



Usecase Diagram Contd...

Class Diagram



Examples from the Lab

Author – Identify team member who wrote this use case. We expect each team member to have at least 1 use case.

Purpose - What is the basic objective of the use-case. What is it trying to achieve?

Requirements Traceability – Identify all requirements traced to this use case - the F_n numbers from Section 3.2 above

Priority - What is the priority. Low, Medium, High. Importance of this use case being completed and functioning properly when system is deployed

Preconditions - Any condition that must be satisfied before the use case begins

Post conditions - The conditions that will be satisfied after the use case successfully completes

<Please be careful while filling up Pre and Post conditions>

Actors – Actors (human, system, devices, etc.) that trigger the use case to execute or provide input to the use case

Extends – If this is an extension use case, identify which use case(s) it extends <Study what “extends” actually means before proceeding>|

Flow of Events

1. Basic Flow - flow of events normally executed in the use-case
2. Alternative Flow - a secondary flow of events due to infrequent conditions
3. Exceptions - Exceptions that may happen during the execution of the use case

Includes (other use case IDs)

Notes/Issues - Any relevant notes or issues that need to be resolved

A sample usecase for template purpose



Use Case #15 (View My Books)

Author – Aman Singh

Purpose – User can see how many books they have added.

Requirements Traceability – F15

Priority – High. User can check their book availability.

Preconditions – User must have to be login before see him/her added books.

Post conditions – All books will showed which added by him/her.

Actors – User.

Extends – U8.

Flow of Events

1. Basic Flow – User open the Application and login. Click on view my book. Get all details.

2. Alternative Flow – None.

Includes – None.



Please note that...

- Associate the actors and use cases -- there shouldn't be any actor or use case floating without any connection
- Be careful on specifying relationships

