ITEC 136

Business Programming Concepts

Week 13, Part 01

Overview

FRANKLIN UNIVERSITY

FOUNDED 1902

Week 13 Overview

- Week 12 review
  - Sorting algorithms for arrays
    - Selection sort
    - Insertion sort
    - Bubble sort
  - Multi-dimensional arrays
    - An array that holds other arrays as data.

Week 13 Overview

- Outcomes
  - List the benefits of object-orientation.
  - Describe classes, methods, and encapsulation and the mechanisms used to implement them.

Week 13 Overview

- Outcomes
  - Apply the principles of encapsulation to solve a given problem.
  - Explain exception handling for error detection and correction.

ITEC 136

Business Programming Concepts

Week 13, Part 02

Object Oriented Concepts

FRANKLIN UNIVERSITY

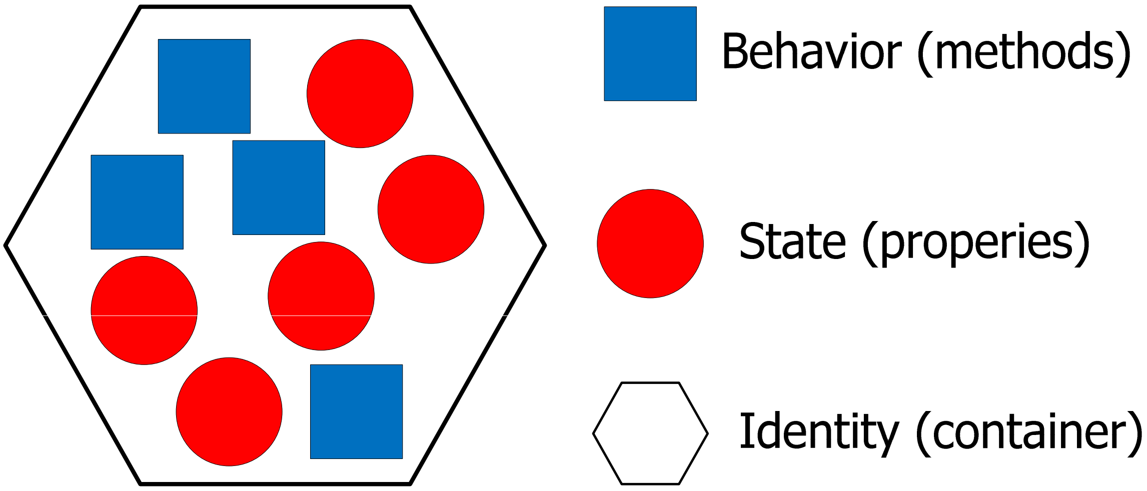
FOUNDED 1902

Object Oriented Concepts

- What is an object?
  - All objects have 3 characteristics
    - **State** – data associated with the object
    - **Behavior** – code associated with the object
    - **Identity** – a location where the object exists in memory

Object Oriented Concepts

- What is an object

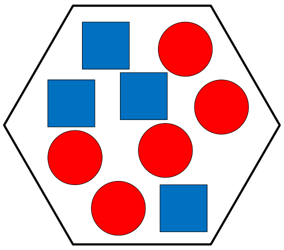


🟦 Behavior (methods)

● State (properies)

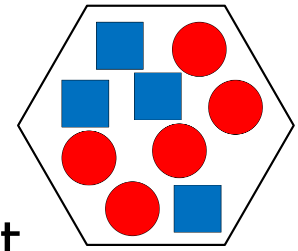
⬡ Identity (container)

Object Oriented Concepts



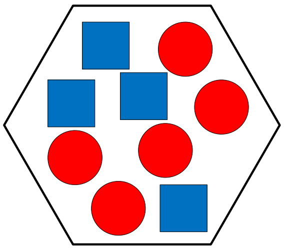
- What is an object?
  - State (properties)
    - Data kept inside the object.
    - The internal representation of the object need not be the same as how it is seen from the outside.
    - Ex: `Date` object in JS represents a date and time as a number of milliseconds elapsed since January 1, 1970.

Object Oriented Concepts



- What is an object?
  - Behavior (method)
    - A function kept inside an object
    - Has access to all the properties of the object as well as any parameters and global variables.

Object Oriented Concepts



- What is an object?
  - Identity (container)
    - Memory location of the object.
    - One variable that holds many other variables (methods and properties) within itself.
    - Very similar to an associative array. In fact, all JS objects are associative arrays.

ITEC 136

Business Programming Concepts

Week 13, Part 03

Custom Objects in JS

FRANKLIN UNIVERSITY

FOUNDED 1902

Custom Objects in JS

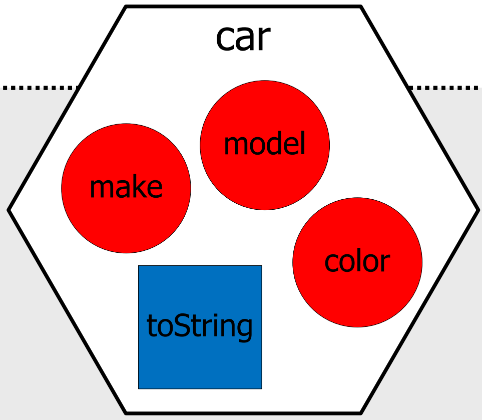
- Let’s build an object!

```
var car = new Object()
car.make = "Chevy";
car.model = "Corvette";
car.color = "Red";
car.toString = function()
{
  return this.color + " " + this.make
    + " " + this.model;
}
alert(car.toString());
```

Custom Objects in JS

- Let’s build an object!

```
var car = new Object()
car.make = "Chevy";
car.model = "Corvette";
car.color = "Red";
car.toString = function()
{
  return this.color + " " + this.make
    + " " + this.model;
}
alert(car.toString());
```



Custom Objects in JS

- Let’s build an object!

```
var car = new Object()
car.make = "Chevy";
car.model = "Corvette";
car.color = "red";
car.toString = function()
{
  return this.color + " " + this.make
    + " " + this.model;
}
alert(car.toString());
```

`make`, `model`, and `color` are properties (state) within the object.

