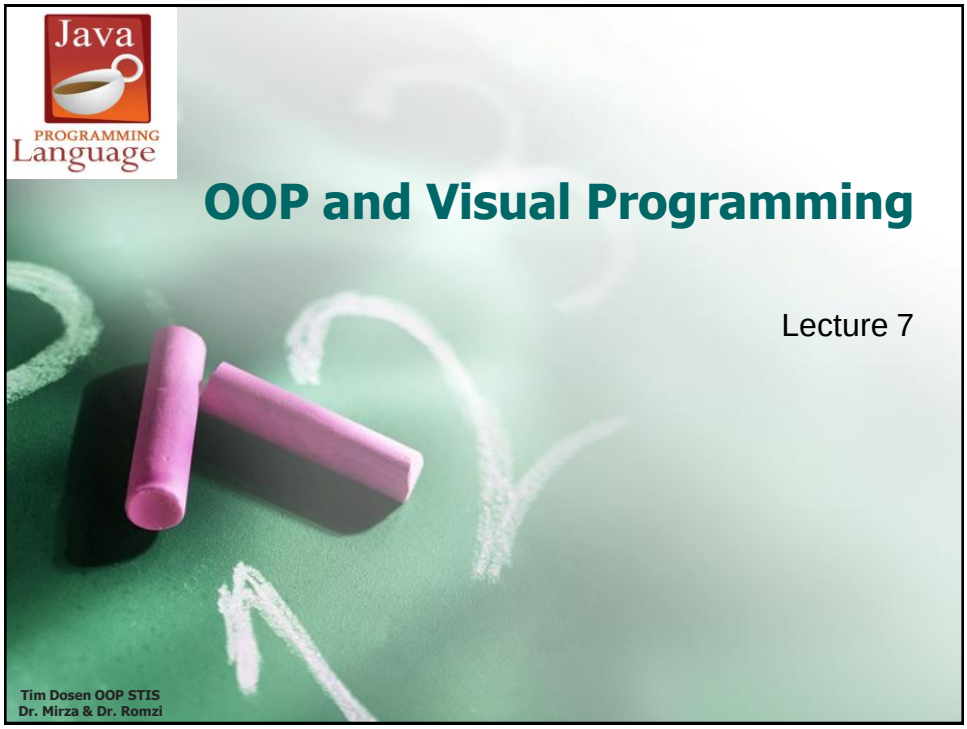
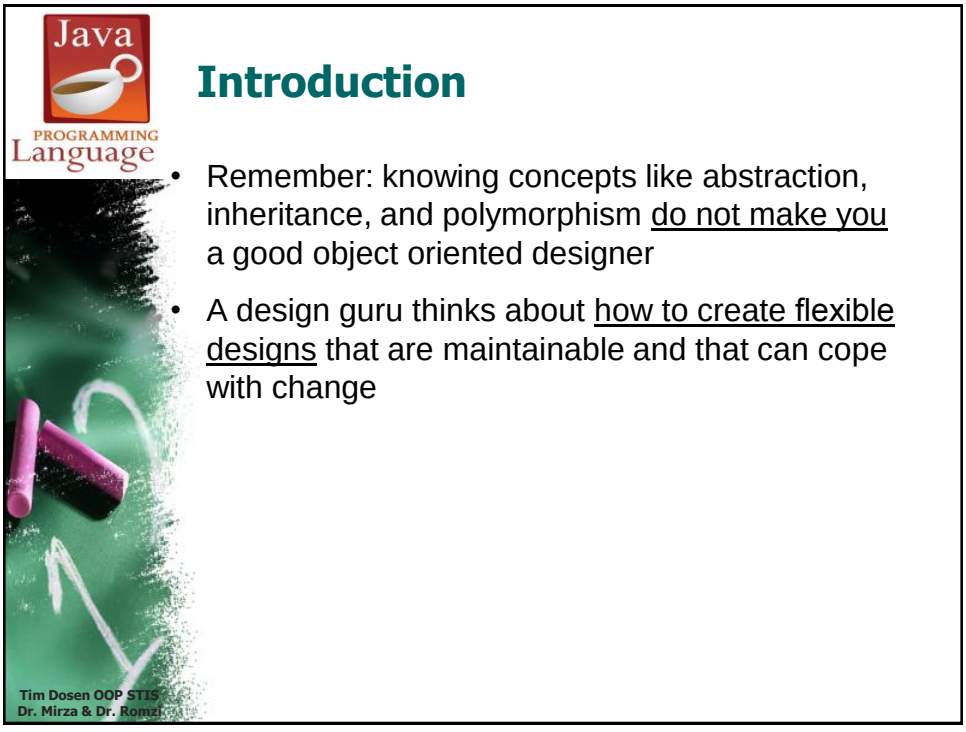
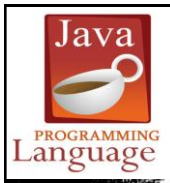

Java
PROGRAMMING
Language
OOP and Visual Programming
Lecture 7
Tim Dosen OOP STIS
Dr. Mirza & Dr. Romzi


Java
PROGRAMMING
Language
Agenda
Part VIII: Design Pattern: Strategy Pattern
• Introduction
• Strategy Pattern by Example
Tim Dosen OOP STIS
Dr. Mirza & Dr. Romzi
Sekolah Tinggi Ilmu Statistik (STIS)


Java
PROGRAMMING
Language
3
Design Pattern
STRATEGY PATTERN
Tim Dosen OOP STIS
Dr. Mirza & Dr. Romzi
Sekolah Tinggi Ilmu Statistik (STIS)

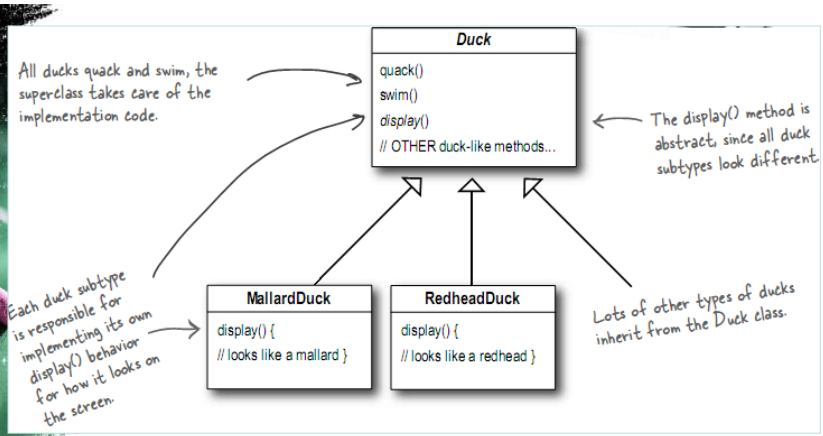

Java
PROGRAMMING
Language
Introduction
• Remember: knowing concepts like abstraction, inheritance, and polymorphism do not make you a good object oriented designer
• A design guru thinks about how to create flexible designs that are maintainable and that can cope with change
Tim Dosen OOP STIS
Dr. Mirza & Dr. Romzi



