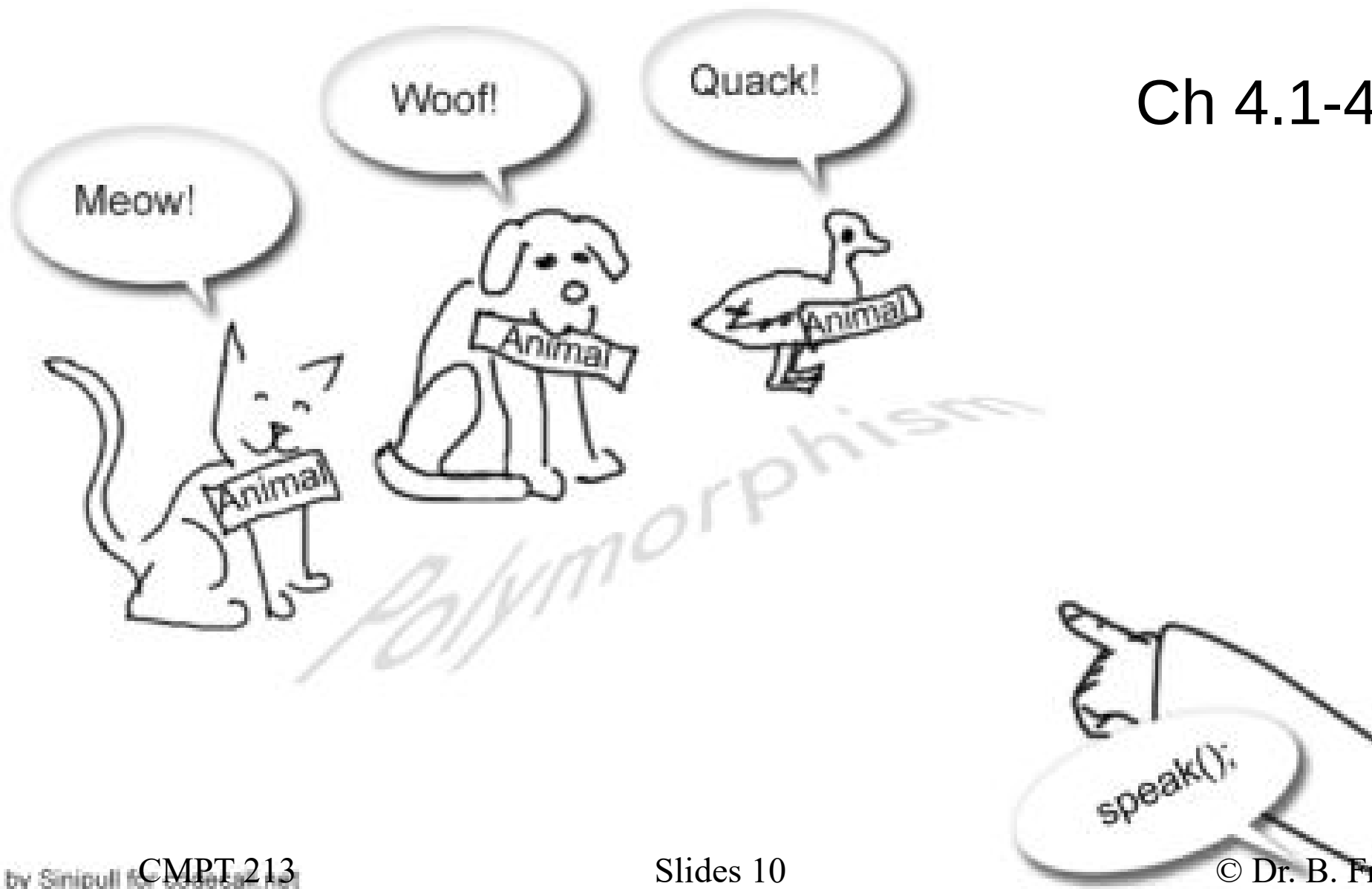


Interface Polymorphism

Ch 4.1-4.5



Topics

- 1) How can we reduce coupling between classes?
- 2) How can one piece of code work on different types of objects?