`toString` is a method (behavior) of the object. Notice different syntax!

Within a method, the keyword “`this`” refers to the current object (`car` in this case)

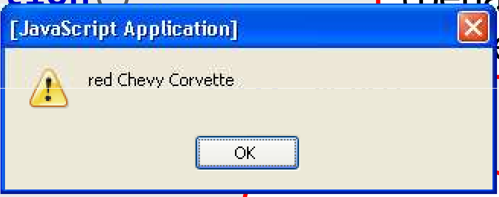
Custom Objects in JS

- Let’s build an object!

```
var car = new Object()
car.make = "Chevy";
car.model = "Corvette";
car.color = "red";
car.toString = function()
{
  return this.color
    + " " + this.mo
}
alert(car.toString());
```

make, model, and color are properties (state) within the object.

toString is a method (behavior) of the object.
e different syntax!



red Chevy Corvette

Within a method, the keyword “this” refers to the current object (car in this case)

Custom Objects in JS

- Let’s make it easier to build objects!
  - Try this: write a function called makeCar that receives a make, model, and color as parameters and returns a car with those properties set and a valid toString() method that reports the state.

Custom Objects in JS

- Solution:

```
function makeCar(make, model, color) {
    var result = new Object();
    result.make = make;
    result.model = model;
    result.color = color;
    result.toString = function() {
        // on next slide
    }
    return result;
}
```

Custom Objects in JS

- Solution:

```
result.toString = function() {
    var str = "";
    for (property in this) {
        if (typeof this[property] != "function")
            str += property + ": " +
               this[property] + "\n";
    }
    return str;
}
```

Custom Objects in JS

- Solution:

```
result.toString = function() {
    var str = "";
    for (property in this) {
        if (typeof this[property] != "function")
            str += property + ": " +
               this[property] + "\n";
    }
    return str;
}
```

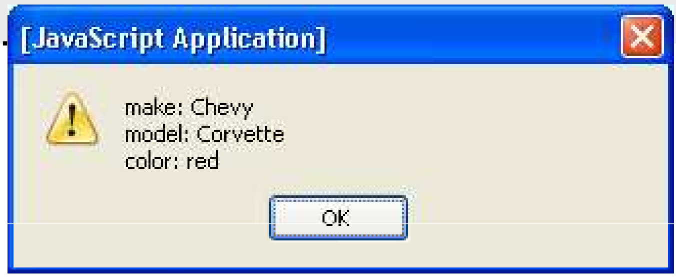
Prevents us from seeing the code of the `toString` function itself.

Custom Objects in JS

- Solution:

```
var car = makeCar("Chevy", "Corvette", "red");
alert(car);
```

Automatically calls the `toString` method.



```
make: Chevy
model: Corvette
color: red
```

Custom Objects in JS

- Let’s improve our object
  - What we want is to create a car object using the keyword **new**:

```
var car = new Car("Chevy", "Corvette", "red");
alert(car);
```

  - Change the name and structure of makeCar.

Custom Objects in JS

- Let’s improve our object!

```
function Car(make, model, color)
{
    this.make = make;
    this.model = model;
    this.color = color;
    this.toString = function()
    {
        // same code as before
    }
}
```

Custom Objects in JS

- Let’s improve our c

```
function Car(make, model, color)
{
    this.make = make;
    this.model = model;
    this.color = color;
    this.toString = function()
    {
        // same code as before
    }
}
```

Name of the function has changed to conform to naming conventions.

Get rid of creating an object and instead assign everything into `this`.

Notice, no return value whatsoever. We’ve build a *constructor*.

Custom Objects in JS

- One final improvement
  - Each car we build has its ***own*** deep copy of the `toString` function. It would be better if there were one ***shared*** shallow copy of the function.
  - Use ***prototypes*** to create shared code in an object.

Custom Objects in JS

- One final improvement

```
function Car(make, model, color) {
    this.make = make;
    this.model = model;
    this.color = color;
}

Car.prototype.toString = function() {
    // same code as before
}
```

Custom Objects in JS

- One final improvement

```
function Car(make, model, color) {
    this.make = make;
    this.model = model;
    this.color = color;
}

Car.prototype.toString = function() {
    // same code as before
}
```

`prototype` is a property of every function (remember, functions are objects too).

Custom Objects in JS

- What is `prototype`?
  - Every constructor function has a property called `prototype`.
  - Anything assigned into `prototype` is automatically received by every object constructed with that function.

Custom Objects in JS

- Ex: A deep array copy

```
Array.prototype.clone = function() {
    var result = new Array(this.length);
    for (i in this) {
        if (this[i] instanceof Array)
            result[i] = this[i].clone();
        else
            result[i] = this[i];
    }
    return result;
}
var arr1 = [1, [2, 3, 4], [5, 6, 7, [8]]];
var arr2 = arr1.clone(); // make a deep copy
```

Custom Objects in JS

- Ex: A deep array copy

```
Array.prototype.clone = function() {
    var result = new Array(this.length);
    for (i in this) {
        if (this[i] i      ceof Array)
            result[i]         [i].clone();
        else
            result
    }
    return result;
}
var arr1 = [1, [2, 3, 4], [5, 6, 7, [8]]];
var arr2 = arr1.clone(); // make a deep copy
```

`clone` is now a function that can be called on *all* arrays, even those created before this code was executed.

