COIS 2240

Software Design & Modelling

Lectures 3 & 4

Concepts of Object Orientation in Java

Taher Ghaleb

TRENT UNIVERSITY

Parts of the content on the slides were sourced from / inspired by the textbook slides

What is object orientation?

- **Procedural paradigm:**
  - Software is organized around the notion of *procedures*
    - *functions* or *routines*
  - *Procedural abstraction*
    - Works as long as the data is simple
  - *Data abstractions*
    - *Records* and *Structures*
    - Group together the pieces of data that describe some entity
    - Helps reduce the system's complexity
- **Object-oriented paradigm:**
  - Organizing procedural abstractions in the context of data abstractions

Example of procedural paradigm

```
if account is of type checking then
   do c
else if account is of type savings then
   do something else
else
   do yet another thing
end if
```

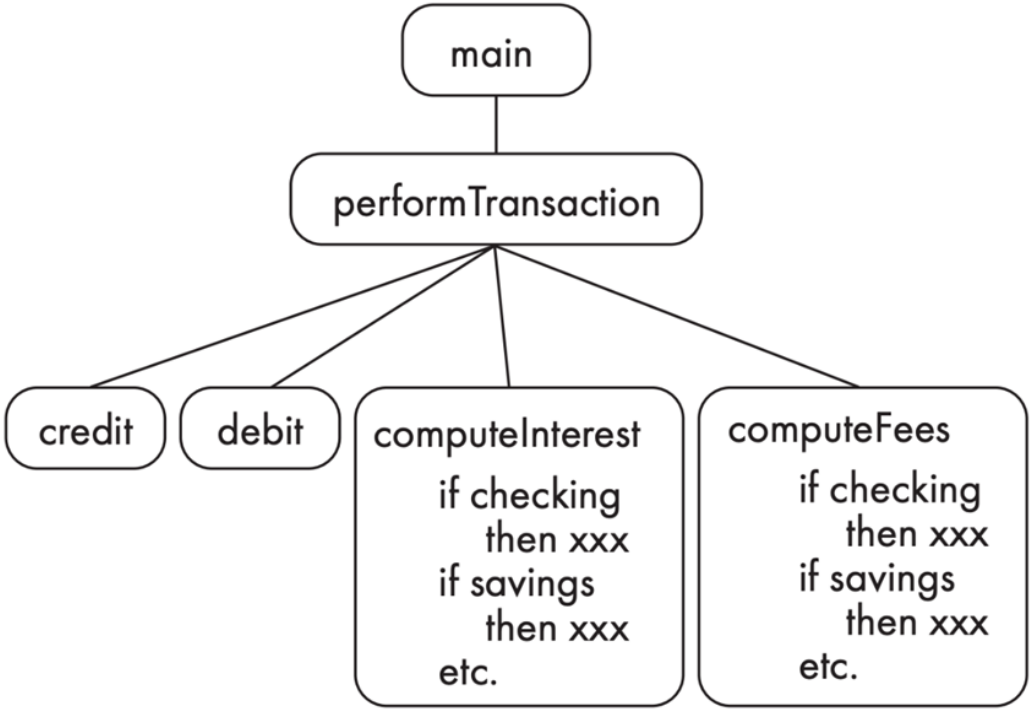
- One client can have several accounts of different types
- Some accounts can be held jointly
- Different account holders might have different rights

