1878

Western

UNIVERSITY · CANADA

Topic 3

Review of Object Orientation (Chapter 2)

Computer Science 2212b
Introduction to Software Engineering
Winter 2014

Jeff Shantz
jeff@csd.uwo

2.1 – What is Object Orientation?

Procedural paradigm:

- Software is organized around the notion of *procedures*
- *Procedural abstraction*
    - Works as long as the data is simple
- *Adding data abstractions*
    - Groups together pieces of data that describe some entity
    - Helps reduce the system’s complexity.
    - Such as *records* and *structures*

A concept or idea not associated with any specific instance

Object-oriented paradigm:

- Organizing procedural abstractions in the context of data abstractions

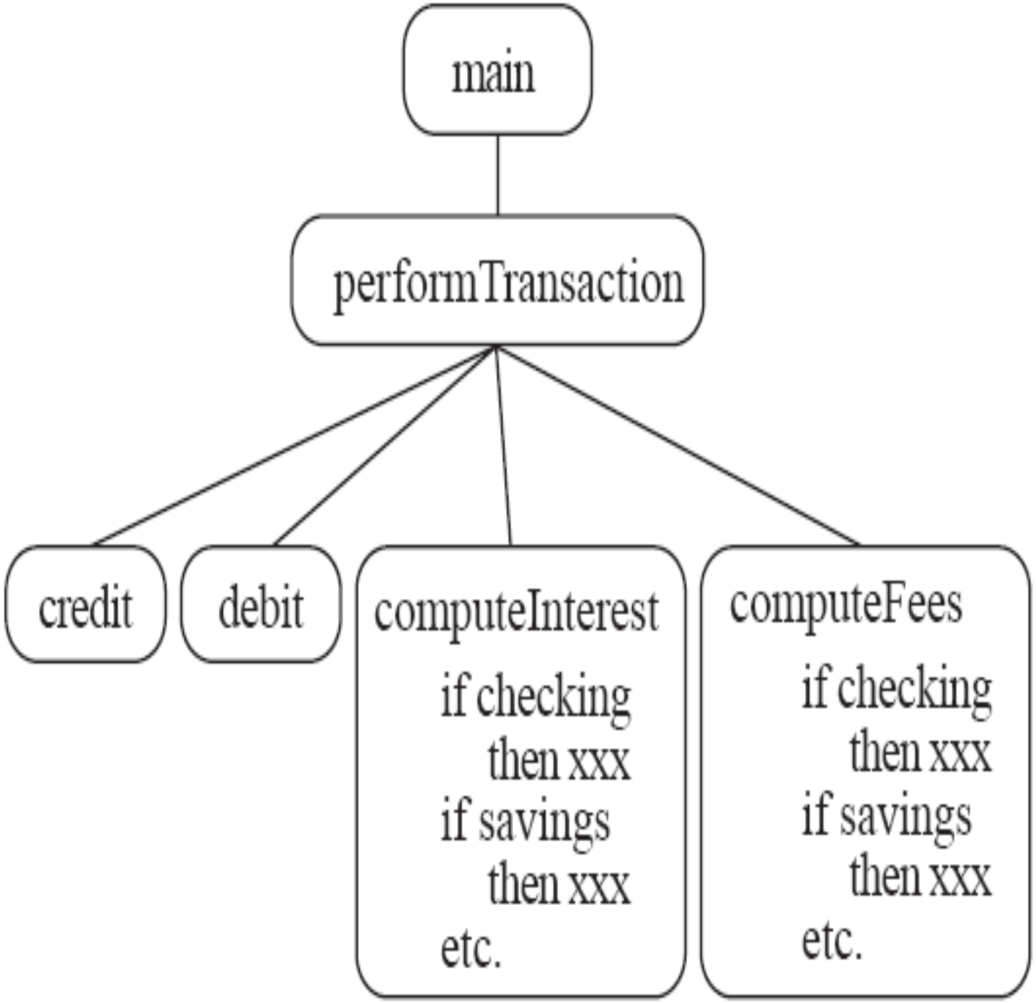
Object-Oriented Paradigm

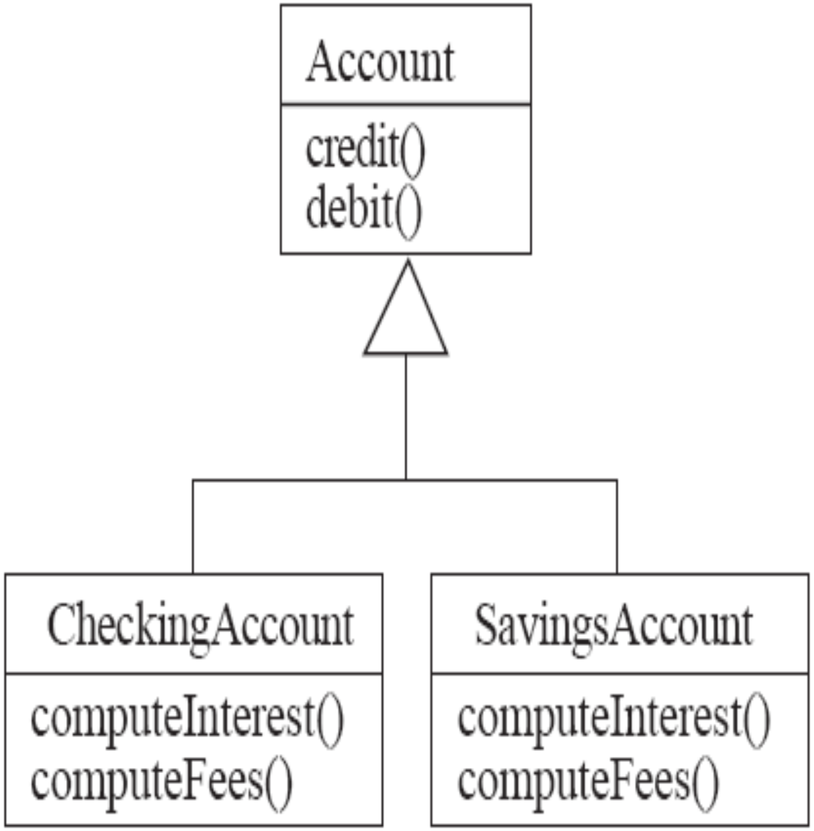
An approach to the solution of problems in which all computations are performed in the context of objects.