Custom Objects in JS

- What is `prototype`?
  - Its an object, and a property of the constructor function. As an object, it can have data and functions within it.
  - All instances share the prototype, and thus any functions within it.

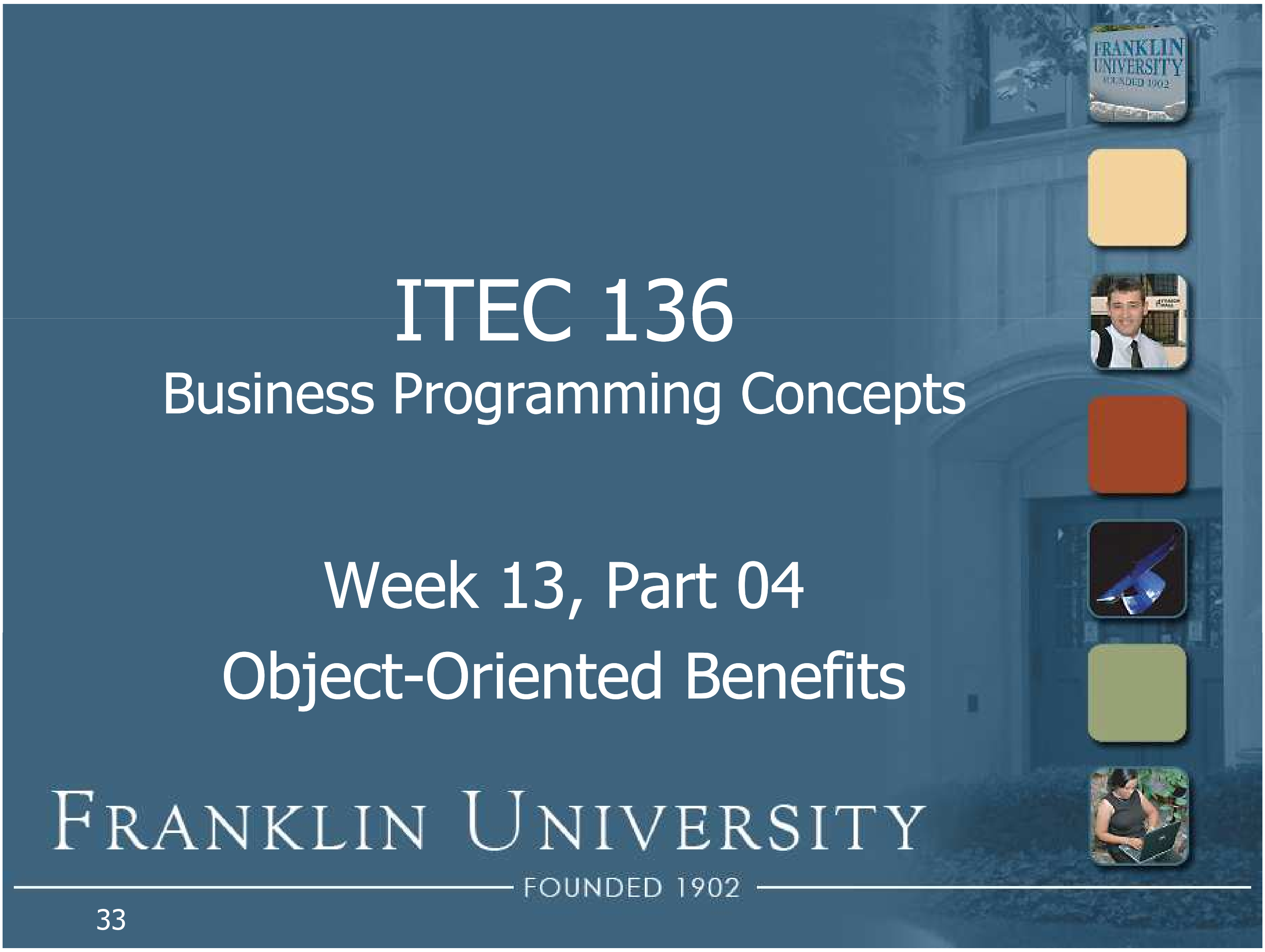
Custom Objects in JS

- Benefits of what we've done:
  - Can reuse the code many times for many different `Car` objects.

```
var car1 = new Car("Toyota", "Prius", "blue");
var car2 = new Car("Chevy", "Corvette", "red");
alert(car1);
alert(car2);
```

Custom Objects in JS

- Benefits of what we've done:
  - Can reuse the code many times for many different `Car` objects.
  - All the data and functions for a `Car` are kept in one single unit.
  - All `Car` objects share their `toString` method (i.e. only one copy exists in memory).



ITEC 136

Business Programming Concepts

Week 13, Part 04

Object-Oriented Benefits

FRANKLIN UNIVERSITY

FOUNDED 1902

Object-Oriented Benefits

- Many benefits to grouping data and methods together:
  - **Increased modularity**: the unit of modularity becomes the object and systems become a set of cooperating objects. Objects are typically smaller, and therefore there are more modules.

Object-Oriented Benefits

- Many benefits to grouping data and methods together:
  - **Simplified analysis**: The real world consists of objects. In the real world, objects have attributes and behaviors. When the method of programming and the real world align, then the process of analyzing the problem becomes simpler.

Object-Oriented Benefits

- Many benefits to grouping data and methods together:
  - **Easier testing**: With increased modularity (i.e. smaller, more tightly focused objects) comes easier testing of those objects. Tests can be written to validate the behavior of each object independently of the entire system.

Object-Oriented Benefits

- Many benefits to grouping data and methods together:
  - **Increased comprehension**: Since objects are kept small (on the order of perhaps a couple of hundred lines of code) programmers are better able to keep the entire state of the object in their working memory at once.