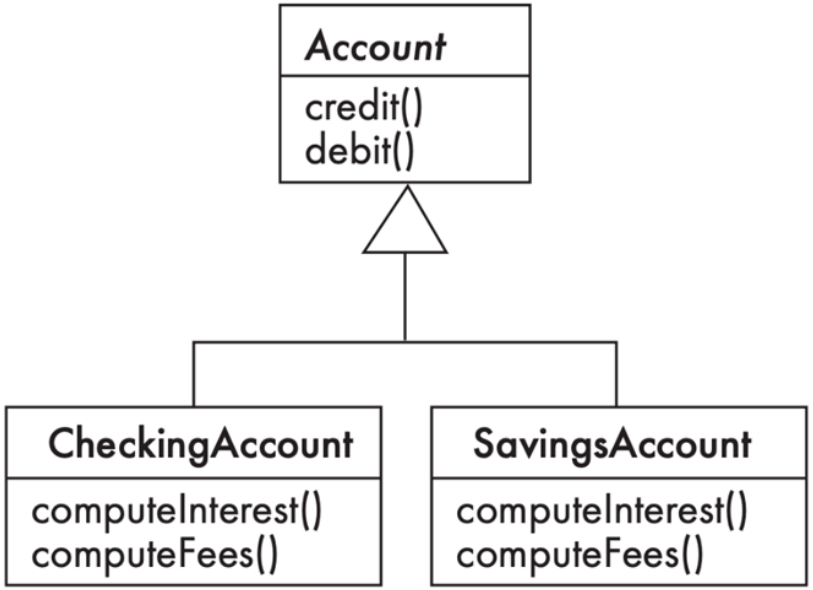
Object-oriented paradigm

An approach in which all computations are performed in the context of objects

- Objects are instances of classes, which:
  - are data abstractions
  - contain procedural abstractions that operate on the objects
- A running program can be seen as a collection of objects collaborating to perform a given task

