What letter is?

- a bird
- part of your head
- an insect
- a drink
- a building extension
- a vegetable
- a body of water
- a farm animal

- J
- I
- B
- T
- L
- P
- C
- U

Programming Paradigms Object-Oriented Programming Structures

Luke Sathrum – CSCI 20

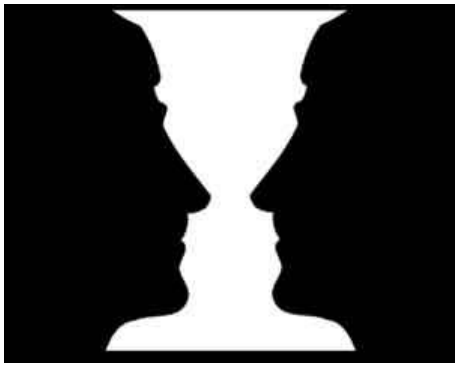
Today's Class

- Defining a Paradigm
- Common Programming Paradigms
- Intro to Object-Oriented Programming (OOP)
- Object-Oriented Programming Terminology
- Object-Oriented Programming Principles
- Introduction to Structures

Programming Paradigms

Paradigms

- What is a paradigm?
  - A way or perspective that we use to interpret what we see
  - “Looking at the same thing but seeing different things”



Programming Paradigms

- Define how the user conceptualizes and interprets complex problems related to programming
- Particular way of looking at a programming problem
- Most programming languages are optimized for a particular programming paradigm
  - Doesn’t mean they don’t/can’t support other paradigms

Programming Paradigms

- There are 4 main **Programming Paradigms**
  - Many more stem from these 4 main ones
- They are
  - Imperative
  - Object-Oriented
  - Logic
  - Functional

Programming Paradigms - Imperative

*"First **do this** and next **do that**"*

- This is the way we have programmed so far
- Comprised of statements that change the programs **state**
  - State is a snapshot of a program at a given time
  - All of the data values associated with it
- Views everything as a sequence of steps to perform
- Used to express algorithms

Programming Paradigms – Object-Oriented

"Send messages between objects to simulate a set of real world phenomena"

- Model your program based on real world phenomena
- View everything as an object
- Data is stored inside these objects
- Objects talk to each other through message passing

Programming Paradigms – Object-Oriented

- Usually objects are grouped into classes
  - The blueprint of the object
- Classes represent concepts
- Objects represent the phenomena
- Classes are organized into inheritance hierarchies

Programming Paradigms - Functional

"Evaluate an expressions and use the resulting value for something"

- Treats computation as the evaluation of mathematical functions
  - Derives from the theory of functions
- Values produces are immutable
  - They cannot change
  - Can instead make a revised copy of the original value
- Emphasizes application of functions
  - All computations are done by calling functions

Programming Paradigms - Functional

- Factorial
- 5! is really
- In ML we write it as

```
fun fac 0 = 1
    | fac n = n * fac (n - 1)
```

Programming Paradigms - Logic

"Answer a question via search for a solution"

- Use mathematical logic for computer programming
- Don't tell the computer how to solve a problem
- Instead, describe the problem domain as facts and rules
  - Using these the computer figures out how to solve the specific problem

Programming Paradigms - Logic

- Some sound logic
- In Prolog
- `witch(X)  <= burns(X) and female(X).`
- `burns(X)  <= wooden(X).`
- `wooden(X) <= floats(X).`
- `floats(X) <= sameweight(duck, X).`
- `female(girl).  {by observation}`
- `sameweight(duck,girl). {by experiment }`
- `? witch(girl).`

Programming Paradigms - Languages

Paradigm	Language(s)
Imperative	C, C++, Java, Python
Object-Oriented	C++, Java, C#, Python, Ruby
Functional	LISP, Scheme, ML
Logic	Prolog, Datalog
More at http://en.wikipedia.org/wiki/List_of_programming_languages_by_type	

Summary

- Programming Paradigms define how view a particular problem
- Languages are usually optimized for a particular paradigm
- Four main ones
  - Imperative
  - Object-Oriented
  - Functional
  - Logical

Introduction to Object-Oriented Programming

Intro to OOP

- The world around us is made up of objects
- Things like
  - A car
  - A chair
  - An apple

Intro to OOP

- These objects have the ability to perform certain actions
- A car can
  - Start
  - Stop
  - Fill Up
  - Open a door
  - Unlock
  - etc…