Object-Oriented Benefits

- Many benefits to grouping data and methods together:
  - **Looser coupling**: *Coupling* is a measure of the degree to which a class depends on other classes to work properly. It is rare that an object acts in isolation of other objects, the connections between objects are clearly defined by the methods.

Object-Oriented Benefits

- Many benefits to grouping data and methods together:
  - **Tighter cohesion**: *Cohesion* is a measure of the degree to which a class models a single concept. Objects are smaller modules of modeling than those found in non-object oriented systems, and hence tend to promote tighter cohesion.

Object-Oriented Benefits

- Many benefits to grouping data and methods together:
  - **Increased reuse**: Because objects are loosely coupled and highly cohesive, they are easier to reuse within the same or different systems.

Object-Oriented Benefits

- Many benefits to grouping data and methods together:
  - **Better maintainability**: All of the aforementioned benefits lead to systems that are much more flexible to change and much easier to fix when bugs are encountered.

FRANKLIN UNIVERSITY

ITEC 136

Business Programming Concepts

Week 13, Part 05

The 5 Pillars of OOP

FRANKLIN UNIVERSITY

FOUNDED 1902

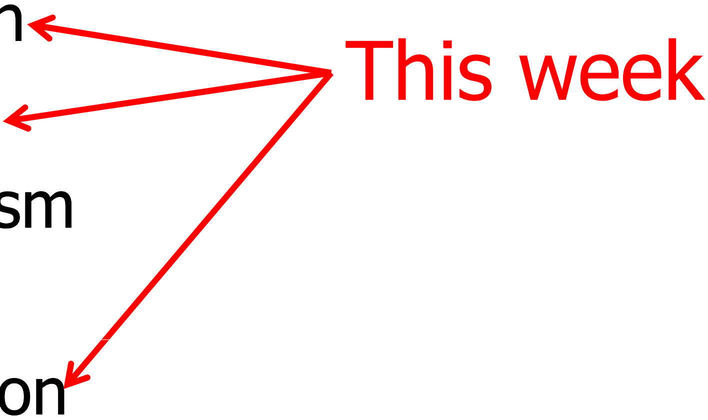
The 5 Pillars of OOP

- Five key concepts in OOP
  - Composition
  - Abstraction
  - Polymorphism
  - Inheritance
  - Encapsulation

C a pie.

The 5 Pillars of OOP

- Five key concepts in OOP
  - Composition
  - Abstraction
  - Polymorphism
  - Inheritance
  - Encapsulation



This week

The 5 Pillars of OOP

- Five key concepts in OOP
  - Composition
  - Abstraction
  - Polymorphism
  - Inheritance
  - Encapsulation

Not covered ☹

The 5 Pillars of OOP

- Abstraction
  - Process of reading a real-world problem description and figuring out how to model it using objects, methods, and properties.

The 5 Pillars of OOP

- Abstraction
  - Nouns can become objects or properties.
  - Verbs can become methods.

The 5 Pillars of OOP

- Abstraction
  - Try it: “A calculator consists of several buttons for entering numbers and several more buttons for entering operations on those numbers. Valid arithmetic operations are add, subtract, multiply, and divide. The equals button displays the current result.”

The 5 Pillars of OOP

- Abstraction
  - **Objects**: calculator
  - **Properties**: current result, buttons
  - **Methods**: add, subtract, multiply, divide, equals

The 5 Pillars of OOP

- Composition/Aggregation
  - Using one or more objects as properties within another object (i.e. objects within objects).
  - Called the "has-a" relationship.
  - Not unusual at all (strings are objects, and they were properties of our `Car` object built previously).

The 5 Pillars of OOP

- Composition/Aggregation
  - Two forms of “has-a”
    - **Aggregation**: the two objects can exist independently of one another, but happen to be connected. *Ex: classes and students*
    - **Composition**: a “whole-part” relationship where the contained object can’t reasonably exist apart from the container. *Ex: students and dates of birth*

FRANKLIN UNIVERSITY

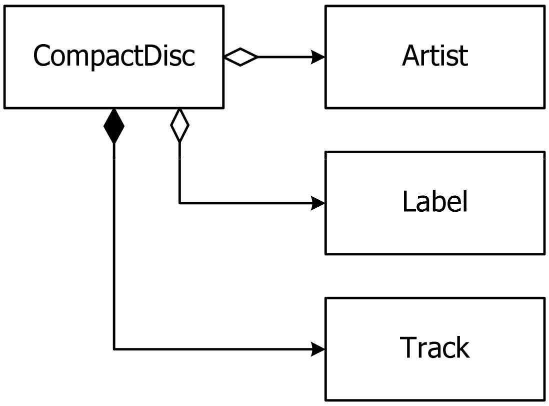
The 5 Pillars of OOP

- Composition
  - Try it: Show the relationships between `CompactDisc`, `Track`, `Artist`, and `Label`.

FRANKLIN UNIVERSITY

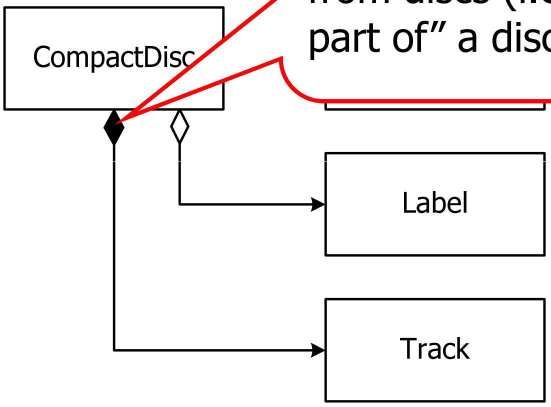
The 5 Pillars of OOP

- Composition
  - Try it: Show the relationships between `CompactDisc`, `Track`, `Artist`, and `Label`.



