# OBJECT ORIENTED SOFTWARE ENGINEERING (CAS775)

**Dr. Ghanshyam S. Bopche**
Assistant Professor
Dept. of Computer Applications

September 19, 2020

# OBJECT ORIENTED SOFTWARE ENGINEERING (CAS775)

**Syllabus**

- **Unit-I**: What is software engineering? Software Development Life Cycles Models, Conventional Software Life Cycle Models, What is Object Orientation? Objects and Classes, Features, Object Oriented Software Life Cycle Models, Object Oriented Methodologies, Object Oriented Modeling, Terminologies
- **Unit-II**: Software Requirements Elicitation and Analysis, Case Study: Library Management System What is Software Requirement? Requirements Elicitation Techniques, Characteristics of a good Requirement, Software Requirements Specification Document, Requirements Change Management, Object Oriented Analysis, Overview of Cost Estimation Techniques, Agile development, Classification of methods, The agile manifesto and principles, Agile project

# OBJECT ORIENTED SOFTWARE ENGINEERING (CAS775) (cont.)

management, Agile Methodology, Method overview, Lifecycle, Work products, Roles and Practices values, Process mixtures, Adoption strategies, Understanding SCRUM

- **Unit-III**: Software Design, Object Oriented Design, What is done in object oriented design? UML, Refinement of Use Case Description, Refinement of classes and relationships, Construction of Details class diagrams, Development of Details Design and Creation, Generating Test cases from User Cases, Object Oriented Design principles for Improving Software Quality.

# OBJECT ORIENTED SOFTWARE ENGINEERING (CAS775) (cont.)

- **Unit-IV**: Software Implementation- Quality and Metrics, Software Implementation Tools and Techniques, What is software quality? Software quality models, Measurement basic, analyzing the metric data, Metrics for measuring size and structure, Measuring software quality object oriented metrics, Overview of Scala for Implementation.
- **Unit-V**: Software Testing and Maintenance, What is software testing? Software verification techniques, Checklist: a popular verification tool, Functional Testing, Structural Testing, Object Oriented Testing, Class testing, State based testing, Mutation testing, Levels of testing, Software testing tools, What is a software maintenance? Categories, Challenges of software maintenance, Maintenance of Object oriented Software, Software rejuvenation, Estimation of maintenance efforts, Configuration management, Regression testing.

# OBJECT ORIENTED SOFTWARE ENGINEERING (CAS775) (cont.)

- **References**:


1. Yogesh Singh, Ruchika Malhotra, "Object-Oriented Software Engineering", PHI, 2012.
2. Timothy C. Lethbridge and Robert Laganiere, "**Object-Oriented Software Engineering**", McGraw-Hill, 2nd ed., 2004.
3. G. Booch, Benjamin/Cummings, "Object-Oriented Analysis and Design with Applications", 3rd Edition, Addison-Wesley, 2007.
4. Roger Pressman, "**Software Engineering: A Practitioner's Approach**", McGraw-Hill Higher Education, 2010.
5. S. Kenneth Rubin, "Essential Scrum: A Practical Guide to the Most Popular Agile Process", Pearson Publication, 2012.
6. Jason Swartz, "Learning Scala Practical Functional Programming for the JVM", O'Reilly Media, December 2014

# What is Object Orientation?

- Object-oriented systems make use of abstraction.
  **Goal**: make software less complex.
- **Abstraction**: something that relieves us from having to deal with details.
  1. **Procedural/Method Abstraction**: achieved by separating the use of a method from its implementation.
     The client can use a method without knowing how it is implemented.

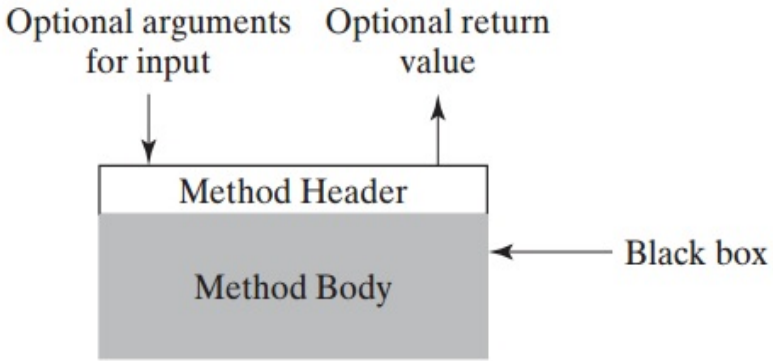



Figure : The method body can be thought of as a black box that contains the detailed implementation for the method.

  2. **Data Abstraction**: the process of hiding certain details and showing only essential information to the user.