Intro to OOP

- These objects also can interact with other objects
- A Car interacts with
  - A human (hopefully)
  - A key
  - A Gas pump
  - A trailer
  - etc…

Intro to OOP

- Object-Oriented Programming (OOP) views a computer program as objects that interact with each other
- A Program is made up objects
  - Can act alone
  - Can interact with one another
- Objects can
  - correspond to real-world objects
  - be an abstraction

Intro to OOP

- Objects model
  - Tangible things
    - A School
    - A Car
  - Conceptual things
    - A Meeting
    - A Date
  - Processes
    - Finding a path through a maze
    - Sorting a deck of cards

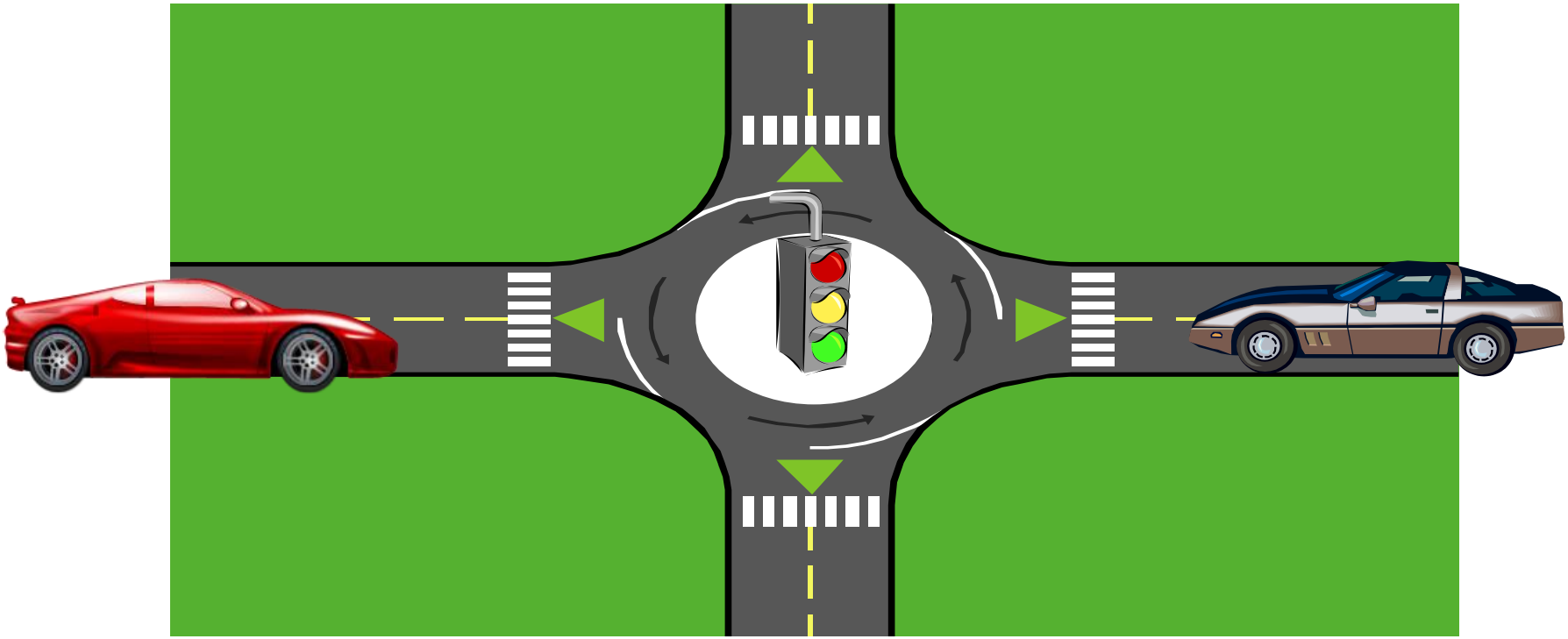
Intro to OOP

- Objects have
  - Behaviors
    - What they can do
    - How they behave
  - Attributes
    - Features that describe them
    - Data

OOP Example

- Think of a program that simulates an intersection
- Have an
  - object for each car
  - object to simulate each lane
  - object for the traffic light
  - etc…
- These objects are distinct but can interact with each other

OOP Example



Summary

- The world is made up of objects
- OOP tries to model these real world objects
- Objects have
  - Behaviors
  - Attributes

Object-Oriented Programming Terminology

OOP Terms – Attributes

- Characteristics of an object
- Basically the data of the object
- A Car might have
  - Name
  - Speed
  - Fuel Level
- Attributes are implemented as variables