A view of the two paradigms



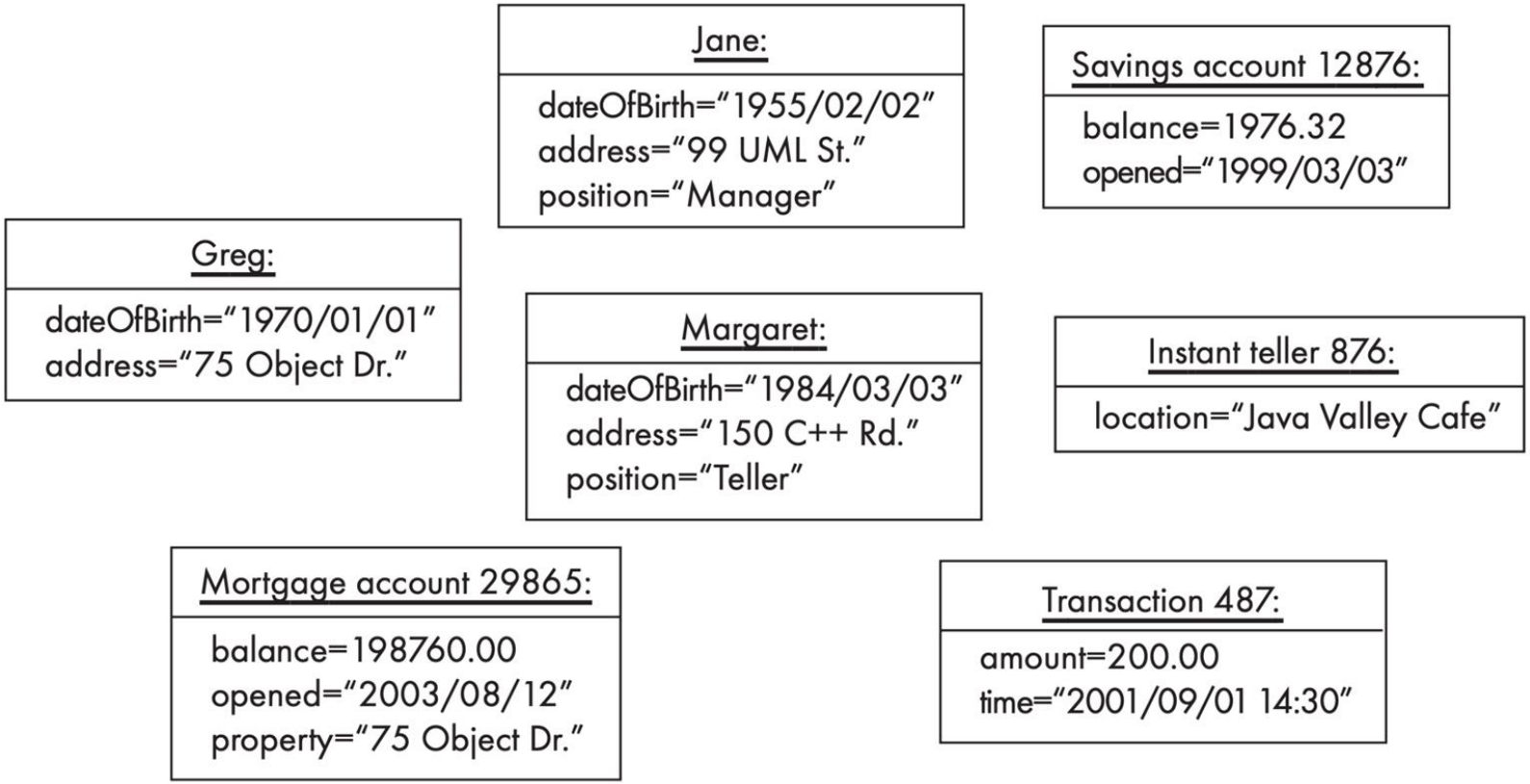


The concept of objects

Object is a chunk of structured data in a running software system

- Has ***properties***
  - *Represent its state*
- Has ***behaviors***
  - *How it acts and reacts (possibly changing its state)*
  - May simulate the behavior of an object in the real world

Examples of objects



Classes

A class is a unit of abstraction in an object oriented (OO) program

- Represents similar objects
  - Its *instances*
- A kind of software module
  - Describes its instances' structure (properties)
  - Contains *methods* to implement their behavior

Class instances

- An object is an instance of a class
- Objects are created using the `new` keyword, which allocates memory and calls the class constructor
  - `ClassName objName = new ClassName();`
  - `new ClassName();`
- Once created, you can call methods: `objName.methodName();`

Is something a class or an instance?

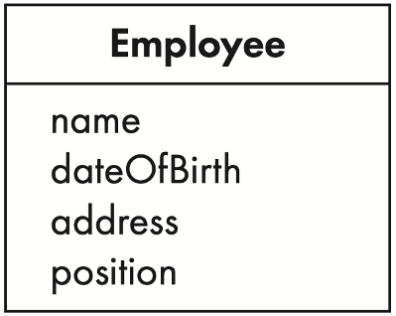
- Something should be a ***class*** if it could have instances
- Something should be an ***instance*** if it is clearly a *single* member of the set defined by a class

A class or instance? If instance, name a suitable class for it.

- *General Motors*
- *Automobile company*
- *Boeing 777*
- *Computer science student*
- *Mary Smith*
- *Game*
- *Board game*
- *Chess*
- *University course SEG 2100*
- *Airplane*
- *The game of chess between Tom and Jane which started at 2:30 pm yesterday*
- *The car with serial number JM 198765T4*

Class members

- Member fields:
  - Often called ***member*** *variables*
  - Describe the class
  - Attributes
    - *Simple data*
  - Associations
    - *Relationships with other classes*
    - *e.g. supervisor (also class Employee), coursesTaken (class Course)*
- Member methods
  - Often called ***member*** methods
  - Implement a certain functionality of the class



Inheritance

Superclasses

- Contain features common to a set of subclasses

Inheritance hierarchies

- Show the relationships among superclasses and subclasses
- A triangle shows a *generalization*

Inheritance

- Subclasses *implicitly* possess all features in superclasses
- Private members are not inherited