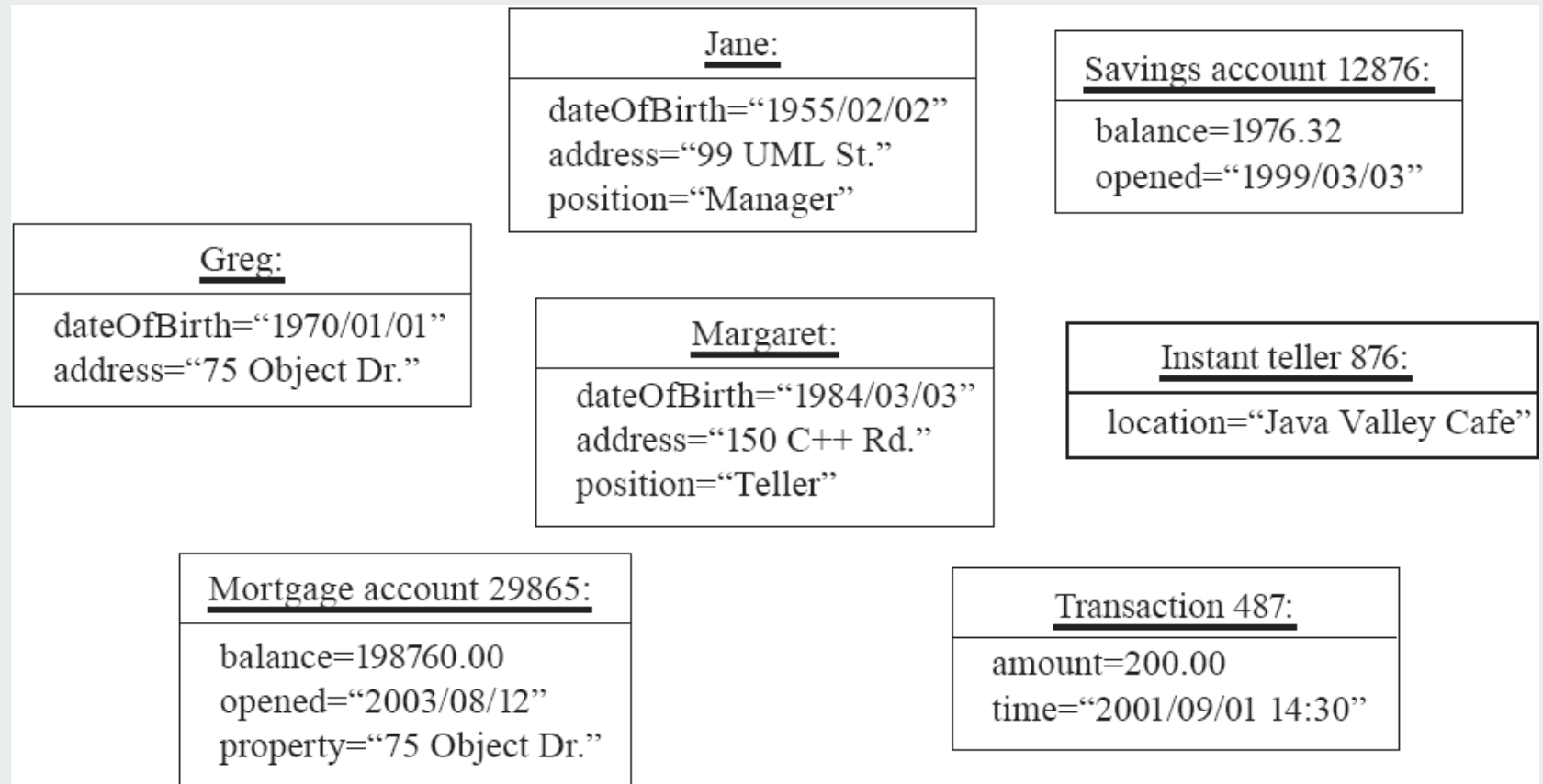


Object Oriented paradigm

An approach to the solution of problems in which all computations are performed in the context of objects.

- The 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

Objects





Objects

Object

- A chunk of structured data in a running software system
- Has *properties*
 - Represent its **state**
- Has *behaviour*
 - How it **acts and reacts**
 - May simulate the behaviour of an object in the real world



Classes

A class:

- A unit of abstraction in an object oriented (OO) program
- **Template or blueprint of objects**
- A kind of software module
 - Describes its object structure (properties)
 - Contains *methods* to implement their behaviour

Car class

```
class Car{  
    String company;  
    int speed;  
  
    void getSpeed(){  
  
        System.out.println(company + "  
        car's speed is " + speed + "  
        Km/hr");  
  
    }  
}
```