The SimUDuck Application

- A company wants to make a highly successful duck pond simulation game, SimUDuck
- The game can show a large variety of duck species swimming and making quacking sounds
- The initial designers of the system used standard OO techniques and created one Duck superclass from which all other duck types inherit

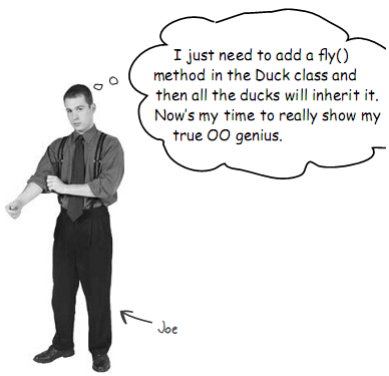
The Duck Superclass

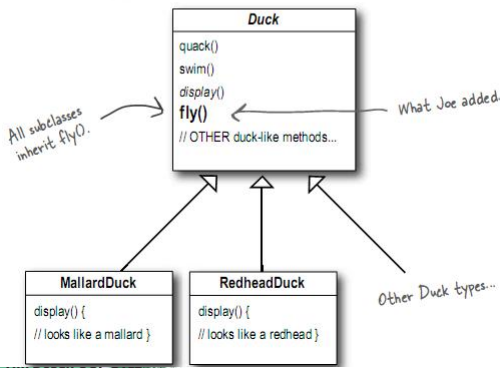


The Story Continued…

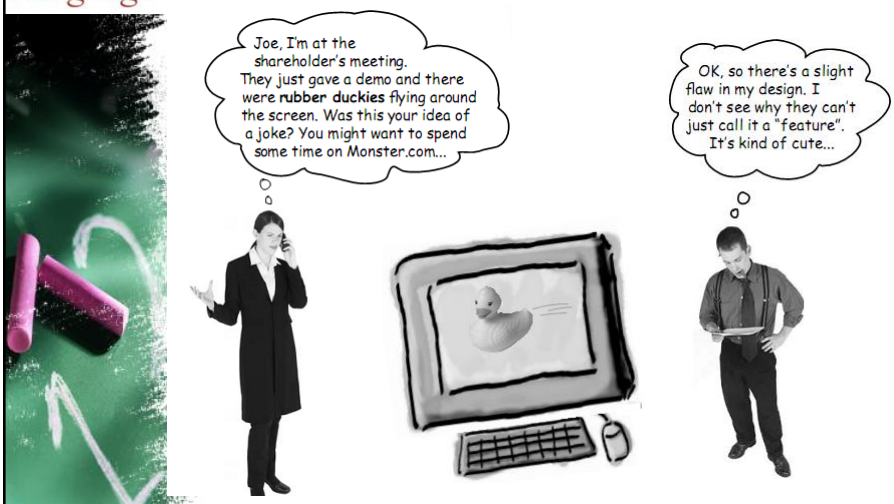
- In the last year, the company has been under increasing pressure from competitors
- After a week long off-site brainstorming session over golf, the company executives think it's time for a big innovation
- They need something really impressive to show at the upcoming shareholders meeting in Maui next week
- The executives decided that flying ducks is just what the simulator needs to blow away the other duck sim competitors

