

CS 635 Advanced Object-Oriented Design & Programming  
Fall Semester, 2018  
Doc 7 Strategy, State  
Sep 18, 2018

Copyright ©, All rights reserved. 2018 SDSU & Roger Whitney, 5500  
Campanile Drive, San Diego, CA 92182-7700 USA. OpenContent (<http://www.opencontent.org/openpub/>) license defines the copyright on this document.

# Strategy Pattern

Favor  
Composition  
over  
Inheritance

# Orderable List

Sorted

Reverse Sorted

Random

OrderableList

SortedList

ReverseList

RandomList

# One size does not fit all



# Issue 1 - Orthogonal Features

Order

Sorted

Reverse Sorted

Random

Threads

Synchronized

Unsynchronized

Mutability

Mutable

Non-mutable

OrderableList

SortedList

ReverseList

RandomList

Mutable

NonMutable

Mutable

NonMutable

Mutable

NonMutable

Synch

Not  
Synch

Synch

Not  
Synch

Synch

Not  
Synch

Synch

Not  
Synch

Synch

Not  
Synch

Synch

Not  
Synch



## Issue 2 - Flexibility



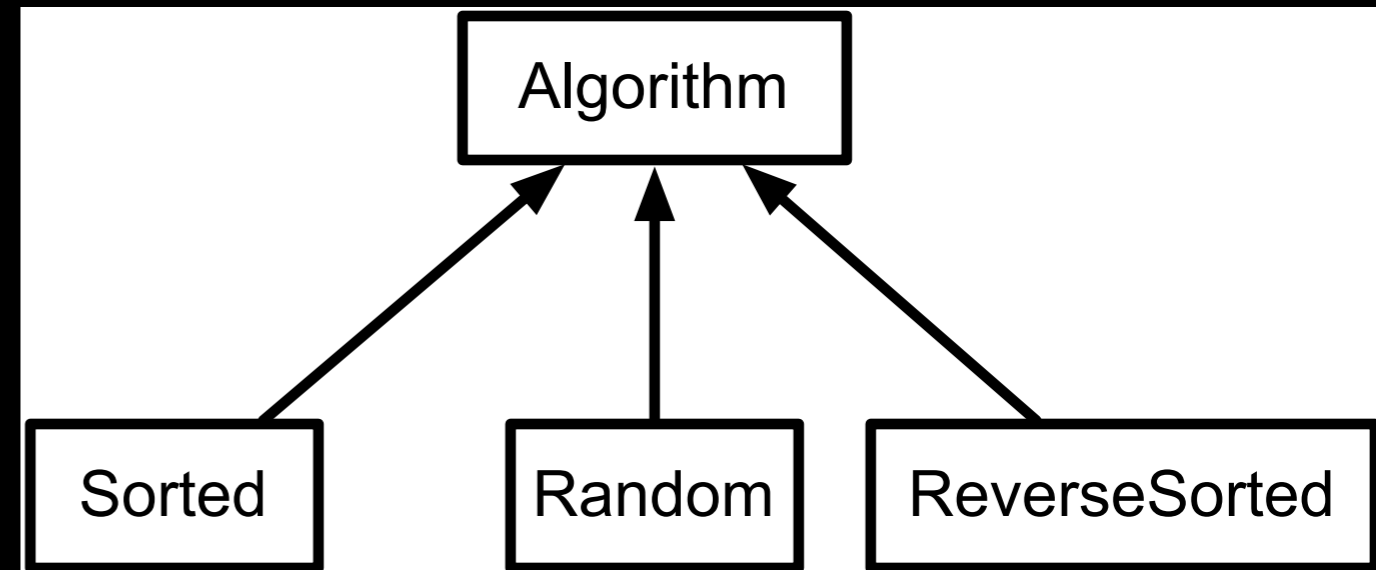
# Change behavior at runtime

```
OrderableList x = new OrderableList();  
x.makeSorted();  
x.add(foo);  
x.add(bar);  
x.makeRandom();
```