The 5 Pillars of OOP

- Composition
  - Try it: Show the relationships between `CompactDisc`, `Track`, `Artist`, and `Label`.

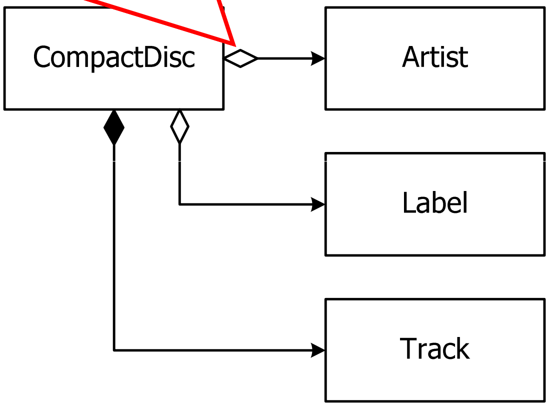


Composition: Filled diamond. Tracks don’t exist separately from discs (i.e. tracks “are a part of” a disc).

The 5 Pillars of OOP

- Composition
  - ationships between
    k, Artist, and
    Label.

Aggregation: Hollow diamond. Artists exist as an entity separate from discs. But a disc "has an" artist.



The 5 Pillars of OOP

- Encapsulation
  - Hiding the implementation details of an object (i.e. the properties and code) behind a simple *interface* defined by the methods.
  - Ex: String objects. Don't know how they work internally, but we have a well defined interface through the API.

The 5 Pillars of OOP

- Encapsulation
  - Try it: A television is a well encapsulated real-world object. What is its interface?

The 5 Pillars of OOP

- Encapsulation
  - Try it: A television is a well encapsulated real-world object. What is its interface?

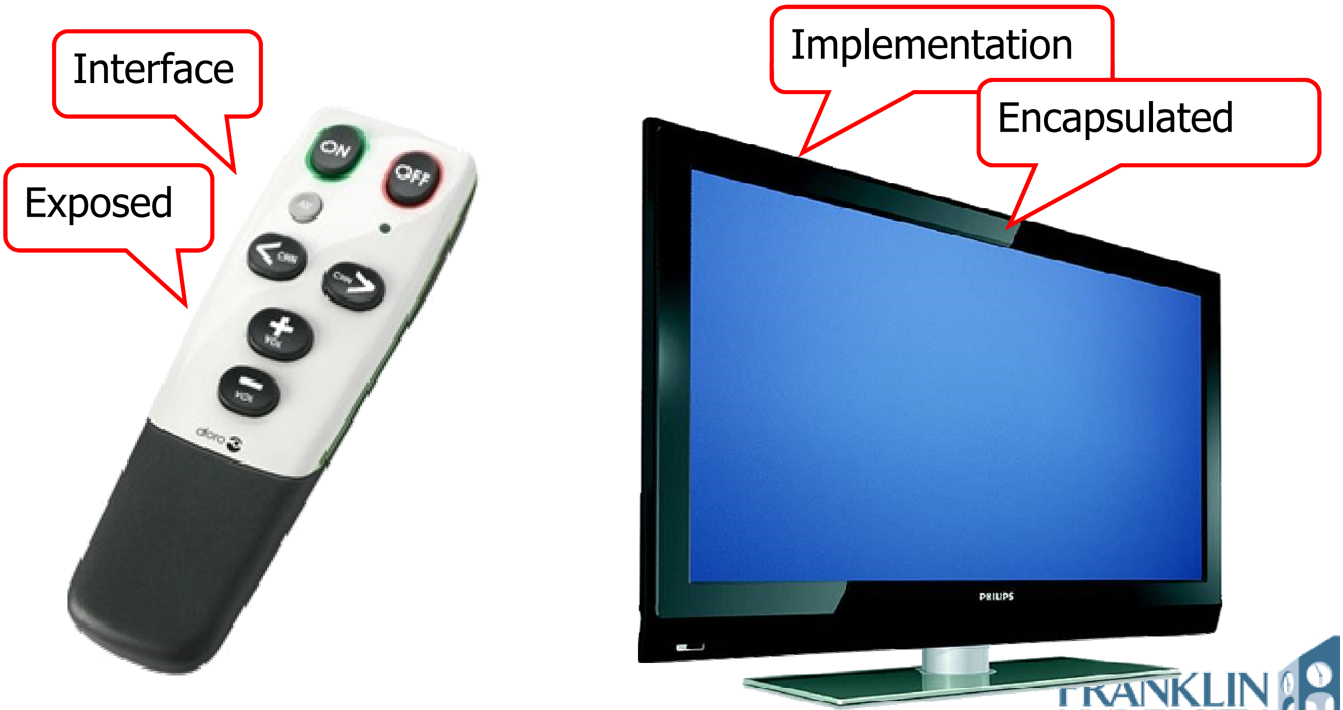
  - Simplest interface: Channel up, channel down, volume up, volume down, power toggle, mute (maybe).

The 5 Pillars of OOP

- Encapsulation
  - Try it: A television is a well encapsulated real-world object. What is its interface?
  - Simplest interface: Channel up, channel down, volume up, volume down, power toggle, mute (maybe).

The 5 Pillars of OOP

The 5 Pillars of OOP



Interface

Exposed

Implementation

Encapsulated

ITEC 136

Business Programming Concepts

Week 13, Part 06

Exception handling

FRANKLIN UNIVERSITY

FOUNDED 1902

Exception Handling

- How do errors get processed?
  - Old way: lots of if/else cases, checking the return values of functions
    - Functions return `true` if everything went as expected.
    - Functions return `false` if something went wrong.
  - Problem: detecting vs. correcting

Exception Handling

- Detecting vs. correcting
  - Can usually *detect* the error in one section of code, but not be able to *correct* it in the same place.
    - Callee function can detect
    - Caller function can correct
  - How does the error get communicated from the callee to the caller?