Now We Need the Ducks to FLY



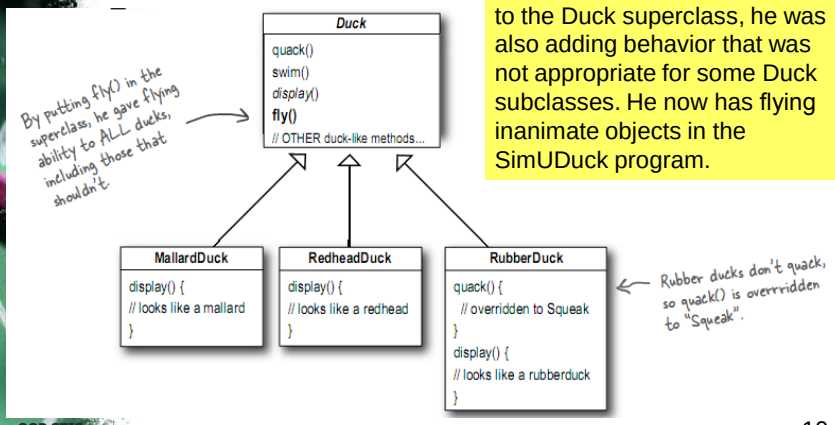


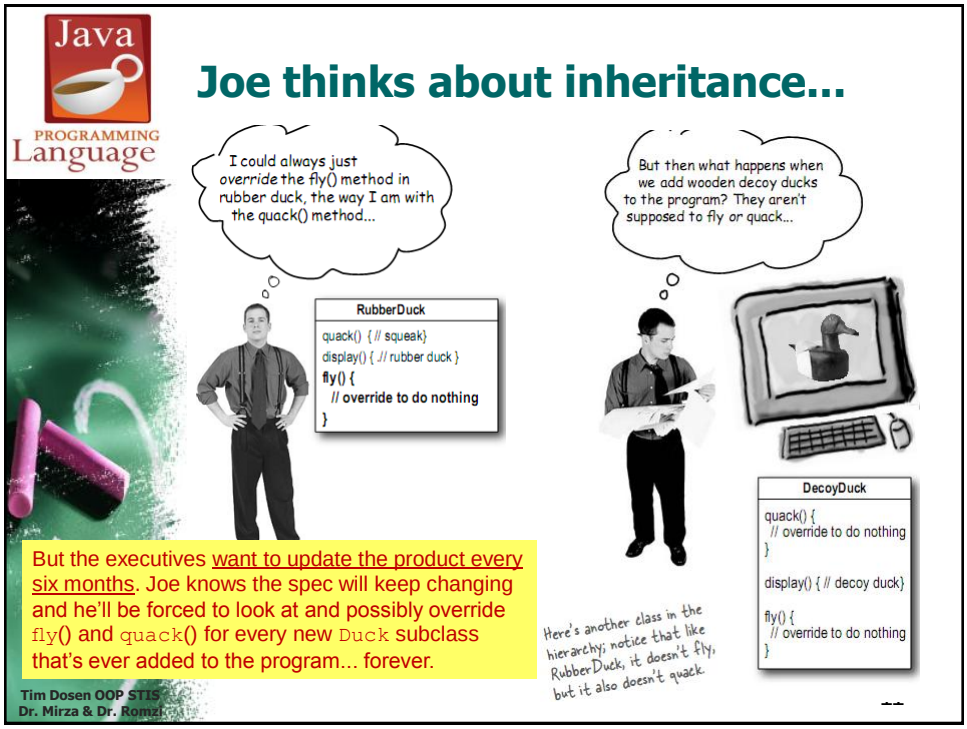
But something went horribly wrong…

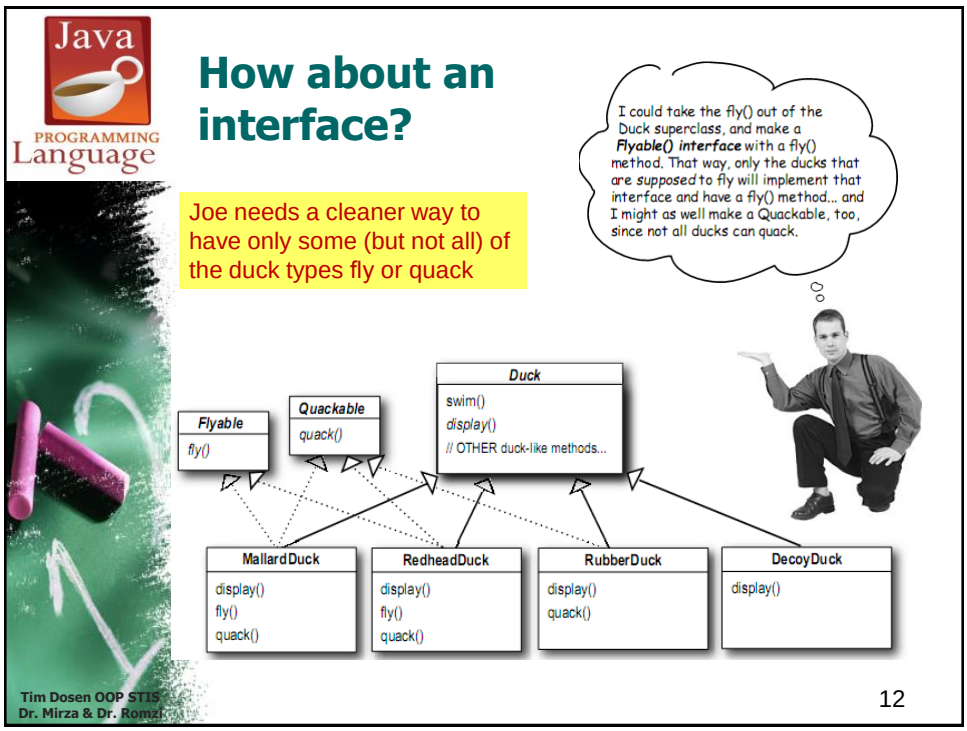


What happened?

Joe failed to notice that not all subclasses of Duck should fly. When Joe added new behavior to the Duck superclass, he was also adding behavior that was not appropriate for some Duck subclasses. He now has flying inanimate objects in the SimUDuck program.




Java
PROGRAMMING
Language
Joe thinks about inheritance...
I could always just override the fly() method in rubber duck, the way I am with the quack() method...
RubberDuck
quack() { // squeak}
display() { // rubber duck }
fly() {
// override to do nothing
}
But then what happens when we add wooden decoy ducks to the program? They aren't supposed to fly or quack...
DecoyDuck
quack() {
// override to do nothing
}
display() { // decoy duck}
fly() {
// override to do nothing
}
But the executives want to update the product every six months. Joe knows the spec will keep changing and he'll be forced to look at and possibly override fly() and quack() for every new Duck subclass that's ever added to the program... forever.
Here's another class in the hierarchy; notice that like RubberDuck, it doesn't fly, but it also doesn't quack.
Tim Dosen OOP STIS
Dr. Mirza & Dr. Romzi


Java
PROGRAMMING
Language
How about an interface?
Joe needs a cleaner way to have only some (but not all) of the duck types fly or quack
I could take the fly() out of the Duck superclass, and make a Flyable() interface with a fly() method. That way, only the ducks that are supposed to fly will implement that interface and have a fly() method... and I might as well make a Quackable, too, since not all ducks can quack.
Duck
swim()
display()
// OTHER duck-like methods...
Flyable
fly()
Quackable
quack()
MallardDuck
display()
fly()
quack()
RedheadDuck
display()
fly()
quack()
RubberDuck
display()
quack()
DecoyDuck
display()
Tim Dosen OOP STIS
Dr. Mirza & Dr. Romzi
12

What would you do if you were Joe?

- We know that not all of the subclasses should have flying or quacking behavior, so inheritance isn't the right answer
- But while having the subclasses implement Flyable and/or Quackable solves part of the problem, it completely destroys code reuse for those behaviors
- At this point you might be waiting for a Design Pattern to come riding in on a white horse and save the day!

The one constant in software development

- What's the one thing you can always count on in software development?

CHANGE

(use a mirror to see the answer)

- No matter how well you design an application, over time an application must grow and change or it will die!

Zeroing in on the problem...

- Luckily, there's a design principle for just this situation

Design Principle

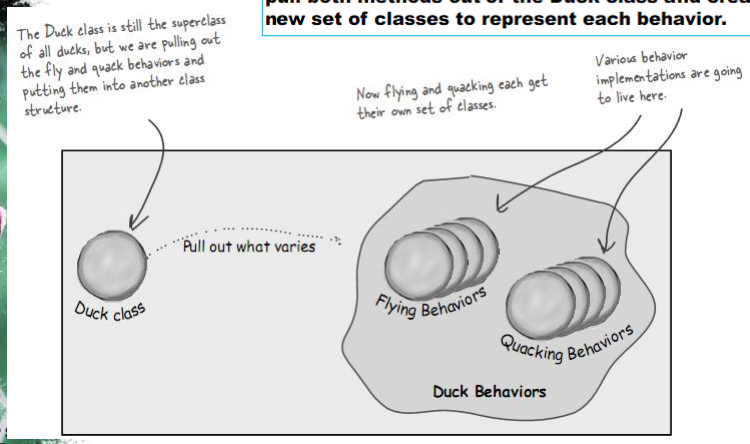
Identify the aspects of your application that vary and separate them from what stays the same.