Multiple Objects



Company = Honda
Speed = 100
Honda car's speed is 100
Km/hr



Company = Jeep
Speed = 500
Jeep car's speed is 500
Km/hr



Company = BMW
Speed = 800
BMW car's speed is 800
Km/hr



Breed: Bulldog
Size: large
Colour: light gray
Age: 5 years

Dog1Object



Breed: Beagle
Size: large
Colour: orange
Age: 6 years

Dog2Object



Breed: German Shepherd
Size: large
Colour: white & orange
Age: 6 years

Dog3Object

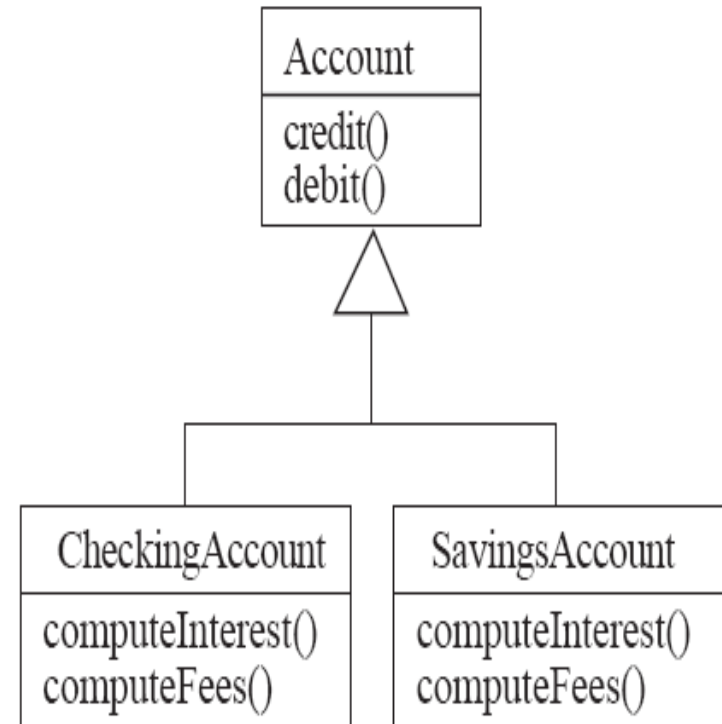
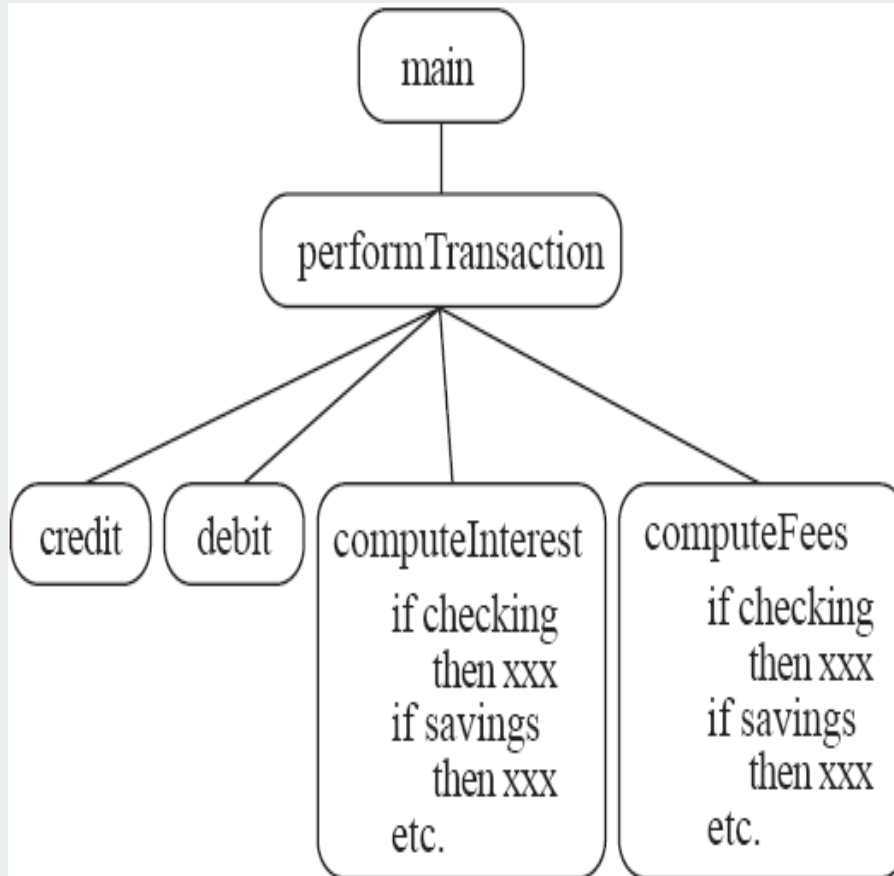




Is Something a Class or an Instance?

- *Student*
 - Class; instances are individual students.
- *Course*
 - Class; instance is CS3004 SE
- *Teacher with Employee ID – CSED100*
 - Instance of **Class Teacher**

A View of the procedural and object-oriented paradigms





Instance Variables

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

- Also called *fields* or *member variables*
- Attributes
 - Simple data
 - E.g. name, dateOfBirth
- Associations
 - Relationships to other important classes
 - E.g. supervisor, coursesTaken



Variables vs. Objects

- A variable

- Refers to an object
- May refer to different objects at different points in time
- Eg:

```
Student s1 = new Student();  
s1.name = "ABC";  
s1.computeGrade();
```

- An object can be referred to by several different variables at the same time

```
Student s2 = new Student();  
s2 = s1;
```

- Type of a variable

- Determines what classes of objects it may contain



Benefits of Object Orientation

- Object technology emphasizes modeling the real world and provides us with the stronger equivalence of the real world's entities (objects) than other methodologies.