Exception Handling

- Detecting errors

```
HourlyEmployee.prototype.setHoursWorked = function(hours)
{
    // Impossible number of hours.
    if (hours < 0 || hours > 24*7) {
        // what to do here?
    }
    else {
        this.hoursWorked = hours;
    }
}
```

Can detect a bad parameter here, but can’t correct for it.

Exception Handling

- Detecting errors: solution

```
HourlyEmployee.prototype.setHoursWorked = function(hours)
{
    // Impossible number of hours.
    if (hours < 0 || hours > 24*7) {
        throw "Bad parameter for hours: " + hours;
    }
    else {
        this.hoursWorked = hours;
    }
}
```

“throw” an error back to the caller. Execution immediately stops. You can throw *any* object.

Exception Handling

- Correcting errors

```
var emp = new HourlyEmployee();
var hours = parseInt(prompt("Enter hours worked"));
emp.setHoursWorked(hours);
```

How do we handle a potential bad input here?

Exception Handling

- Correcting errors: solution

```
var emp = new HourlyEmployee();
var done = false;
while (!done) {
    done = true;
    var hours = parseInt(prompt("Enter hours worked"));
    try {
        emp.setHoursWorked(hours);
    } catch (exception) {
        alert(exception);
        done = false;
    }
}
```

Handling the exception means another trip through the loop.

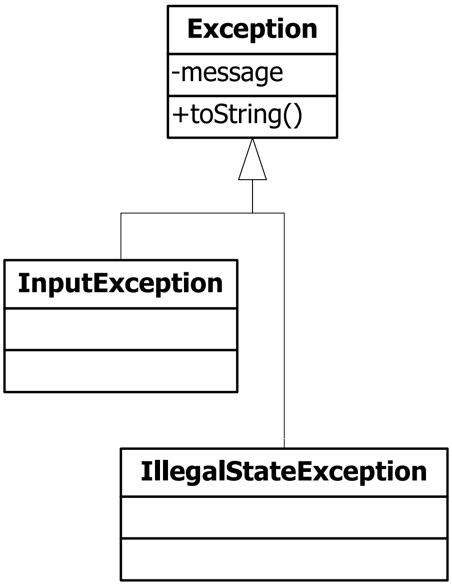
Exception Handling

- Try/catch/finally syntax

```
try {
    // code here that may throw an exception
}
catch (exception) {
    // do something to fix the error
}
finally {
    // code here is always executed regardless of
    // whether an exception is thrown/caught or not.
}
```

Exception Handling

- Exception objects
  - Can throw any kind of object.
  - Different object types can permit us to distinguish between different error conditions in a `catch` block.



Exception Handling

- Throwing exceptions, revised

```
HourlyEmployee.prototype.setHoursWorked = function(hours)
{
    if (hours < 0 || hours > 24*7) {
        throw new IllegalArgumentException(
            "Bad parameter for hours: " + hours);
    }
    else {
        this.hoursWorked = hours;
    }
}
```

A custom object that captures the message and the type of error.

Exception Handling

- Catching exceptions, revised

Using custom exception objects permits more choices of corrective action based on the type of exception.

```
try {
    emp.setHoursWorked(hours);
}
catch (ex) {
    log.debug(exception);
    if (ex instanceof IllegalArgumentException) {
        // correct this kind of error
    } else if (ex instanceof FoolishUserException) {
        // correct another kind of error
    } //.. and so on
}
```

Exception Handling

- Flow of control
  - Code is executing normally
  - An exception is thrown, terminating the current function.
  - The exception keeps propagating up the call stack until a try/catch block is found