The IsA Rule

Java allows single class inheritance

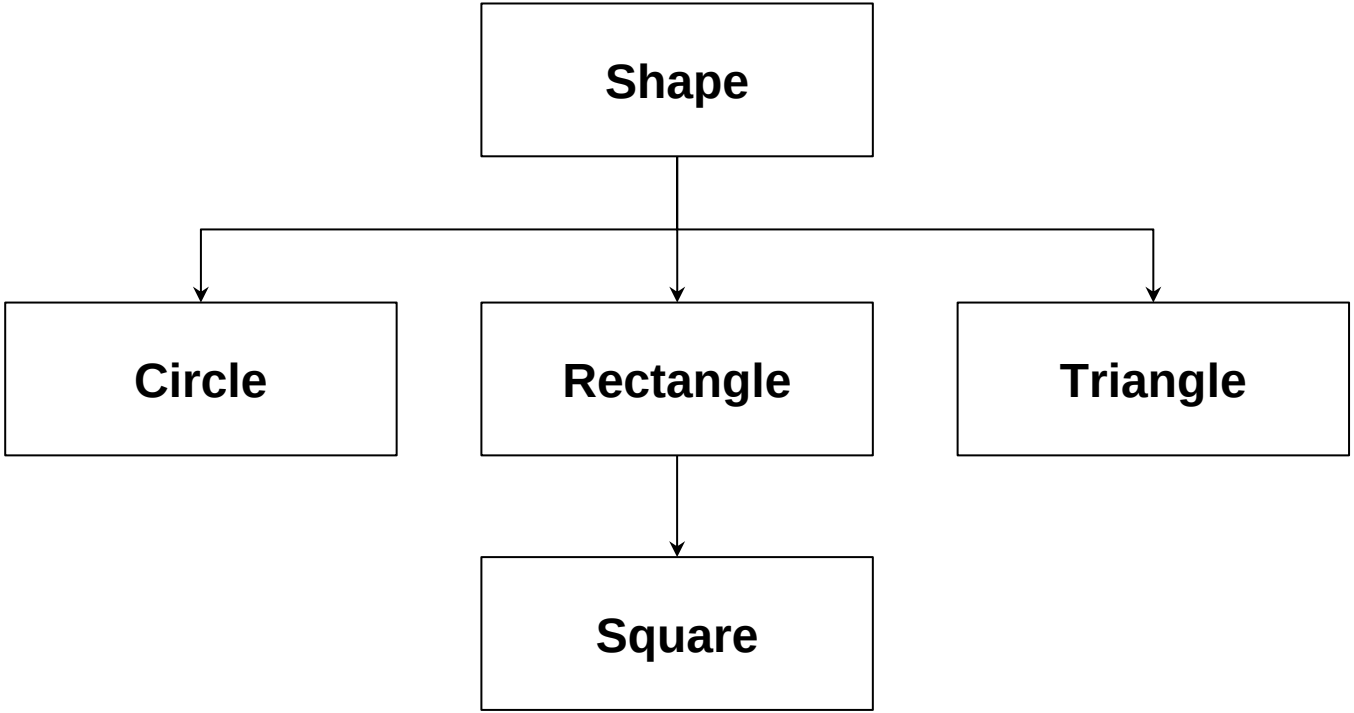
Always check generalizations to ensure they obey the *isa* rule

- "A checking account ***is an*** account"
- "A circle ***is a*** shape"

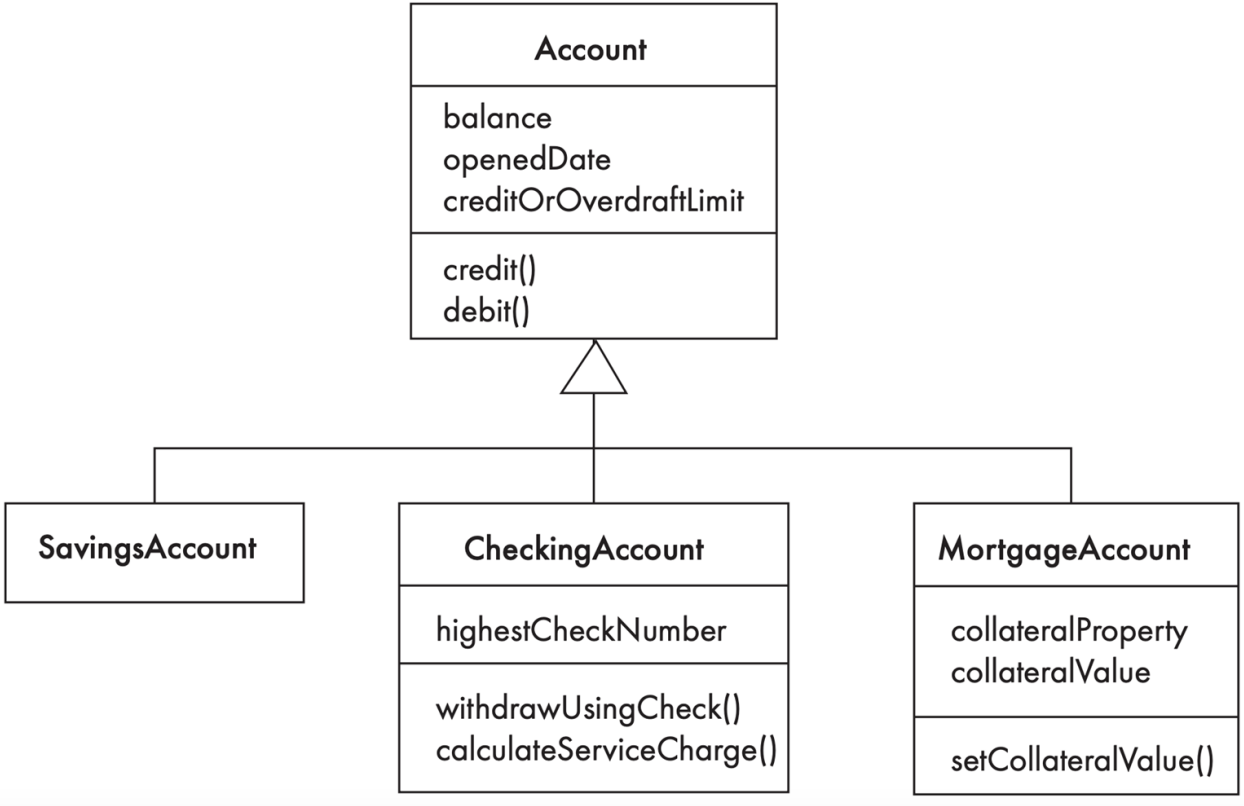
Should class 'Province' be a subclass of class 'Country'?

