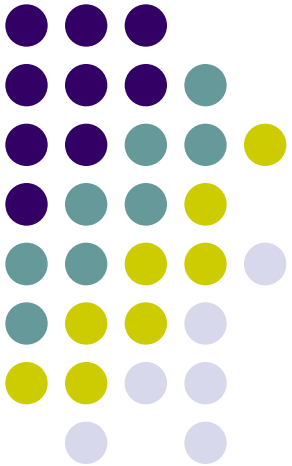
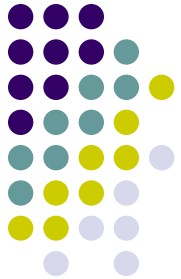
CMSC 202H

Fall 2011, Honors Section

Designing with Objects

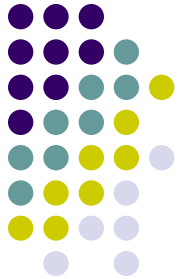


Programming & Abstraction



- All programming languages provide some form of ***abstraction***.
  - Also called ***information hiding***
  - Separating how one uses a program module from its implementation details
- Procedural languages:
  - Data abstraction: using data types
  - Control abstraction: using functions
- Object-oriented languages:
  - Data and control abstraction: using classes
- Our world is object-oriented.

Procedural vs. OO Programming



Procedural

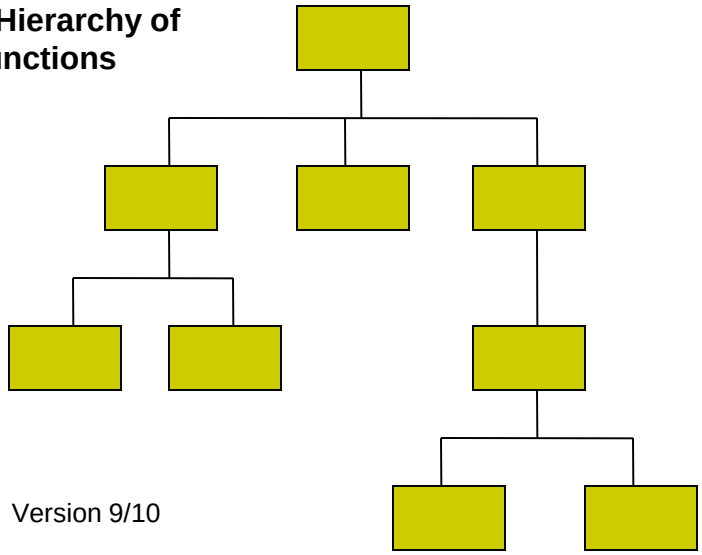
Examples: C, Pascal, Basic, Python

Modular units: functions

Program structure: hierarchical

Data and operations are not bound to each other.

A Hierarchy of Functions



Object-Oriented (OO)

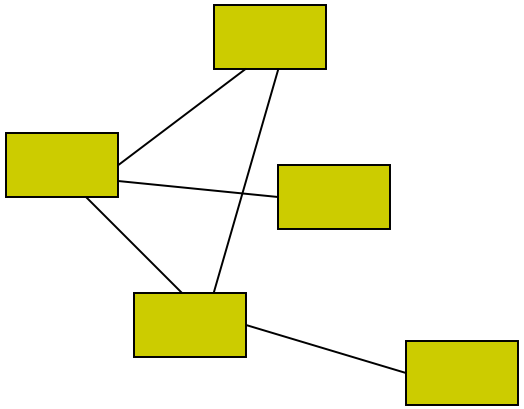
Examples: Java, C++, Ruby

Modular units: objects

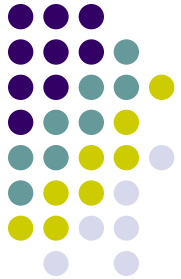
Program structure: a graph

Data and operations are bound to each other.

A Collection of Objects

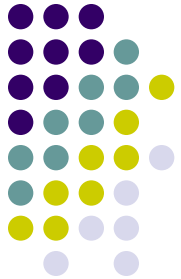


Encapsulation



- We will use the term *encapsulation* in two different ways in this class (and in the text)
  - Definition #1: "Inclusion" (“bundling"):
    - bundling of structure and function
  - Definition #2: “Exclusion” (“access control”)
    - Strict, explicit control of how our objects can be used

Procedural vs OO: Example



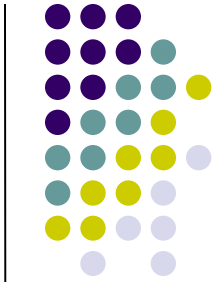
- Procedural:

```
if (shape.type == SQUARE)
  area = shape.width ** 2;
else if (shape.type == TRIANGLE)
  area = (shape.width * shape.height) / 2;
else if (shape.type == CIRCLE)
  area = ((shape.width / 2) ** 2) * PI;
```