- 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

A View of the Two Paradigms





2.2 – Classes and 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

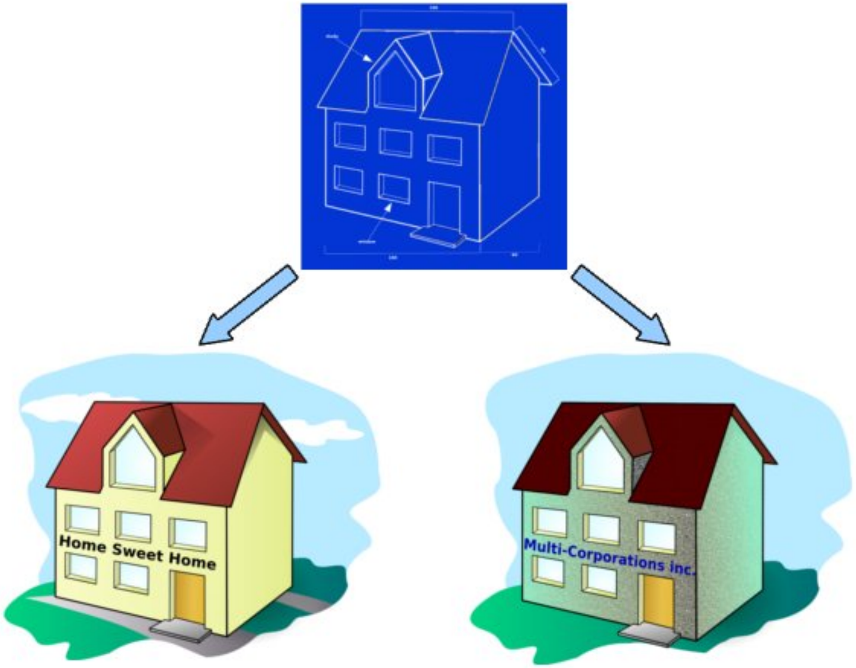
Classes and Objects

Class

- A unit of abstraction in an object-oriented (OO) program
- Represents similar objects
  - Its *instances*
- Is a kind of software module
  - Describes its instances' structure (properties)
  - Contains *methods* to implement their behaviour

Class vs. Object

- 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



Class vs. Object

	Class or Instance?
Film	Class; instances are individual films.
Reel of Film	Class; instances are physical reels
Film reel with serial number SW19876	Instance of **ReelOfFilm**
Science Fiction	Instance of the class **Genre**
Science Fiction Film	Class; instances include ‘Star Wars’
Showing of Star Wars in the Empire Theatre at 7 PM	Instance of **ShowingOfFilm**

Class vs. Object

	Class or Instance?
General Motors	
Automobile Company	
Boeing 777	
Computer Science Student	
Mary Smith	
Game	
Board Game	
Chess	

Class vs. Object

	Class or Instance?
CS 2212b	
Airplane	
The game of chess between Tom and Jane which started at 2:30 PM yesterday	
The car with serial number 198765T4	

Naming Classes

- Use *capital* letters
  - `BankAccount` not `bankAccount`

- Use *singular* nouns
  - `Person` not `People`

- Use the right level of generality
  - `Municipality` not `City`

- Make sure the name has only *one* meaning
  - `bus` has several meanings

2.3 – Instance Variables

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

- **Attributes**
  - Simple data
  - e.g. `name`, `dateOfBirth`

- **Associations**
  - Relationships to other important classes
  - e.g. `supervisor`, `coursesTaken`
  - More on these in Chapter 5

Instance Variables

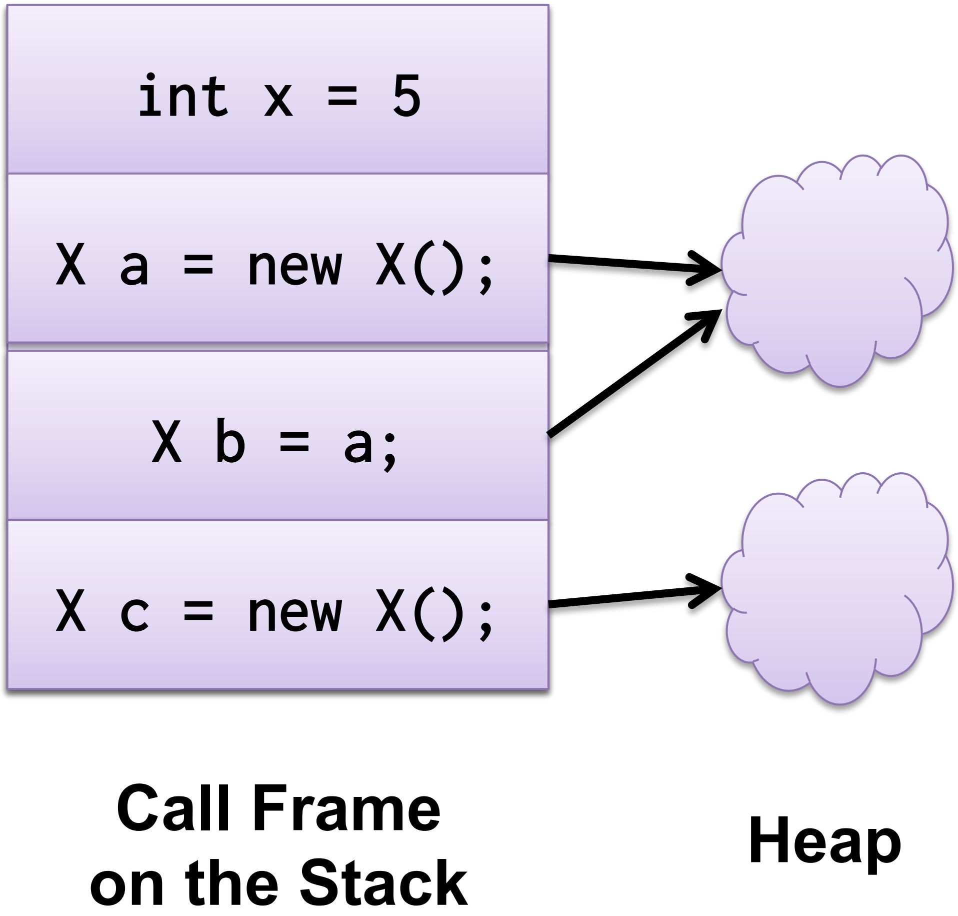
Variable

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

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

***Type* of a variable**

- Determines what classes of objects it may contain



```
int x = 5
X a = new X();
X b = a;
X c = new X();
```