# Procedural abstraction and the procedural paradigm

- **Procedural paradigm**
  - Entire system will be organized around the notion of procedures or functions or routines.
  - One "main" procedure calls several other procedures, which in turn call others.
- **Procedural abstraction**
  - Make the programmer's view of the system much simpler.
  - **How?** when using a certain procedure, a programmer does not need to worry about all the details of how it performs its computations; he or she only needs to know how to call it and what it computes.
- **Note**: procedural paradigm works very well for the programs that perform calculations with relatively simple data.

# Data abstraction

- Help in reducing system's complexity to a certain extent.
- Records and Structures
  - Data abstractions of very first kind
  - **Idea**: group together the pieces of data that describe some entity, so that programmers can manipulate that data as a unit.
- **Note**: despite the use of data abstraction, programmers still have to write complex code in many different places.
- **Example**: Consider a banking system that is written using the procedural paradigm, but using records representing bank accounts. The software has to manage accounts of different types, such as current/checking, savings and mortgage accounts. Each type of account will have different rules for the computation of fees, interest, etc.

# Data abstraction (cont.)

Such a system would have procedures like the following pseudocode in many different places:

*if account is of type current then*
→do something
*else if account is of type saving then*
→do something else
*else*
→do yet another thing
endif

# The Object-oriented Paradigm

- Organizes procedural abstractions in the context of data abstractions.
- **Definition**: The object-oriented paradigm is an approach to the solution of problems in which all computations are performed in the context of objects. The objects are instances of programming constructs, normally called classes, which are data abstractions and which contain procedural abstractions that operate on the objects.
- Object-oriented paradigm) involves programming using objects (collection of objects collaborating to perform a given task).

# The Object-oriented Paradigm (cont.)

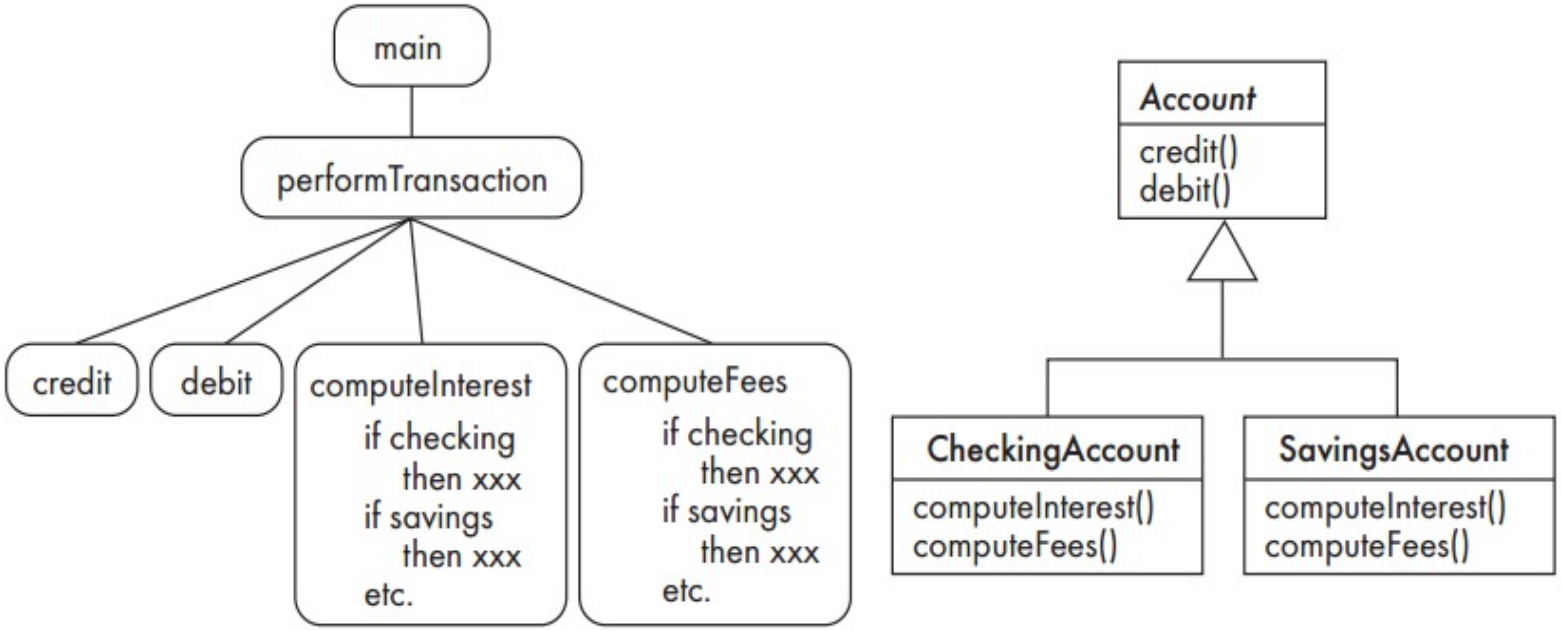



Figure : Organizing a system according to the procedural paradigm (left) or the object-oriented paradigm (right).