- No, it violates the *isa* rule
  - "A province ***is a*** country" is invalid!

Example inheritance hierarchy



Inherited features must make sense



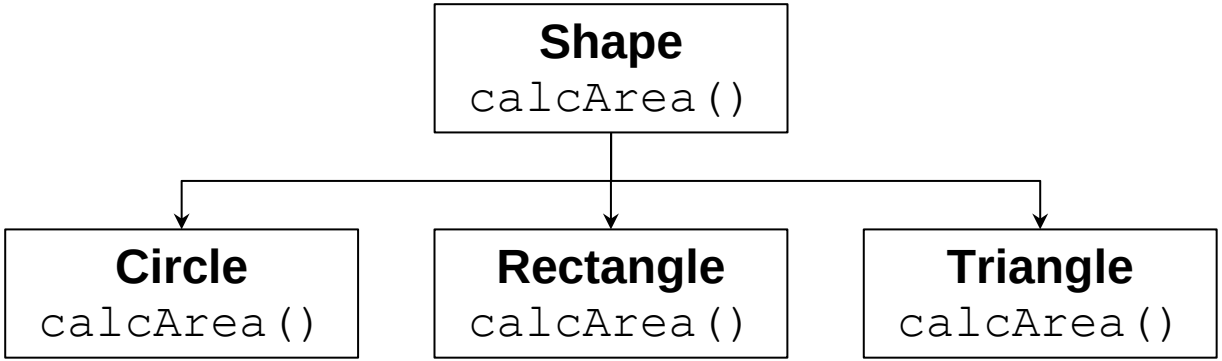
Polymorphism

An *abstract operation may be performed in different ways* in different classes

- Requires *multiple methods with the same name*
- The choice of which one to execute depends on the object created
- Reduces the need for programmers to code many if-else or switch statements
- Forms of polymorphism:
  - Overriding
  - Overloading

Overriding

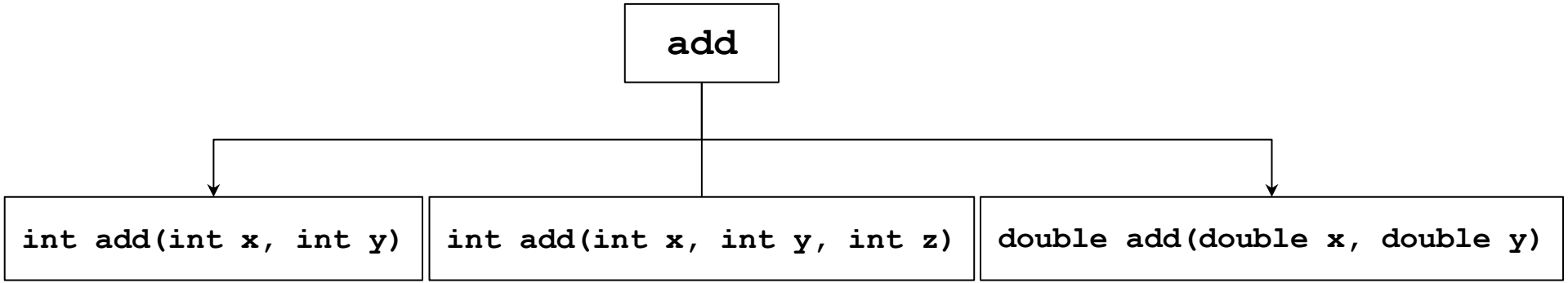
A method would be inherited, but a subclass contains a new version instead