Call Frame on the Stack

Heap

Class Variables

Value is *shared* by all instances

- 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
  - Lookup tables and similar structures

Do not over-use class variables

2.4 – Methods, Operations, and Polymorphism

Operation

- A higher-level procedural abstraction that specifies a type of behaviour
- Independent of any code which implements that behaviour
  - e.g. calculating area (in general)

Methods, Operations, and Polymorphism

Method

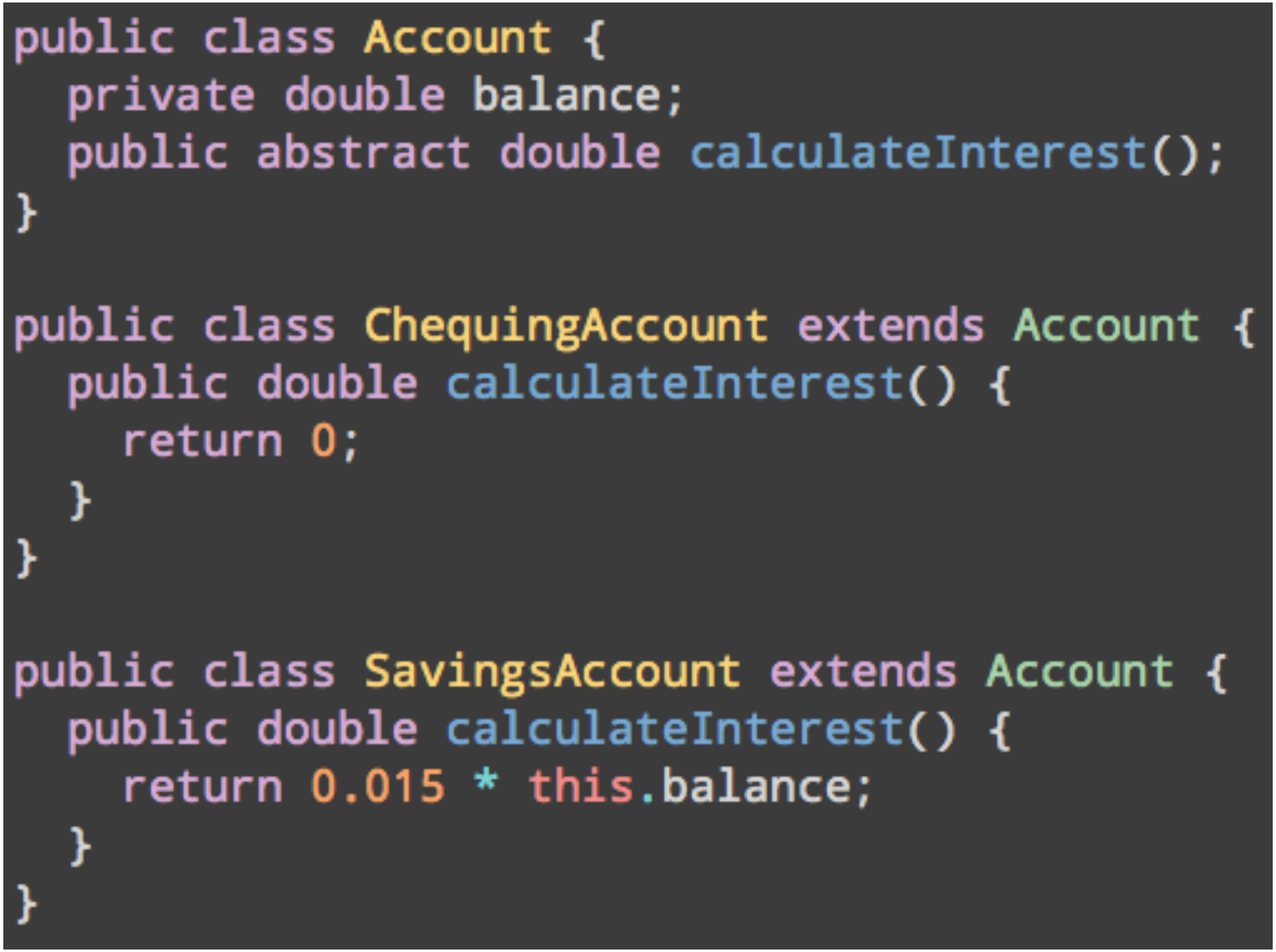
- A procedural abstraction used to implement the behaviour of a class.
- Several different classes can have methods with the same name
    - They implement the same abstract operation in ways suitable to each class
    - e.g. calculating area in a rectangle is done differently from in a circle

Methods, Operations, and Polymorphism

Polymorphism

- **A property of object oriented software by which an abstract operation may be performed in different ways in different classes.**
  - Requires 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