- Take the parts that vary and encapsulate them, so that later you can alter or extend the parts that vary without affecting those that don't

Separating what changes from what stays the same

We know that fly() and quack() are the parts of the Duck class that vary across ducks.

To separate these behaviors from the Duck class, we'll pull both methods *out* of the Duck class and create a new set of classes to represent each behavior.



Designing the Duck Behaviors

Design Principle

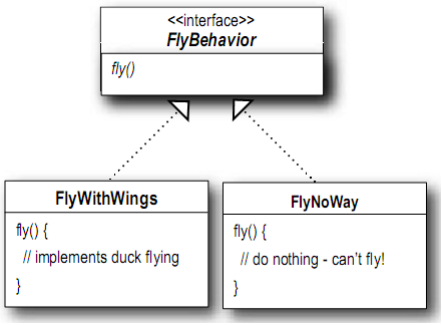
Program to an interface, not an implementation.

- We'll use an interface to represent each behavior (`FlyBehavior` and `QuackBehavior`) and each implementation of a behavior will implement one of those interfaces
- Duck subclasses will use behavior represented by the interfaces, so that the actual implementation of the behavior won't be locked into the Duck subclass

Applying the Design Principle

From now on, the Duck behaviors will live in a separate class—a class that implements a particular behavior interface.

That way, the Duck classes won't need to know any of the implementation details for their own behaviors.



Program to an Interface

"Program to an *interface*" really means "Program to a *supertype*."

- You can program to an interface, without having to actually use a Java interface
- The point is to exploit polymorphism by programming to a supertype so that the actual runtime object isn't locked into the code
- And we could rephrase "program to a supertype" as:
  - *"the declared type of the variables should be a supertype (an abstract class or interface), so that the objects assigned to those variables can be of any concrete implementation of the supertype, which means the class declaring them doesn't have to know about the actual object types!"*

Example

Programming to an implementation would be:

```
Dog d = new Dog();
d.bark();
```

Declaring the variable "d" as type Dog (a concrete implementation of Animal) forces us to code to a concrete implementation.

But **programming to an interface/supertype** would be:

```
Animal animal = new Dog();
animal.makeSound();
```

We know it's a Dog, but we can now use the animal reference polymorphically.

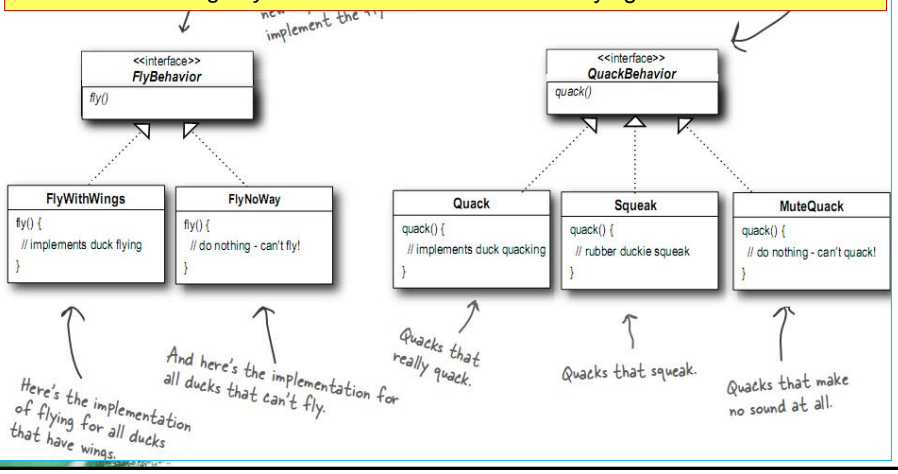
Even better, rather than hard-coding the instantiation of the subtype (like new Dog()) into the code, **assign the concrete implementation object at runtime:**

```
a = getAnimal();
a.makeSound();
```

We don't know WHAT the actual animal subtype is... all we care about is that it knows how to respond to makeSound().

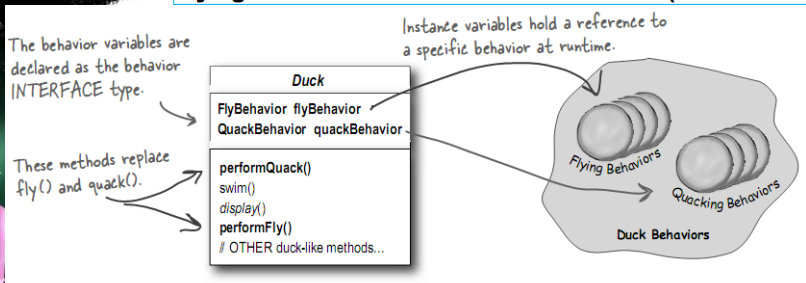
Implementing the Duck Behaviors

With this design, other types of objects can reuse our `fly` and `quack` behaviors because these behaviors are no longer hidden away in our Duck classes!
And we can add new behaviors without modifying any of our existing behavior classes or touching any of the `Duck` classes that use flying behaviors.



Integrating the Duck Behavior

The key is that a Duck will now *delegate* its flying and quacking behavior, instead of using quacking and flying methods defined in the Duck class (or subclass).



❷ **Now we implement performQuack():**

```
public class Duck {
   QuackBehavior quackBehavior;
   // more

   public void performQuack() {
      quackBehavior.quack();
   }
}
```

Each Duck has a reference to something that implements the QuackBehavior interface.

Rather than handling the quack behavior itself, the Duck object delegates that behavior to the object referenced by quackBehavior.

2

Mallard Duck Class

```
public class MallardDuck extends Duck {

    public MallardDuck() {
        quackBehavior = new Quack();
        flyBehavior = new FlyWithWings();
    }

    public void display() {
        System.out.println("I'm a real Mallard duck");
    }
}
```

A MallardDuck uses the Quack class to handle its quack, so when performQuack is called, the responsibility for the quack is delegated to the Quack object and we get a real quack.

And it uses FlyWithWings as its FlyBehavior type.

Remember, MallardDuck inherits the quackBehavior and flyBehavior instance variables from class Duck.