- **Procedural paradigm**: the code is organized into procedures that each manipulate different types of data.
- **Object-oriented paradigm**: the code is organized into classes that each contain procedures for manipulating instances of that class alone.

# The Classes and Objects

## Object

- An object represents an entity in the real world that can be distinctly identified. For example, a student, a desk, a circle, a button, and even a loan can all be viewed as objects.
- An object has a unique identity, state (properties or attributes), and behavior (actions).
  - A circle object, for example, has a data field radius, which is the property that characterizes a circle.
  - The behavior of an object (also known as its actions) is defined by methods. To invoke a method on an object is to ask the object to perform an action. For example, a circle object may invoke getArea() method to return its area.
- Objects of the same type are defined using a common class.

# The Classes and Objects (cont.)

## Classes and and their instances

- Units of data abstraction in an object-oriented program.
- A class is a template, blueprint, or contract that defines what an object's data fields and methods will be.
- An object is an instance of a class. One can create many instances of a class.
- All the objects with the same properties and behavior are instances of one class.

# The Classes and Objects (cont.)

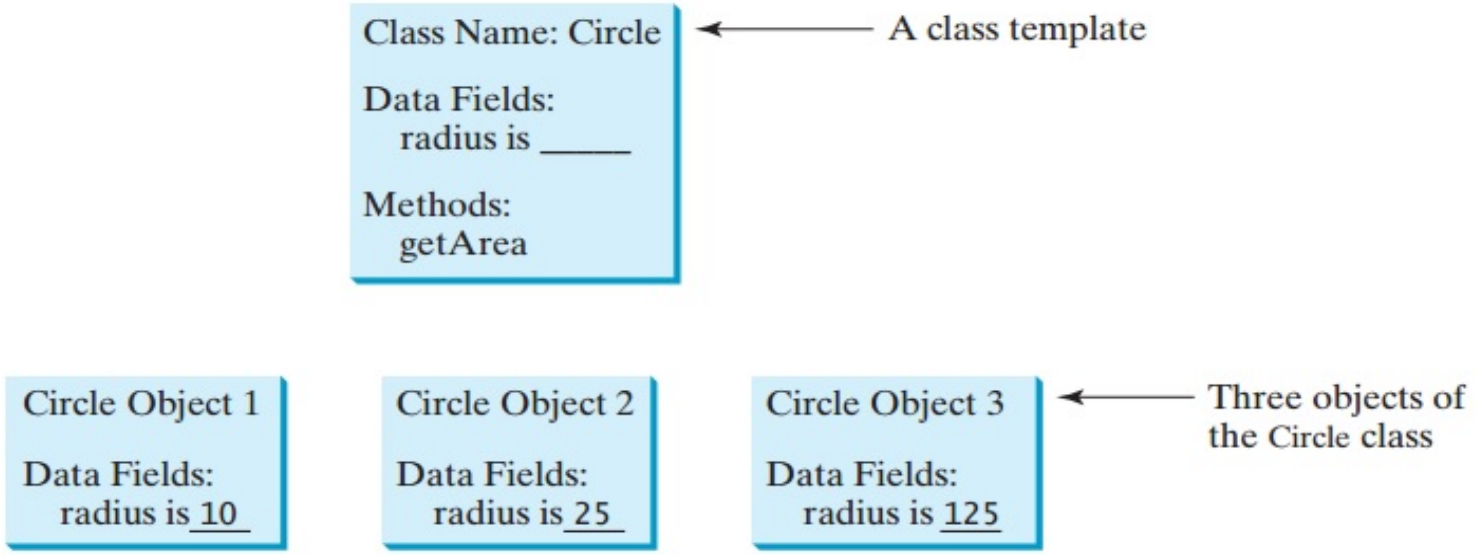



Figure : A class is a template for creating objects.

## Instances variables

- A variable is a place/memory location where you can store data.
- Each class declares a list of variables corresponding to data that will be present in each instance; such variables are called instance variables.

# Attributes and associations

- There are two groups of instance variables:
  - those used to implement attributes, and
  - those used to implement associations.
- **Attribute**
  - A simple piece of data used to represent the properties of an object.
  - For example, each instance of class Employee might have the attributes: name, dateOfBirth, socialSecurityNumber, telephoneNumber, address.
- **Association**
  - Represents the relationship between instances of one class and instances of another.
  - For example, class Employee in a business application might have the following relationships:
    - supervisor (association to class Manager)
    - tasksToDo (association to class Task

# Variables versus objects

- At any given instant, a variable can refer to a particular object or to no object at all.
- Variables that refer to objects are therefore often called references.
- During the execution of a program, a given variable may refer to different objects.
- The type of a variable determines what classes of objects it may contain.
- Variables can be local variables in methods; these are created when a method runs and are destroyed when a method returns.
- However, objects temporarily referenced by such variables may last much longer than the lifetime of the method as long as some other variable also references the object.