Example: Polymorphism

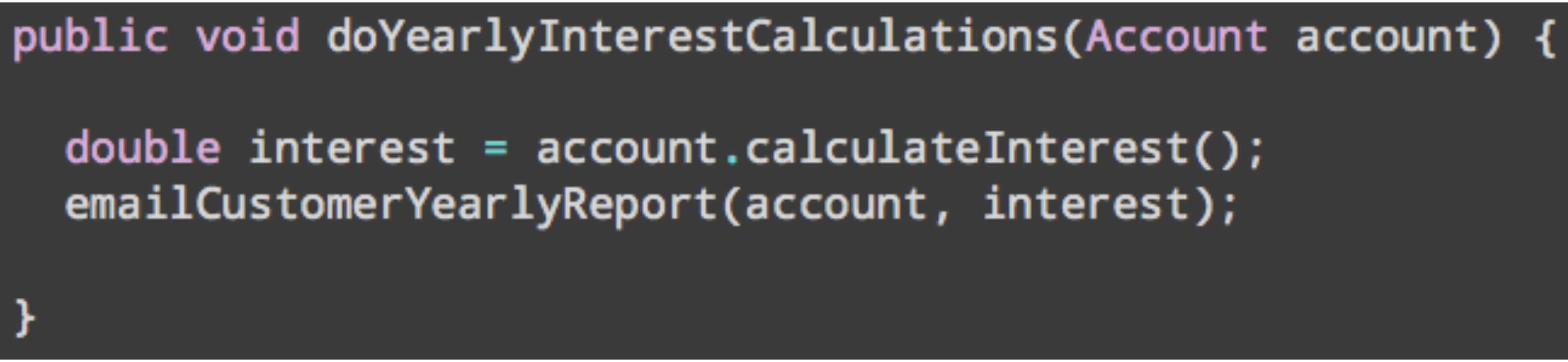


```java
public class Account {
  private double balance;
  public abstract double calculateInterest();
}

public class ChequingAccount extends Account {
  public double calculateInterest() {
    return 0;
  }
}

public class SavingsAccount extends Account {
  public double calculateInterest() {
    return 0.015 * this.balance;
  }
}
```

Example: Polymorphism

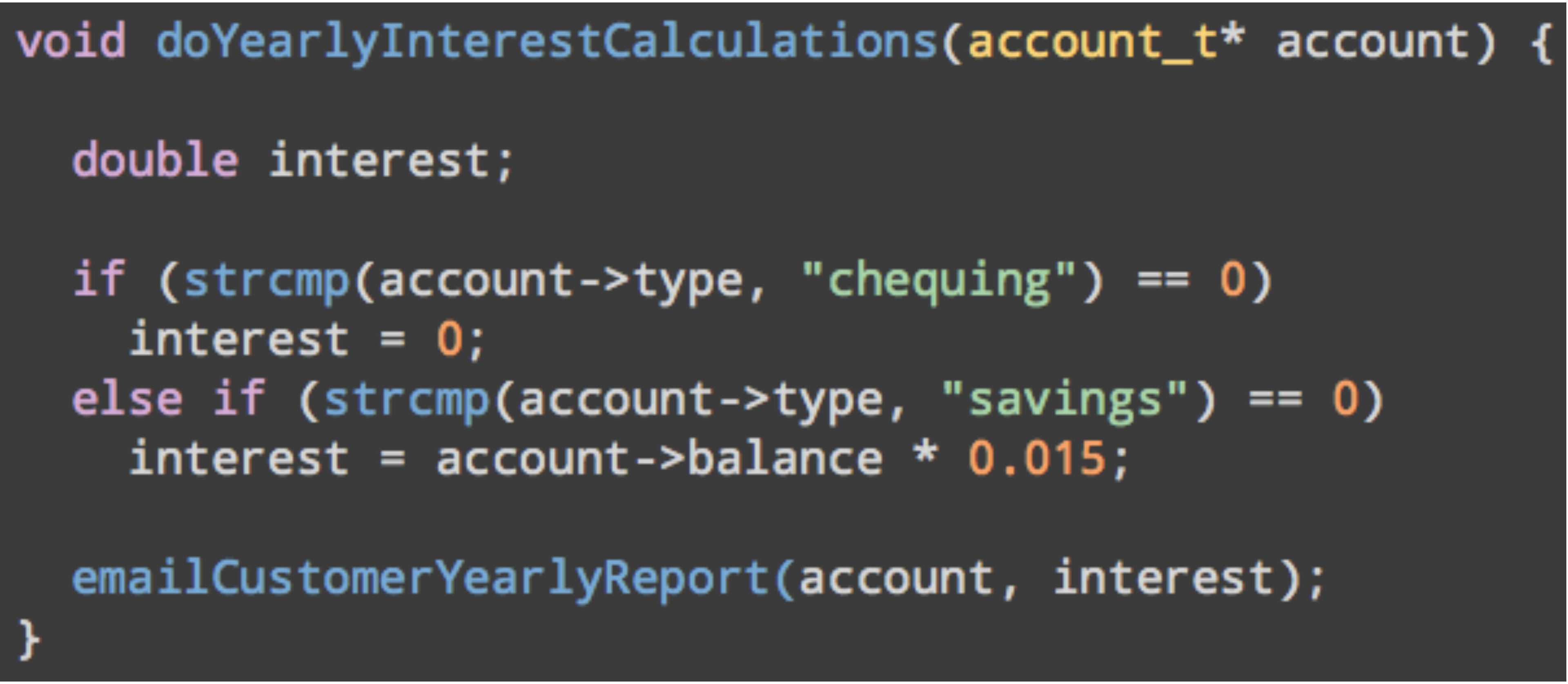


```
public void doYearlyInterestCalculations(Account account) {

  double interest = account.calculateInterest();
  emailCustomerYearlyReport(account, interest);

}
```

Example: Polymorphism



```c
void doYearlyInterestCalculations(account_t* account) {

  double interest;

  if (strcmp(account->type, "chequing") == 0)
    interest = 0;
  else if (strcmp(account->type, "savings") == 0)
    interest = account->balance * 0.015;

  emailCustomerYearlyReport(account, interest);
}
```