- Method visibility cannot be reduced
- Which method to run? Dynamic binding

Overloading

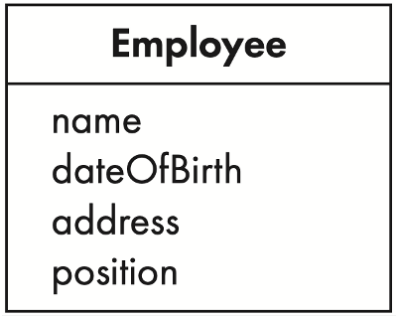
Multiple methods with the same name, but with different parameter lists (*number*, *types*, and/or *order*)



Instance variables

Variables defined inside a class corresponding to data present in each instance

- Also called ***fields*** or ***member*** *variables*
- Attributes
  – Simple data



- Associations
  – Relationships with other important classes
  – e.g. supervisor (also class Employee), coursesTaken (class Course)

Class variables

A *class variable's* value is *shared* by all instances of a class

- Also called a ***static*** variable
- If one instance sets the value of a class variable, then all the other instances see the same changed value
- Class variables are useful for:
  - Default or 'constant' values (e.g. `PI`)
  - Lookup tables and similar structures (e.g. `enum`)

Caution: *do not overuse class variables*

Operations & Methods

Operation

- A higher-level procedural abstraction (e.g., `calculateArea`)

Method

- A procedural abstraction used to implement the behavior of a class
- Several different classes can have methods with the same name
  - They implement the same abstract operation in ways suitable to each class
  - e.g. calculating the area for a rectangle and a circle

Naming conventions

Class

- Use *nouns* with the first letter *capitalized*
  - e.g. **`BankAccount`** not **`bankAccount`** or **`BANKACCOUNT`**
- Use *singular* nouns (e.g. **`Student`** not **`Students`**)
- Use the right level of generality (e.g. **`Vehicle`**, not **`Car`**)
- Use name with only *one* meaning (e.g. '**`Spring`**' has several meanings)

Members, static variables, parameters, local variables

- The first word is lowercase, and each subsequent word is firstCap
  - (e.g. **`studentAge`**, **`getName(int name)`**)

Constants

- All caps with an underscore (e.g. **`MAX_SIZE`**)

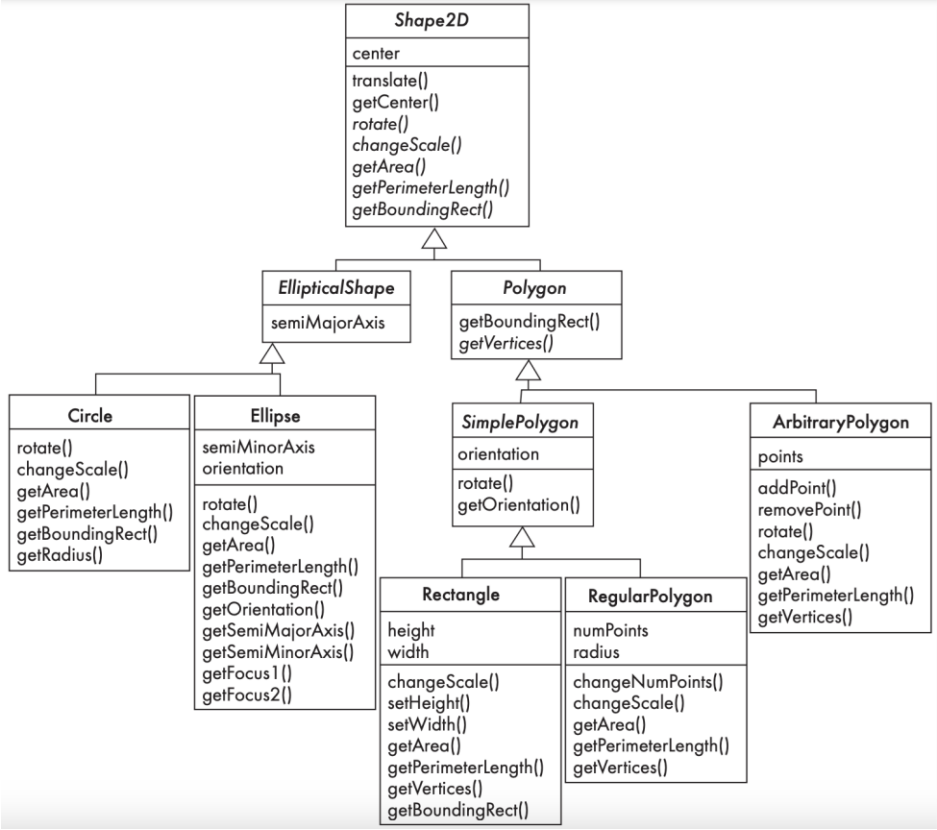
Member Access Modifiers

- *private* (within the class)
- *public* (everywhere)
- *protected* (within package, or outside package through child class)
- Default (no modifier) (within package, no access outside package)

