Java By Abstraction: Chapter 9

Inheritance

Some examples and/or figures were borrowed (with permission)
from slides prepared by Prof. H. Roumani

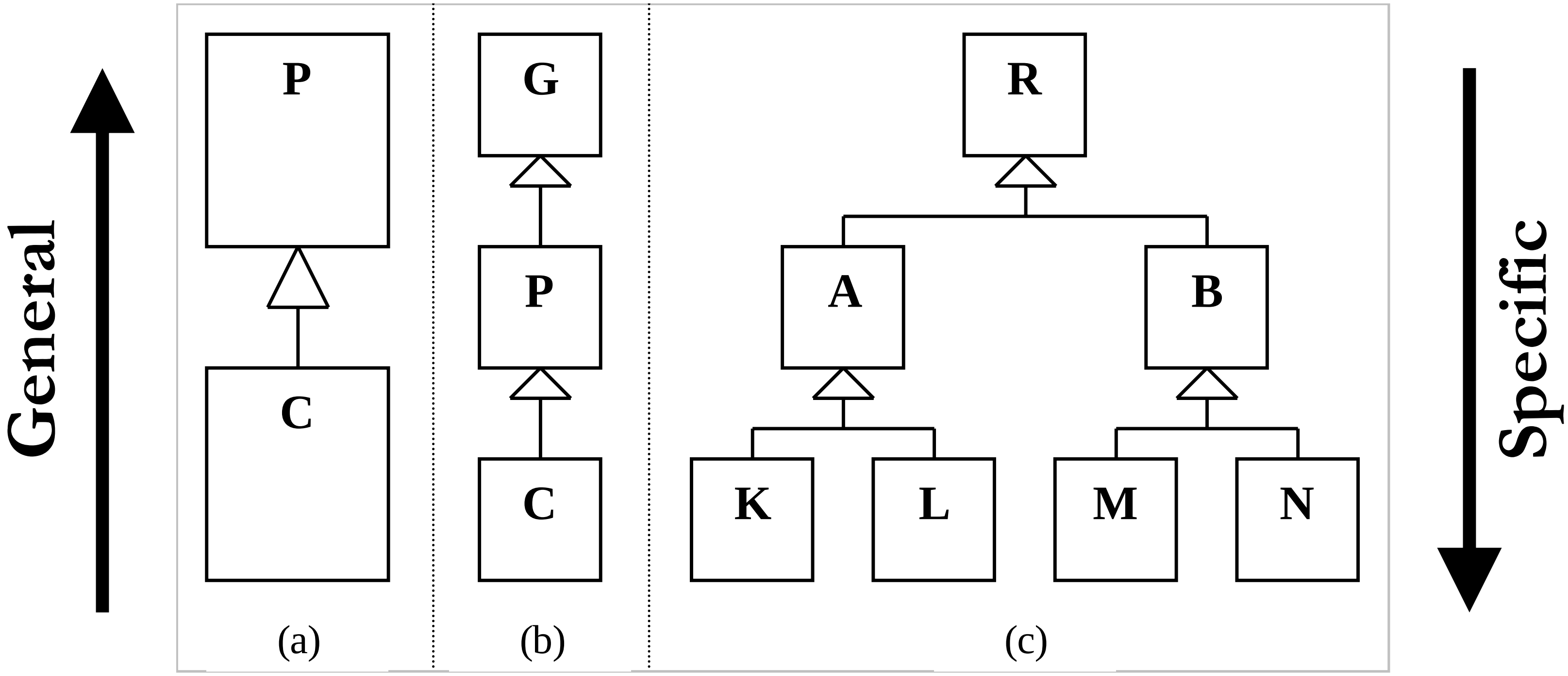
What is Inheritance?

- A thing sometimes can be described as a specialized type of another thing
  - E.g., a car is a particular type of vehicle
  - E.g., a dog is a particular type of animal
  - E.g., a laptop is a particular type of computer
  - E.g., a cell phone is a particular type of telephone
- Similarly, a class sometimes can be described as an extension or abstraction of another class
- The extended class (child) inherits all the features of the original class (parent) and can implements new/different features for its particular purpose