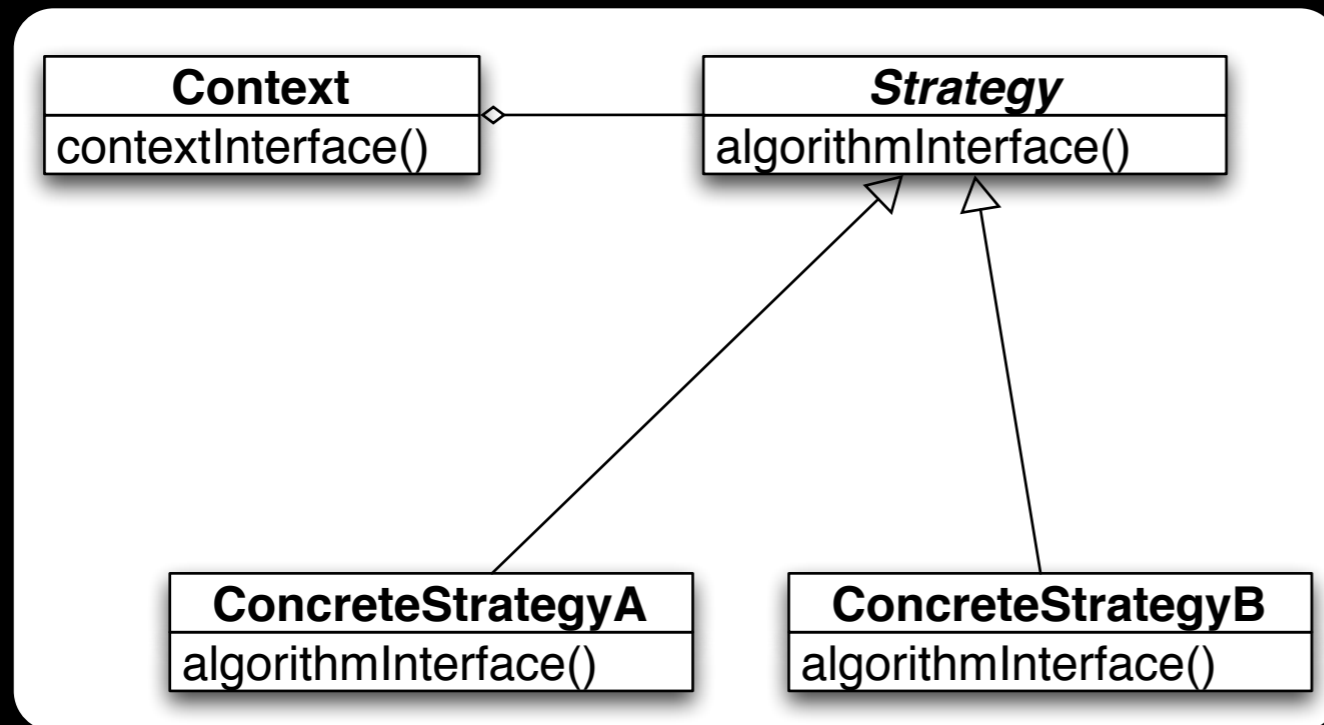
Configure objects behavior at runtime

# Strategy Pattern

```
class OrderableList {  
    private Object[ ] elements;  
    private Algorithm orderer;  
  
    public OrderableList(Algorithm x) {  
        orderer = x;  
    }  
  
    public void add(Object element) {  
        elements = ordered.add(elements,element);  
    }  
}
```



# Structure



The algorithm is the operation

Context contains the data

How does this work?

# Prime Directive Data + Operations



# How does Strategy Get the Data?

Pass needed data as parameters in strategy method

Give strategy object reference to context

Strategy extracts needed data from context



# Example - Java Layout Manager

```
import java.awt.*;
class FlowExample extends Frame {

    public FlowExample( int width, int height ) {
        setTitle( "Flow Example" );
        setSize( width, height );
        setLayout( new FlowLayout( FlowLayout.LEFT) );

        for ( int label = 1; label < 10; label++ )
            add( new Button( String.valueOf( label ) ) );
        show();
    }

    public static void main( String args[] ) {
        new FlowExample( 175, 100 );
        new FlowExample( 175, 100 );
    }
}
```

# Example - Smalltalk Sort blocks

```
| list |
```

```
list := #( 1 6 2 3 9 5 ) asSortedCollection.
```

```
Transcript
```

```
  print: list;
```

```
  cr.
```

```
list sortBlock: [:x :y | x > y].
```

```
Transcript
```

```
  print: list;
```

```
  cr;
```

```
  flush.
```

# Java Sorting

How to sort a Collection in Java?