Using *this* keyword

- A reference to the current object
- Used to:
  - Distinguish between class attributes and parameters
  - Call another constructor of the same class
  - Return the current object

A hierarchy of class inheritance with polymorphism



Constructors

- A special method that is used to initialize objects
- Has the same name as the class and does not have a return type
- A default constructor is created if no constructor explicitly defined
- Constructors can be overloaded

Using *super* in Constructors

- Used to:
  - call the constructor of the superclass (parent class)
  - call a method in the parent class
- Must be the first statement in the constructor if used
- If no `super()` call is made, a default one is created

Object reference

- Variable(s) pointing to the memory address where an object is stored
- Cloning creates a copy of an object
  - By implementing the `Cloneable` interface and override the `clone()` method
  - By defining a cloning mechanism from scratch

Properties (Setters / Getters)

Java

```java
public class Car {
    private String model;
    public String getModel() { return model; }
    public void setModel(String model) { this.model = model; }
}
```

C#

```csharp
public class Car {
    public string Model { get; set; }
}
```

Final Classes and Methods

- A *final* class cannot be subclassed (no other class can extend it)

- A *final* method cannot be overridden by subclasses, making its implementation unchanged in subclasses

Abstract classes and methods

An operation should be declared to exist at the highest class in the hierarchy where it makes sense

- The *operation* may be *abstract* (has no implementation) at that level
- If so, the *class* also must be *abstract*
  - **No instances can be created**
  - The **opposite** of an abstract class **is** a ***concrete*** class
- If a superclass has an abstract operation then its subclasses at some level must have a concrete method for the operation
  - Leaf classes must have or inherit concrete methods for all operations
  - Leaf classes must be concrete

Interfaces

- An *interface* is a completely "abstract class" that is used to group related methods with empty bodies
- It enahnces the generalization of software components and connecting them to each other
- Java allows a class to implement multiple interfaces using the *implement* keyword

Packages

- Used for organizing code, and if used, must be declared at the 1st line
  - E.g., ***package mysystem;***
- Contains classes and interfaces only
- Importing classes from existing packages using the *import* keyword
  - E.g., ***import java.util.List;***
  - ***import java.util.*;***

Cohesion and coupling

Cohesion

- Refers to how closely related and focused the responsibilities of a single class or module are
- High cohesion means a class has methods and attributes that are all related to a single purpose or functionality

Coupling

- Refers to the degree of interdependence between different classes or modules
- Low coupling means that classes are independent and have minimal knowledge of each other

We strive for high cohesion and low coupling

Java documentation (Javadoc) and comments

- **Single-Line Comments**: for brief explanations in the code
- **Multi-Line Comments**: for longer explanations, or comment out code

Javadoc

- Looking up unknown classes and methods will get you a long way towards understanding code
- Java documentation can be automatically generated by a program called **`javadoc`**
- Documentation is generated from the code and its comments
- You should format your comments as:
  - These may include embedded HTML

```
//
/*
*/
/**
*
*
*/
```

Concepts that define object orientation (I)

Identity

- Each object is *distinct* from other objects, and *can be referred to*
- Two objects are distinct *even if they have the same data*

Classes

- The code is organized using classes, each of which describes a set of objects

Inheritance

- The mechanism where features in a hierarchy inherit from superclasses to subclasses

Polymorphism

- The mechanism by which several methods can have the same name and implement the same abstract operation

Concepts that define object orientation (II)

Abstraction

- Object -> something in the world
- Class -> objects
- Superclass -> subclasses
- Operation -> methods
- Attributes and associations -> instance variables

Modularity

- Code can be constructed entirely of classes

Encapsulation

- A class is a container for its features (variables and methods) and defines an interface that controls access to them
- Details can be hidden in classes (*information hiding)*
  – Programmers do not need to know all the details of a class