-> we want to encapsulate the things change

Interface

- An Interface specifies a set of **public** methods, but, *does not normally implement them*

- It's a contract for providing methods.

```
public interface LetterGrader {  
    String getGrade(double percent);  
    double getMinPercentForGrade(String grade);  
}
```

- "Interface" can refer to two things:
 - An interface in Java
(such as "The LetterGrader interface")
 - The *set of methods of a class*
(such as "The class's public interface")

Interface Usage

- To implement an interface, a class must both:
 - Say it "implements" the interface
 - implement all methods specified by the interface

```
public class EasyLetterGrader implements LetterGrader {  
    private static final double BREAK_POINT = 70;
```

```
    @Override
```

```
    public String getGrade(double percent) {  
        if (percent >= BREAK_POINT) {  
            return "A+";  
        } else {  
            return "B";  
        }  
        // Code seems incomplete :)  
    }  
}
```

```
    @Override
```

```
    public double getMinPercentForGrade(String grade) {  
        if (grade.compareToIgnoreCase("A+") == 0) {  
            return BREAK_POINT;  
        } else {  
            return 0;  
        }  
    }  
}
```

@Override is an..
annotation

Tells Java that this method..
MUST override a
method in the base
class/ interface

-takes a Logic error turns into Compile time error
-check if the spelling is correct

Concrete Types

- Concrete Type
 - the exact instantiated class of an object
(not a more general interface or base class).
- Example
 - LetterGrader is an Interface (not instantiatable),
so *not* a concrete type.
 - BAD: LetterGrader oops = new LetterGrader();
compile time error
bcuz LetterGrader is not a concrete type
- Example
 - EasyLetterGrader is an instantiatable class,
so is a concrete type
 - GOOD: LetterGrader good = new EasyLetterGrader();

Polymorphism

- Polymorphism Example:
 - A variable of type LetterGrade can reference any object of class type which **implements the LetterGrader interface**

```
LetterGrader g = new EasyLetterGrader();  
computeClassGrades(g);  
g = new HardLetterGrader();  
computeClassGrades(g);
```

- **(Subtype) Polymorphism**
If S is a subtype of type T, then ..
**a variable of type T can safely reference
an object of type S in any context**

- **The exact method to execute is selected at runtime (late binding).**
- Ex: Does g.getGrade() call **EasyLetterGrader.getGrade()**, or **HardLetterGrader.getGrade()** ?

Polymorphism Example

```
class MarkingSystem {  
    double[ ] marks = {74, 85, 25, 55, 93, 1};  
  
    void printLetterGrades() {  
        LetterGrader grader = new EasyLetterGrader();  
        String[] grades = gradeEachStudent(grader);  
  
        for (String grade : grades) {  
            System.out.println("Grade: " + grade);  
        }  
    }  
  
    String[ ] gradeEachStudent(LetterGrader grader) {  
        String[ ] letterGrades = new String[marks.length];  
        for (int i = 0; i < marks.length; i++) {  
            letterGrades[i] = grader.getGrade(marks[i]);  
        }  
        return letterGrades;  
    }  
}
```

gradeEachStudent is using strategy pattern:
have a hole in the algorithm

No idea what type of
LetterGrader is passed;
just that the object..