# Instance variables versus class variables

- Each class declares a list of variables corresponding to data that will be present in each instance; such variables are called instance variables.
- If you want all the instances of a class to share data, use static variables, also known as class variables.
- Static variables store values for the variables in a common memory location. Because of this common location, if one object changes the value of a static variable, all objects of the same class are affected.
- Java supports static methods as well as static variables.
- Static methods can be called without creating an instance of the class.

# Instance variables versus class variables (cont.)

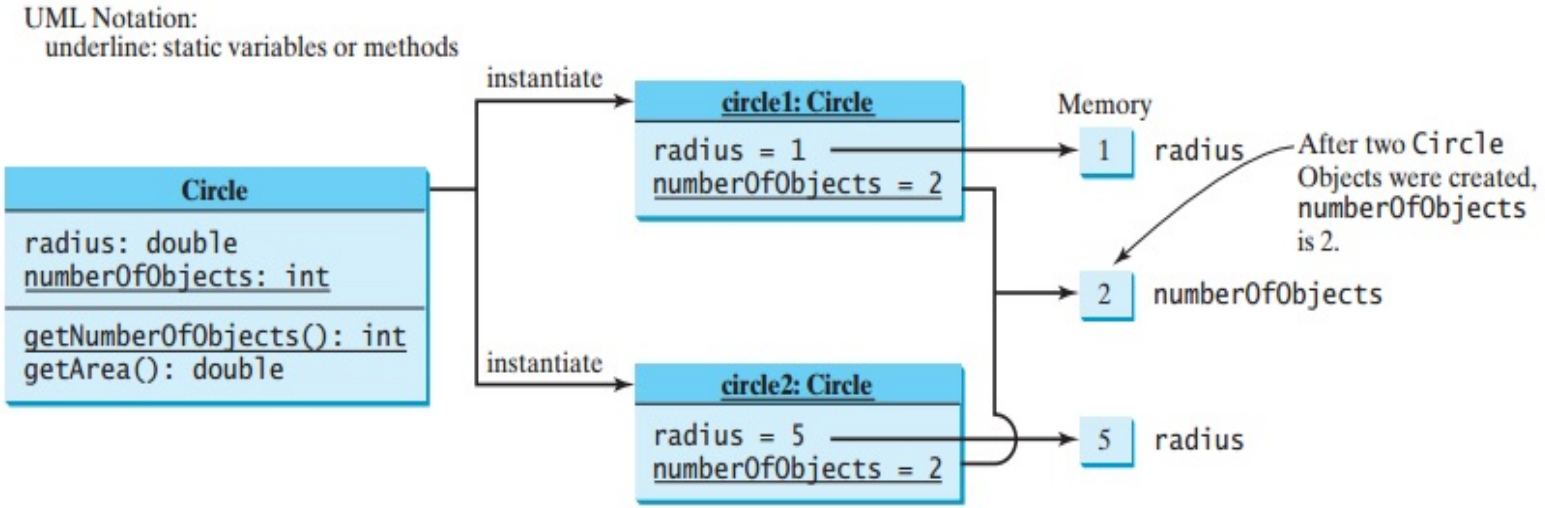



Figure : Instance variables belong to the instances and have memory storage independent of one another. Static variables are shared by all the instances of the same class.

- Class variables useful for storing the following types of information:
  - Default or "constant" values that are widely used by methods in a class.
  - Lookup tables and similar structures used by algorithms inside a particular class.

# Methods, operations and polymorphism

- Methods are procedural abstractions used to implement the behavior of a class.

