CS 5004

LECTURE 3

MORE COMPLEX DATA

INTERFACES, PROTOCOLS, TYPES, ABSTRACT CLASSES & EQUALITY

KEITH BAGLEY

SPRING 2022

Northeastern University
Khoury College of Computer Sciences

440 Huntington Avenue ▪ 202 West Village H ▪ Boston, MA 02115 ▪ **T** 617.373.2462 ▪ **khoury.northeastern.edu**

AGENDA

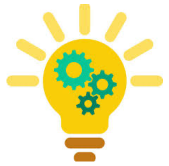
- Exception Handling Review
- Equality & Comparison
- Method Overriding & Dynamic Dispatch
- Types & Sub-types
- Protocols & Interfaces
- Inheritance & Subclasses
- Subtyping vs. Subclassing

REVIEW: EXCEPTION HANDLING

- We raised exceptions last week (using Java throw keyword).
- Let's briefly review handling exceptions with our SimpleFraction class

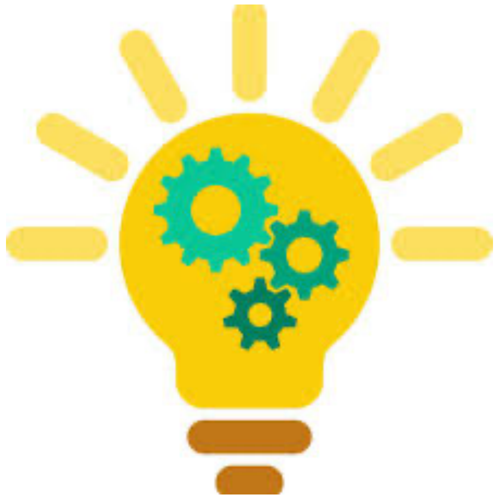
EQUALITY AND COMPARISON

- Objects have
  - Behavior
    - What can it do?
  - Attributes
    - What does it know?
  - State
    - What condition is it in?
  - Unique Identity
    - Who is it, really?

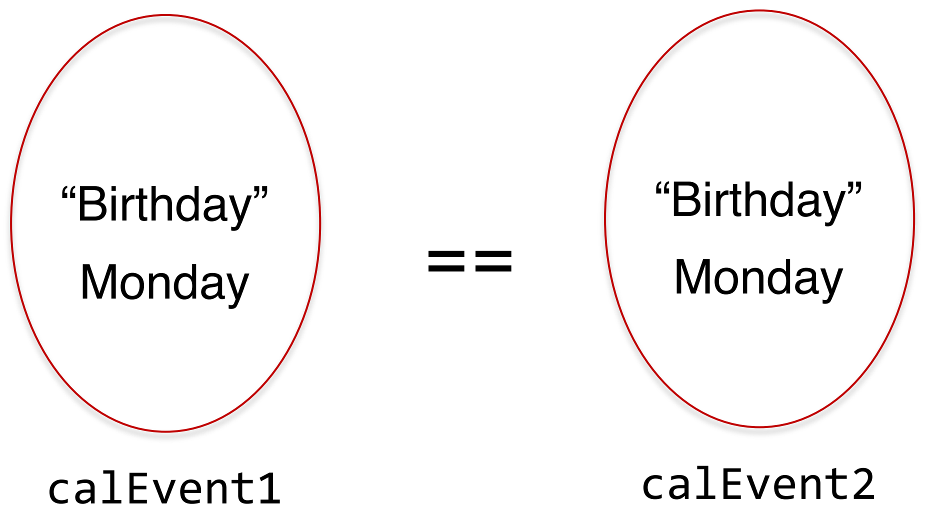


EQUALITY AND COMPARISON

- When we compare objects, we might use any (or all) of the previous – except Behavior – to determine what we mean when we say “equal to” in English



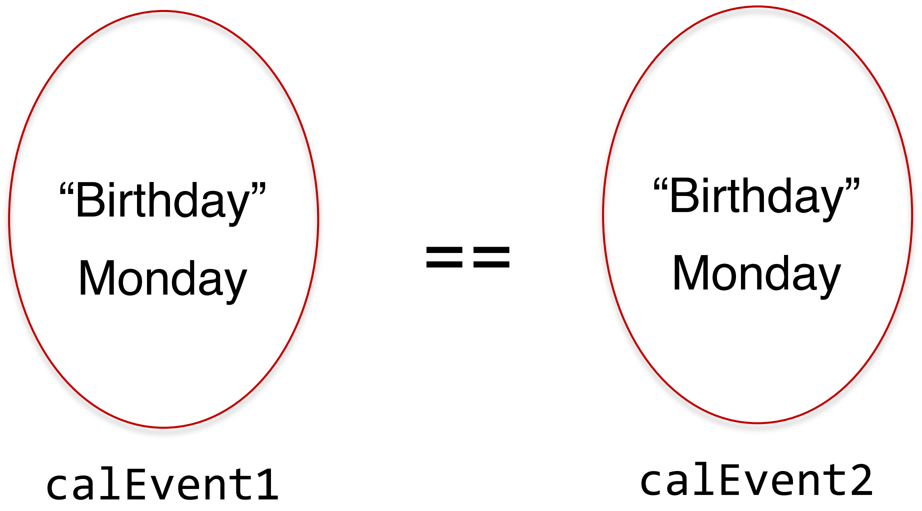
EQUALITY AND COMPARISON



What are we saying here?

```
calEvent1 = new CalendarEvent(DAY.MONDAY, “Birthday”);
calEvent2 = new CalendarEvent(DAY.MONDAY, “Birthday”);
if (calEvent1 == calEvent2)  // ?????
```

EQUALITY AND COMPARISON: ==



== works for basic types like int, but probably NOT what you intend for objects

== essentially is asking: "is this the same object" (IDENTITY), not
"Are these equal"?

```
calEvent1 = new CalendarEvent(DAY.MONDAY, "Birthday");
calEvent2 = new CalendarEvent(DAY.MONDAY, "Birthday");
if (calEvent1 == calEvent2)  // false
```

EQUALITY – WE NEED TO KNOW WHAT WE MEAN

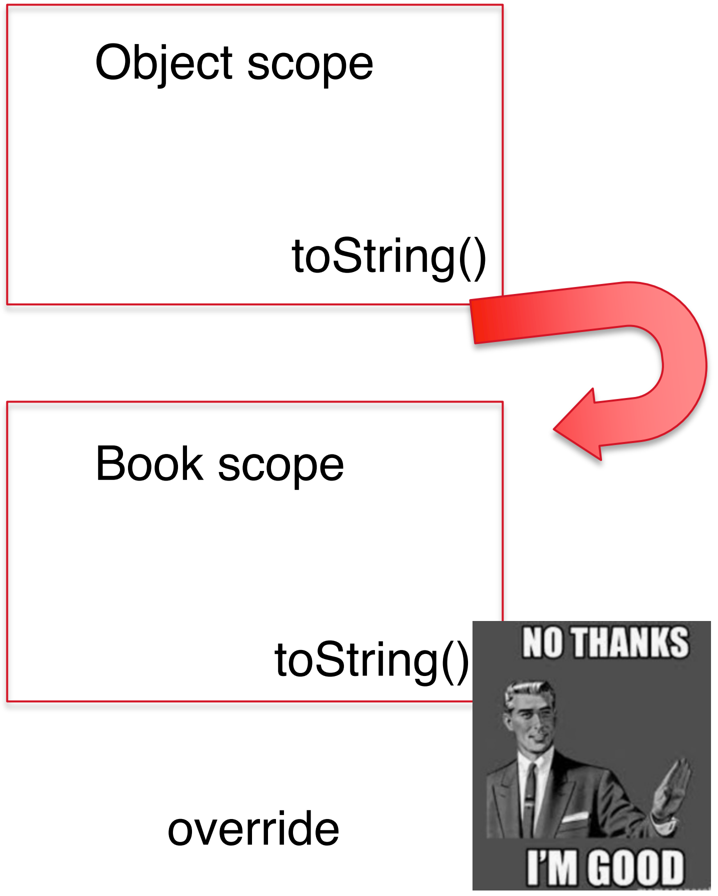
- The previous example didn't "fail" necessarily
  - We need to define what WE mean by equality for our classes & objects
  - The previous example was examining IDENTITY