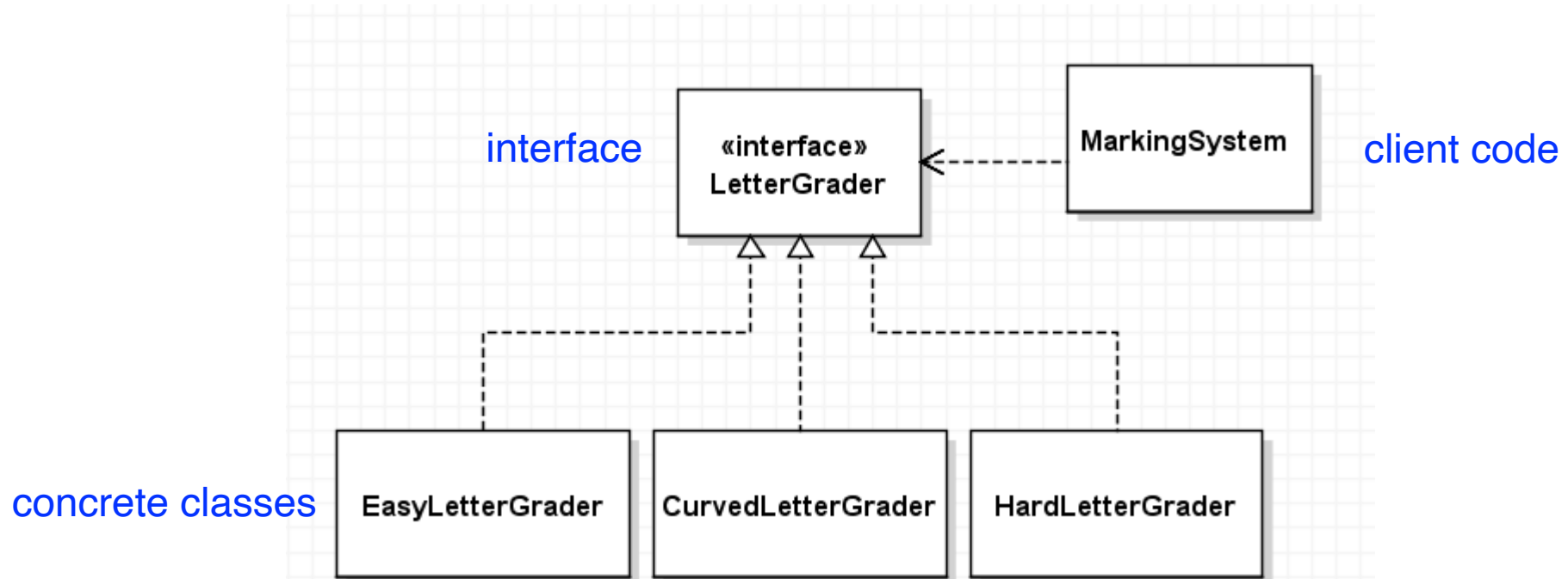
implements the LetterGrader interface

It can only use..

methods in the
LetterGrader interface

*note: it doesn't care what types of grader its using,
if it cares -> it violates the Lioskv's principle,
violates the subtype polymorphism

Terminology



principles used here:

- programming to interface, not implementation
- > code doesn't depend on concrete classes
- encapsulate things that change (therefore we have different types of grading)

Why Use Polymorphism?

- late binding

Exact method (concrete type) determined at runtime.

- give loose coupling

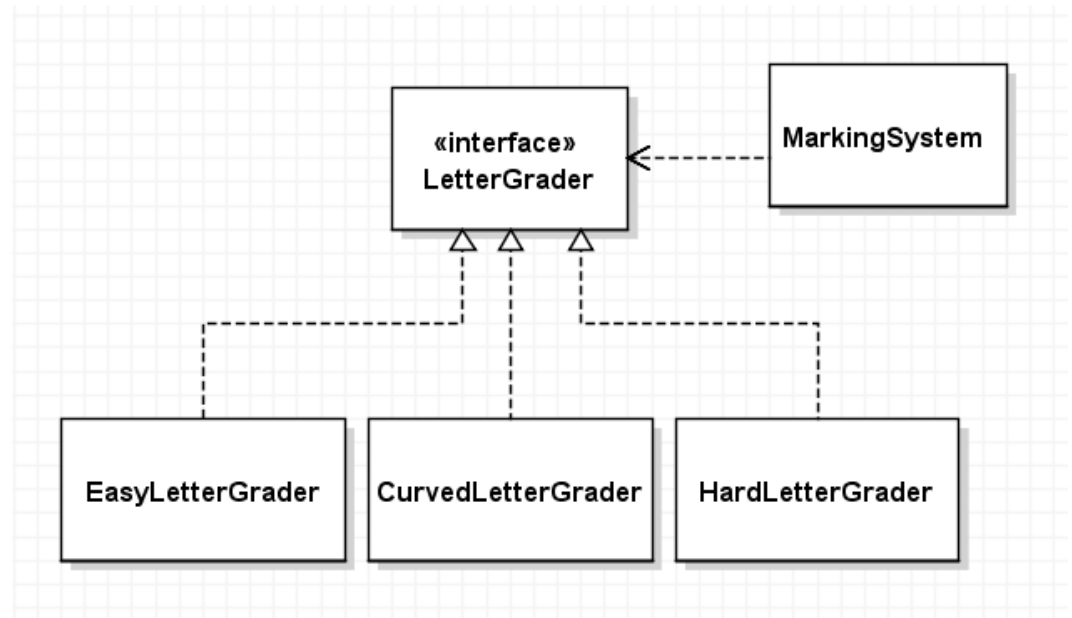
works with any object implementing the Interface so independent of object's concrete type.