OOP Terms – State

- The current values (state) of our object's attributes
- My car has
  - 5 gallons of fuel left
  - Is a Honda Pilot
  - Is currently traveling at 15 mph

OOP Terms – Behaviors

- The actions an object can take
- A Car can
  - Start
  - Stop
  - Turn
  - Turn on Lights
  - Turn off Lights
- Behaviors are implemented as functions

OOP Terms – Class

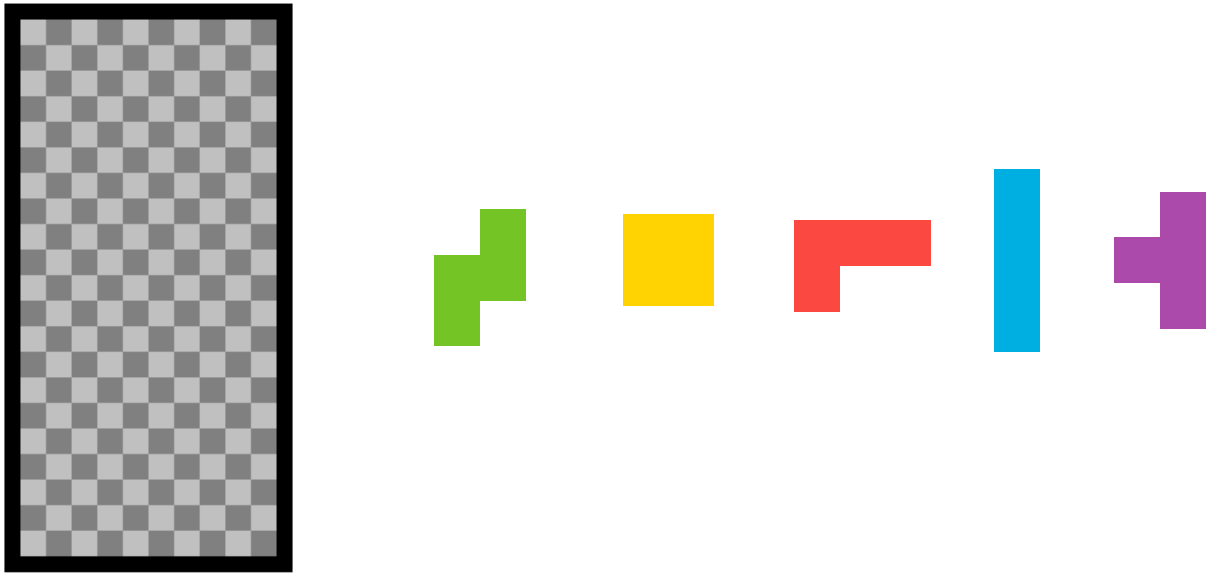
- A blueprint for an object
- A class defines an object
  - Defines the attributes
  - Defines the behaviors
- Helps each related object have the same behavior
- Each object shares a class definition but has its own state

OOP Terms - Instance

- Each object created is an instance of a class blueprint
- So we can have **`MyCar`** and **`YourCar`** be an instance of a Car class
- Remember
  - **`MyCar`** and **`YourCar`** share the same behaviors
    - start, stop, turn etc…
  - **`MyCar`** and **`YourCar`** have different attribute states
    - **`MyCar`** is a Honda Pilot, **`YourCar`** is a Ford Mustang, etc…

OOP Example - Tetris

- We'll think of a Tetris game being objects that interact with each other
- What are the game's objects?
  - Board
  - Piece(s)

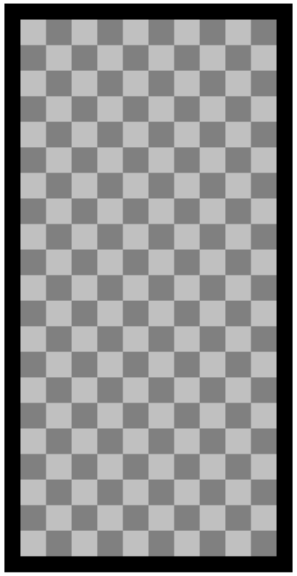


OOP Example - Tetris

- What are the attributes?

- Piece
  - Orientation
  - Position
  - Shape
  - Color

- Board
  - Size
  - Rows
  - Columns

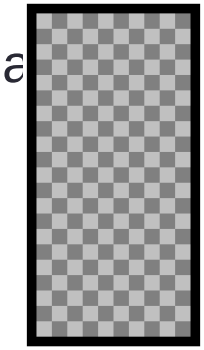


OOP Example - Tetris

- What are the behaviors?