    - E.g. "Is this the same object?"
  - Most likely, what we really want is an approach to asking if the objects have equivalent values

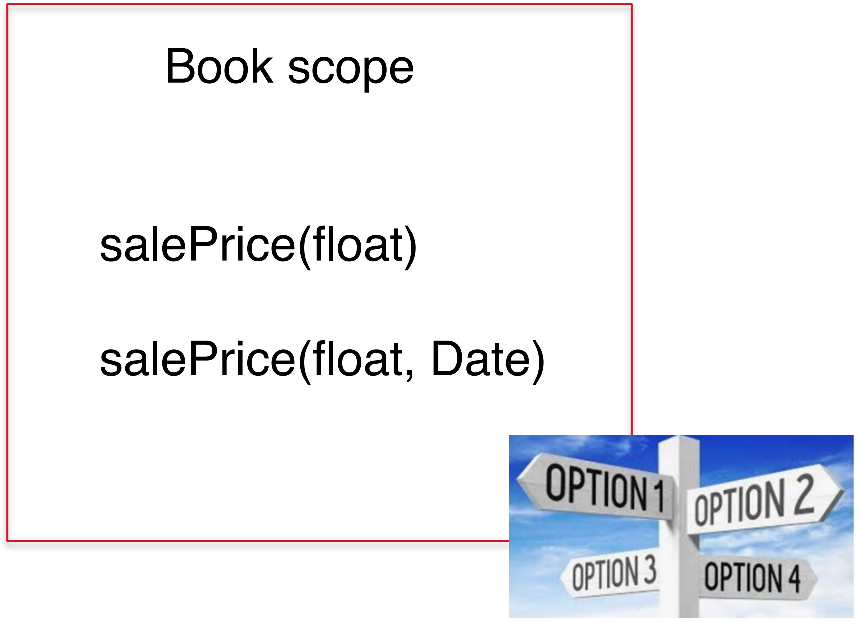
OVERRIDES

- Java allows for method overrides so we can supply an “alternate version” of a method for the specific type we’re developing
  - Note – this is not “overloading” – the topic we covered previously
    - Overload is a compile-time, static polymorphic approach within a single scope
    - Override (our topic today) is a runtime polymorphic approach for “dynamic dispatch” based on runtime objects, not compile-time type of variable.
- As with `toString()`, we can override the `equals()` method for comparing objects

CLARIFY: OVERRIDE VS. OVERLOAD

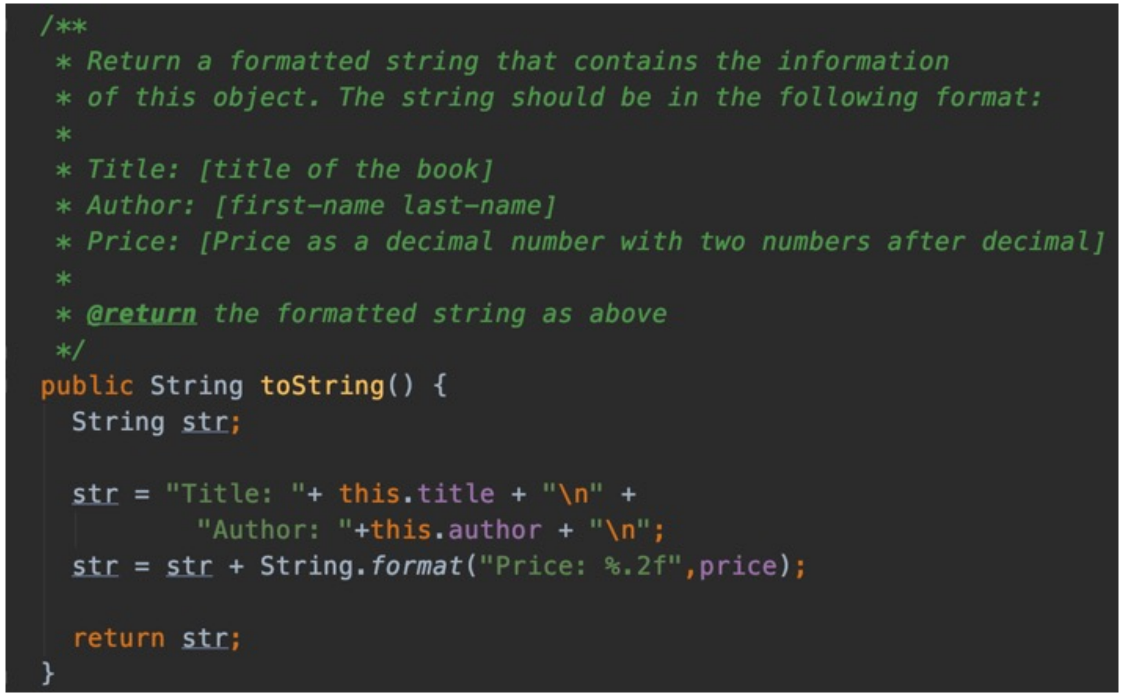


override

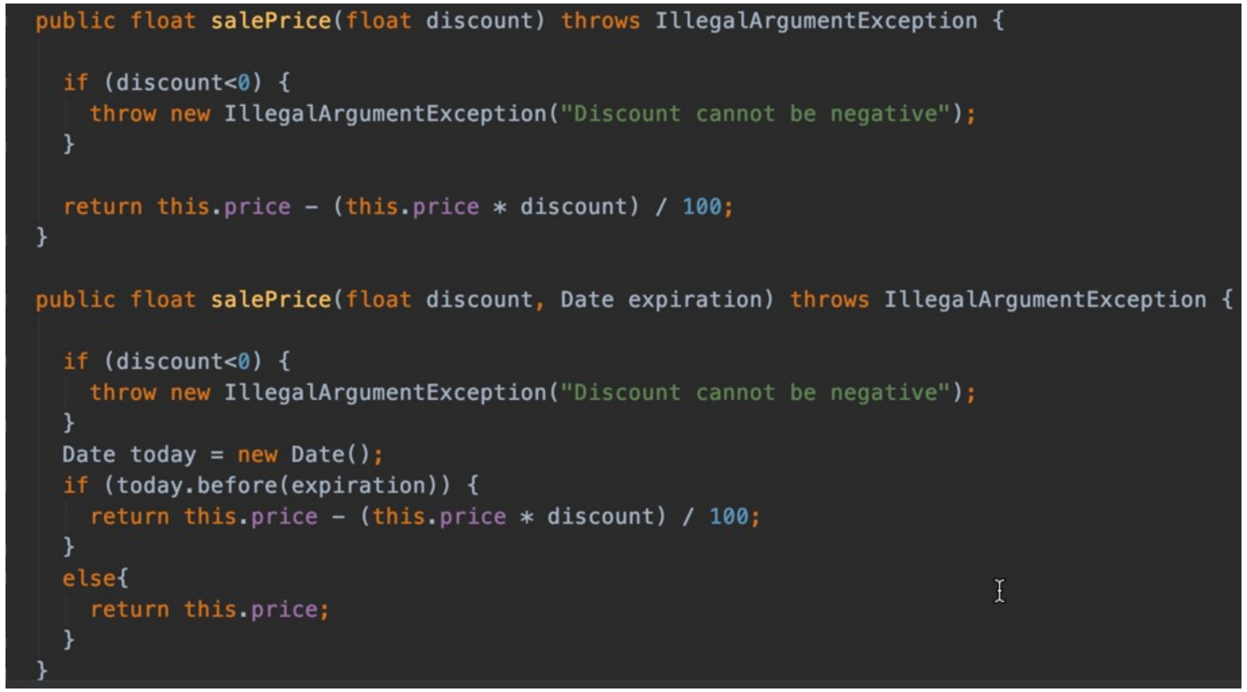


overload

CLARIFY: OVERRIDE VS. OVERLOAD



```java
/**
 * Return a formatted string that contains the information
 * of this object. The string should be in the following format:
 *
 * Title: [title of the book]
 * Author: [first-name last-name]
 * Price: [Price as a decimal number with two numbers after decimal]
 *
 * @return the formatted string as above
 */
public String toString() {
  String str;

  str = "Title: "+ this.title + "\n" +
          "Author: "+this.author + "\n";
  str = str + String.format("Price: %.2f",price);

  return str;
}
```