OO:

```
shape.getArea();
```

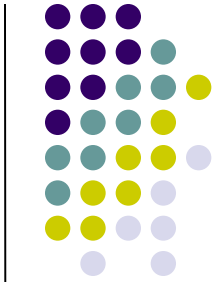
Objects are Models



Model Type I: Objects as Models of the Real World

- Programming “objects” are accurate representations of physical entities
- Goal is realism
- We want to create representations of “things”, with:
  - Observable attributes
  - Physical functionality
- E.g.: Physical vending machine:
  - A given bin has physical capacity limit
    - `float binDepth, spiralTurnSize;`

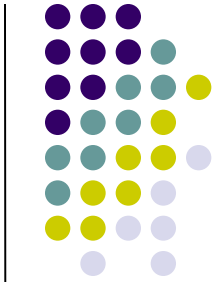
Objects are Models



Model Type II: Objects as Models of Conceptual Entities

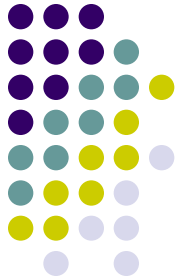
- We think programmatically from the outset
- Goal is to simplify programming
- E.g.: generic vending machine slot representation
  - A bin is an unbounded array
    (a pysical vending machine cannot recalibrate once built)
  - Might also want to track “numItemsVended”, e.g., to find popular positions.
    - Not restricted to what is a “physical attribute”

Objects are Models



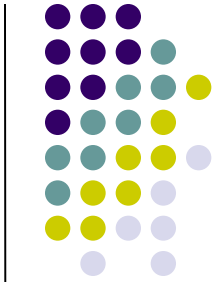
- Should we use physical or abstract model?
  - Humans are better at relating to physical
  - Computers can take advantage of abstract
- General rule: start with concrete, then start abstracting, extending
- Don’t be dogmatic!

What Parts to Model:



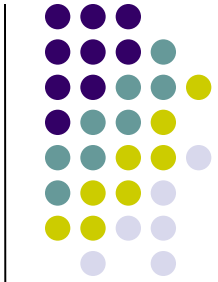
- Depends on purpose of model
- Can not (and should not) model everything
- Can try to anticipate likely future needs, though
  - Running Theme #1 (RT1): “Multiply-by-PI Rule”
- E.g.: Bank Account
  - Account type
  - Account number
  - Name
  - Address
  - Balance

What Parts to Model:



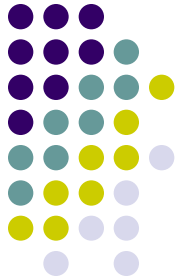
- ... But what about:
  - Transaction history
  - Number of overdrafts
  - Interest rate
    - This might be function of account type (stored in bank-wide object)
    - …Or, might also want to be able to set individually per-account
- Also, might want to extract personal info (name, address, etc.) into separate object

What Parts to Model:



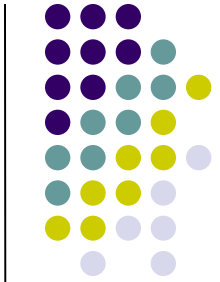
- What about methods?
  - Again, same considerations of:
    - Physical vs. abstract models
    - Designing for future needs

Order of Design Tasks



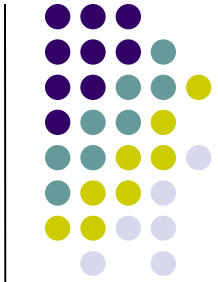
- In general, do data layout first
  - I.e., consider “complete” set of attributes you want to model; aka:
    - state variables
    - properties
    - instance variables (Java’s term)
- Then do top-level design of functionality needed (methods)
  - Easier once you have attributes in mind
- After initial design, we refine iteratively

Related Concepts



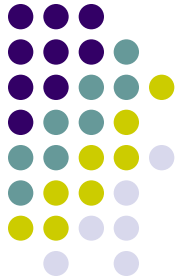
- Composition
  - A mechanism for incorporating other, existing objects as components of our new object
- Inheritance
  - A way to incrementally build on the design of existing objects to create related, more complex objects

Classes



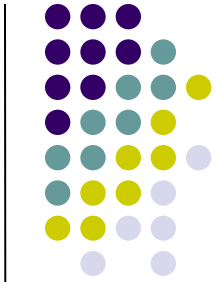
- All objects are unique.
- Objects that are identical *except for their state* can be grouped into a class.
  - Class of cars
  - Class of birds
  - Class of playing cards
  - Class of bakers

Classes



- A class definition serves as a template for creating one or more related objects.
  - A Car class can be used to create different car objects
  - A Bird class can be used to create several birds
- The blueprint defines
  - The class's state/attributes as variables
  - The class's behavior, as methods