~~ArrayList~~ List method - `sort(Comparator<? super E> c)`

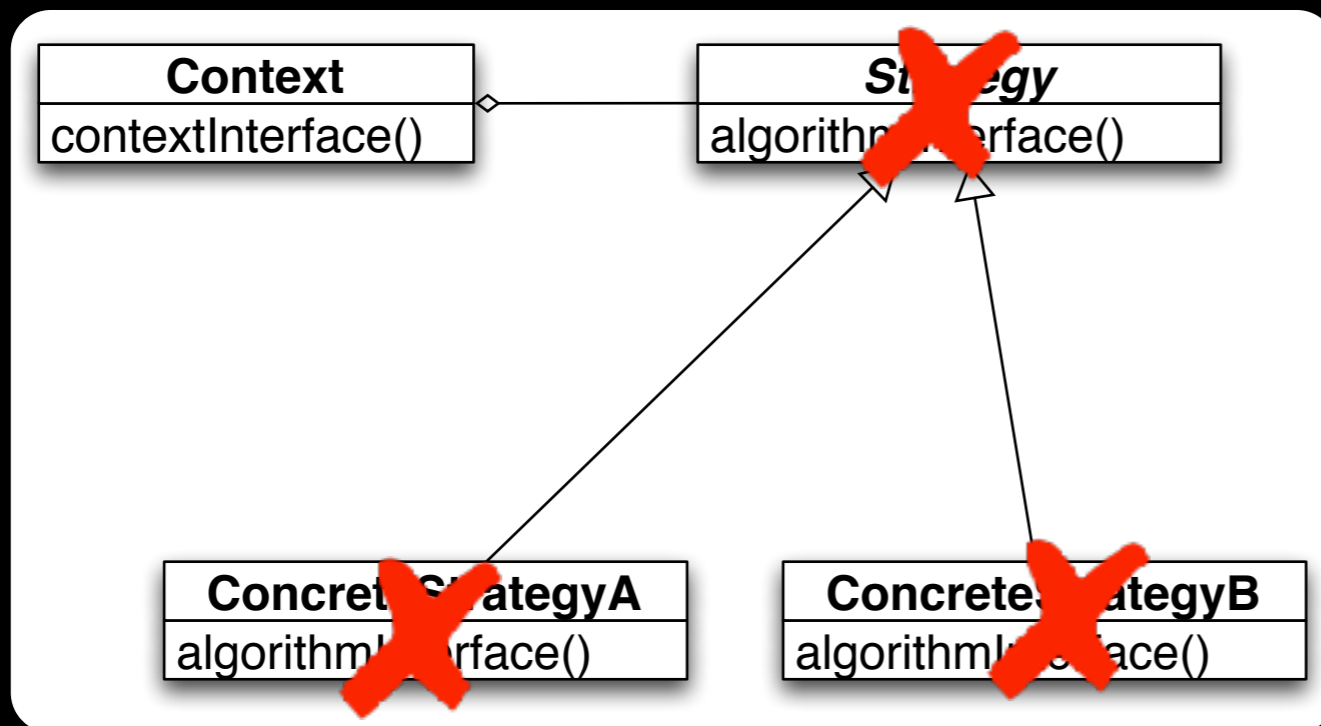
Create a subclass of Comparator

Pass in comparator object to sort method

```
List<Students> students = new ArrayList<>();  
add students  
students.sort(new SortByGPA());
```

# Lambda & Strategy Pattern

If strategy only contains one method  
Can replace Strategy classes with a lambda



In Java may need to define lambda type

# Java Sorting Using Lambda

```
List<Students> students = new ArrayList<>();
```

```
add students
```

```
students.sort( (a, b) -> (a.gpa() <= b.gpa()) ? -1 : 1);
```

# Costs

Clients must be aware of different Strategies

Communication overhead between Strategy and Context

Increase number of objects

# Benefits

Alternative to subclassing of Context

Eliminates conditional statements

Replace in Context code like:

```
switch ( flag ) {  
    case A: doA(); break;  
    case B: doB(); break;  
    case C: doC(); break;  
}
```