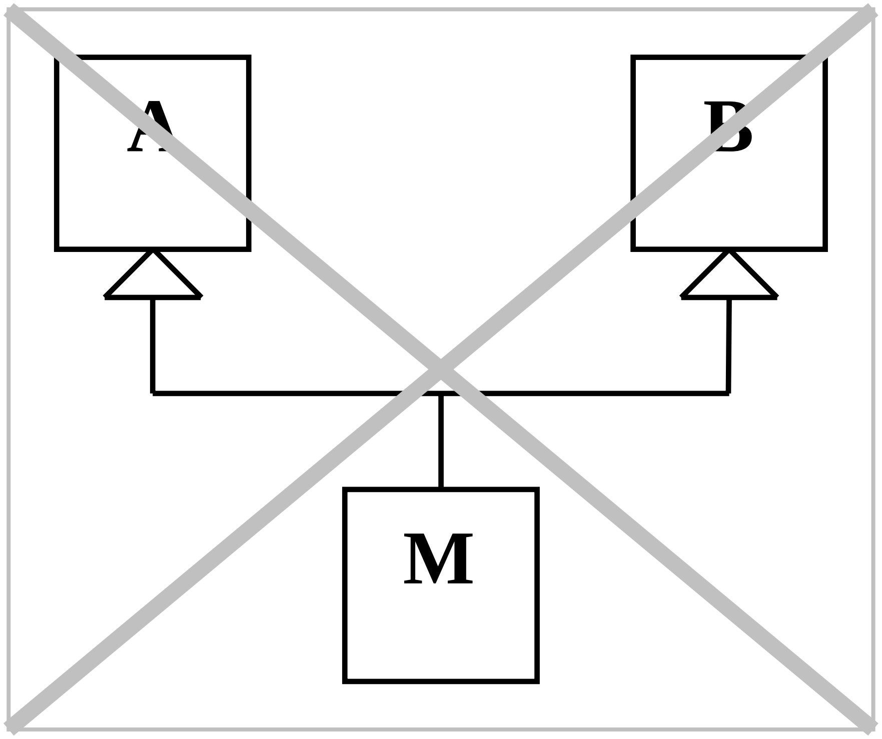
Definition and Terminology

- Child (class) = Subclass
- Parent (class) = Superclass
- When *C* inherits from *P*, every feature of *P* is in *C*
- "*C* inherits from *P*" = "*C* extends *P*"
- Inheritance = "*is-a*" relationship = specialization
- Inheritance hierarchy: (graphical) organization of classes related by inheritance