- Design Heuristic:

code to an Interface,
not a concrete type

-> makes the code extensible

- Extensible:
Reuse code without
re-write to support
new classes.



NOTE: MarkingSystem only depends on the interface LetterGrader, not the other concrete types
-> this is called “programming to an interface, not implementation”
-> give loose coupling

Types of Polymorphism

- **adhoc polymorphism (a form of static polymorphism)**
 - Function or operator **overloading**
 - Write numerous functions, **each for a different specific type**
 - Compiler/interpreter picks the function to call based on the type of arguments.
- **parametric polymorphism**
 - Java's generics
 - Write one general implementation that **can work for any type**
- **subtype polymorphism**
 - Done using inheritance or interfaces with method **overriding**
 - The exact method to execute chosen at runtime (late binding).

```
void paint(Car c) {...};  
void paint(House h){...};  
...  
Car myCar = ...  
paint(myCar);
```

Static
(not at runtime)

```
int a = 1 + 3;  
String b = "hi" + "all";
```

```
class ArrayList<E> {  
    void add(E element) {...  
    E get(int idx) {...}  
}
```

Static

```
Object obj = ...;  
obj.toString();
```

Runtime

adhoc: write different versions for different things, and select between them in compile time

parametric: write once for everything

Interface Details

- Interface methods are **automatically public**
 - can provide “**default**” implementation of function.
- Can declare **constants** (automatically public static final)

```
public interface CardDeck {  
    int NUM_CARDS = 52;  
    // ...  
}
```

Comparable Review

- Can write algorithms for interface types.

```
interface Comparable<Type> {  
    int compareTo(Type obj);  
}
```

This is not quite perfect.
Comparable is a generic type, so
isAscending() should have the heading

public static <T extends Comparable<T>>
 boolean isAscending(T[] array) {

```
public class InOrder {  
    public static void main(String[] args) {  
        Long[] data = new Long[5];  
        for (int i = 0; i < data.length; i++) {  
            data[i] = i;  
        }  
  
        System.out.println("In order? "  
            + isAscending(data));  
    }  
  
    public static boolean  
    isAscending(Comparable[] array) {  
        for(int i = 0; i < array.length - 1; i++) {  
            Comparable first = array[i];  
            Comparable second = array[i+1];  
            if (first.compareTo(second) > 0) {  
                return false;  
            }  
        }  
        return true;  
    }  
}
```

this is subtype polymorphism bcuz elements are extended from Comparable
we can call the same function with different types of data

Comparator Review

- An idiom is.. [a common practice](#)
- For creating anonymous classes make a function which creates it.

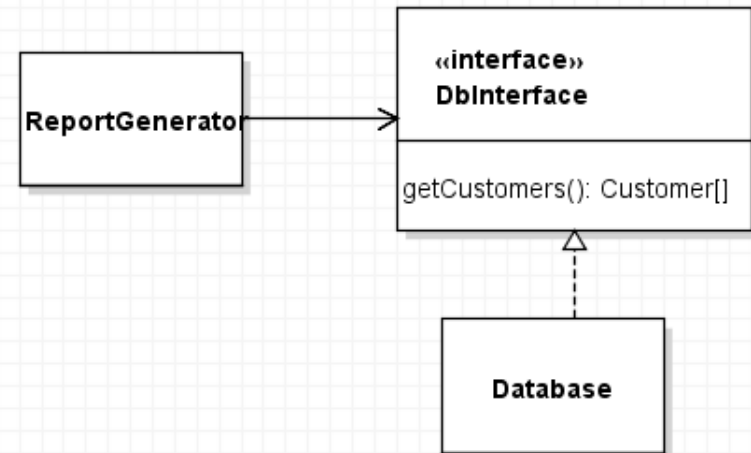
```
public interface FileFilter {  
    boolean accept(File path);  
}
```

```
private void addFolder(File directory) {  
    FileFilter filter = createExtensionFilter();  
    File[] files = directory.listFiles(filter);  
    //..  
}  
  
private FileFilter createExtensionFilter() {  
    return new FileFilter() {  
        @Override  
        public boolean accept(File path) {  
            return path.isDirectory()  
                || hasAcceptedExtension(path);  
        }  
    };  
}
```