With code like:

```
strategy.do();
```

Gives a choice of implementations

# Refactoring: Replace Conditional Logic with Strategy

Conditional logic in a method controls which of several variants of a calculation are executed

so

Create a Strategy for each variant and make the method delegate the calculation to a Strategy instance



# Replace Conditional Logic with Strategy

```
class Foo {  
    public void bar() {  
        switch ( flag ) {  
            case A: doA(); break;  
            case B: doB(); break;  
            case C: doC(); break;  
        }  
    }  
}
```



```
class Foo {  
    private strategy;  
    public void bar() {  
        strategy.do(data);  
    }  
}
```

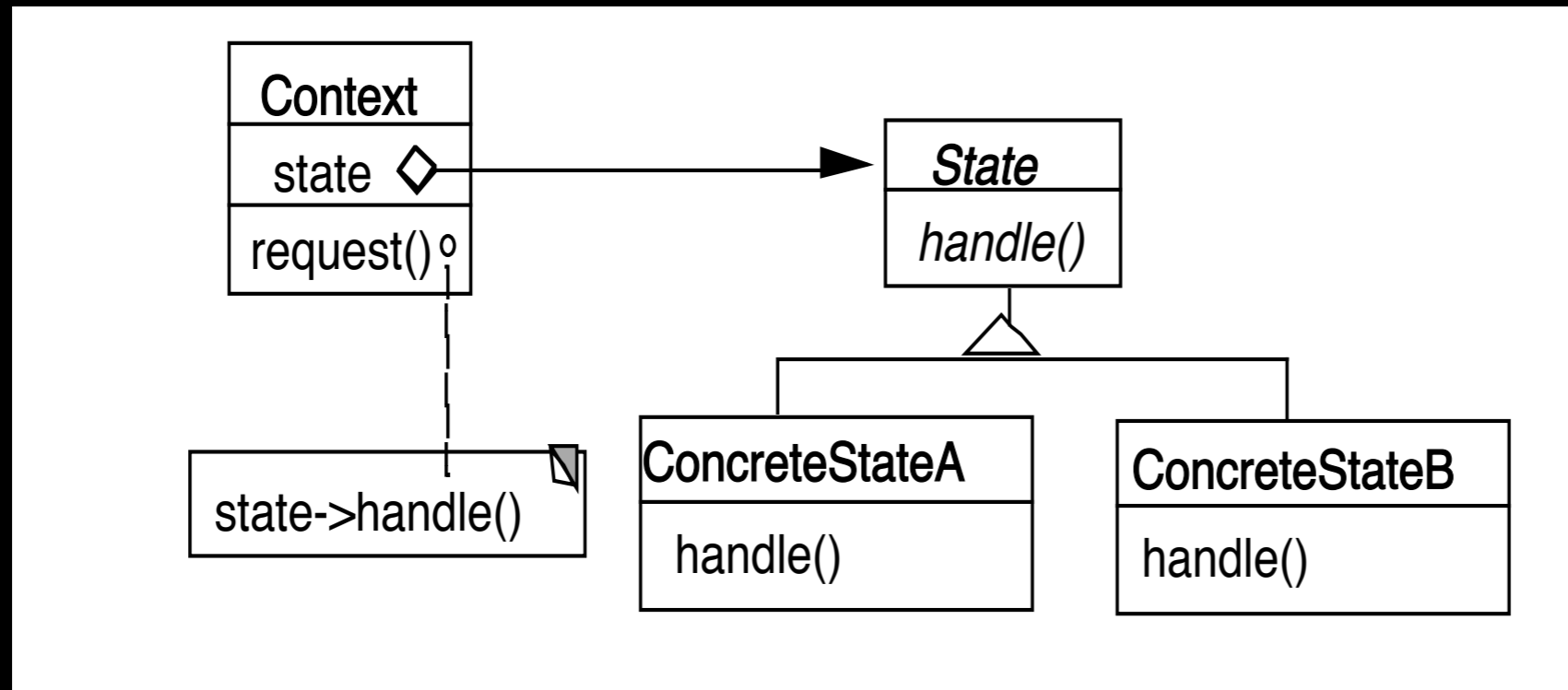
State

# State Pattern

Allow an object to alter its behavior when its internal state changes

The object will appear to change its class

# Structure



# Grade Program

## Operations

View assignment dates

Log in

View grades

Post grades

## States

Not logged in

View dates, Log in

Invalid operations

View & post grades

Logged in - student

View dates & grades

Invalid operations