2.5 – Organizing Classes into Inheritance Hierarchies

Superclasses

- Contain features common to a set of subclasses

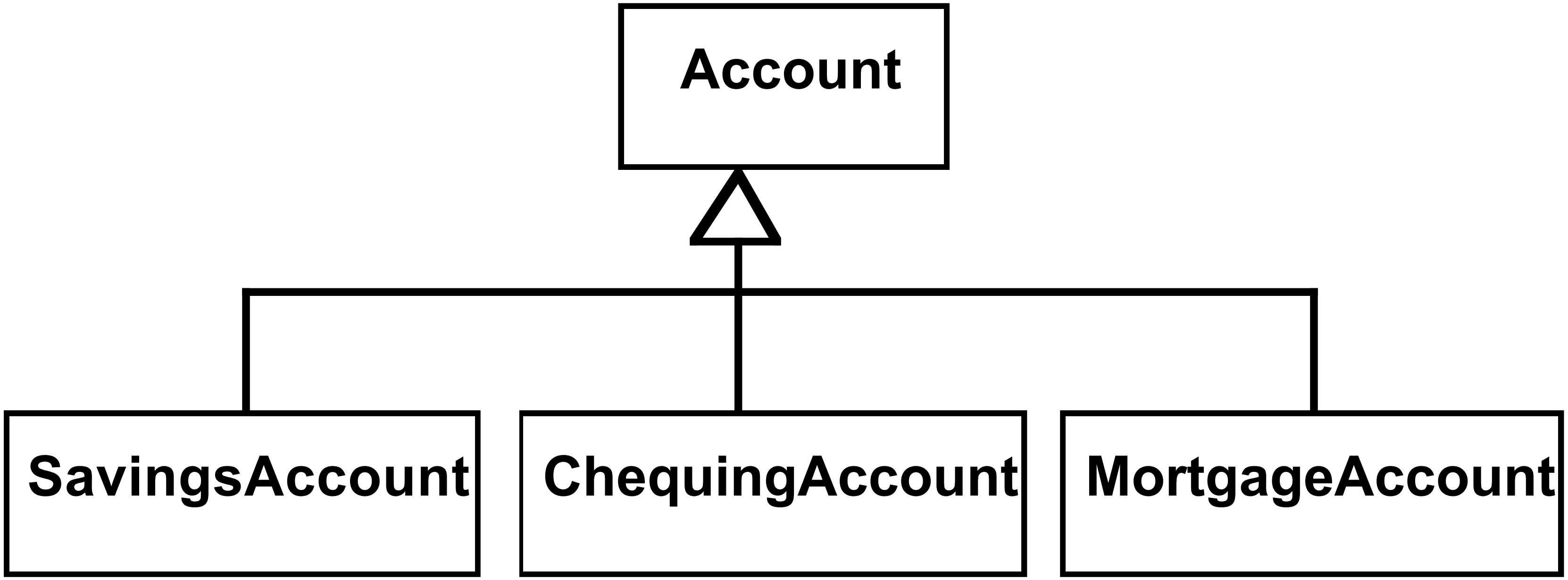
Inheritance hierarchies

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

Inheritance

- The *implicit* possession by all subclasses of features defined in its superclasses

Example: Inheritance Hierarchy



Inheritance

- The *implicit* possession by all subclasses of features defined in its superclasses

The IS-A Rule

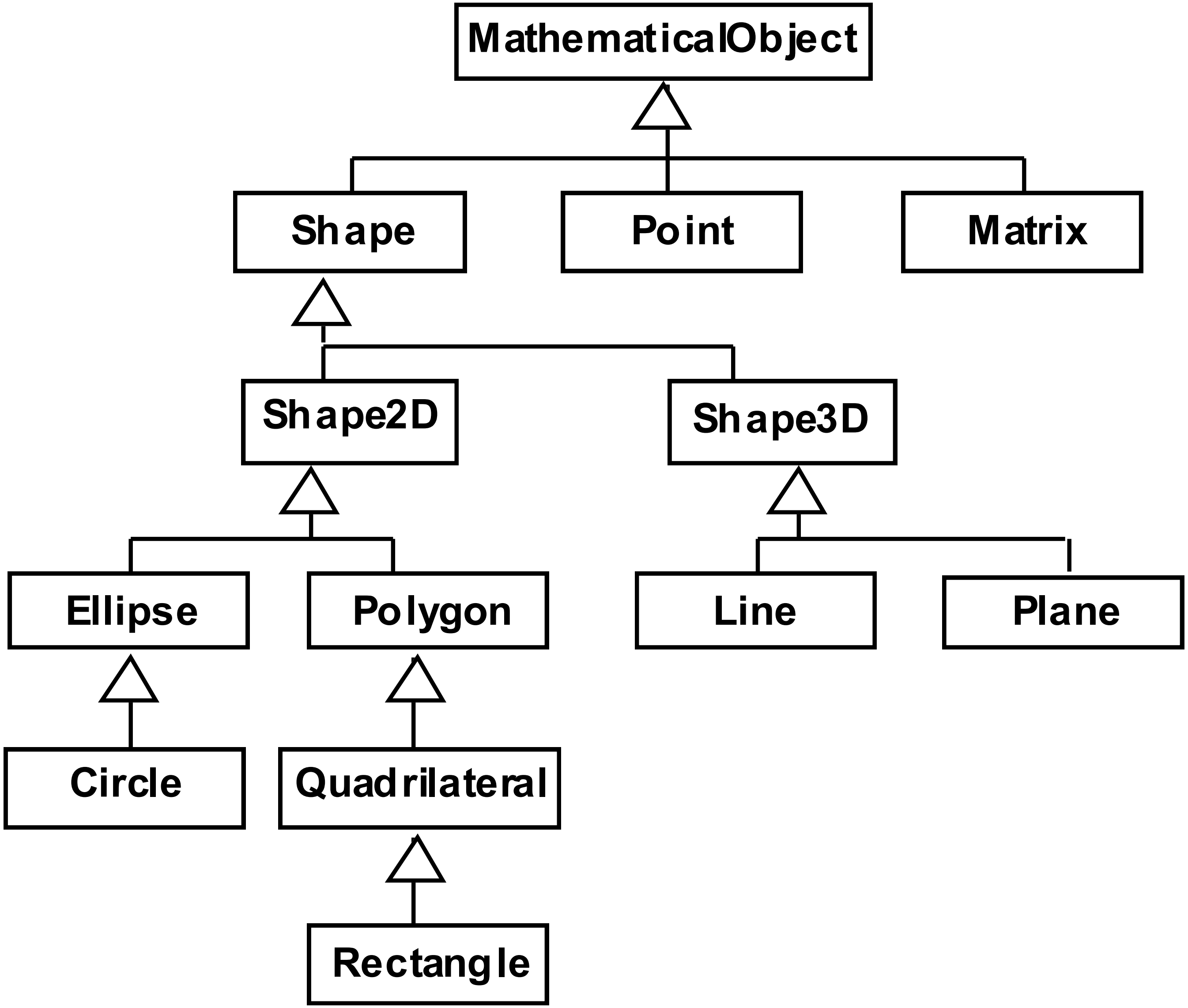
Always check generalizations to ensure they obey the IS-A rule

