David M. Keil, Framingham State University

CSCI 252 Computer Science II Using Java

5. Inheritance and polymorphism

1. Inheritance in object-oriented design
2. Inheritance in Java
3. Interface types
4. Polymorphism

Inquiry

- Does object-oriented design have a class concept analogous to *module hierarchies*?
- What is a *taxonomy*?

Topic objective

Explain and implement the notions of an *inheritance hierarchy* and of *polymorphic behavior*

1. Object-oriented design and inheritance

- What are some relationships among *concepts*?
- What distinguishes *is-a* from *has-a*?

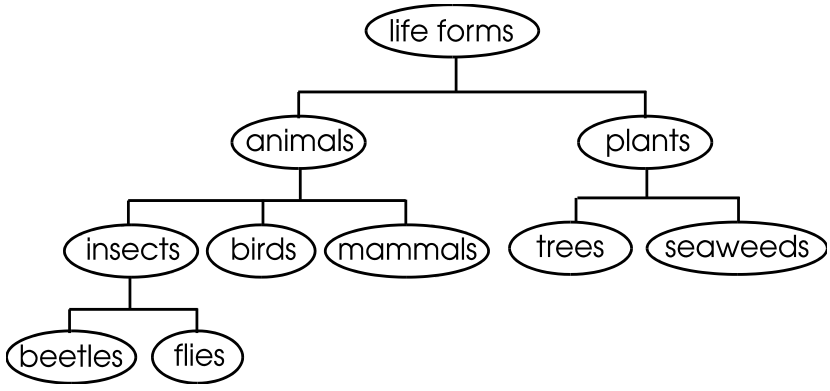
Subtopic objective

5.1a Explain inheritance*

Extending classes with inheritance

- The concept “car” is a *kind of* vehicle in that all cars have the features of vehicles
- Cars *extend* the concept of vehicles, or *inherit* the features of vehicles; e.g., to move, steer, stop, transport cargo
- Java classes may inherit similarly, including inheriting behaviors or operations (as methods)

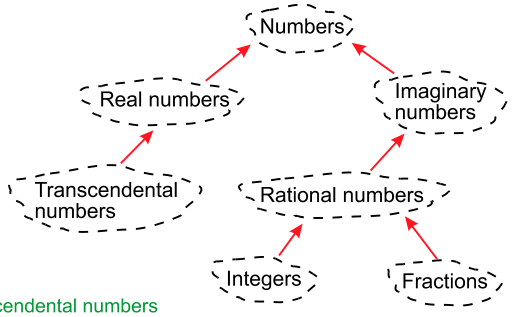
A taxonomy uses inheritance

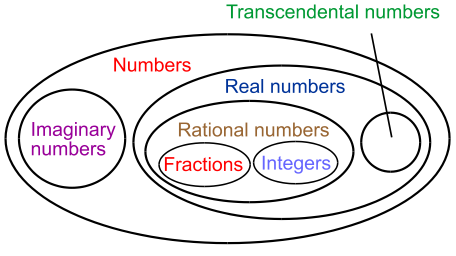


- *Example:* a tree is a *kind of* plant

Numeric subclasses

Numbers
- A. Reals
  - 1. Transcendentals
  - 2. Rationals
    - a. Integers
    - b. Fractions
- B. Imginaries





These three notations for inheritance are equivalent

Object-oriented design and inheritance

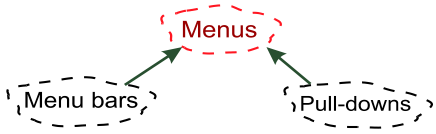
- A class that embodies a concept that is a *subcategory* of some other concept may inherit from the class that represents that other concept
- The *is-a* or *kind-of* relationship is an *inheritance* relationship

Base and derived classes

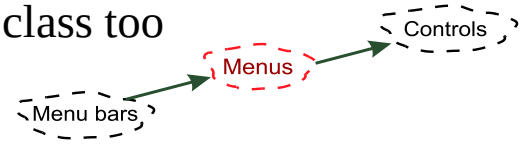
- *Derived class* inherits members from base class
- *Base class* encapsulates a more general category
- Derived class = subclass = descendant
- Base class = superclass = ancestor
- Derived classes are sometimes called *subclasses*

A base class may...