Post grades, log in

Logged in - instructor

View dates & grades,  
post grades

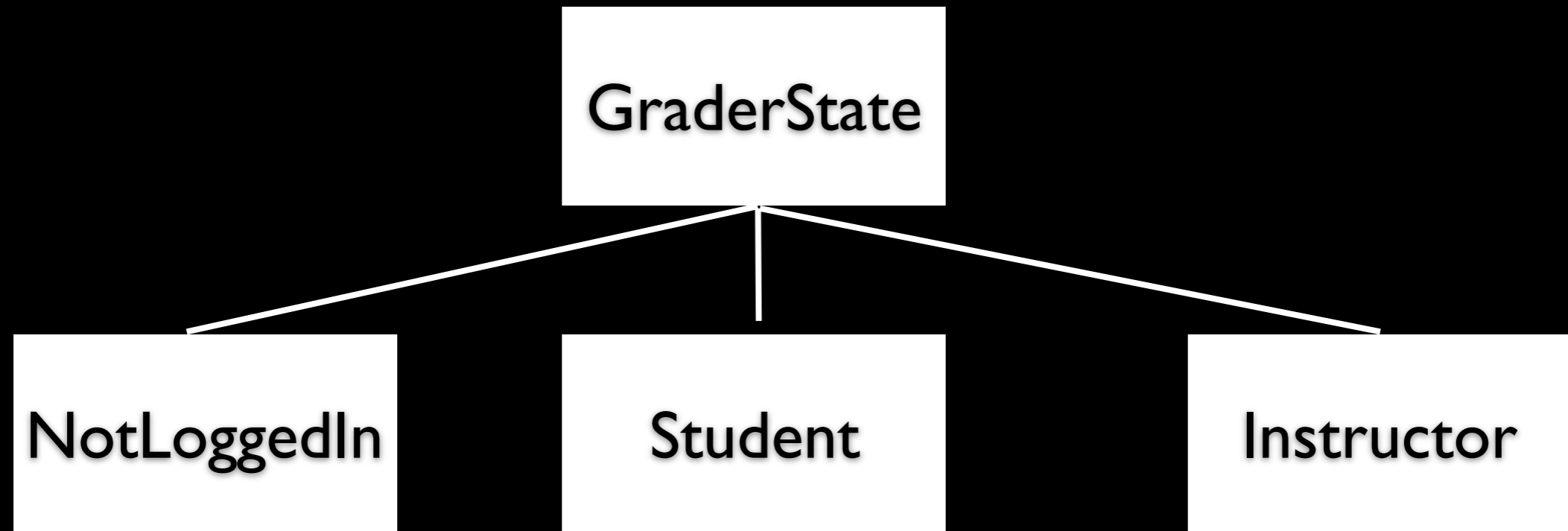
Invalid operations

log in

```
public class Grader {
    static final int NOT_LOGGED_IN = 0;
    static final int STUDENT = 1;
    static final int INSTRUCTOR = 2;
    int state = NOT_LOGGED_IN;
```

```
    public viewGrades() {
        if (state == NOT_LOGGED_IN)
            redirectToLogin();
        if (state == STUDENT)
            showStudentGrade();
        if (state == INSTRUCTOR)
            showAllGrades();
    }
}
```

```
    public postGrades() {
        if (state == NOT_LOGGED_IN)
            redirectToLogin();
        if (state == STUDENT)
            showError();
        if (state == INSTRUCTOR)
            getGradeFile();
    }
}
```



```
public class GraderState {  
    public GraderState login() {...}  
    public GraderState viewGrades() {}  
    public GraderState postGrades() {}  
}
```

```

public class Grader {
    GraderState state = new NotLoggedIn();

    public void login() {
        state = state.login();
    }

    public viewGrades() {
        state = state.viewGrades();
    }
}