- Piece
  - Be created
  - Fall
  - Rotate
  - Stop at collision

- Board
  - Be created
  - Remove rows
  - Check for end of ga



OOP Example - Tetris

- We will have 1 instance of a board
  - An object created from a class
- We will have many piece objects
  - All created from the same class
- To make these objects interact with each other I'd need some kind of interface object
  - This would take input from the user, etc
  - Helps my objects interact with each other

Summary

- Objects have
  - Attributes
  - State
  - Behaviors
- Objects are defined by a class
- We create instances of our objects from the class

Object-Oriented Programming Principles

This Video

- Object-Oriented Programming Principles
  - Encapsulation
  - Polymorphism
  - Inheritance

OOP Design Principles – Encapsulation

- Hides unneeded information from the user
- Packs everything into a nice little ‘capsule’
- Other terms for encapsulation are
  - Black Box
  - Information Hiding
- OOP allows the programmer to enforce encapsulation

OOP Design Principles – Encapsulation

- Example
  - You turn a key to start your car
    - Do you know exactly how the engine turns on?
  - You create a piece for a Tetris game
    - Do you know how the piece is being represented in memory?

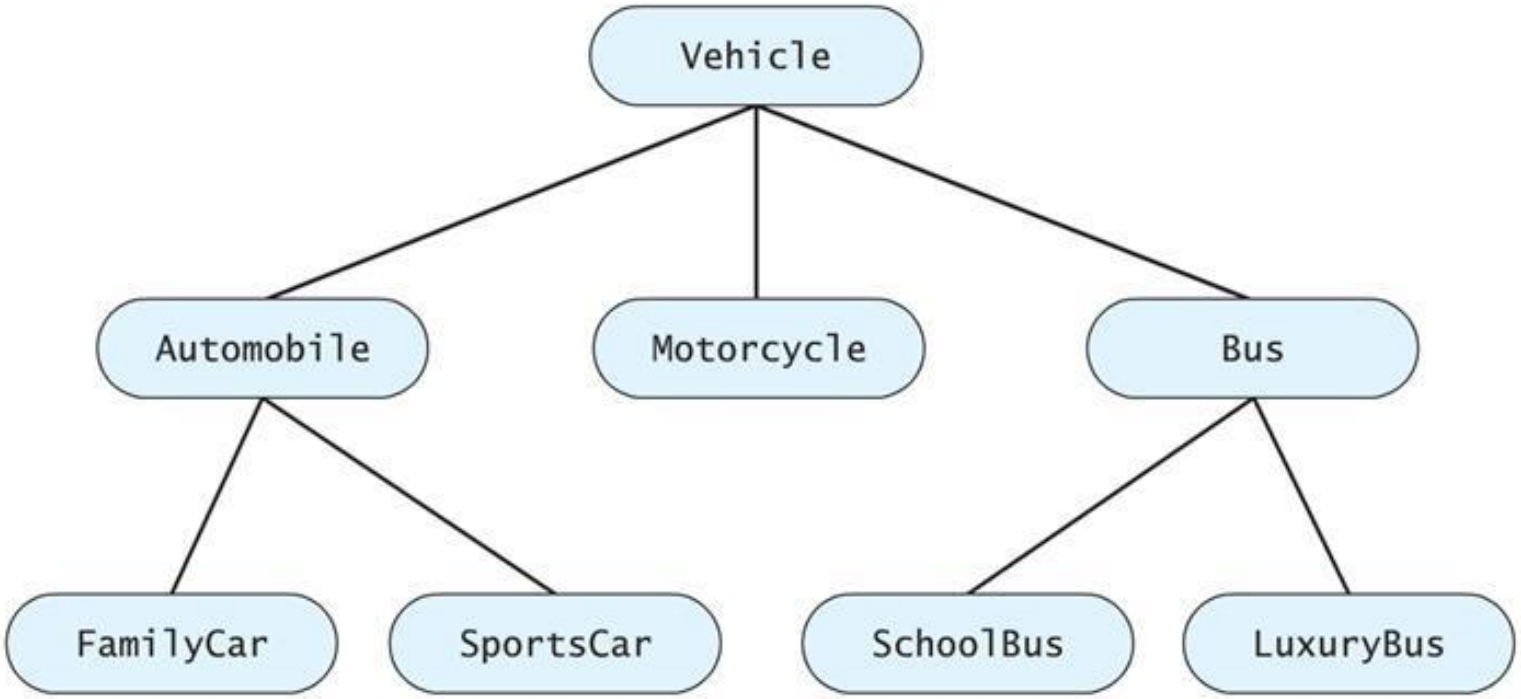
OOP Design Principles – Polymorphism

- Greek for “many forms”
- Allows a program instruction to mean different things in different contexts
- Much like in English
  - “Go play your favorite sport”
    - Could be baseball
    - Could be soccer
- Allows one behavior, used as an instruction, to cause different actions
  - Depends on the object that is performing the action