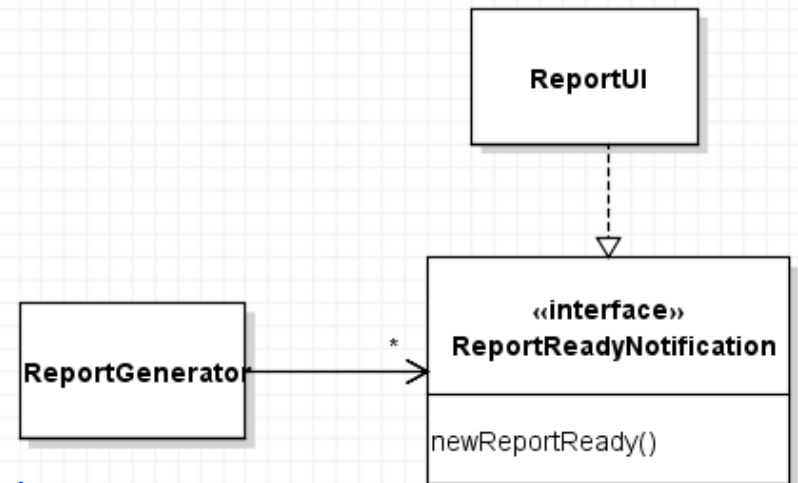
Using Interfaces

interface Dbinterface here isolates ReportGenerator and specific Database, give loose coupling, make it flexible

- Interface for Dependencies
 - A class may need the services of another object to do its job.
 - It can. define the interface it needs



- Interface for Services Offered
 - A class may provide services to another object.
 - It can. define the interface it provides



case 1. creating a service/interface that i need

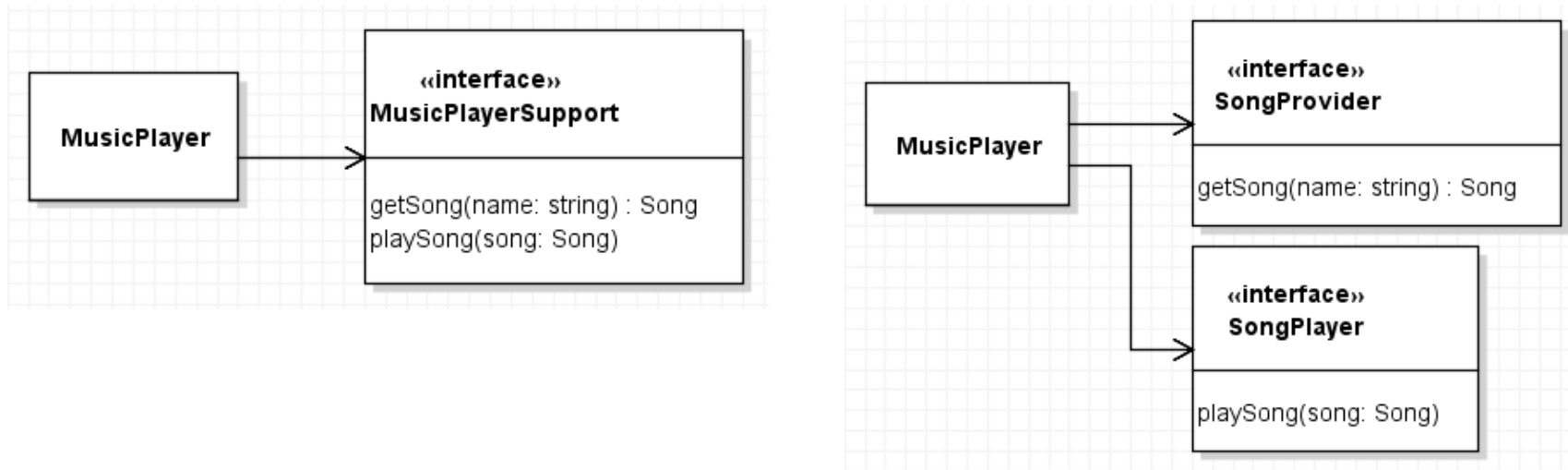
case 2.

in any case, both provide abstraction

big idea: use interface for isolation

Narrow Interfaces

- Prefer using a few small interfaces rather than one big one:



- Design Principle: [interface segregation](#)
 - Prefer small interfaces rather one large one.
 - Client code should not be forced to implement methods they do not need.
 - Client code can provide targeted functionality.