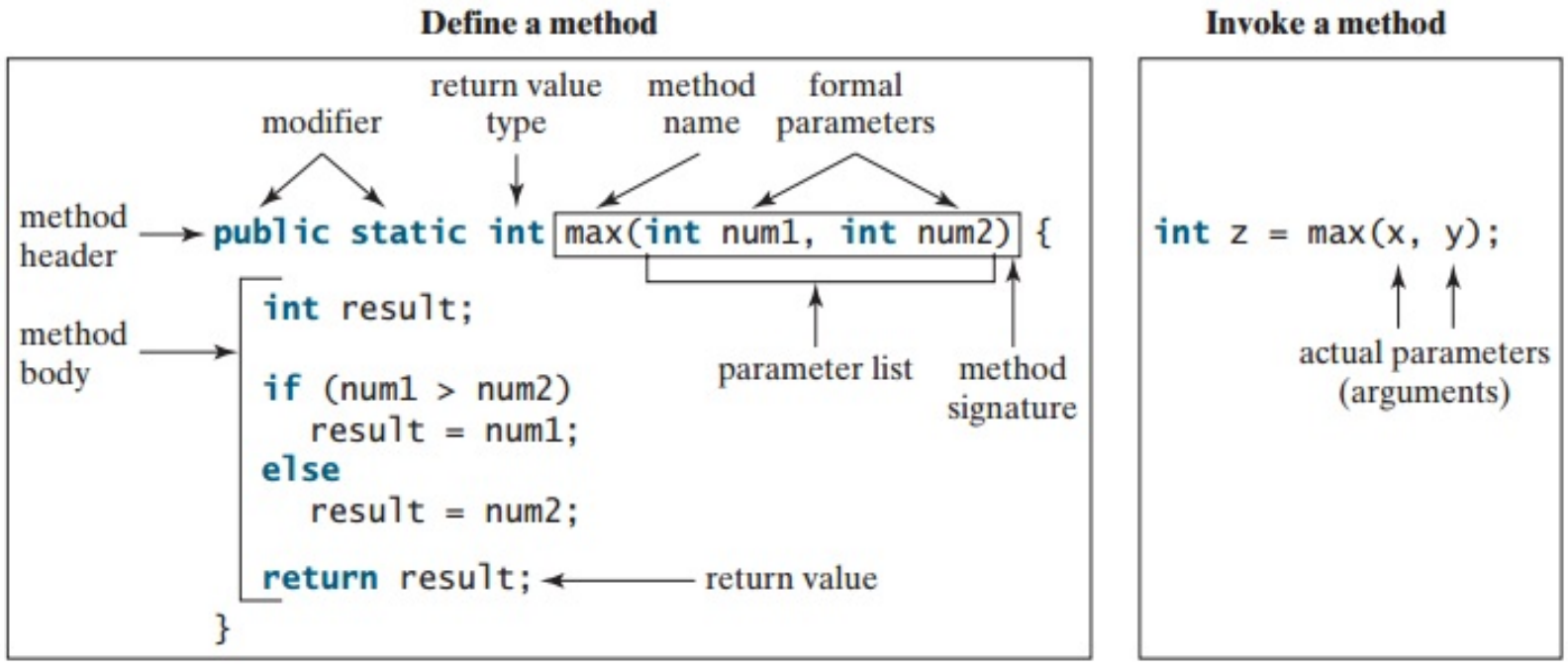



Figure : A method definition consists of a method header and a method body.

# Methods, operations and polymorphism (cont.)

- **Operation**
  - An operation is a higher-level procedural abstraction.
  - It is used to discuss and specify a type of behavior, independently of any code that implements that behavior.
  - Several different classes can have methods with the same name that implement the abstract operation in ways suitable to each class.

# Methods, operations and polymorphism (cont.)

- **Polymorphism**
  - One of the three pillars of object-oriented programming (Other two- encapsulation, and inheritance).
  - Polymorphism is a property of object-oriented software by which an abstract operation may be performed in different ways, typically in different classes.
  - An operation is said to be polymorphic, if the running program decides, every time an operation is called, which of several identically named methods to invoke.

# Inheritance

- If several classes have attributes, associations or operations in common, it is best to avoid duplication by creating a separate *superclass* that contains these common aspects.
- Inheritance allows us to derive new classes from existing classes. If you have a complex class, it may be good to divide its functionality among several specialized subclasses.

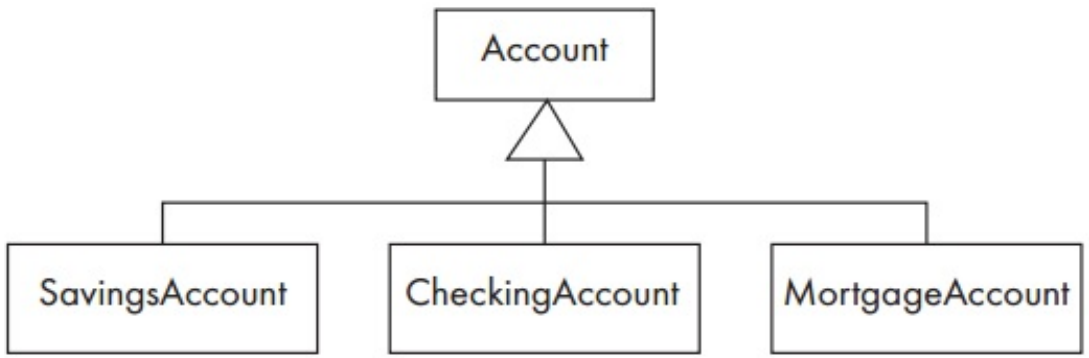



Figure : Basic inheritance hierarchy of bank accounts.

# Inheritance (cont.)

- Inheritance is an important and powerful feature of object oriented software engineering for reusing software code.
- Inheritance is the implicit possession by a subclass of features defined in a superclass. Features include variables and methods.
- The relationship between a subclass and an immediate superclass is called a *generalization*.
- The subclass is called a *specialization*.
- A hierarchy with one or more *generalizations* is called an *inheritance hierarchy*.