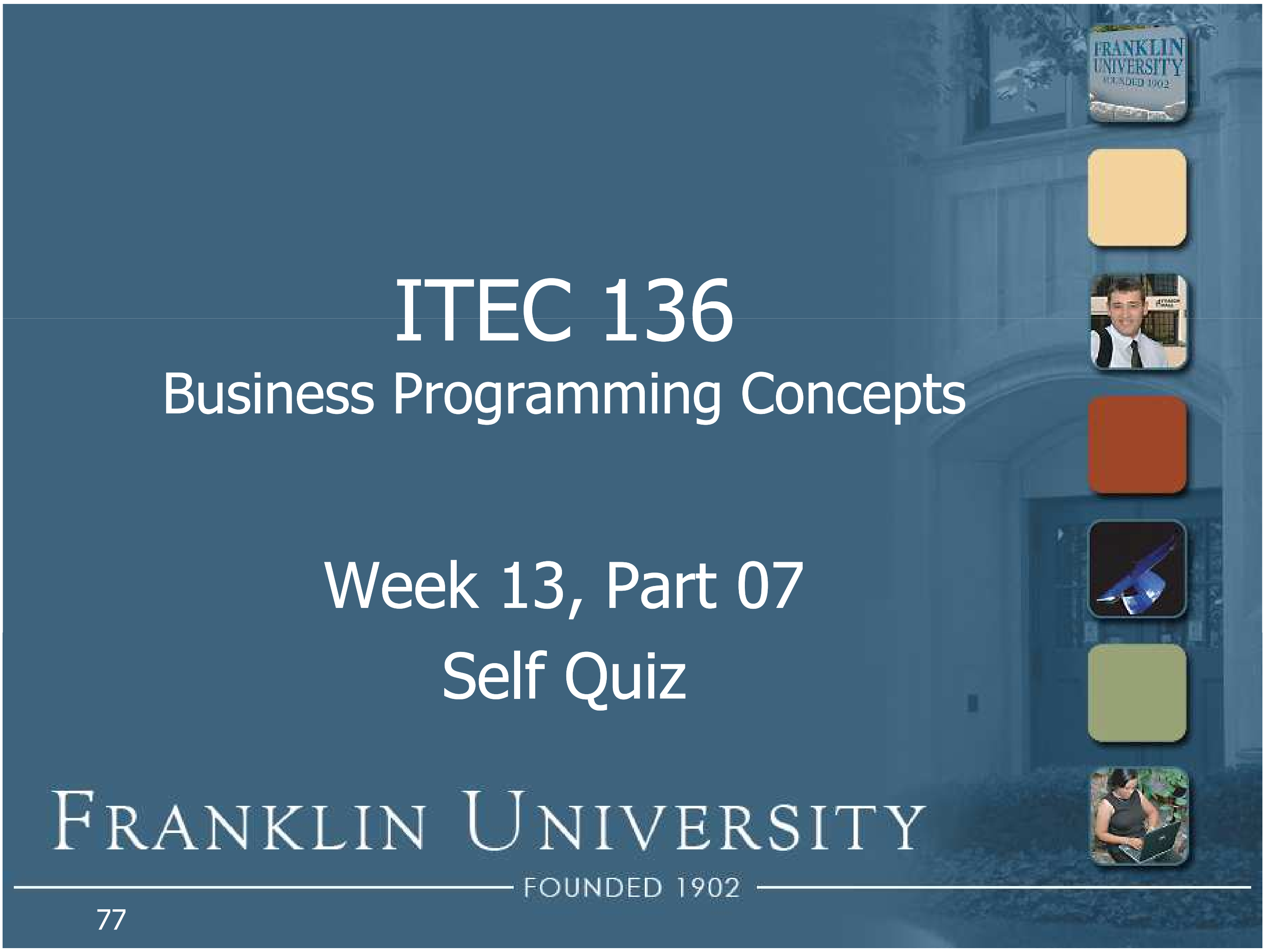
Exception Handling

- Flow of control
  - Catch block is executed
  - Finally block is executed

Questions?

Next Week

- Testing and debugging
  - A more thorough approach



ITEC 136

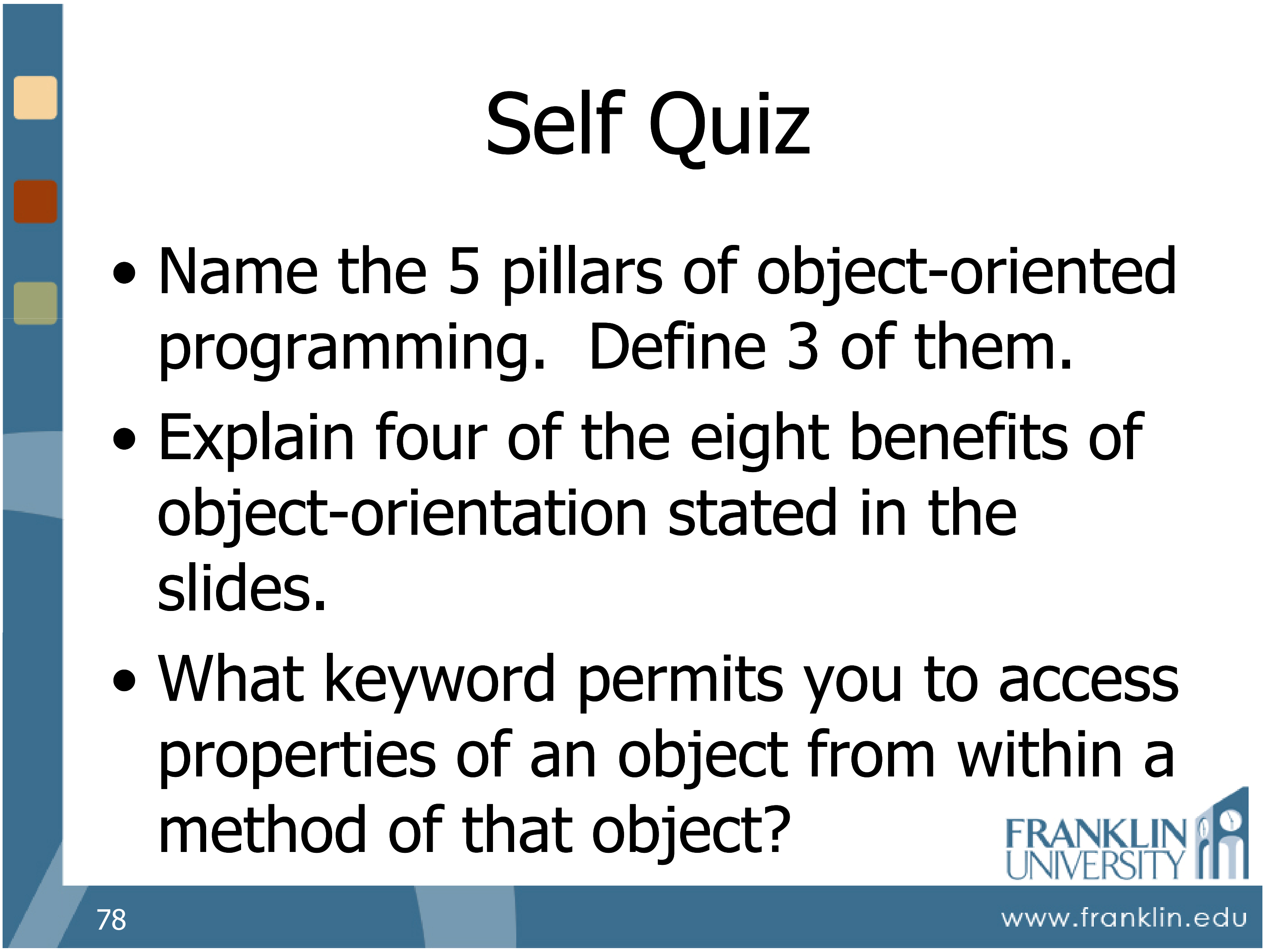
Business Programming Concepts

Week 13, Part 07

Self Quiz

FRANKLIN UNIVERSITY

FOUNDED 1902



Self Quiz

- Name the 5 pillars of object-oriented programming. Define 3 of them.
- Explain four of the eight benefits of object-orientation stated in the slides.
- What keyword permits you to access properties of an object from within a method of that object?