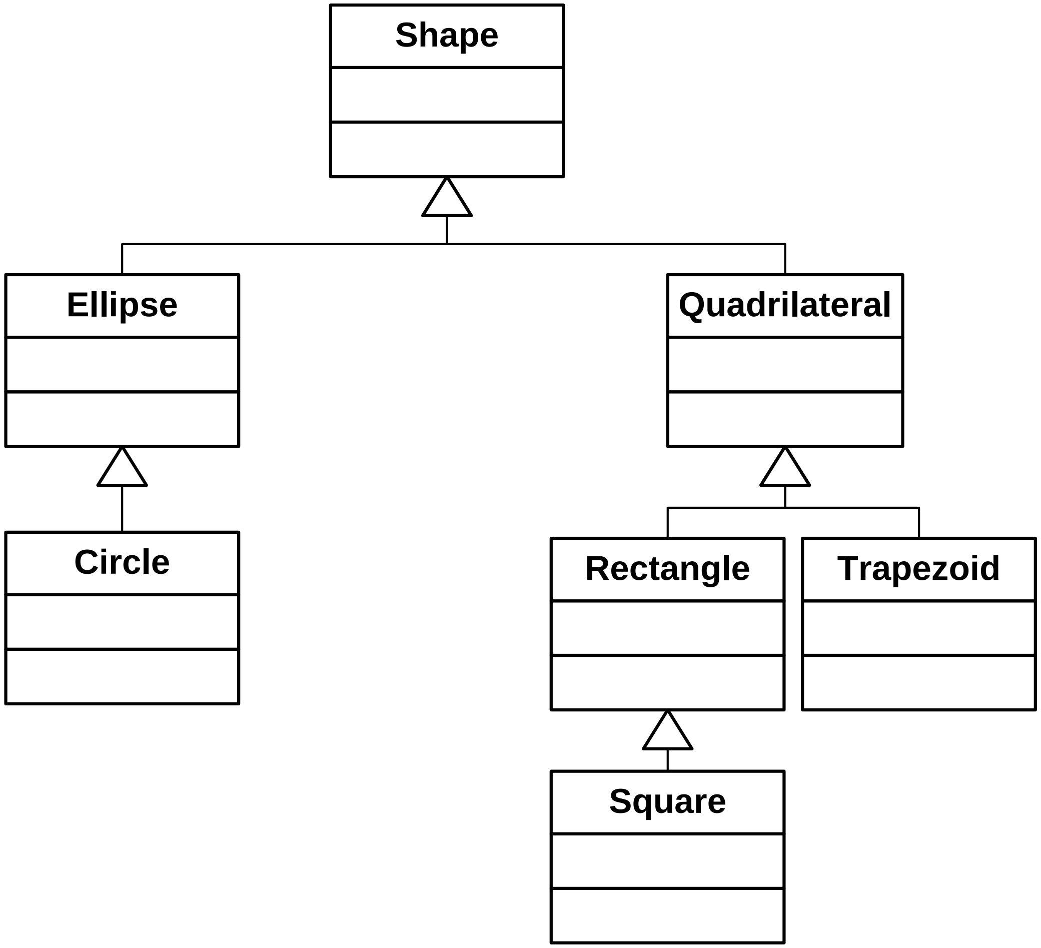
UML Representation



No Multiple Inheritance



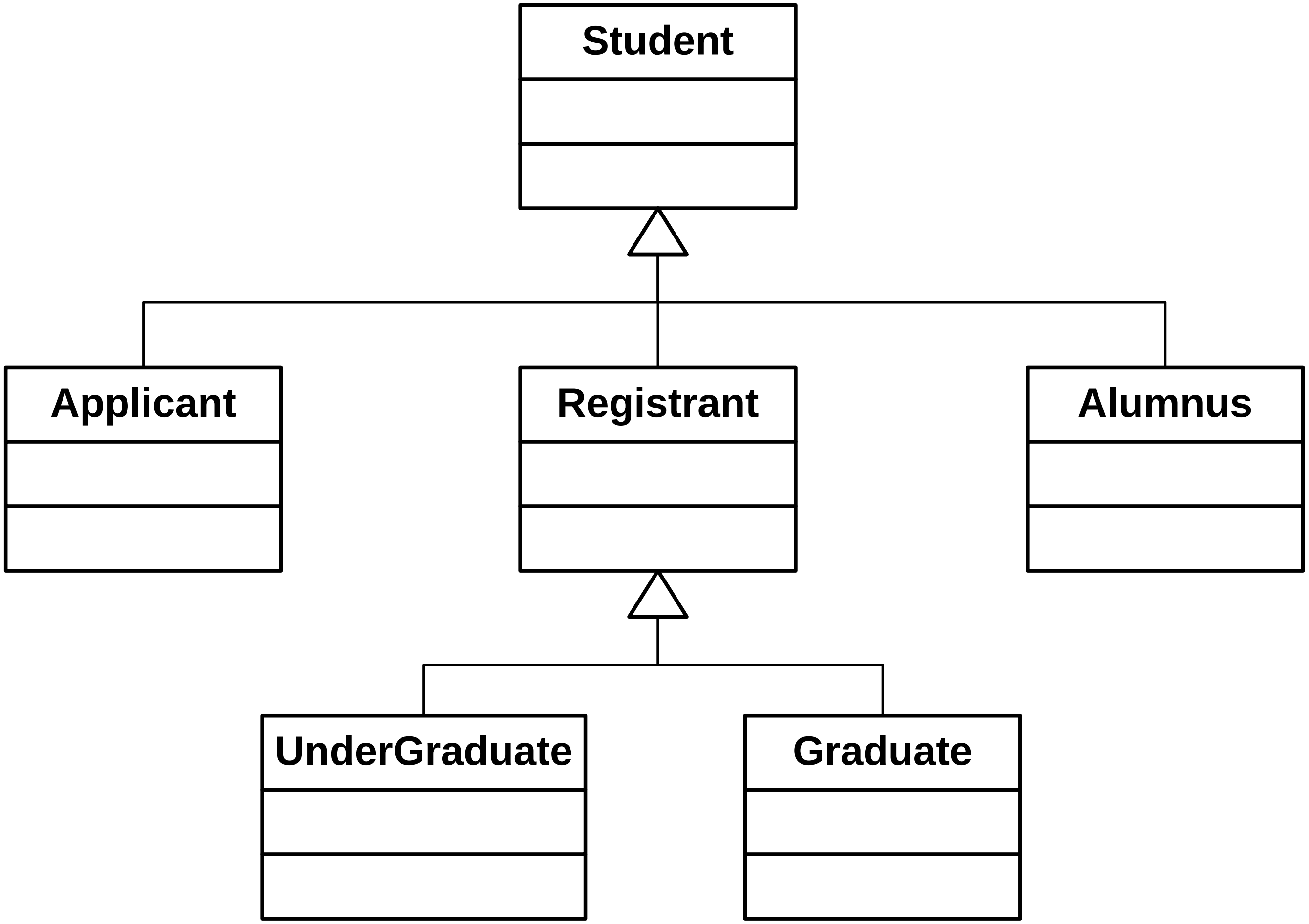
Example Shape Hierarchy



Example

- Situation: The University wants a program to manage information about past, present, and future students
- Task: Give a UML class diagram illustrating the inheritance hierarchy
  - Identify the specific types of students
  - Identify how they relate using “is-a” relationships

Student Inheritance Hierarchy



Overriding or Shadowing Parent Methods

- Child class sometimes requires a method with specialized implementation to take advantage of features not available in the parent class
- Overriding:
  - Child class keeps parent method's signature **and return type**
- Shadowing:
  - Child class keeps parent method's name only (number or types of parameter are different)
  - Like overloading, but spans parent and child classes

Inheritance Example: CreditCard