- “A checking account ***is an*** account”
- “A village ***is a*** municipality”

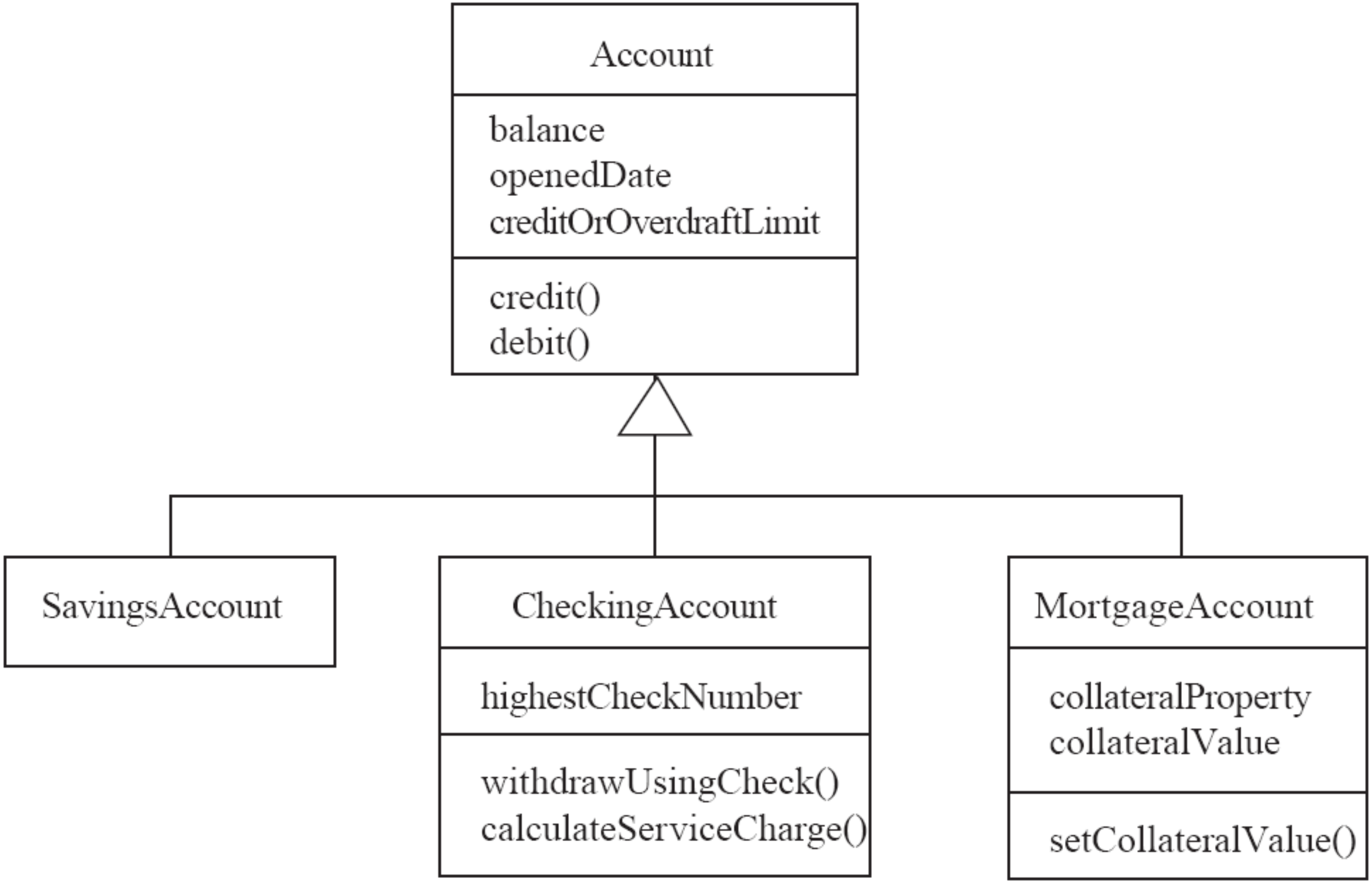
Should ‘Province’ be a subclass of ‘Country’?

- No, it violates the IS-A rule
- “A province ***is a*** country” is invalid!

Example: IS-A Relationships



Make Sure All Inherited Features Make Sense in All Subclasses



Account

balance
openedDate
creditOrOverdraftLimit

credit()
debit()

SavingsAccount

CheckingAccount

highestCheckNumber

withdrawUsingCheck()
calculateServiceCharge()

MortgageAccount

collateralProperty
collateralValue

setCollateralValue()

Which of the following would not form good superclass-subclass pairs? Look for violations of IS-A, poor naming, etc.