# Inheritance (cont.)

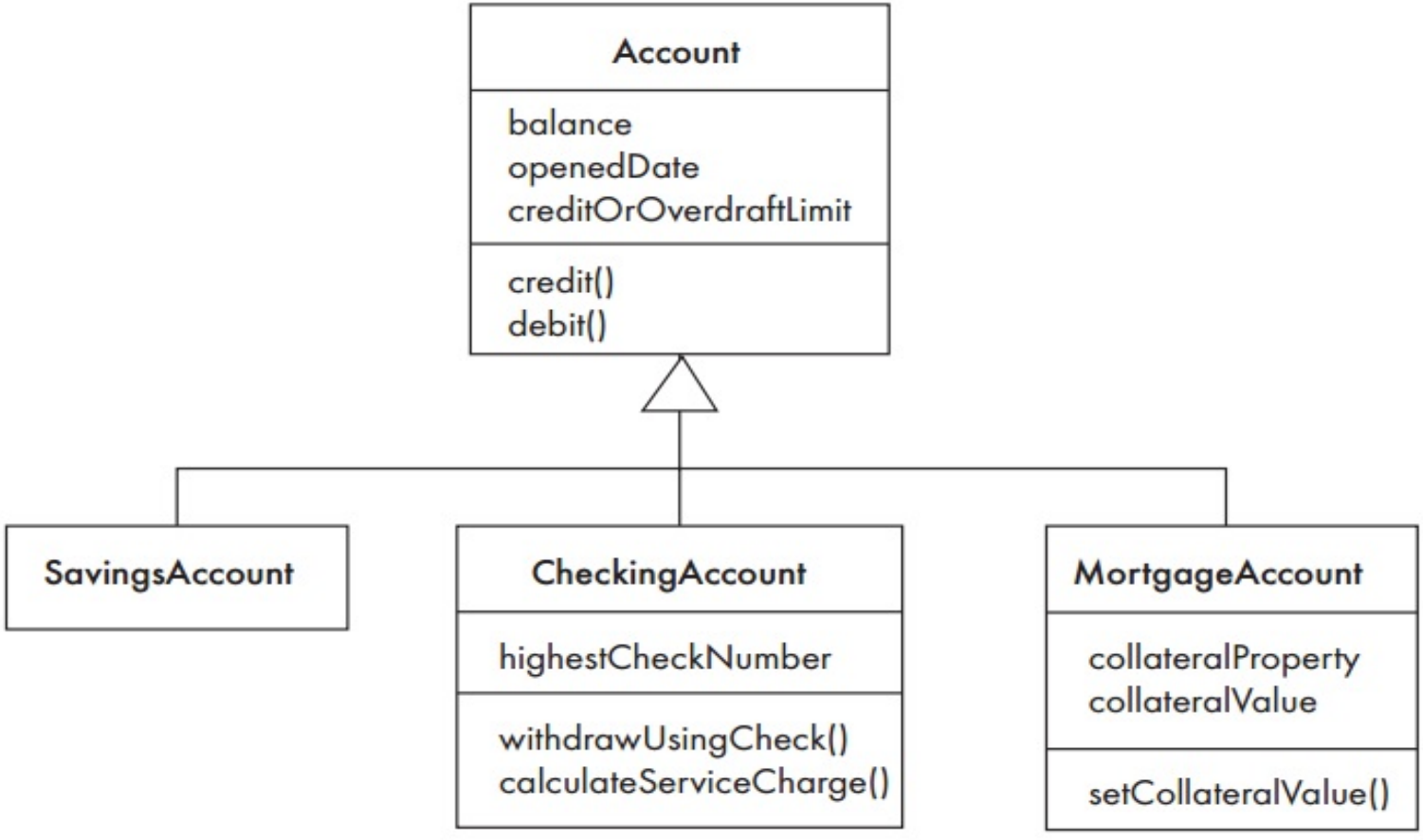
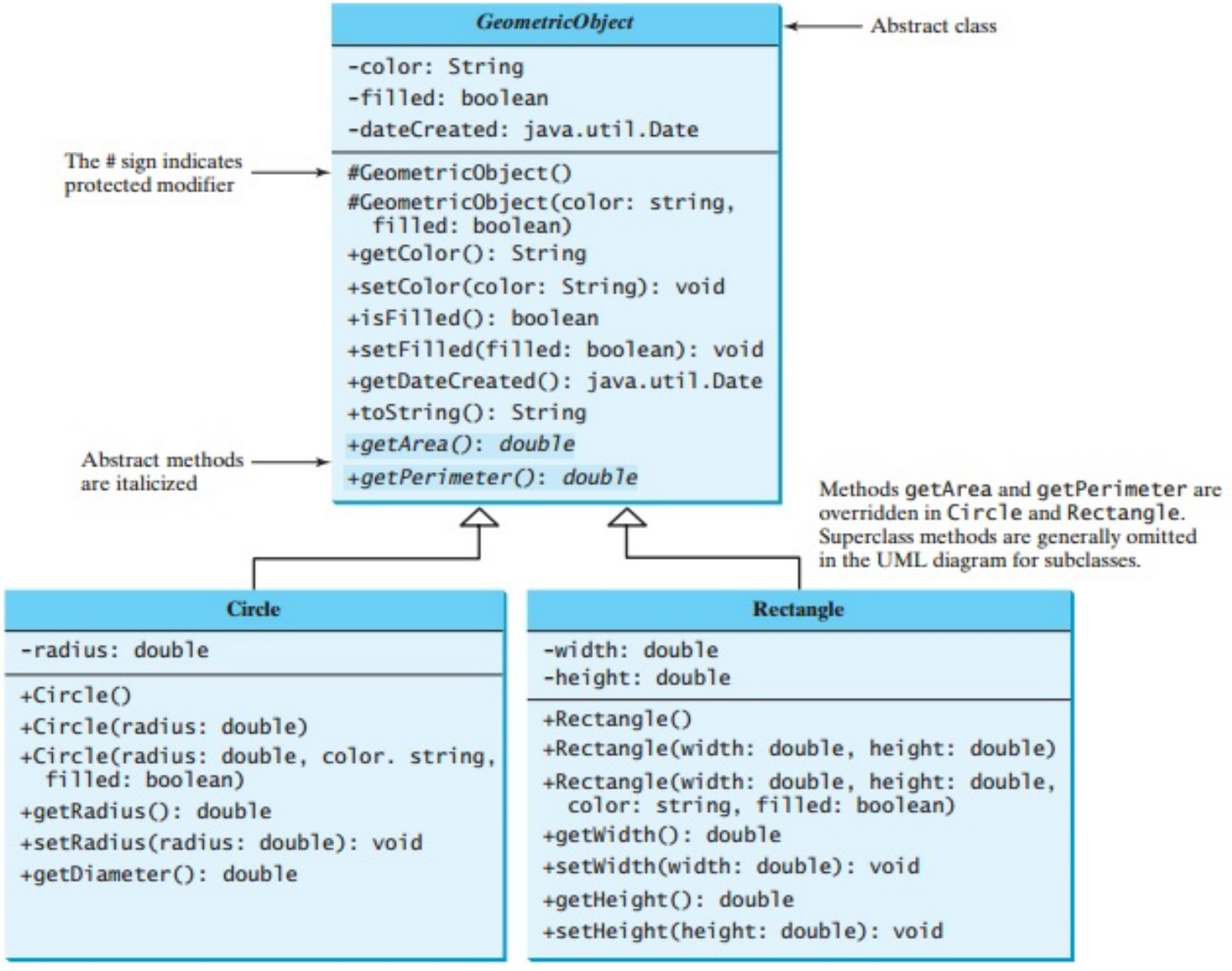