- CreditCard class:
  - Charge purchases
  - Pay balance
- RewardCard class:
  - (similar features of CreditCard class)
  - Earn reward points

Inheritance Example: CreditCard

- Some features are common:
  - Credit limit
  - Card balance
  - Issue date
  - Expiry date
  - Card number
  - Holder's name
- Some features are unique to RewardCards
  - Points balance

Inheritance Example: CreditCard

- Examine the API of CreditCard and RewardCard
- Identify inherited features
- Identify overridden features

- Other inheritance hierarchies are detailed on pages 357 - 359

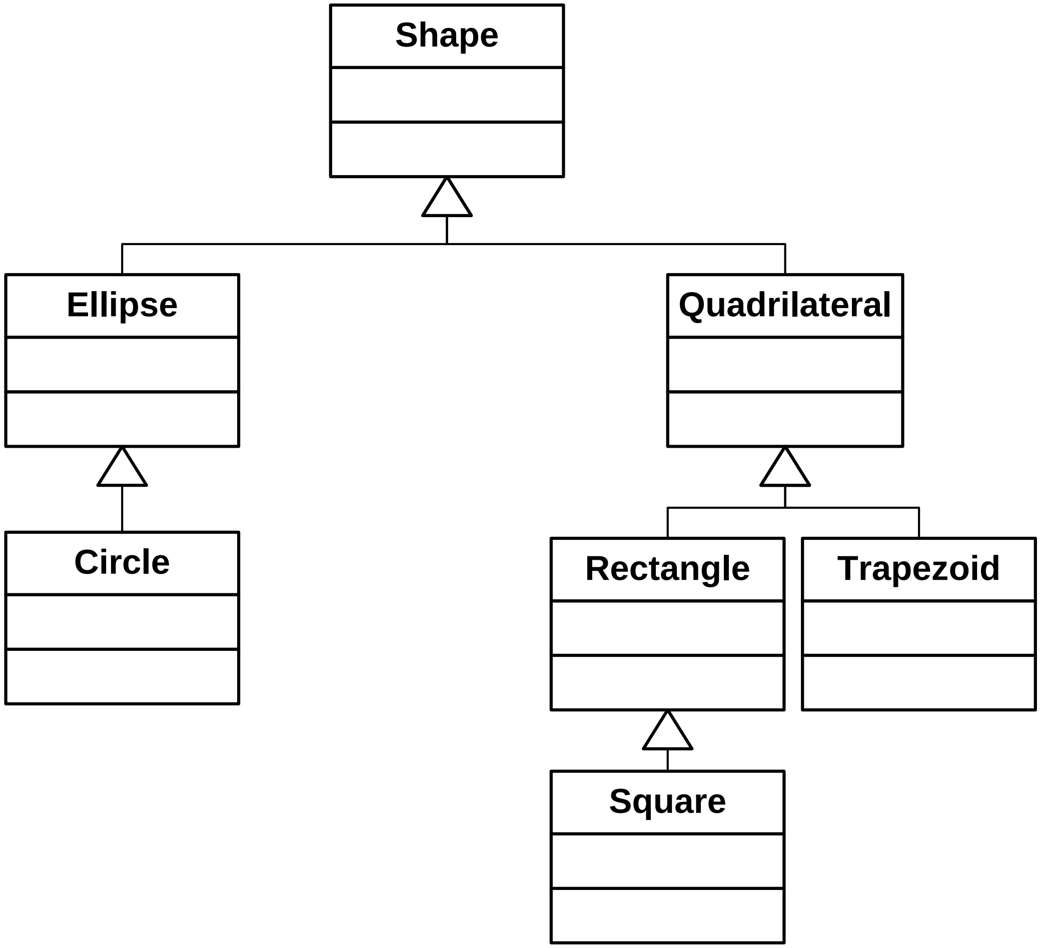
The Substitutability Principle

- “When a parent is expected, a child is accepted”
- This allows the same code to process both parent classes and their (grand) children
- For example, a program intended to handle CreditCard objects will be able to handle RewardCard objects without modification