Organizing Classes into Inheritance Hierarchies

- Super classes

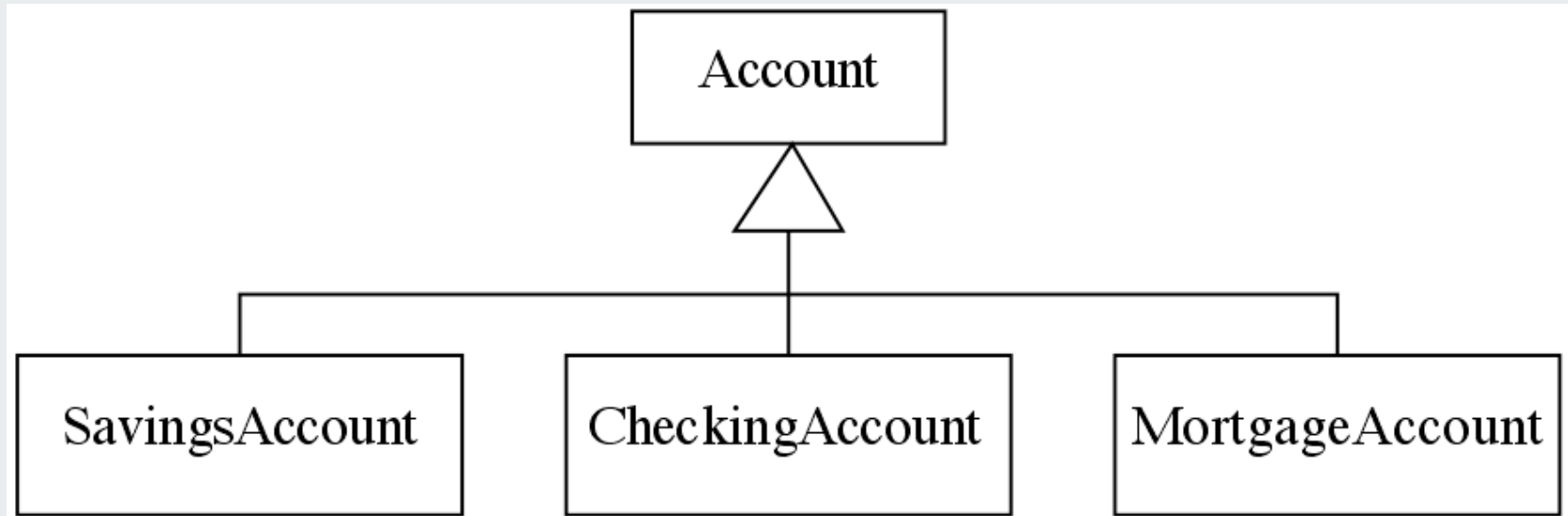
- Contain features **common** to a set of subclasses