Figure : Inheritance hierarchy of bank accounts showing some attributes and operations

# Abstract classes and abstract methods

- **Abstract class**: a class that cannot have any instances.
  - Any class, except a leaf class, can be declared abstract.
  - A class that has one or more abstract methods must be declared abstract.
  - **Purpose**: to hold features that will be inherited by its subclasses.
- **Concrete Class**: a class which is not abstract (instances of a class can be created).
  - Leaf classes must be concrete.
  - It is also possible to have concrete classes higher in the inheritance hierarchy.
- **Abstract method** : defined without implementation.
  - Implementation is provided by the subclasses.
  - A class that contains abstract methods must be defined abstract.

# Abstract classes and abstract methods (cont.)



# Method Overriding

- A subclass inherits methods from a superclass.
- Sometimes it is necessary for the subclass to modify the implementation of a method defined in the superclass. This is referred to as method overriding.

```
public class Test {
  public static void main(String[] args) {
    A a = new A();
    a.p(10);
    a.p(10.0);
  }
}

class B {
  public void p(double i) {
    System.out.println(i * 2);
  }
}

class A extends B {
  // This method overrides the method in B
  public void p(double i) {
    System.out.println(i);
  }
}
```

# Method Overriding (cont.)

Reasons for overriding methods

- **Overriding for restriction**
  - The overriding method prevents a violation of certain constraints that are present in the subclass, but were not present in the superclass.
- **Overriding for extension**
  - The overriding method does basically the same thing as the version in the superclass, but adds some extra capability needed in the subclass.
- **Overriding for optimization**
  - The overriding method in the subclass has exactly the same effect as the overridden method, except that it is more efficient.