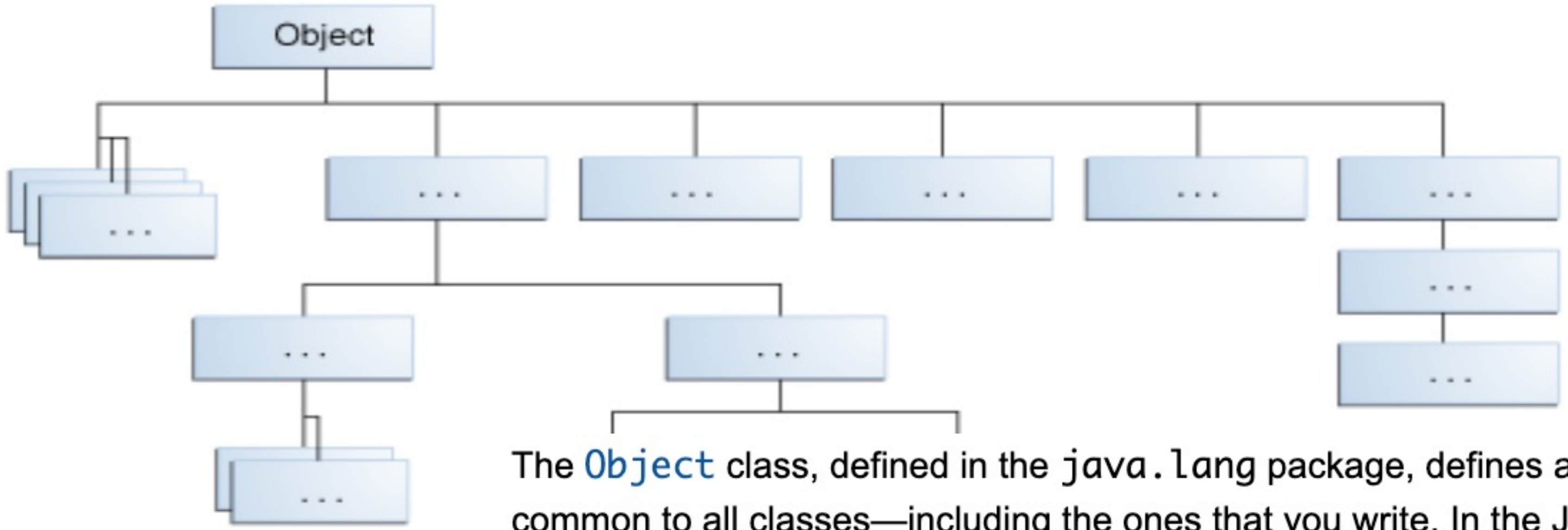


```java
public float salePrice(float discount) throws IllegalArgumentException {

  if (discount<0) {
    throw new IllegalArgumentException("Discount cannot be negative");
  }

  return this.price - (this.price * discount) / 100;
}

public float salePrice(float discount, Date expiration) throws IllegalArgumentException {

  if (discount<0) {
    throw new IllegalArgumentException("Discount cannot be negative");
  }
  Date today = new Date();
  if (today.before(expiration)) {
    return this.price - (this.price * discount) / 100;
  }
  else{
    return this.price;
  }
}
```

Override toString()

Overload salePrice()

WHY? (ALMOST) EVERYTHING DERIVES FROM THE OBJECT CLASS IN JAVA



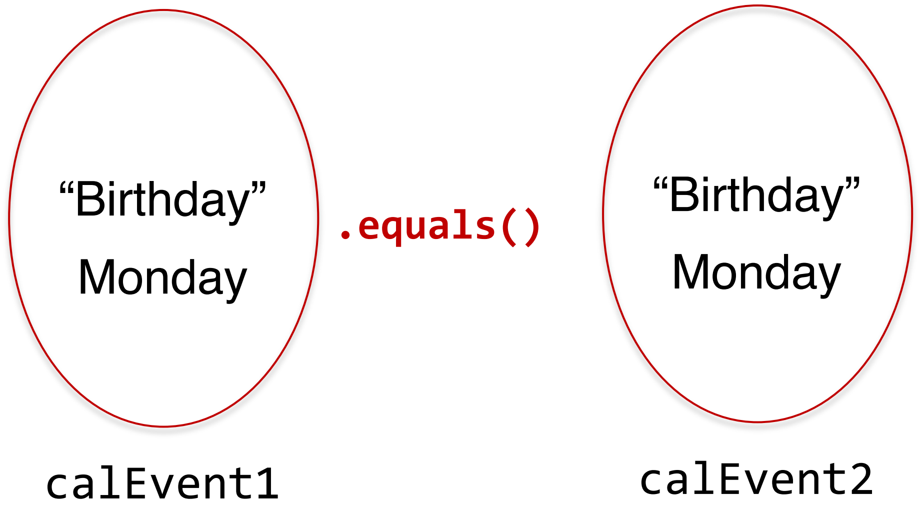
The `Object` class, defined in the `java.lang` package, defines and implements behavior common to all classes—including the ones that you write. In the Java platform, many classes derive directly from `Object`, other classes derive from some of those classes, and so on, forming a hierarchy of classes.

All Classes in the Java Platform are Descendants of Object

From: https://docs.oracle.com/javase/tutorial/java/IandI/subclasses.html

So every class already has equals(), toString(), etc.

EQUALITY AND COMPARISON: EQUALS()



== works for basic types like int, but probably NOT what you intend for objects

== essentially is asking: "is this the same object" (same identity), not
Are these equal

```
calEvent1 = new CalendarEvent(DAY.MONDAY, "Birthday");
calEvent2 = new CalendarEvent(DAY.MONDAY, "Birthday");
if (calEvent1.equals(calEvent2))  // true or false - depends on our override
```

EQUALS() RECIPE

- Writing an equals() method can follow a generally standard conceptual form (or recipe), with some adjustments depending on the implementation language.
  - E.g.: strongly/statically typed languages like Java may require type coercion for checks, etc., dynamic languages typically will not
- Steps:
  - Check for identity. If current object is the same as the "other" object, they are equal
  - Check for common class (and possibly class hierarchy). If not same class or not in same class hierarchy, the two objects are not "equal"
  - Compare relevant instance data. If current object's data == other object's data (the relevant assets for equality, that is), they are equal. If not, they are not equal.

EQUALS() RECIPE – JAVA VERSION

- public boolean equals(Object o) {

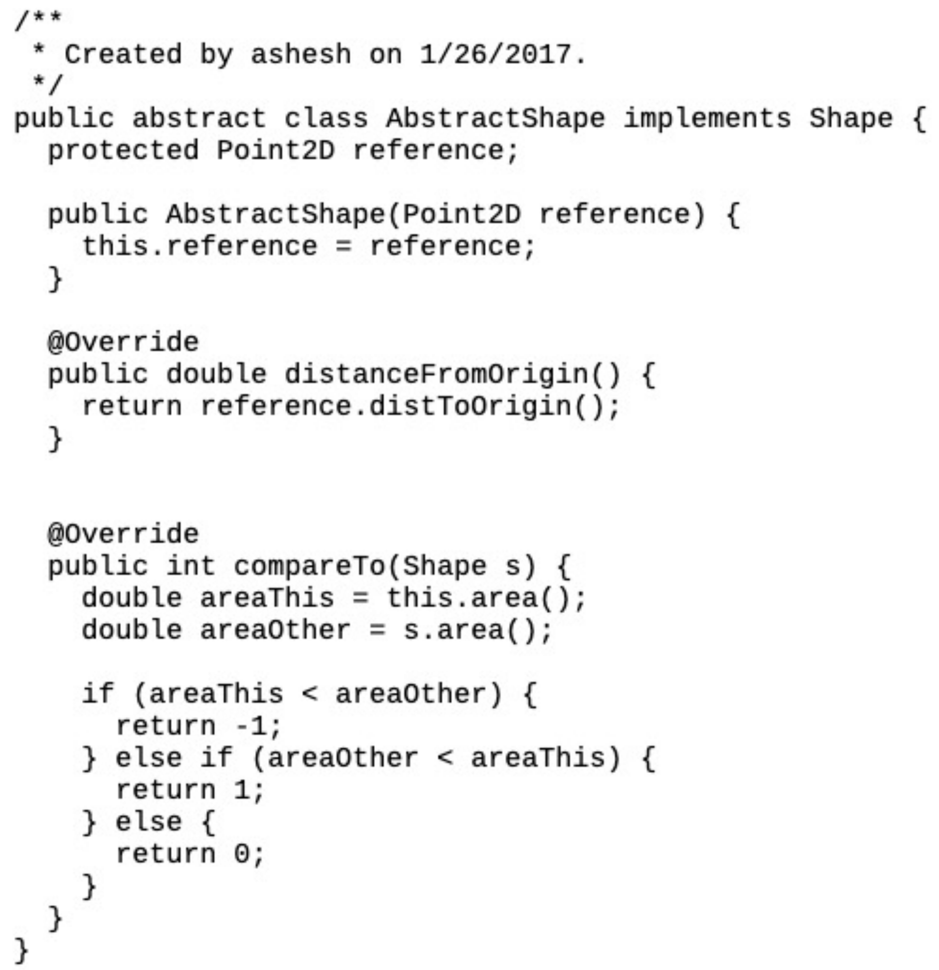
```
 if (this == o) return true;  // check identity
 if (!(o.getClass() == this.getClass())) return false; // not same class
 ClassX x = (ClassX) o;
 if( // x’s stuff == o’s stuff, return true else return false){}
```