Money – CanadianDollars	
Bank – Account	
OrganizationUnit – Division	
SavingsAccount – ChequingAccount	
People – Customer	
Student – GraduateStudent	
Continent – Country	

Review Question 1

Organize the following set of items into inheritance hierarchies of classes. Hints:

- You will have several distinct hierarchies
- You will need to add additional classes to act at superclasses. You will also need to change some names and you will discover that two items may correspond to a single class
- Think of important attributes present in your classes. Make sure that attributes in a superclass will be present in each of its subclasses
- Remember the IS-A rule
- Some items put there to confuse you and should be considered instances or attributes and not classes

Review Question 1

- Currency
- Financial instrument (some thing you use to pay for purchases)
- Cheque
- Visa
- Bank account
- US dollars
- Exchange Rate
- Credit Card

- Credit Union
- MasterCard
- Bank branch
- Bank
- Debit card
- Bank machine (ATM)
- Loan
- Canadian dollars
- Credit card company

Review Question 1: Sample Solution

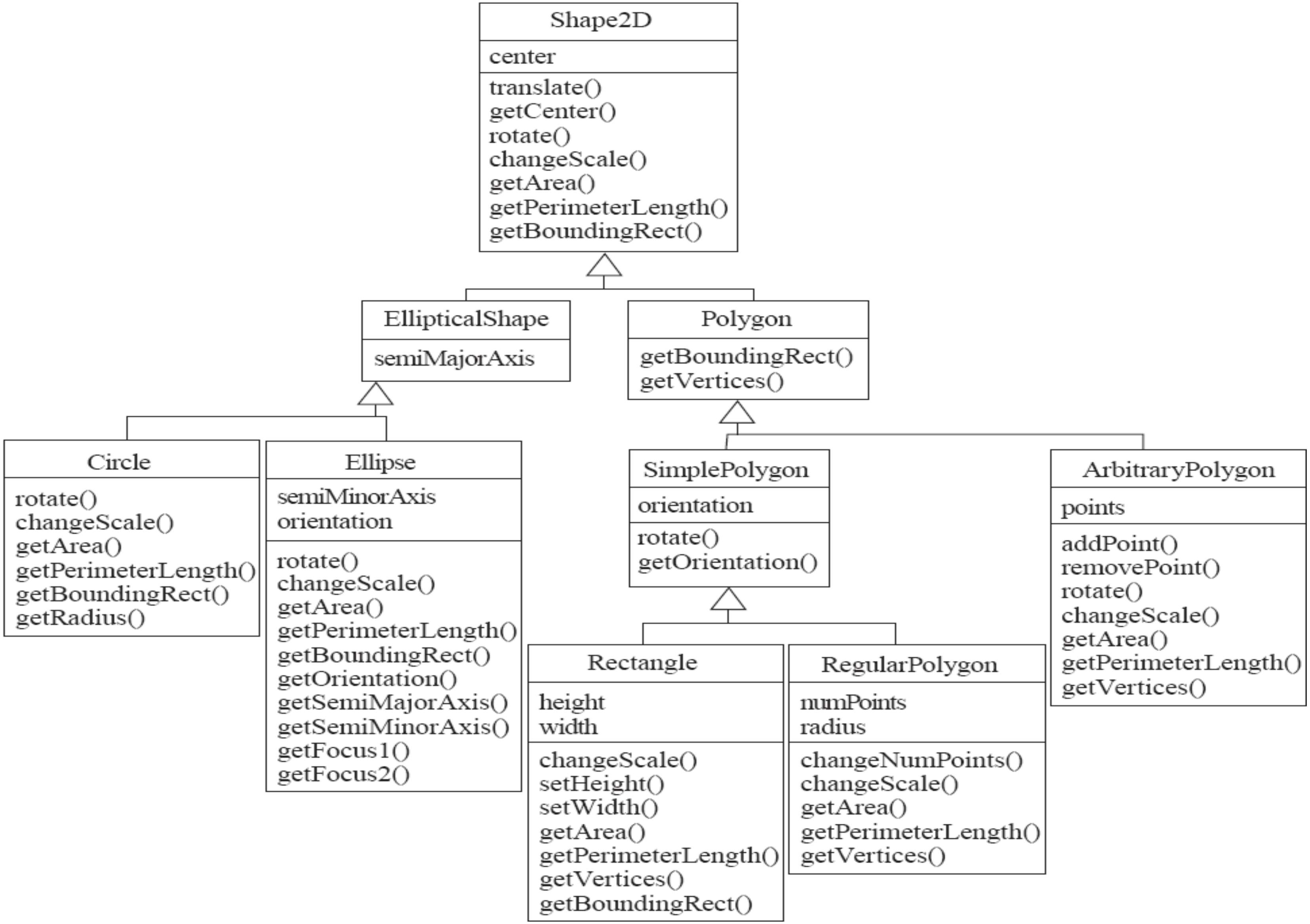
Review Question 2

- Vehicle
- Airplane
- Jet Engine
- Transmission
- Car
- Amphibious Vehicle
- Electric Motor
- Truck
- Sports Car
- Engine
- Wheel
- Bicycle

Hint: You may need to add a class or two

Review Question 2: Sample Solution

2.6 – Inheritance, Polymorphism, and Variables



Shape2D
center
translate()
getCenter()
rotate()
changeScale()
getArea()
getPerimeterLength()
getBoundingRect()

EllipticalShape
semiMajorAxis

Polygon
getBoundingRect()
getVertices()

Circle
rotate()
changeScale()
getArea()
getPerimeterLength()
getBoundingRect()
getRadius()

Ellipse
semiMinorAxis
orientation
rotate()
changeScale()
getArea()
getPerimeterLength()
getBoundingRect()
getOrientation()
getSemiMajorAxis()
getSemiMinorAxis()
getFocus1()
getFocus2()

SimplePolygon
orientation
rotate()
getOrientation()

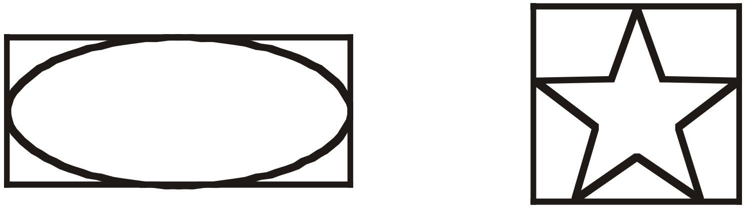
ArbitraryPolygon
points
addPoint()
removePoint()
rotate()
changeScale()
getArea()
getPerimeterLength()
getVertices()

Rectangle
height
width
changeScale()
setHeight()
setWidth()
getArea()
getPerimeterLength()
getVertices()
getBoundingRect()