Substitutability Example

- The following is correct:
    - CreditCard cc1 = new CreditCard(9, “Adam”);
    - CreditCard cc2 = new RewardCard(9, “Adam”);
    - Subsequently, any method that can be called on a CreditCard can also be called on a RewardCard
- The following is NOT correct (why?):
    - RewardCard rc = new CreditCard(9, “Adam”);

Example Shape Hierarchy



Example Shape Hierarchy

- Ellipse: a rounded shape
  - Circle: an ellipse whose height and width are equal
- Thus, a circle is an ellipse, but an ellipse is not necessarily a circle

- Quadrilateral: a four-sided shape
  - Rectangle: a quadrilateral with four sides meeting at 90º
    - Square: a rectangle with four sides of equal length
- Thus, a square is a rectangle, but a rectangle is not necessarily a square

instanceof Operator

- Used to test if a reference points to an instance of the parent or child class
  - CreditCard cc1 = new CreditCard(9, “Adam”);
  - CreditCard cc2 = new RewardCard(9, “Adam”);

  - cc1 **instanceof** CreditCard → true
  - cc2 **instanceof** RewardCard → true
  - cc2 **instanceof** CreditCard → true (by substitutability)
  - cc1 **instanceof** RewardCard → false

Early and Late Binding

- Binding: validation of a method call
- Early binding:
  - Occurs at compile-time
  - Binding failure results in a compile-time error (i.e., cannot find method)
- Late binding:
  - Applicable only when (explicit) inheritance is used
  - Occurs at run-time

Binding Example One

- CreditCard cc2 = new RewardCard(9, “Adam”);

  cc2. getBalance();
- Early binding:
  - Verifies “getBalance()” method in CreditCard class
- Late binding:
  - Determines cc2 points to a RewardCard object
  - Cannot find “getBalance()” method in RewardCard because “getBalance()” was not overridden in RewardCard
  - Calls “getBalance()” method in CreditCard class instead

Binding Example Two

- CreditCard cc2 = new RewardCard(9, “Adam”);
  cc2.charge(500.00);
- Early binding:
  - Verifies “charge(double amount)” is a method in the CreditCard class
- Late binding:
  - Determines cc2 points to a RewardCard object
  - Calls “charge(double amount)” method in RewardCard class

Polymorphism

- The ability of a method to take on various forms

- Occurs when early binding targets a method in a parent class and late binding targets the method with the same signature in a (grand) child class
  - E.g.: the “charge(double amount)” method from the previous example

The Need to Cast

- Wrong:
  - CreditCard cc2 = new RewardCard(9, “Adam”);

    balance = cc2.getPointBalance();
  - Early binding will fail because CreditCard does not have a “getPointBalance()” method
- Correct:
  - CreditCard cc2 = new RewardCard(9, “Adam”);

    if (cc2 instanceof RewardCard)

    { balance = ((RewardCard)cc2).getPointBalance();

    }

Abstract Classes and Interfaces

- Interfaces:
  - Define only method signatures
  - Methods have no implemented body
  - Allow implementer to define class requirements to other implementers
- Abstract classes:
  - Only some (not all) methods are implemented
  - Allow implementers implement some methods and define requirements for others

Abstract Classes and Interfaces (Client View)

- Classes: public class *ClassName*
- Abstract: public abstract class *ClassName*
- Interface: public interface *InterfaceName*
- Interface names appear in *italics* in the API
- Both can be used as types for declarations
- Neither can be instantiated
    - Look for a class that extends it or a (static) method that returns a pre-made instance of it
    - E.g., Try to create an instance of Calendar

Obligatory Inheritance

- The Object class is the root of all inheritance hierarchies
- The Object class defines methods applicable to and required by all Java classes.
  - equals(Object other)
  - toString()
  - …
- To ensure all classes have these methods, all classes implicitly extend the Object class