SIDEBAR: COMPARETO()

- There is also a compareTo() method for comparing ordering sequence of objects. We get this method from Object (like toString()) and can override it if we want
  - Less than, equals, greater than
  - CompareTo method must return negative number if current object is less than other object, positive number if current object is greater than other object and zero if both objects are equal to each other.
  - CompareTo must be in consistent with equals method e.g. if two objects are equal via equals() , there compareTo() must return zero otherwise if those

SIDEBAR: COMPARETO()

- Example from Module 3 code from Amit & John
- Allows us to "order" objects sequentially based on the criteria WE set.
- **Don't worry about compareTo() for now,** we'll cover this method again later when we talk about Java generics



```java
/**
 * Created by ashesh on 1/26/2017.
 */
public abstract class AbstractShape implements Shape {
  protected Point2D reference;

  public AbstractShape(Point2D reference) {
    this.reference = reference;
  }

  @Override
  public double distanceFromOrigin() {
    return reference.distToOrigin();
  }


  @Override
  public int compareTo(Shape s) {
    double areaThis = this.area();
    double areaOther = s.area();

    if (areaThis < areaOther) {
      return -1;
    } else if (areaOther < areaThis) {
      return 1;
    } else {
      return 0;
    }
  }
}
```

LET’S CREATE EQUALS FOR CIRCLE

- What does it mean for our Circle to be “equal to” another Circle?
- Let’s override the equals method (and hashcode) to make it work
  - hashcode() override isn’t strictly needed for what we are doing today, but the general rule is: **IF** you override equals() override hashcode() too

REVIEW OF TERMS : ABSTRACTION

- Abstraction (the concept) allows us to ignore unessential details
  - Verb: **Abstraction** as an activity: the *process* of thinking where unessential details are ignored, and conceptual "things" are produced to help us solve problems
  - Noun: **Abstraction** as an entity: The resulting "things" that are produced from the process of abstraction

REVIEW OF TERMS: TYPE

- A Type is the description of external behavior
  - Keep this in mind for our discussion of interfaces
- In languages like Java, we can describe different “kinds” of types
  - Primitive types
  - Abstract data types (like Queues, Sets, etc.)
  - User defined types (and types provided by system libraries)

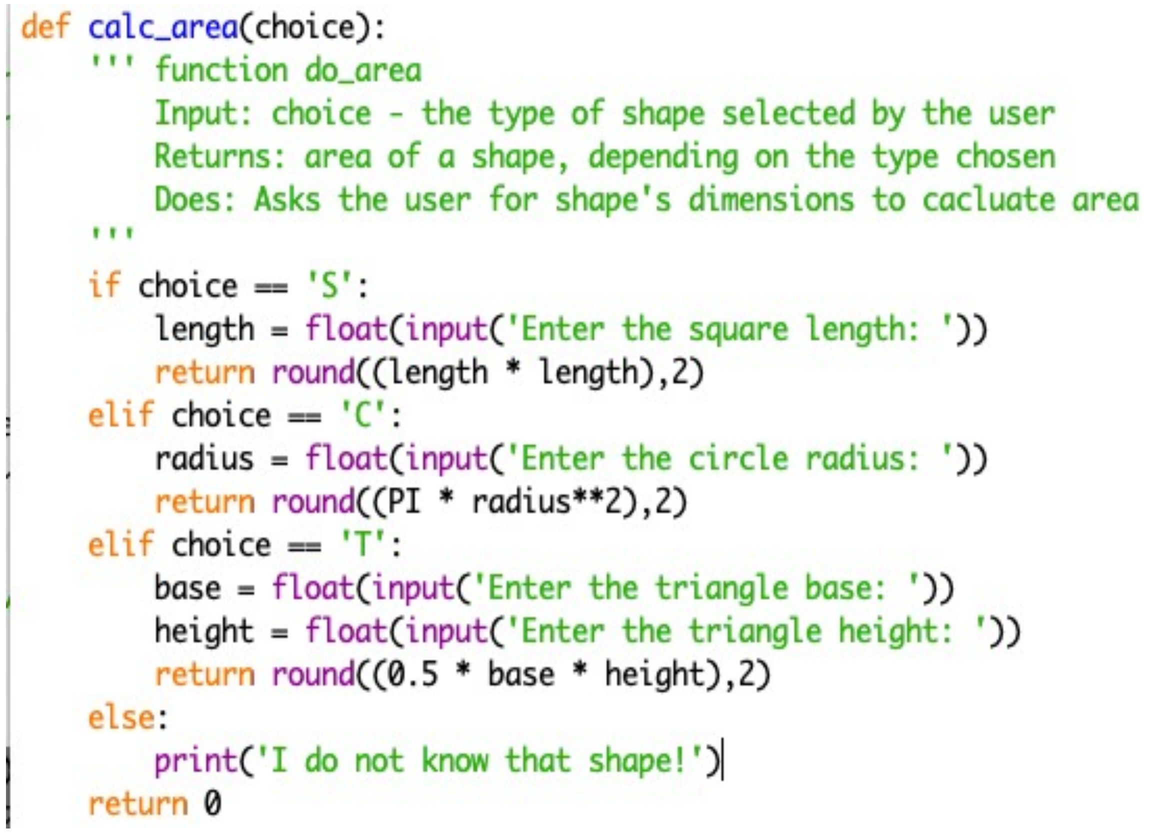
REVIEW: ENUM

- An enum is a finite set of conceptual values
  - Typically, we are not concerned about the *underlying value* how the enumeration is represented
  - We do care about the *conceptual* value
  - We may care about *sequencing* (predecessor/successor relationship)
  - Good for representing a variation in data values. **Not** good for variation in **behavior**