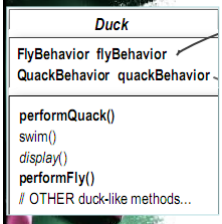
Wait a second, didn't you say we should NOT program to an implementation? But what are we doing in that constructor? We're making a new instance of a concrete Quack implementation class!

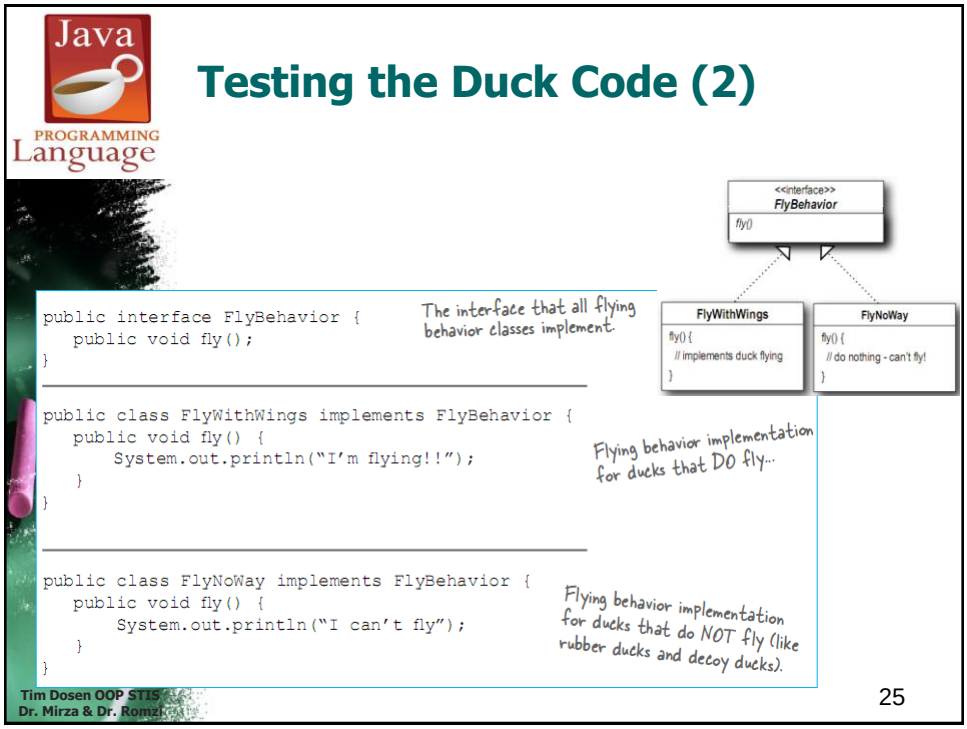
Testing the Duck Code (1)

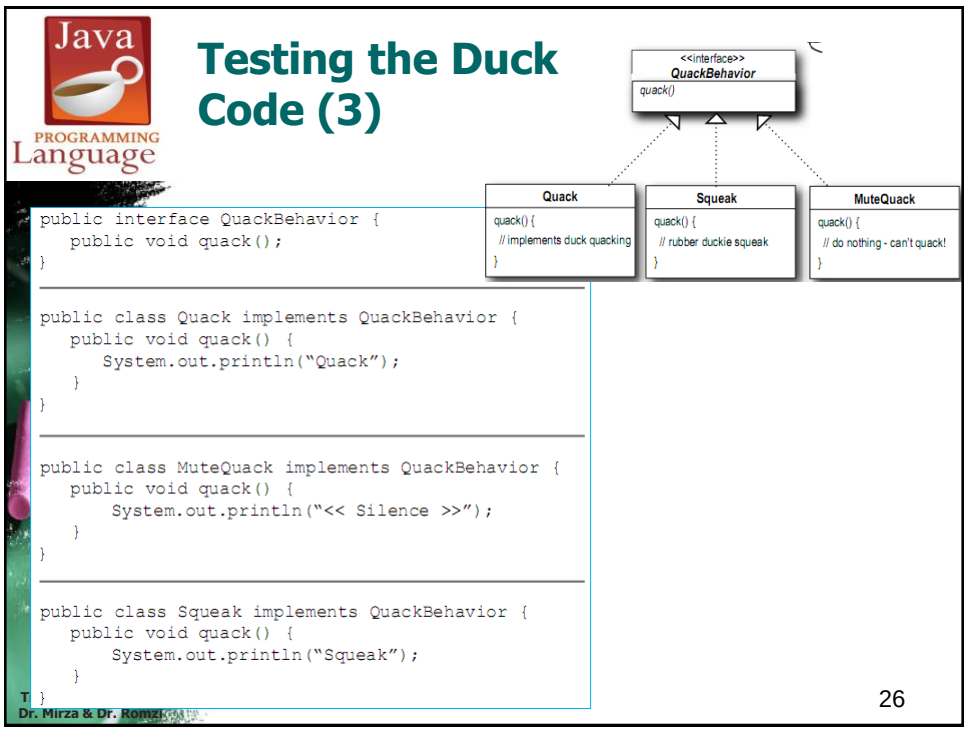
```
public abstract class Duck {

    FlyBehavior flyBehavior;
    QuackBehavior quackBehavior;
    public Duck() {
    }

    public abstract void display();

    public void performFly() {
        flyBehavior.fly();
    }

    public void performQuack() {
        quackBehavior.quack();
    }

    public void swim() {
        System.out.println("All ducks float, even decoys!");
    }
}
```