# Interfaces

- An interface is a classlike construct that contains only constants and abstract methods.
- In many ways an interface is similar to an abstract class, but the intent is to specify common behavior for objects.
- Interface neither have instance variables nor concrete methods.
- Interface is a named list of abstract operations.
- Several implementing classes of an interface that must implement the abstract operations.
- In Java, a class can implement multiple interfaces, but can have only one superclass.
- **Interfaces in Java**: Comparable, ActionListener, Cloneable, Runnable, etc.

# Features of Object Oriented Language

- **Identity**
  - The language must allow a programmer to refer to an object without having to refer to the instance variables contained in the object.
  - Every object has a unique identity; therefore objects that contain instance variables with the same values must be recognized as different objects.
- **Classes**
  - The programmer must be able to organize the code into classes, each of which describes the structure and function of a set of objects.

# Features of Object Oriented Language (cont.)

- **Inheritance**
  - There has to be a mechanism to organize classes into inheritance hierarchies, where features inherit from superclasses to subclasses.
- **Polymorphism**
  - There has to be a mechanism by which several methods, in related classes, can have the same name and implement the same abstract operation.
  - There must consequently be a dynamic binding mechanism that allows the choice of which method to run to be made during execution of the program.

# Features of Object Oriented Language (cont.)

- **Abstraction**
  There are many abstractions in an object-oriented program as follows:
  - **Object**: an abstraction of something of interest to the program, normally something in the real world such as a bank account.
  - **Class**: an abstraction of a set of objects; at the same time it also acts as an abstract container for the methods that operate on those objects.
  - **Superclass**: an abstraction of a set of subclasses.
  - **Interfaces**: an abstraction of a set of implementing classes. Compared to superclasses, interface provides even better abstraction since it has fewer details defined (only abstract operations).
  - **Method**: procedural abstraction that hides its implementation (user can call the method without having to know the implementation).
  - **Operation**: an abstraction of a set of methods.

# Features of Object Oriented Language (cont.)

  - **Attributes** and **Associations**: abstractions of the underlying instance variables used to implement them.
- **Modularity**
  - An object-oriented system can be constructed entirely from a set of classes.
  - Each class takes care of a particular subset of the functionality (functionality related to a given type of data), rather than having the functionality spread out over many parts of the system.
- **Encapsulation**
  - A class acts as a container to hold its features (variables and methods) and defines an interface that allows only some of them to be seen from outside.

# Features of Object Oriented Language (cont.)

- **Information hiding**
  - Software developers using some feature of a programming language or system do not need to know all the details; they only need to know sufficient details to use the feature.
  - Abstraction, modularity and encapsulation each help provide information hiding.

# Metrics for measuring the quality and complexity of a program

## Goals of measurement in Software Engineering

- Better prediction of the time and effort required for development, and
- Improved control of aspects of quality such as reliability, usability and maintainability.

**Metric**: a well-defined method and formula for computing some value of interest to a software engineer.

# Metrics for measuring the quality and complexity of a program (cont.)

- **Lines of code (LOC)**
  - Simple metric that describe the amount of work developers have accomplished in terms of the number of lines of code they have written.
  - Not a reasonable metric to predict future maintenance effort.
- **Uncommented lines of code**
  - Only the lines containing actual source code statements are included.
  - Less biased compared to LOC.
- **Percentage of lines with comments**
  - Code is said to be more maintainable if it has lots of informative comments.
  - A well-structured code with better choices for variable and method names can be self-documenting and therefore require fewer comments.

# Metrics for measuring the quality and complexity of a program (cont.)

- **Number of classes**
  - Good indicator of the overall size of a design.
  - The number of classes can be affected significantly by the quality of the design.
- **Number of methods per class**
  - If a class has a very large number of methods it is often a sign that it is too complex.
- **Number of public methods per class**
  - Number of public methods in a class should be very small.
- **Number of public instance variables per class**
  - Ideally this should be zero.
  - It is good practice to make them all as private as possible.

# Metrics for measuring the quality and complexity of a program (cont.)

- **Number of parameters per method**
  - A low number is better.
  - Most methods should take zero or one parameter.
- **Number of lines of code per method**
  - It is considered better to have more, but smaller methods.
- **Depth of the inheritance hierarchy**
  - Very complex inheritance hierarchies can be quite difficult to maintain.
  - At the same time, having no inheritance at all limits opportunities for reuse.
- **Number of overridden methods per class**
  - A number too high here suggests problems in the design.
  - A subclass is supposed to be a specialization of its superclass, not something completely different.

# Object-oriented Software Development Life Cycle

The Object-oriented Software Development Life Cycle (SDLC) consists of three macro processes:

1. **Object-oriented Analysis**
   - Develops an object-oriented model of the application domain.
2. **Object-oriented Design**
   - Develops an object-oriented model of the software system.
3. **Object-oriented Implementation**
   - Realizes the software design with the object-oriented programming language that supports direct implementation of objects, classes, and inheritance.