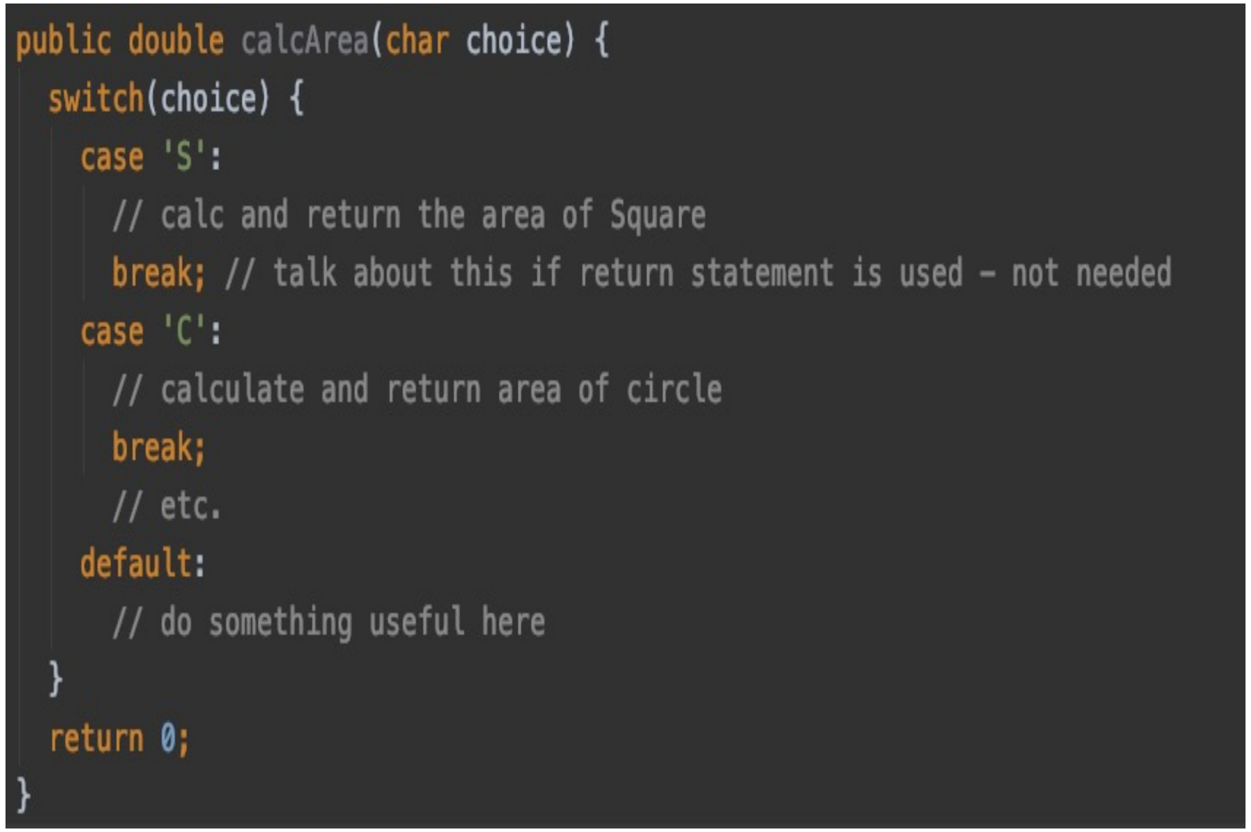
HOW DO WE EFFECTIVELY MODEL VARIATIONS IN BEHAVIOR?

- An interface is a “protocol” (in other languages) it describes the external behavior expected by anything that implements it. Effectively forms the “contract” for external behavior.
  - “Clean” basis for our User Defined Types
- An Abstract Class is a class designed to be inherited from, but NOT instantiated itself. Abstract classes can be conceptual, but most often placeholder for common elements/operations the rest of the inheritance hierarchy can reuse.

VARIATION IN BEHAVIOR - WHAT'S "WRONG" WITH THIS APPROACH?



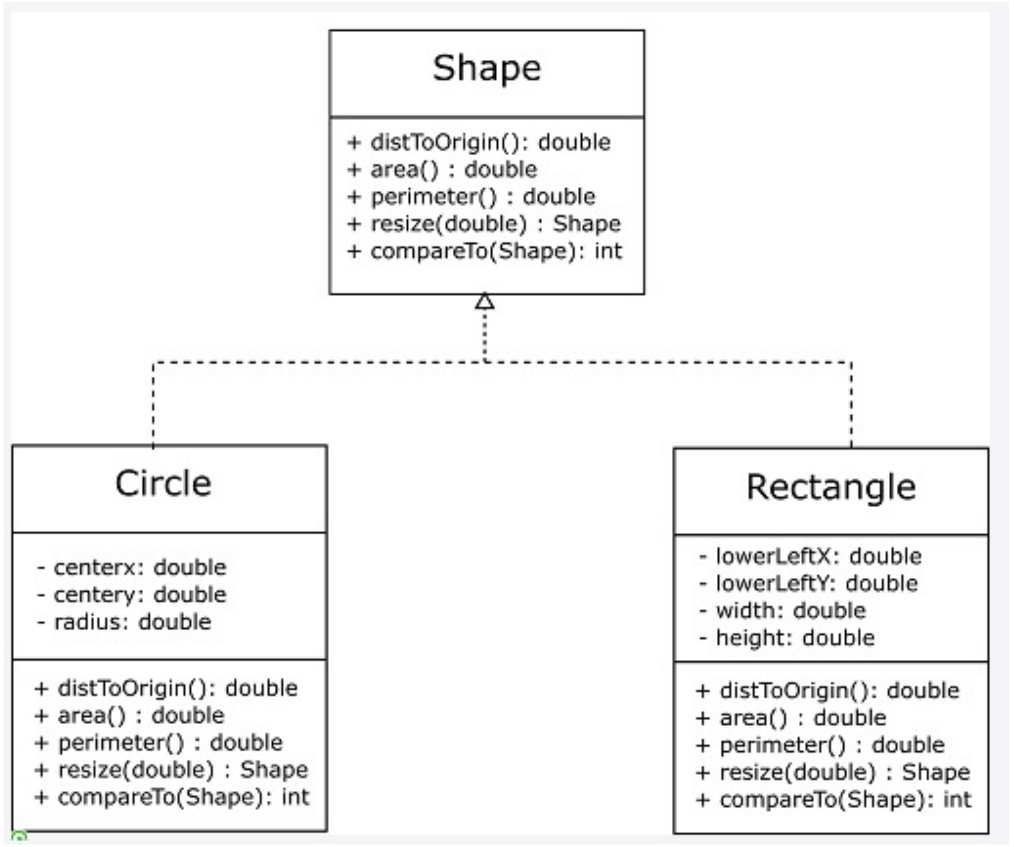
```python
def calc_area(choice):
    ''' function do_area
        Input: choice - the type of shape selected by the user
        Returns: area of a shape, depending on the type chosen
        Does: Asks the user for shape's dimensions to cacluate area
    '''
    if choice == 'S':
        length = float(input('Enter the square length: '))
        return round((length * length),2)
    elif choice == 'C':
        radius = float(input('Enter the circle radius: '))
        return round((PI * radius**2),2)
    elif choice == 'T':
        base = float(input('Enter the triangle base: '))
        height = float(input('Enter the triangle height: '))
        return round((0.5 * base * height),2)
    else:
        print('I do not know that shape!')
    return 0
```



```java
public double calcArea(char choice) {
  switch(choice) {
    case 'S':
      // calc and return the area of Square
      break; // talk about this if return statement is used - not needed
    case 'C':
      // calculate and return area of circle
      break;
      // etc.
    default:
      // do something useful here
  }
  return 0;
}
```

VARIATION IN BEHAVIOR: OO APPROACH

- All Shapes can answer their area, perimeter, etc.
  - But different TYPES of shapes have different algorithms to DO what they need
  - Same operation list, variation in behavior
  - **Interfaces** are the best option here
    - Interface (protocol) is our contract
    - Specifies the Type
    - Operation list tells what the type does
    - *Is-A*, or *Is-A-Kind-Of*



PROTOCOLS & ABSTRACT CLASSES

- A Java **interface** is a “protocol” (in other languages) it describes the external behavior expected by anything that implements it. Effectively forms the “contract” for external behavior.
  - “Clean” basis for our User Defined Types
- An **Abstract Class** is a class designed to be inherited from, but NOT instantiated itself. Abstract classes can be conceptual, but most often placeholder for common elements/operations the rest of the inheritance hierarchy can reuse.