Self Quiz

- How is a constructor different from other functions?
- How do you write code such that methods are *shared* between objects generated from the same constructor?

Self Quiz

- What is the difference between composition and aggregation?
- Why is it important to separate the implementation of an object from its interface? What “pillar” is this?

Self Quiz

- Give two reasons that exceptions are useful in programming.
- What keyword lets you alter the flow of control in a function by generating an exception?
- What keyword(s) lets you handle an exception?

Self Quiz

- Write a constructor for a Book object that takes a title, author, and ISBN as parameters. It should make properties out of each parameter.
- Write a constructor for a Library object (no parameters). It should create an empty array to hold books.

Self Quiz

- Write methods for the Library object that will allow you to
  - Add a book to the collection
  - Look up a book by author
  - Look up a book by title
  - Look up a book by ISBN

ITEC 136

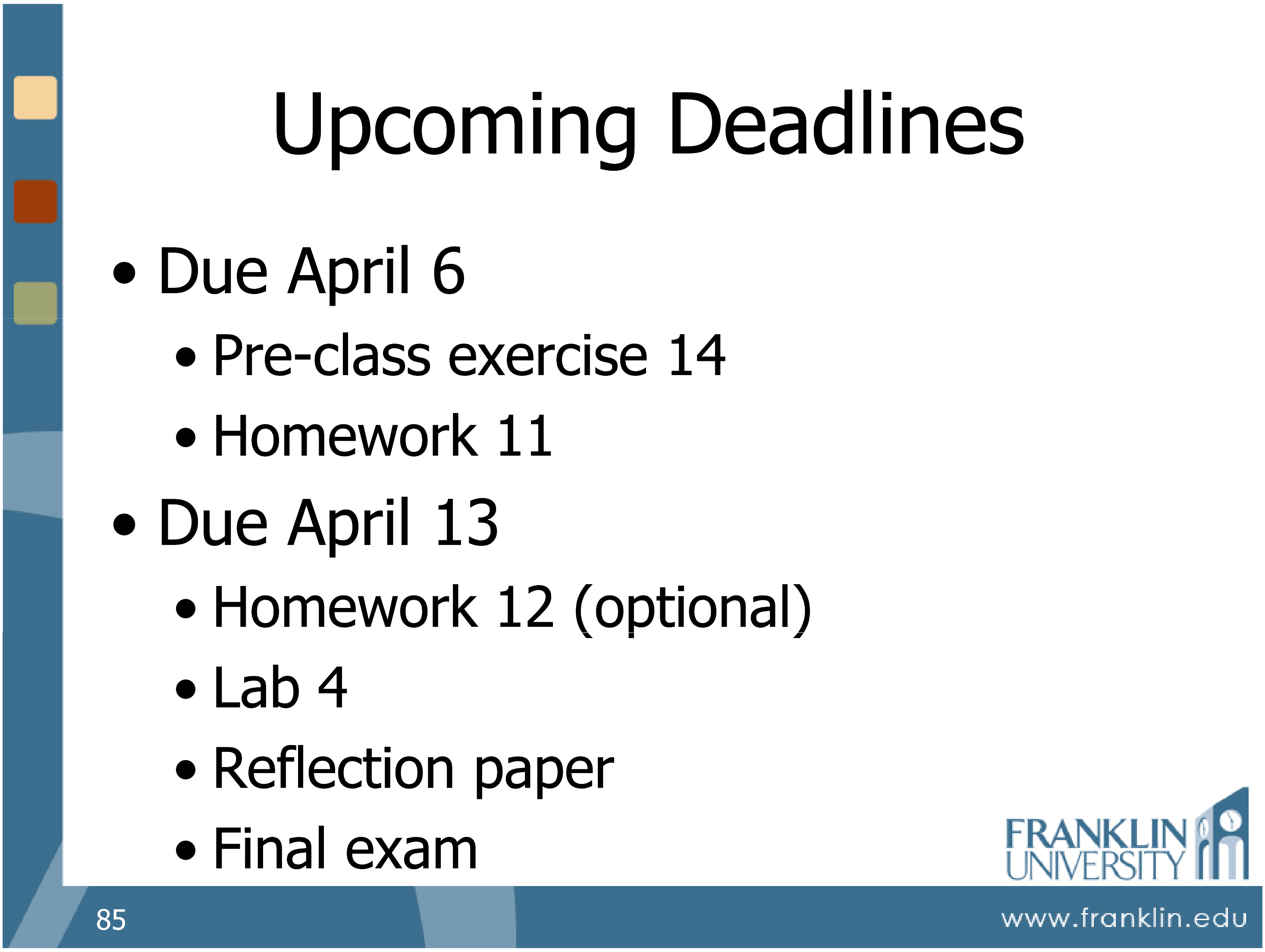
Business Programming Concepts

Week 13, Part 08

Upcoming deadlines

FRANKLIN UNIVERSITY

FOUNDED 1902



Upcoming Deadlines

- Due April 6
  - Pre-class exercise 14
  - Homework 11
- Due April 13
  - Homework 12 (optional)
  - Lab 4
  - Reflection paper
  - Final exam