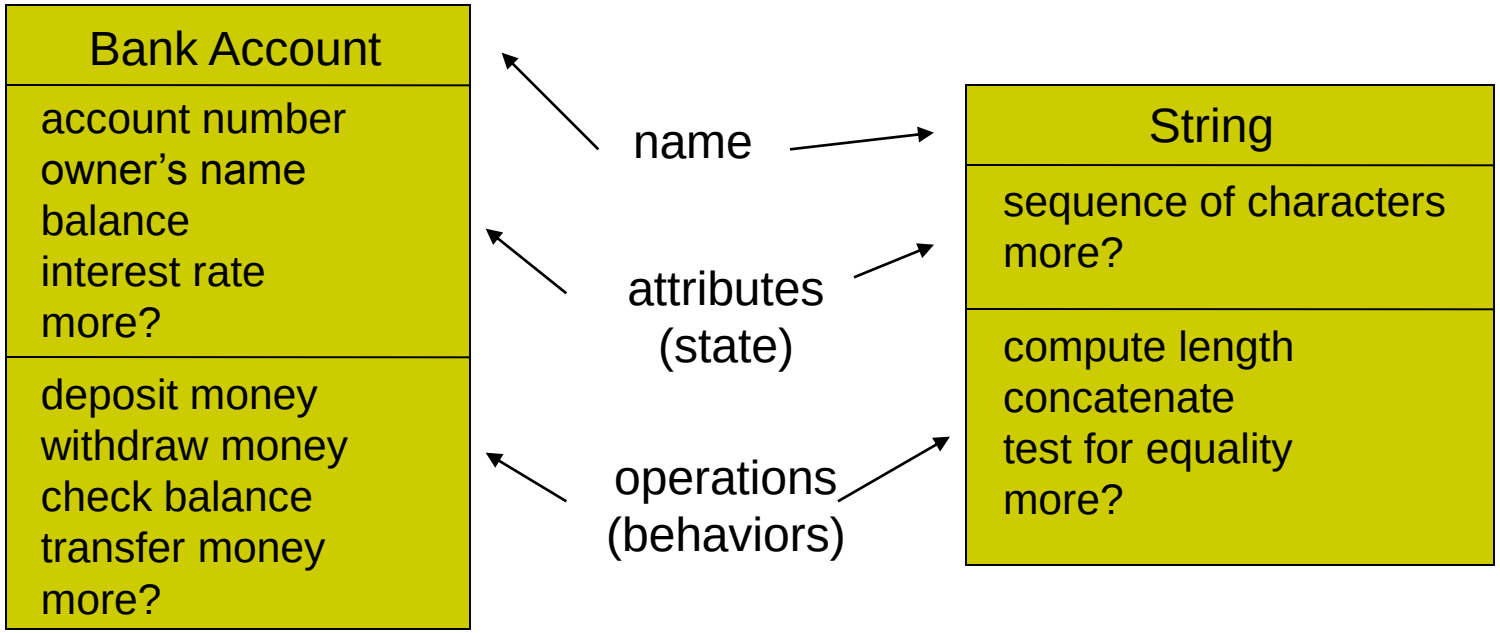
A Class is a Model



- A class is a model for a group of similar objects.
- The focus of the class is the common behavior of the objects the class models.
- A class's behavior may also be referred to as services, operations, actions, or commands.
- The model also contains common attributes of the objects.
  - The attributes exist to support the model's behaviors.
  - The actual stored values vary from object to object

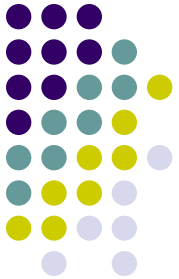
Objects, Classes

- If a *class* is:
  - A data type defining a template for a set of:
    - Attributes - make up the object's "state"
    - Operations - define the object's "behaviors"



Objects, Classes

- …Then an *object* is a particular "instance" of a class.



Bergeron's Account

12-345-6
Ryan Bergeron
$1,250.86
1.5%

Frey's Account

65-432-1
Dennis Frey
$5.50
2.7%

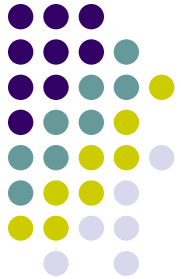
Mitchell's Account

43-261-5
Susan Mitchell
$825.50
2.5%

For any of these accounts, one can

- deposit money
- withdraw money
- check the balance
- transfer money

Classes vs. Instances



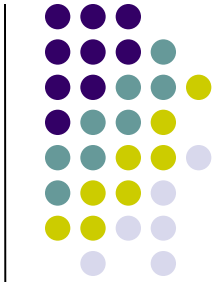
There is always inherent tension between:

- Designing classes to be more inclusive/general, w/discriminating attributes in each object

vs.