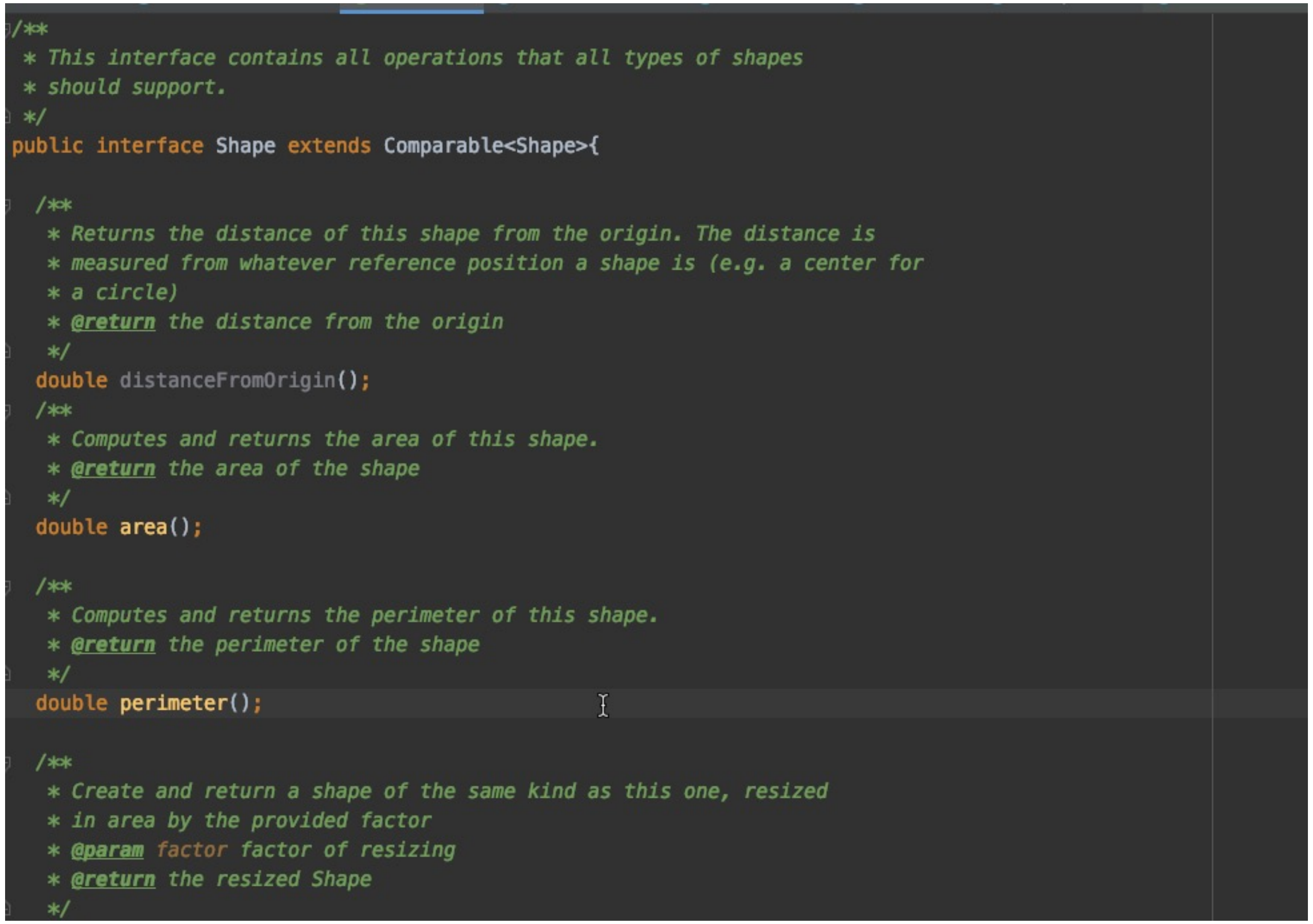
INTERFACES

- Interfaces allow us to create a specification for common operations
  - They allow us to determine the <u>external</u> behavior of our user-defined types, without specifying how that behavior should be implemented
  - They allow us to deal generically with any object that implements the interface
  - They also give us the beginnings of what is required for Design By Contract

INTERFACES & DBC

- Design By Contract
  - Expect a certain condition to be guaranteed on entry (precondition). Client code is obligated to fulfill this, or service code can ignore the request or raise an exception.
  - Guarantee a specific result on exit: (postcondition). Service code is obligated to provide this.
  - Maintain any class invariants on entry and exit.
- Interfaces give us the external specification for what is required (preconditions) and guaranteed (post-conditions)

INTERFACES



```
/**
 * This interface contains all operations that all types of shapes
 * should support.
 */
public interface Shape extends Comparable<Shape>{

  /**
   * Returns the distance of this shape from the origin. The distance is
   * measured from whatever reference position a shape is (e.g. a center for
   * a circle)
   * @return the distance from the origin
   */
  double distanceFromOrigin();
  /**
   * Computes and returns the area of this shape.
   * @return the area of the shape
   */
  double area();

  /**
   * Computes and returns the perimeter of this shape.
   * @return the perimeter of the shape
   */
  double perimeter();

  /**
   * Create and return a shape of the same kind as this one, resized
   * in area by the provided factor
   * @param factor factor of resizing
   * @return the resized Shape
   */
```

NB: This is the Shape Interface from Week 3 “Module Notes” in Canvas, NOT the Week 1 code we explored in our first lecture

Interfaces is a contract of **what** we promise to provide **IF** we implement a Type that adheres to this protocol (group of related methods)

Interfaces do not tell us **how** we should implement the methods

INTERFACES

- Think of interfaces as an abstract specification for some type.
- **Concrete** types will **implement** interfaces
  - Java keyword `implements` is used to specify a class’ intention for this
  - The concrete types may be conceptually unrelated to each other
  - Interfaces allow for cross-hierarchy polymorphism
- In the previous slide, any Shape that implements the interface should be able to:
  - Answer its distance from the origin
  - Answer its area
  - Answer its perimeter, and
  - Resize itself

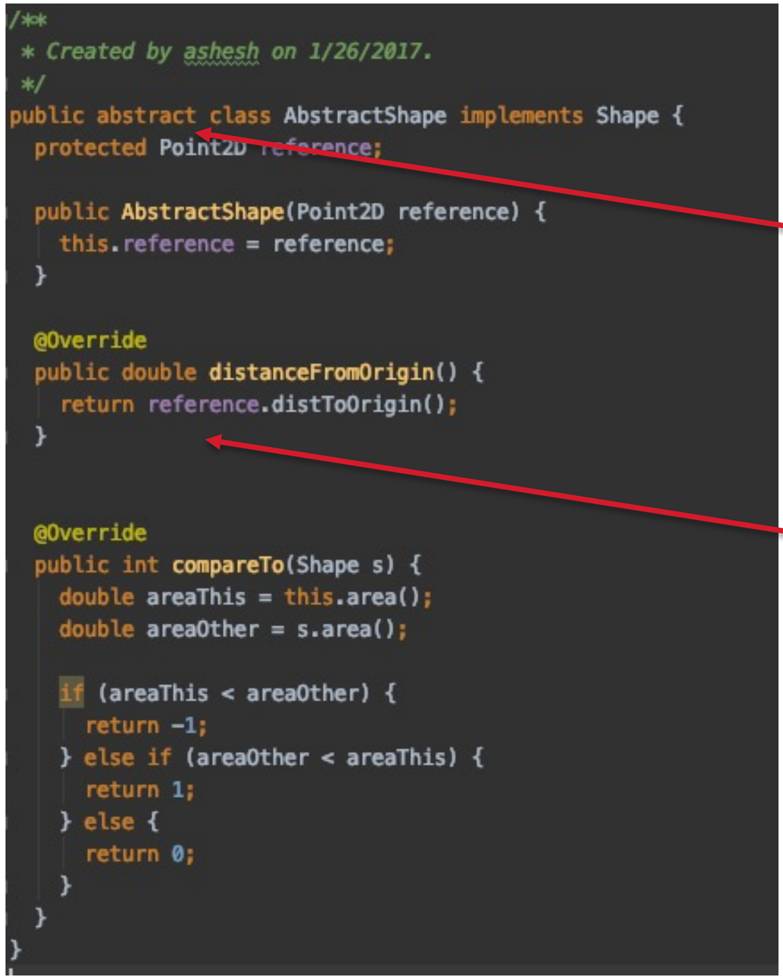
LET’S DO: REFACTOR OUR CIRCLE WITH INTERFACES

- Let’s refactor our Circle class to make use of interfaces.
- Then we’ll extend our program to handle other types of shapes

INHERITING INTERFACES

- Interfaces are abstractions that specify external behavior
- Interfaces can inherit from other interfaces
- Abstract classes can also be specified to implement an interface
  - Abstract classes are classes that we use in an inheritance hierarchy as a conceptual "placeholder" and for code reuse
- At some point, however, a **concrete** class will be specified to implement the interface (or extend an abstract class) and actually be ready to be instantiated