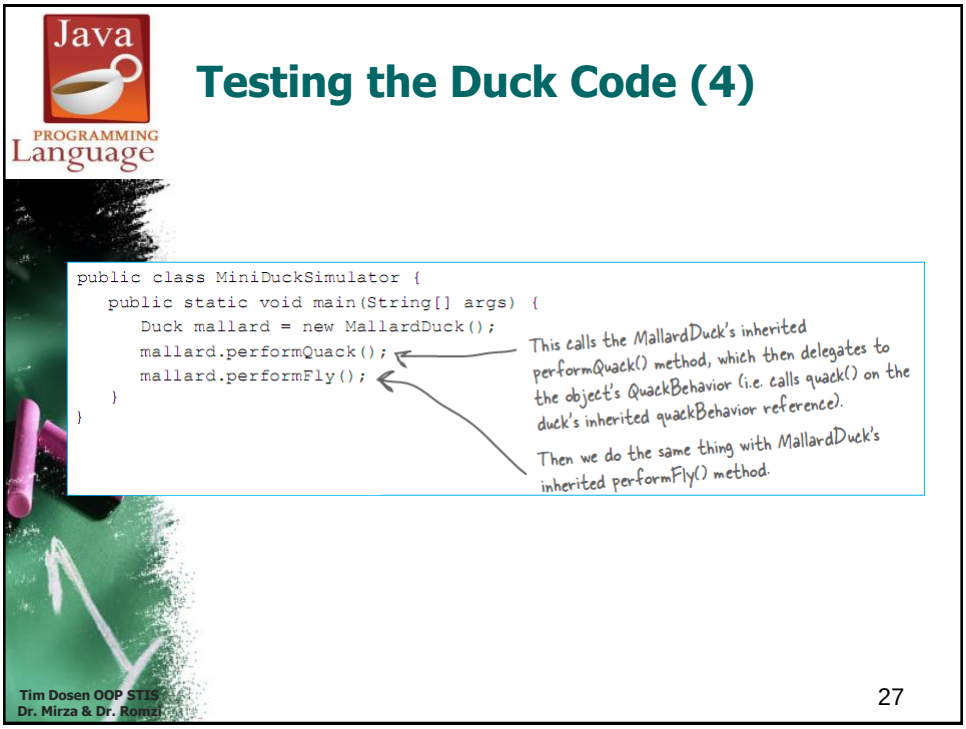
Declare two reference variables for the behavior interface types. All duck subclasses (in the same package) inherit these.

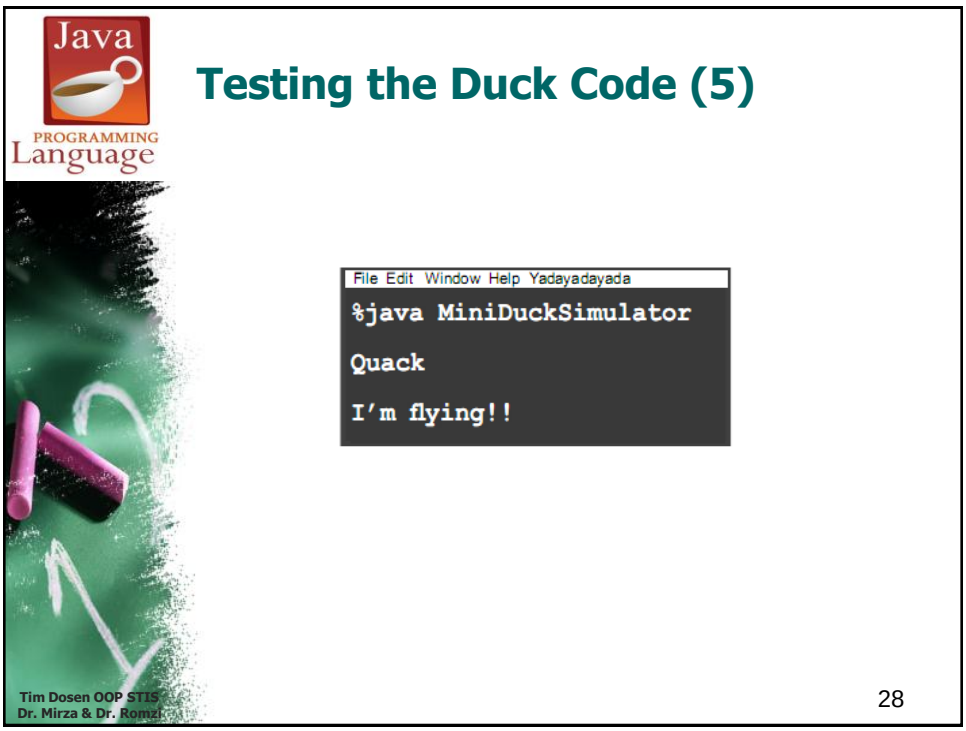
Delegate to the behavior class.




Java
PROGRAMMING
Language
Testing the Duck Code (2)
<<interface>>
FlyBehavior
fly()
FlyWithWings
fly() {
// implements duck flying
}
FlyNoWay
fly() {
// do nothing - can't fly!
}
public interface FlyBehavior {
public void fly();
}
The interface that all flying behavior classes implement.
public class FlyWithWings implements FlyBehavior {
public void fly() {
System.out.println("I'm flying!!");
}
}
Flying behavior implementation for ducks that DO fly...
public class FlyNoWay implements FlyBehavior {
public void fly() {
System.out.println("I can't fly");
}
}
Flying behavior implementation for ducks that do NOT fly (like rubber ducks and decoy ducks).
Tim Dosen OOP STIS
Dr. Mirza & Dr. Romzi
25


Java
PROGRAMMING
Language
Testing the Duck Code (3)
<<interface>>
QuackBehavior
quack()
Quack
quack() {
// implements duck quacking
}
Squeak
quack() {
// rubber duckie squeak
}
MuteQuack
quack() {
// do nothing - can't quack!
}
public interface QuackBehavior {
public void quack();
}
public class Quack implements QuackBehavior {
public void quack() {
System.out.println("Quack");
}
}
public class MuteQuack implements QuackBehavior {
public void quack() {
System.out.println("<< Silence >>");
}
}
public class Squeak implements QuackBehavior {
public void quack() {
System.out.println("Squeak");
}
}
Dr. Mirza & Dr. Romzi
26


Java
PROGRAMMING
Language
Testing the Duck Code (4)
public class MiniDuckSimulator {
public static void main(String[] args) {
Duck mallard = new MallardDuck();
mallard.performQuack();
mallard.performFly();
}
}
This calls the MallardDuck's inherited performQuack() method, which then delegates to the object's QuackBehavior (i.e. calls quack() on the duck's inherited quackBehavior reference).
Then we do the same thing with MallardDuck's inherited performFly() method.
Tim Dosen OOP STIS
Dr. Mirza & Dr. Romzi
27

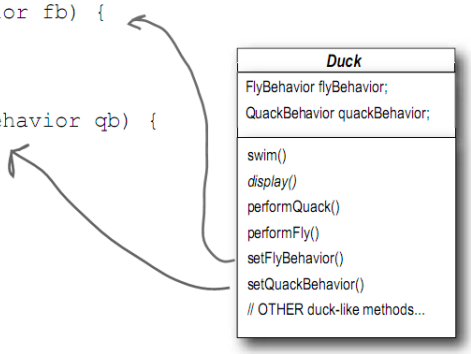

Java
PROGRAMMING
Language
Testing the Duck Code (5)
File Edit Window Help Yadayadayada
%java MiniDuckSimulator
Quack
I'm flying!!
Tim Dosen OOP STIS
Dr. Mirza & Dr. Romzi
28