OOP Design Principles – Inheritance

- A way of organizing classes
  - Remember: Classes are the blueprints for objects
- Each Level of the class becomes more specialized
- EX
  - Shape -> Quadrilateral -> Square
  - Vehicle -> Car
  - Vehicle -> Bus
  - Vehicle -> Motorcycle
- Allows a programmer to avoid repeating instructions for each class

OOP Design Principles – Inheritance



Summary

- Three main OOP Principles
  - Encapsulation
  - Polymorphism
  - Inheritance

Introduction to Structures

Structures

- Useful to have a collection of types as a single item
- Think of a car
- Has distinct pieces of information associated with it
  - Year
    - Integer
  - Doors
    - Integer
  - Horsepower
    - Double
  - Model
    - String

Structures

- Currently we don’t have a way to group this information
- Structures give us a way to group different types together

My Car	
Year	2005
Doors	4
Horsepower	255
Model	Pilot

Structures - Syntax

```
struct Auto {
  int year;
  int doors;
  double horse_power;
  string model;
};
```

- Use the keyword **`struct`** to define a structure
- Then you give your structure a name
  - Called the structure tag, in this case **`Auto`**
  - Should start with an uppercase letter

Structures - Syntax

```
struct Auto {
  int year;
  int doors;
  double horse_power;
  string model;
};
```

- The structure body is enclosed in curly braces
  - Each identifier in the body is called a member name
- Notice the semicolon at the end of the structure body

Structures - Syntax

- We place the structure definition above **`main()`**
  - In the same place you would place global constants
- Think of the structure tag as a type that you can then use throughout your program
- To create a structure variable:

```
Auto my_car;
```

Structures – Member Variables

- Inside our new structure variable **my_car** we now have 4 member variables
  - **year**
  - **doors**
  - **horse_power**
  - **model**
- To get/set those values we use the dot operator

```
my_car.year = 2005;
cout << my_car.model << endl;
```

Structures

- Can be thought of in one of two ways
  - A collection of member values
  - A single (complex) value
    - This means you can use the assignment operator on the entire structure
    - They must share the same structure definition
  - So assuming I have `Auto car1, car2;`
  - I can do the following

```
car1 = car2;
```

Summary

- Structures allow us to group values together
  - Allows us to mix types
- Use the keyword `struct` to define a structure
- Defined above `main()`
- Use it much like a built-in type
- Set member variables using the dot operator

Sample Code

- Creating a Structure
  - `structure.cpp`

Initializing Structures Sub-Structures

Initializing Structures

- You can initialize a structure when it is declared
- Use the equal sign followed by a list of member values
  - Enclose these values in curly braces
- Given

```
struct Date {
  int month;
  int day;
  int year;
};
```

- Initialize

```
Date my_date = {12, 31, 2013};
```

Initializing Structures

```
Date my_date = {12, 31, 2013};
```

- Notes
  - The initializing values must be given in the order of your member variables
  - If you don’t fully initialize a structure
    - It fills how ever many you gave for initializing it

Structures in Structures

- We can have member names be their own structures
- Let’s say we want a structure to hold a person’s information
  - Height
  - Weight
  - Birthdate
- Birthdate is made up of 3 distinct values
  - Month
  - Day
  - Year

Structures in Structures

- The syntax

```
struct Date {
  int month;
  int day;
  int year;
};
```

```
struct PersonInfo {
  double height;
  int weight;
  Date birthday;
};
```

Structures in Structures

- Create a persons structure the normal way

```
PersonInfo person_one;
```

- To access the birthday data we need another dot

```
person_one.birthday.year
```

Summary

- We can initialize structures on declaration
- You can use a structure inside another structure

Sample Code

- Initializing Structures and Using Sub-Structures
  - `structure_concepts.cpp`

Review

- Four Main Programming Paradigms
- Objects have behaviors and attributes
  - Attributes have state
- Objects are made from a class
- Three OOP Principles
- Structures allow us to