ABSTRACT CLASSES



```
/**
 * Created by ashesh on 1/26/2017.
 */
public abstract class AbstractShape implements Shape {
  protected Point2D reference;

  public AbstractShape(Point2D reference) {
    this.reference = reference;
  }

  @Override
  public double distanceFromOrigin() {
    return reference.distToOrigin();
  }


  @Override
  public int compareTo(Shape s) {
    double areaThis = this.area();
    double areaOther = s.area();

    if (areaThis < areaOther) {
      return -1;
    } else if (areaOther < areaThis) {
      return 1;
    } else {
      return 0;
    }
  }
}
```

Abstract classes cannot be instantiated

- Use the modifier abstract
- Abstract classes have uncompleted (abstract) method definitions, or have not implemented required methods from interfaces

Abstract classes can provide some code that will be reused in derived classes (subclasses)

INHERITANCE

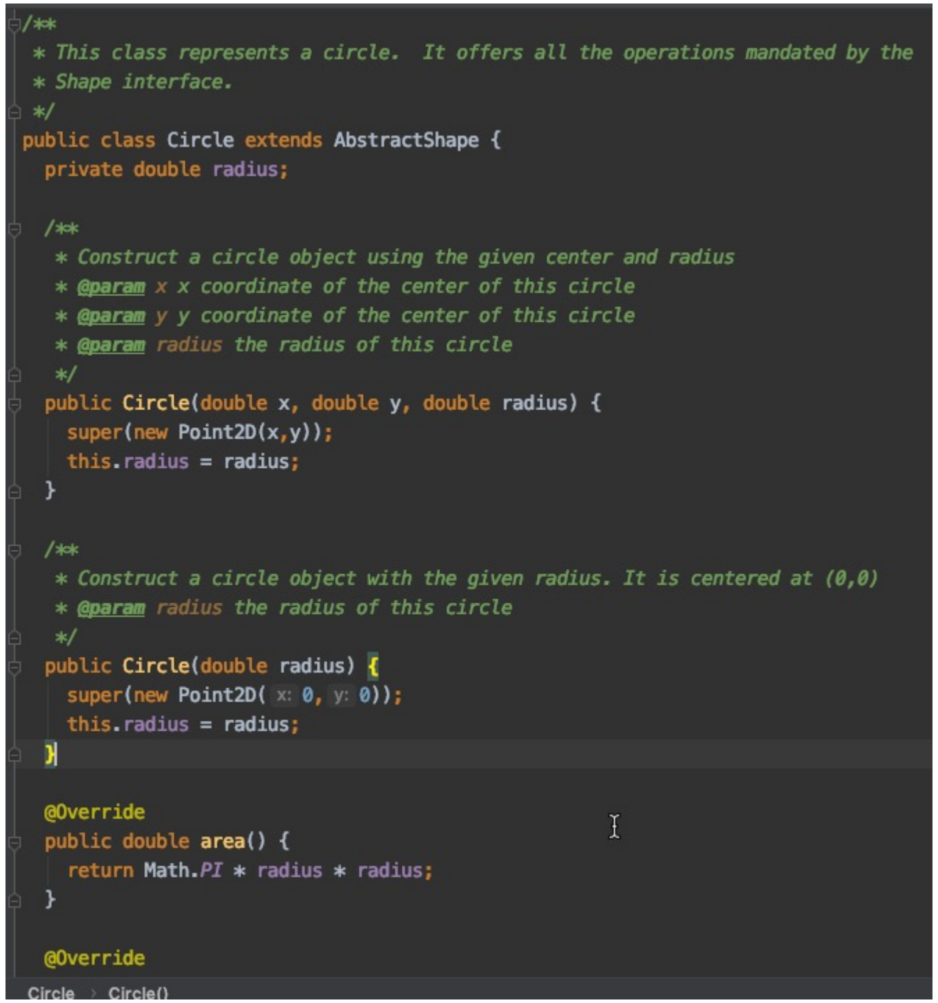
- Inheritance is the processes by which a new class (subclass aka derived class aka child class) is created from another existing class
  - Not talking about runtime objects here, but the actual class specification
- The new class reuses features (attributes & operations) from the existing class (superclass aka base class aka parent class)
  - Code reuse is one focus of inheritance. Conceptual hierarchies is another
  - Subclasses can also add/extend new features to what they get from their superclasses
- In Java, the **`extends`** keyword is used to specify that one class inherits from an existing class
  - Java supports single inheritance (but **classes can implement multiple** interfaces)

WHERE DO ABSTRACT CLASSES COME FROM?

- Bottom-up “Harvesting”
  - Notice commonality across 2 or more classes that implement the same TYPE. Gather common reusable elements, push them into a common superclass. This inserts a super-class/subclass relationship into the existing supertype/subtype implementation
- Top-Down “Design for Reuse”
  - Have some knowledge of the domain. Accept the pros/cons of using inheritance to share structure & fabric throughout the hierarchy.
  - Create an Abstract class to hold common elements for the subclasses. Often this will follow supertype/subtype, but sometimes it will NOT (those cases are simply for code reuse).

<I JUST WANNA KNOW, WHAT'S THE NAME OF THE PLAYER ON FIRST BASE?>*
*TRANSLATED FROM BLEHLISH
NO, WHAT PLAYS ON SECOND BASE.
<I'M NOT ASKING YOU WHO'S ON SECOND.>
WHO'S ON FIRST.
<I DON'T KNOW!>
HE'S ON THIRD.
<HOW DID I GET TO THIRD?!?>
YOU SAID HIS NAME.
little-vampires.com
<IF I SAID THE THIRD BASEMAN'S NAME, WHO DID I SAY IS PLAYING THIRD?>
NO. WHO'S ON FIRST.
<WHAT?!?>
WHAT'S ON SECOND.
<I DON'T KNOW!!>
HE'S ON THIRD.
<THERE I GO, BACK TO THIRD BASE!>
© 2015 Rebecca Hicks

INHERITANCE - CONCRETE CLASSES

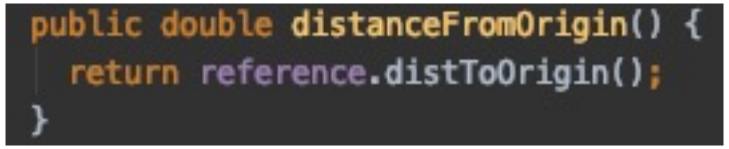


```
/**
 * This class represents a circle.  It offers all the operations mandated by the
 * Shape interface.
 */
public class Circle extends AbstractShape {
  private double radius;

  /**
   * Construct a circle object using the given center and radius
   * @param x x coordinate of the center of this circle
   * @param y y coordinate of the center of this circle
   * @param radius the radius of this circle
   */
  public Circle(double x, double y, double radius) {
    super(new Point2D(x,y));
    this.radius = radius;
  }

  /**
   * Construct a circle object with the given radius. It is centered at (0,0)
   * @param radius the radius of this circle
   */
  public Circle(double radius) {
    super(new Point2D( x: 0, y: 0));
    this.radius = radius;
  }

  @Override
  public double area() {
    return Math.PI * radius * radius;
  }

  @Override
```

Circle › Circle()

Concrete classes are intended to be instantiated
They must implement all abstract methods (either from abstract classes or from interfaces)

Subclasses can choose to override methods from their superclass, or they might accept the implementation provided



```
public double distanceFromOrigin() {
  return reference.distToOrigin();
}
```

Circle reuses `AbstractShape’s` version of `distanceFromOrigin()`

SUBTYPING VS SUBCLASSING

- Remember: protocols/contracts (Java interfaces) are TYPE specifications
  - The protocol tells us what the type does, NOT how the type does it, or what kind of internal representation it uses
  - In CS5001, we learned about Abstract Data Types (like Queues and Stacks). Recall we chose a Python List for one implementation, but that was NOT the only alternative
- Remember: classes are ONE style of implementation. In Java it is THE style of implementation. In other languages (like C) there is no such thing as a class, so there is no such thing as "subclassing".
  - Types and subtyping is a concept that is consistent across all languages, and even hardware