- Inheritance hierarchies

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



An Example Inheritance Hierarchy



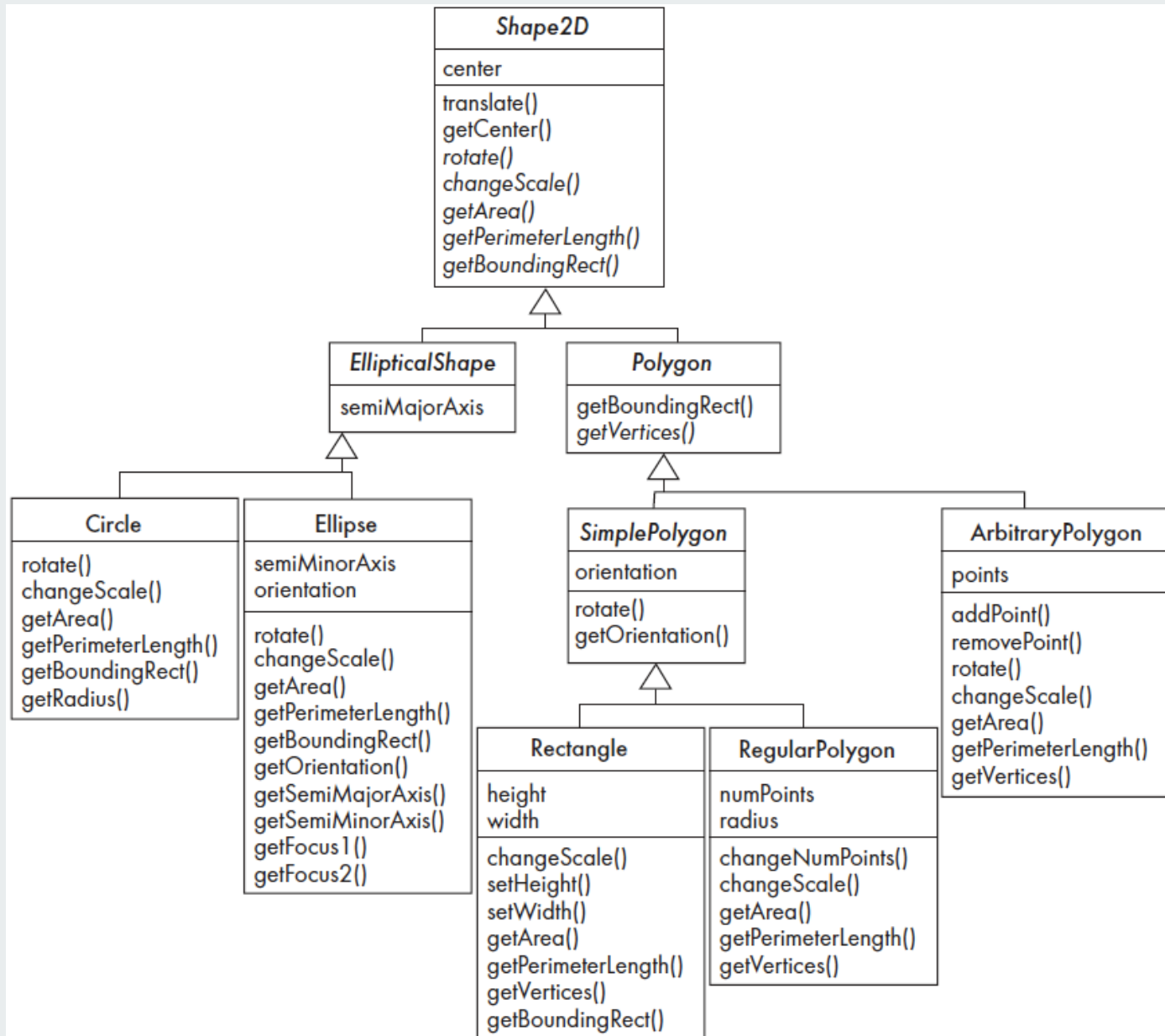
- Inheritance

- Subclasses implicitly have the features defined in its superclasses



The Is A Rule

- Always check generalizations to ensure they obey the is a rule
 - “A checking account *is an* account”
 - “A student is a person”
- Is ‘State’ a subclass of ‘Country’?
 - No, it violates the is a rule





Polymorphism

- A property of object oriented software by which an *abstract operation may be performed in different ways* in different classes.
 - Requires that there be *multiple methods of the same name*
 - The choice of which one to execute depends on the object that is in a variable
 - Reduces the need for programmers to code many if-else or switch statements

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



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* (lacking 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 super class 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



Overriding

- A method would be inherited, but a subclass contains a new version instead
 - For restriction
 - For extension
 - For optimization



How a decision is made on which method to run?

```
Rectangle obj = new Rectangle();  
obj.getBoundingRect();  
Shape2D obj= new Rectangle();  
obj.getBoundingRect();
```

1. If there is a concrete method for the operation in the current class, run that method.
2. Otherwise, check in the immediate super class to see if there is a method there; if so, run it.
3. Repeat step 2, looking in successively higher super classes until a concrete method is found and run.
4. If no method is found, then there is an error



Dynamic binding

- Occurs when decision about which method to run can only be made at *run time*
 - Needed when:
 - A variable is declared to have a superclass as its type, and
 - There is more than one possible polymorphic method that could be run among the type of the variable and its subclasses



Concepts that Define Object Orientation

- The following are necessary for a system or language to be OO
 - **Identity**
 - Each object is *distinct* from each other object, 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.