- Making each class more specific, and being able to assume more defaults, but having more partitioning of set of instances

Classes vs. Instances

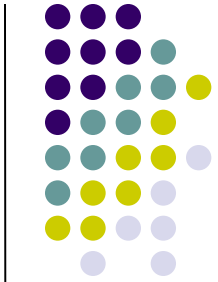


- You can have class Bird, w/attribute “canFly”
- …or you can have classes FlyingBird and FlightlessBird, with “canFly” being implicit in instances of each class.
- Silly, extreme example:

  class AccountsWithBalanceOf_199_97

  which only has instances of accounts with a balance of $199.97

Common vs. Different

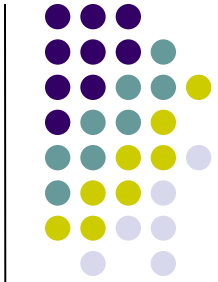


Note—in Java:

- Set of methods is common to all objects in a class
- Set of instance variable *definitions* are common to all objects in a class
- **BUT**: Instance variable *assignments* are unique to each object in a class

So: methods define what is common across class, instance variables define what differs

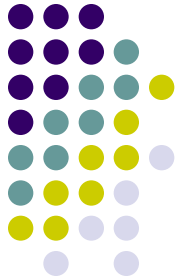
Class Examples



What services/behaviors might be appropriate for the following things?

- A red-green-yellow traffic light
- A garage door opener
- A bank account

A Class is a Type
An Object is a Variable



Variables of any particular class type may be created just like variables of built-in types.

- All objects of type Bakery have similar (class) properties.

You can create as many objects of the class type as you like.

- There is more than one Bakery in Baltimore.

OOP challenge is to define the classes and create the objects that match the problem.

- Do we need an Oven class?

Class Interface

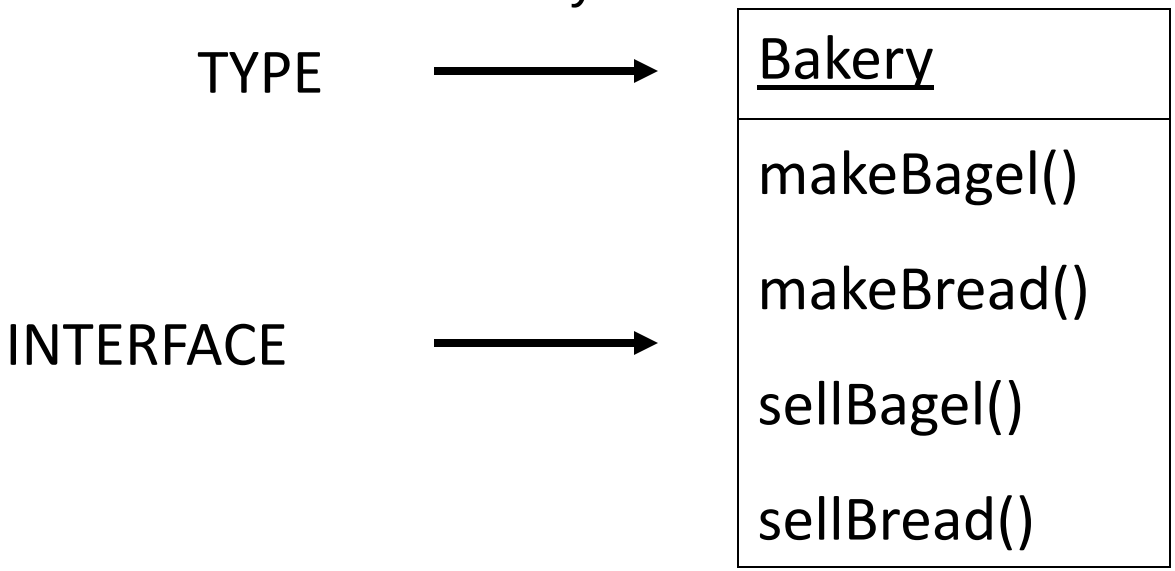
The requests you can make of an object are determined by its ***interface*.**

The interface is determined by its class type.

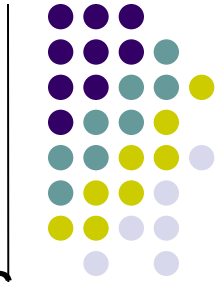
IMPORTANT

Do you need to know how a bakery works to get bread?

Who needs to know how a bakery works?



Implementation



Code and hidden data in the object that satisfies requests comprise the ***implementation***.

What's hidden in a bakery?

Each request in the interface has an associated method. When a particular request is made, that method is called.

In OO-speak we say that you are ***sending a message*** to the object, which responds to the message by executing the appropriate code.