CONCEPT: SUBTYPING VS SUBCLASSING

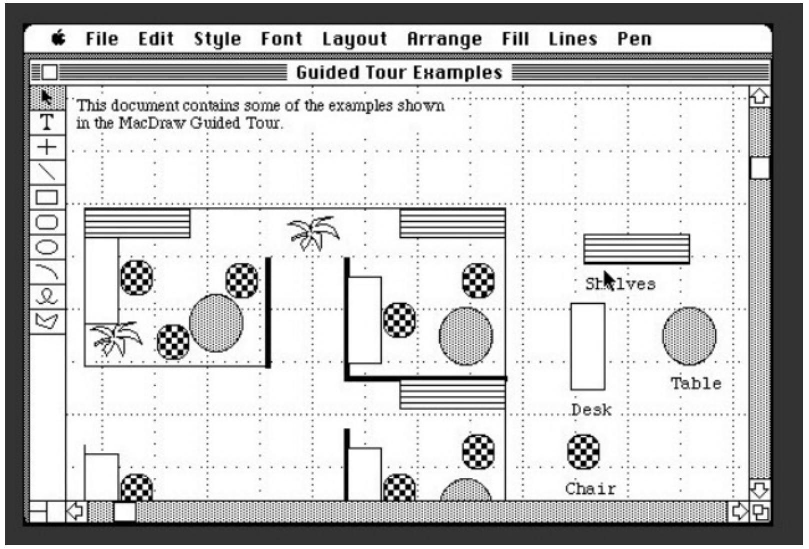
- Consider the Type: Lock
  - Simple protocol:

```
lock()
unlock()
```

Subtypes: Hardware cylinder lock, Keycard lock (hardware/software), Cyber security lock
Implementation is different. No “subclassing”. Classes are a OO software mechanism

CONCEPT: SUBTYPING VS SUBCLASSING

- MacDraw
  - Apple's "wow factor" drawing program from1984
  - Written in Pascal (non-OO)


File Edit Style Font Layout Arrange Fill Lines Pen
Guided Tour Examples
This document contains some of the examples shown in the MacDraw Guided Tour.
Shelves
Table
Desk
Chair

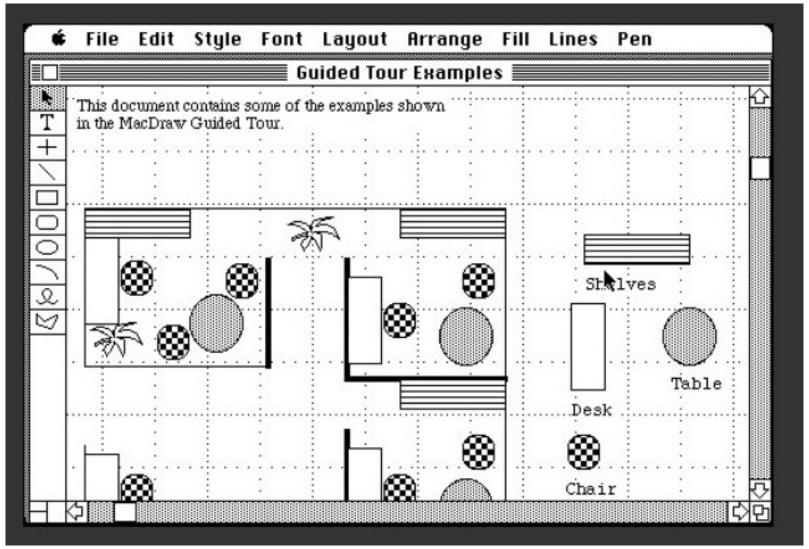
MacDraw / Programming language

Pascal

Elements: Circles, Rectangles, Freehand shapes, other Shapes

SUBTYPING VS SUBCLASSING – COMIC BOOK

- MacDraw – Written in Pascal

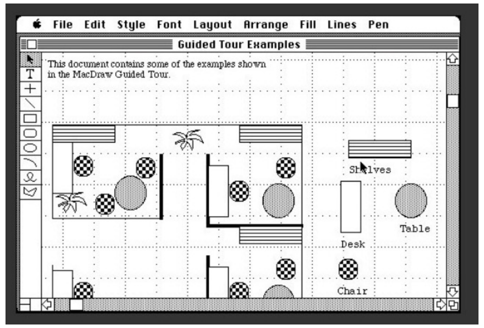

File Edit Style Font Layout Arrange Fill Lines Pen
Guided Tour Examples
This document contains some of the examples shown in the MacDraw Guided Tour.
Shelves
Table
Desk
Chair

Without subclasses, Apple will NEVER be able to create a drawing program with Shapes! Muhahahah!

Elements: Circles, Rectangles, Freehand shapes, other Shapes

SUBTYPING VS SUBCLASSING – COMIC BOOK

- MacDraw – Written in Pascal



This document contains some of the examples shown in the MacDraw Guided Tour.

Without subclasses, Apple will NEVER be able to create a drawing program with Shapes! Muhahahah!

Elements: Circles, Rectangles, Freehand shapes, other Shapes

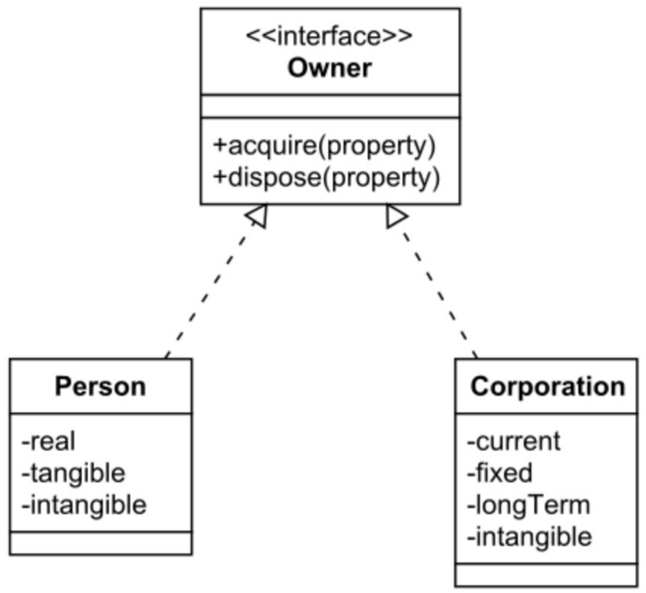
Someone needs a tutorial on subtyping vs. subclassing. Ben, take care of this please.

Sure thing, Stretcho. I’m going to LOVE tutoring Doom on this topic. It’s Clobberin’ Time!

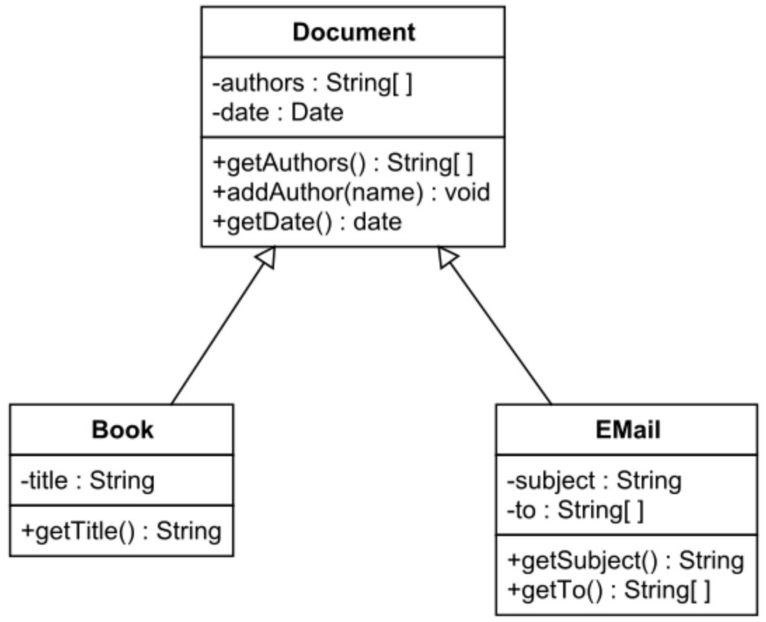
YOU TRY!

- Given the Shape Interface, create a concrete Triangle class that implements the Shape protocol. Feel free to use an abstract class for code reuse if you wish.

A TOUCH OF UML



UML “Realization” -> “implements”



UML “Generalization” -> “Inheritance/Extends”

Diagrams from http://www.cs.utsa.edu/~cs3443/uml/uml.html

Q & A

THANKS!

- Stay safe, be encouraged, & see you next week!