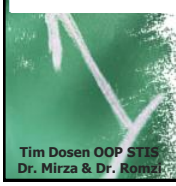
Setting behavior dynamically

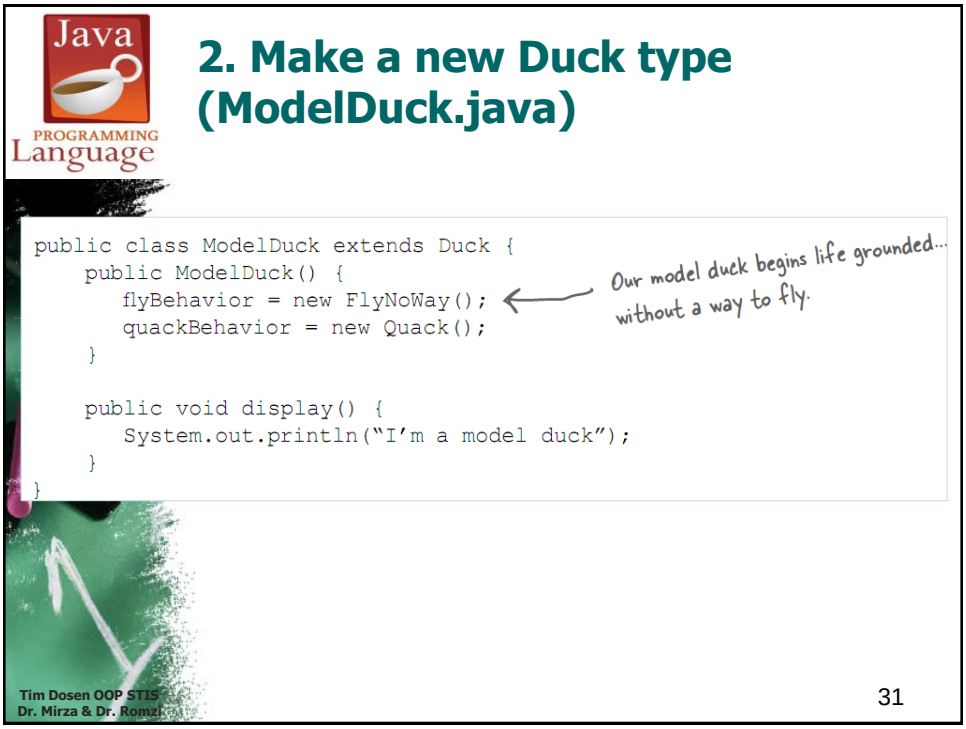
Imagine you want to set the duck's behavior type through a setter method on the duck subclass, rather than by instantiating it in the duck's constructor.

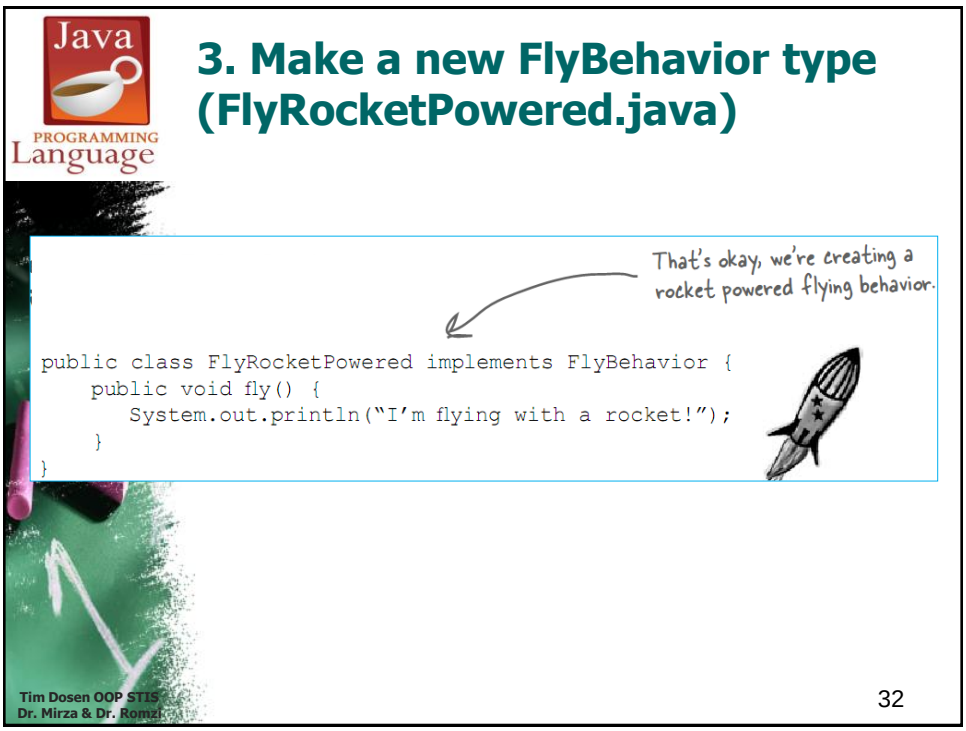
1. Add two new methods to the Duck class

```
public void setFlyBehavior(FlyBehavior fb) {
    flyBehavior = fb;
}

public void setQuackBehavior(QuackBehavior qb) {
    quackBehavior = qb;
}
```






Java
PROGRAMMING
Language
2. Make a new Duck type (ModelDuck.java)
public class ModelDuck extends Duck {
public ModelDuck() {
flyBehavior = new FlyNoWay();
quackBehavior = new Quack();
}
public void display() {
System.out.println("I'm a model duck");
}
}
Our model duck begins life grounded... without a way to fly.
Tim Dosen OOP STIS
Dr. Mirza & Dr. Romzi
31


Java
PROGRAMMING
Language
3. Make a new FlyBehavior type (FlyRocketPowered.java)
That's okay, we're creating a rocket powered flying behavior.
public class FlyRocketPowered implements FlyBehavior {
public void fly() {
System.out.println("I'm flying with a rocket!");
}
}
Tim Dosen OOP STIS
Dr. Mirza & Dr. Romzi
32

4. Change the test class

```
public class MiniDuckSimulator {
   public static void main(String[] args) {
      Duck mallard = new MallardDuck();
      mallard.performQuack();
      mallard.performFly();

      Duck model = new ModelDuck();
      model.performFly();
      model.setFlyBehavior(new FlyRocketPowered());
      model.performFly();
   }
}
```

The first call to performFly() delegates to the flyBehavior object set in the ModelDuck's constructor, which is a FlyNoWay instance.