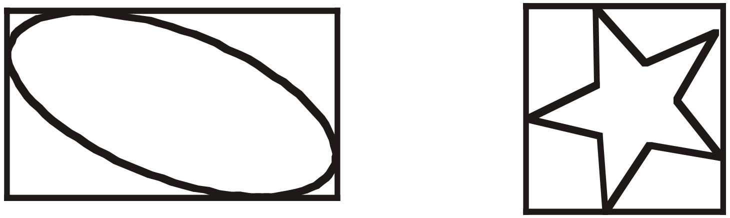
RegularPolygon
numPoints
radius
changeNumPoints()
changeScale()
getArea()
getPerimeterLength()
getVertices()

Some Operations in the Shape Example

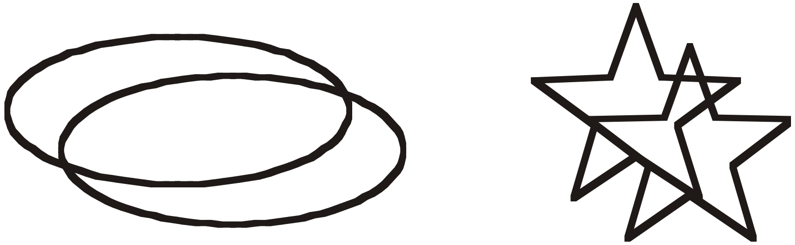
Original objects
(showing bounding rectangle)



Rotated objects
(showing bounding rectangle)

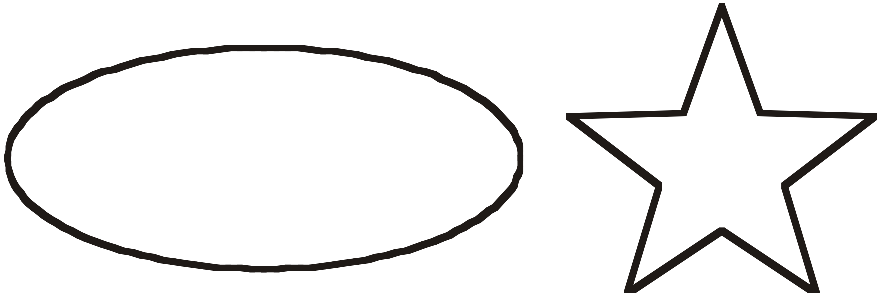


Translated objects
(showing original)



Scaled objects
(50%)

Scaled objects
(150%)



Abstract Classes and Methods

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

- Operation may be *abstract* (no implementation) at that level
- If so, the class also must be *abstract*
  - No instances can be created
  - **Opposite of an abstract class:** a *concrete* class

Abstract Classes and Methods

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

- 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

Overriding

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

- For restriction
  - e.g. scale(x,y) would not work in Circle
- For extension
  - e.g. SavingsAccount might charge an extra fee following every debit
- For optimization
  - e.g. The getPerimeterLength method in Circle is much simpler than the one in Ellipse

How a Decision is Made About Which Method to Run

1. If there is a concrete method for the operation in the current class, run that method.

2. Otherwise, check in the immediate superclass to see if there is a method there; if so, run it.

3. Repeat step 2, looking in successively higher superclasses until a concrete method is found and run.

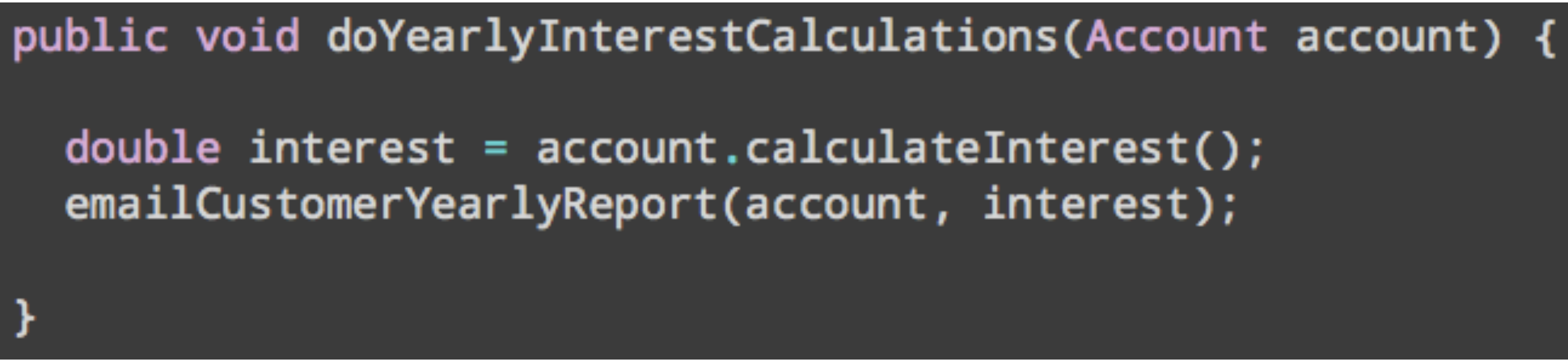
4. If no method is found, then there is an error
    - In Java and C++ the program would not have compiled

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

Example: Polymorphism



```
public void doYearlyInterestCalculations(Account account) {

  double interest = account.calculateInterest();
  emailCustomerYearlyReport(account, interest);

}
```

2.7 – Concepts that Define Object Orientation

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

Concepts that Define Object Orientation

Necessary for a system or language to be OO

- **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.

Other Key Concepts

Abstraction

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

Other Key Concepts

Modularity

- Code can be constructed entirely of classes

Encapsulation

- Details can be hidden in classes
- This gives rise to *information hiding*:
  - Programmers do not need to know all the details of a class

Programming Style

- **Adhere to good object-oriented principles**
  - e.g. the IS-A rule
- **Prefer private as opposed to public**
- **Prefer composition over inheritance**
- **Do not mix UI code with non-UI code**
  - Interact with the user in separate classes
    - This makes non-UI classes more reusable

2.10 – Difficulties/Risks in OOP

Language evolution and deprecated features:

- Java is evolving, so some features are ‘deprecated’ at every release
- But the same thing is true of most other languages

Efficiency can be a concern in some object-oriented systems

- Java can be less efficient than other languages
  - VM-based
  - Dynamic binding

Western

UNIVERSITY · CANADA