Review Questions

- Can the full type of an object be just an Interface type?
 - **No**: An object's concrete type cannot be an Interface. An Interface cannot be instantiated, only implemented by other classes.
 - Are the following two ideas identical?
 - A class which has the same methods as an Interface
 - A class which implements the interface?
 - no: for polymorphism to work, a class must “implement” the interface as well
- 1 + 2 is okay
1 == 2, not correct

Interface Details

- An Interface can [inherit from another interface](#)

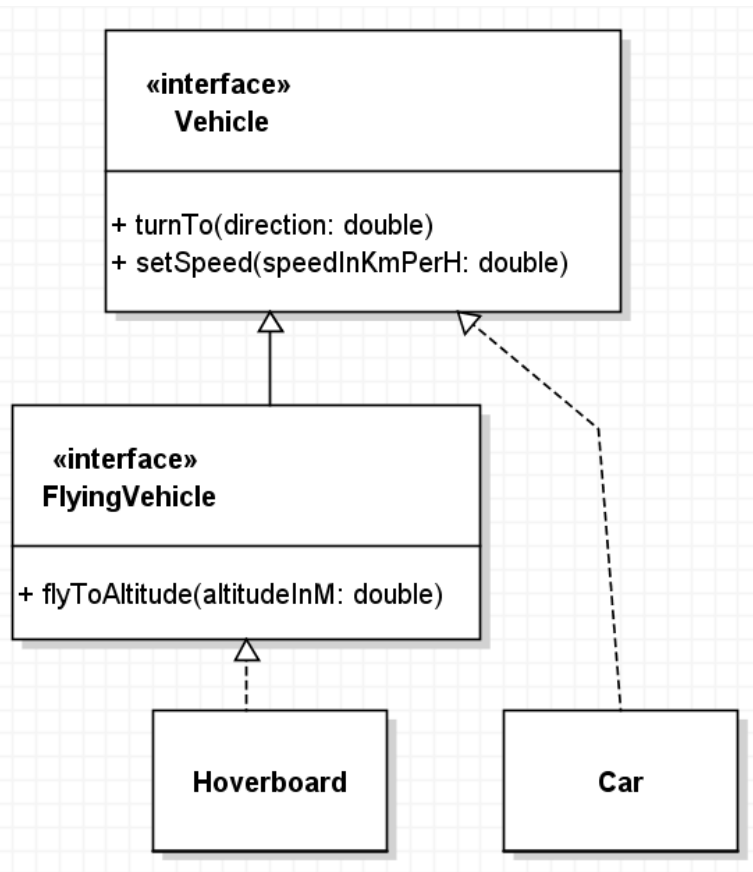
```
public interface Vehicle {  
    void turnTo(double direction);  
    void setSpeed(double speedInKmPerH);  
}
```

```
public interface FlyingVehicle extends Vehicle {  
    void flyToAltitude(double altitudeInM);  
}
```

- A class implementing FlyingVehicle must also implement all of Vehicle's methods too.

Exercise

- Which of the following statements work?



```
public static void main(String[] args) {
```

```
    Vehicle v1;
```

```
    v1 = new Vehicle(); not work
```

```
    v1 = new Car(); yes bcuz support subtype polymorphism
```

```
    v1 = new Hoverboard(); yes (but need to implements  
                           methods of both Vehicle & Flying Vehicle)
```

```
    FlyingVehicle v2;
```

```
    v2 = new Vehicle(); not work
```

```
    v2 = new Car(); not work
```

```
    v2 = new Hoverboard(); yes
```

```
    Car v3;
```

```
    v3 = new Vehicle(); not work (not working reversely)
```

```
    v3 = new Car(); yes
```

```
    v3 = new Hoverboard(); not work
```

```
}
```

Summary

- Interface: A set of methods & constants
 - How to define, implement, and use an interface
- Concrete Type: the instantiated type of an object
- Polymorphism
 - Static (compile time): Ad-hoc and parametric polymorphism
 - Runtime: subtype polymorphism
 - Example uses
- Interface Segregation Principle
 - Define narrow interfaces which provide targeted functionality