This invokes the model's inherited behavior setter method, and...voila! The model suddenly has rocket-powered flying capability!

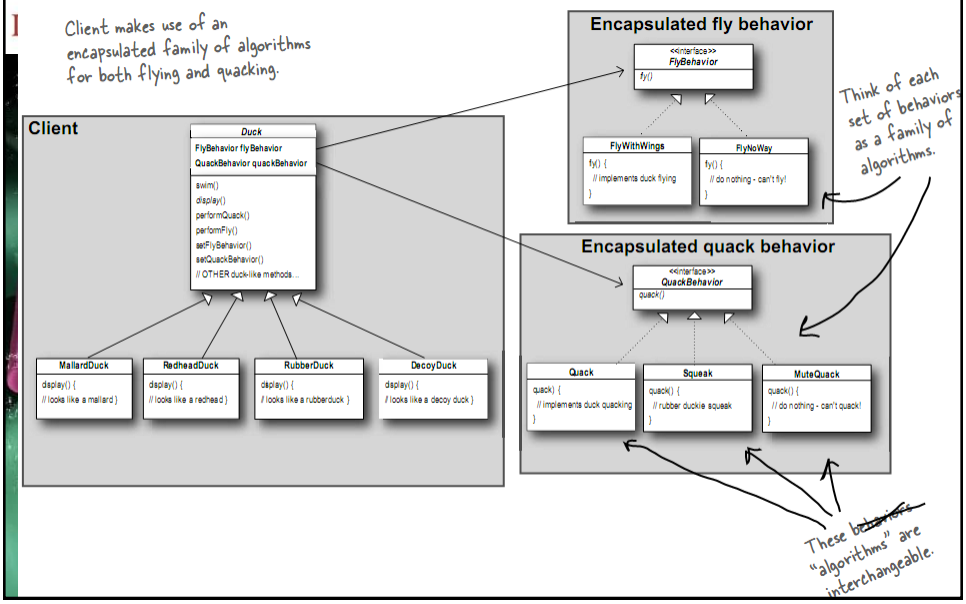
If it worked, the model duck dynamically changed its flying behavior! You can't do THAT if the implementation lives inside the duck class.

Run it!

```
File Edit Window Help Yabadabadoo
%java MiniDuckSimulator
Quack
I'm flying!!
I can't fly
I'm flying with a rocket
```


The Big Picture on encapsulated behaviors

Client makes use of an encapsulated family of algorithms for both flying and quacking.



HAS-A can be better than IS-A

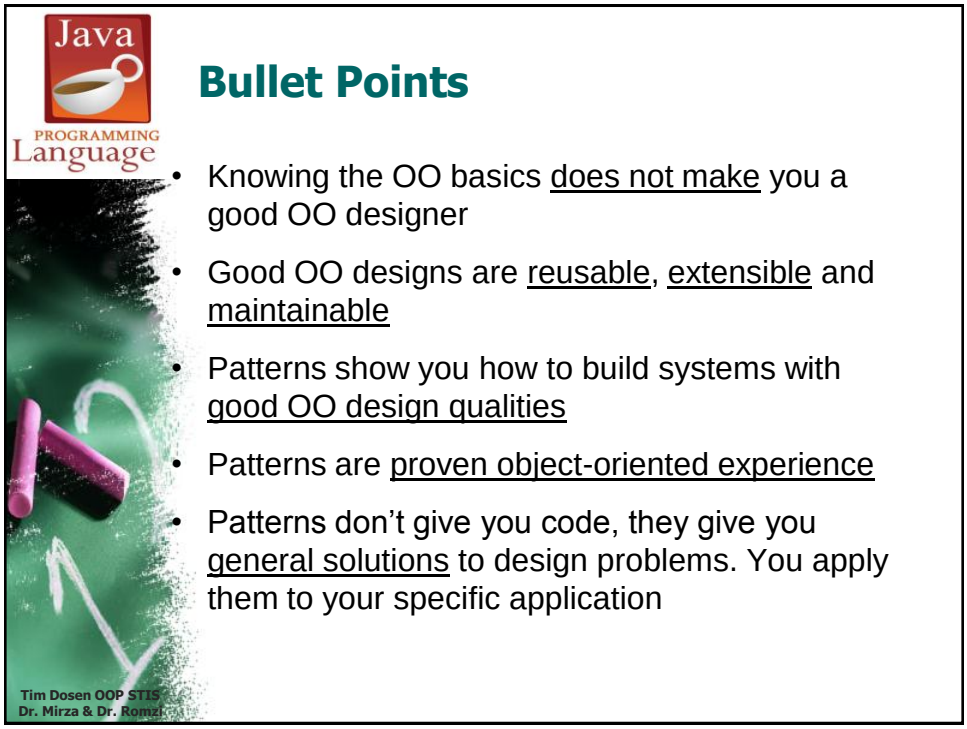
Design Principle

Favor composition over inheritance.

Strategy Pattern

The Strategy Pattern defines a family of algorithms, encapsulates each one, and makes them interchangeable. Strategy lets the algorithm vary independently from clients that use it.


Java
PROGRAMMING
Language
Tools for your Design Toolbox
OO Basics
Abstraction
Encapsulation
Polymorphism
Inheritance
We assume you know the OO basics of using classes polymorphically, how inheritance is like design by contract, and how encapsulation works. If you are a little rusty on these, pull out your Head First Java and review, then skim this chapter again.
OO Principles
Encapsulate what varies.
Favor composition over inheritence.
Program to interfaces, not implementations.
We'll be taking a closer look at these down the road and also adding a few more to the list
Throughout the book think about how patterns rely on OO basics and principles.
OO Patterns
Strategy – defines a family of algorithms, encapsulates each one, and makes them interchangeable. Strategy lets the algorithm vary independently from clients that use it.
Tim Dosen OOP STIS
Dr. Mirza & Dr. Romzi
37


Java
PROGRAMMING
Language
Bullet Points
• Knowing the OO basics does not make you a good OO designer
• Good OO designs are reusable, extensible and maintainable
• Patterns show you how to build systems with good OO design qualities
• Patterns are proven object-oriented experience
• Patterns don't give you code, they give you general solutions to design problems. You apply them to your specific application
Tim Dosen OOP STIS
Dr. Mirza & Dr. Romzi