```

```

public class Student : GraderState {
    public GraderState login() {
        displayError(); // already login
        return self;
    }

    public GraderState viewGrades() {
        fetchStudentsGrades();
        display
    }

    public GraderState postGrades() {
        logStudentAttempt();
        displayError();
        return this;
    }
}

```



# Who defines state Transitions - Context

```
class Context {  
    private AbstractState state = new StartState();  
  
    public Bar foo(int x) {  
        int result = state.foo(x);  
        if (someConditionHolds() )  
            state = nextState();  
        return result;  
    }  
}
```

# Who defines state Transitions - States

```
class Context {  
    private AbstractState state = new StartState();  
  
    public void foo(int x) {  
        state = state.foo(x);  
    }  
}
```

What if foo returns a value?

# Who defines state Transitions - States

```
class Context {  
    private AbstractState state = new StartState();  
  
    public int foo(int x) {  
        return state.foo(x, this);  
    }  
  
    protected void setState(AbstractState newState) {  
        state = newState;  
    }  
}
```

# Sharing State Objects

Stateless state

- State objects without fields

- Can be shared by multiple contexts

Can store data in context and pass as arguments

Large number of state transitions can be expensive

- Only create state once & reuse same object

# State Verses Strategy

Rate of Change

## **Strategy**

Context usually contains just one strategy object

## **State**

Context often changes state objects

# State Verses Strategy

Exposure of Change

## Strategy

Strategies all do the same thing

Client do not see change in behavior of Context

## State

States act differently

Client see the change in behavior

# Changing Class - No Need for Context

Language Dependent Feature

Smalltalk & Lisp

```
class Truthful extends Oracle {  
  
    public boolean foo(int x) {  
        int result = state.foo(x);  
        this.changeClassTo(Random);  
        return result;  
    }  
}
```

# Java/C++/C# Example - Single Dispatch

```
public class Parent {  
}
```

```
public class Child extends Parent {  
}
```

```
public class Bar {
```

```
public foo(Parent x) {  
    return "Parent";  
}
```

```
public foo(Child x) {  
    return "Child";  
}
```

```
Bar test = new Bar();
```

```
Parent x = new Parent();  
test.foo(x);
```

Parent

```
x = new Child();  
test.foo(x);
```

Parent

```
Child y = new Child();  
test.foo(y);
```

Child

Actual type of foo's argument is not used  
Just the declared type

Only runtime selection(dispatch) is on  
type actual type of receive of method



# Multiple Dispatch - Julia

```
foo(a::Integer,b::Integer) = "Integer,Integer"
```

```
foo(a::Integer,b::Number) = "Integer,Number"
```

```
foo(a::Number,b::Integer) = "Number,Integer"
```

```
foo(a::Number,b::Number) = "Number,Number"
```

```
foo(a::Number,b::Complex) = "Number,Complex"
```

```
foo(1,2) Integer,Integer
```

```
foo(1,2.3) Integer,Number
```

```
foo(2//3,1) Number,Integer
```

```
foo(2.3,2im + 2) Number,Complex
```

# Why Important

```
function power(n::Number,exponent::Real)
    Some complicated process for float exponent
end
```

```
function power(n::Number,exponent::Complex)
    Deal with complex exponent
end
```

```
function power(n::Number, exponent::Integer)
    if exponent == 0
        return 1
    result = n
    for k in 1:n-1
        result *= n
    end
    result
end
```

# Open & Closed

A module is open if can

Add/remove fields

Add/remove methods

A module is closed if

It can be used by other modules

# Open-Close Principle

Module should be open for extension

But closed for modification

# Multiple Dispatch & State Pattern - Julia

```
function viewGrades(user::NotLoggedIn)
    goToLoginPage(user)
end
```

```
function viewGrades(user::Student)
    getAndDisplayGrades(user)
end
```

# Multiple Dispatch & State Pattern - Clojure

```
(defmulti view-grades (fn [user] (:state user)))
```

```
(defmethod view-grades :not-logged-in  
  [user]  
  (go-to-log-in-page user))
```

```
(defmethod view-grades :student  
  [user]  
  (student-grade user))
```

```
(defmethod view-grades :instructor  
  [user]  
  (all-course-grades user)))
```