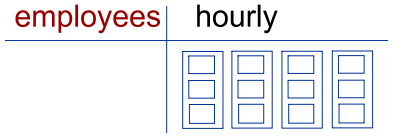
- have multiple derived classes



- be a derived class too



- have no instances (abstract base class)



Base and derived class member names

- Duplicate member identifiers in a derived class override base-class member identifier
- To access base-class identifiers, use *super* with dot

Refactoring

- Definition: modifying program implementation without modifying behavior
- Purpose: improve performance, readability, maintainability, simplicity

2. Inheritance in Java

- How may a Java program use an attribute of an object to select among behaviors?

Subtopic objectives

5.1b Design an inheritance hierarchy*

5.1c Explain issues raised by inheritance

5.2a Define and use a derived class*†

5.2b Predict behavior of derived-class objects

Inheritance in Java

- Classes may have *kind-of* relationships by use of *extends* keyword:
  ```
  class Hourly extends Employee
  ```
- Above, *Hourly* is a subclass, *Employee* is a superclass, *Hourly* automatically inherits all members of *Employee*
- *Object* is a superclass of all other classes
- *Object* methods include *clone(), toString(), equals()*

Inheritance replaces *switch* logic

- Without inheritance, each employee would have extra data members and a flag:

```
enum pay_category {HOURLY, SALARIED};
public class Employee {
  int ID, pay_category;
  double salary, hours, rate;
};
```

- A loop would calculate weekly pay for an employee using *switch*:

```
switch(pay_category)
{
  HOURLY: pay = hours * rate; break;
  SALARIED: pay = salary / 52; break;
};
```


Overriding

- A method of a derived class *overrides* a method of the same name in the base class
- *Example:*

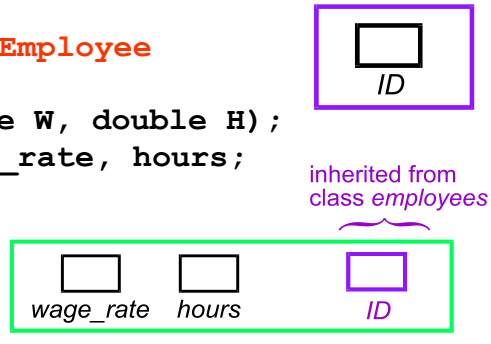
An *Employee* example

- *Problem:* to calculate paychecks for an employee roster that includes salaried and hourly employees
- Some employees have a *salary*
- Others have an hourly *wage rate* and an *hours worked* value

Inheritance with employee classes

```
class Employee
{
  Employee(int I) { ID = I; };
  private int ID;
};

class Hourly extends Employee
{
  hourly(int I, double W, double H);
  private double wage_rate, hours;
};
```



Base-class constructor runs first

```
class Game
{
   public Game() { out.print("Games"); }
};

class BoardGame extends Game
{
   public BoardGame()
   { super(); out.println("BoardGames"); };

public static void main(String[] args)
{
   BoardGame g = new BoardGame();
}
```

Output:
games
board_games

Inheritance example

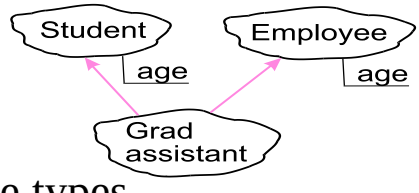
- Instances of subclasses of classes with automatic display methods inherit the auto-display behavior
- For example, a program may extend the *JComponent* class for graphical objects displayed in frames
- An instance of *JComponent* automatically displays itself when the frame is resized or moved

Derived classes and encapsulation

- Derived classes don't have access to private fields and methods of base classes
- Derived-class constructors' first statement must be a call to base-class constructor, using *super*, to initialize base-class fields

Multiple inheritance

- Some concepts may have more than one *is-a* relationship
  - teaching assessments are both student and employees
  - birds are both walkers and fliers
- The diamond inheritance problem: *age* member name is ambiguous
- Solution: not multiple inheritance, but interface types



3. Interface types

- Is a type name ever impossible to instantiate?

Subtopic objective

5.3 Test an interface type†

Interface types (classes)

- A kind of overloading that supports reusability of code via the *is-a* relationship
- Implemented using the *interface* keyword
- All methods of an interface class are *abstract* and *public*, with signatures but not definitions
- All methods are public but undefined (abstract)
- Interface types lack data members (fields)
- To implement an interface class, declare a class that uses it and that defines its abstract methods

Interface-class example

```
public interface Comparable
{
  boolean is_greater_than(Comparable x);
}
public Batter implements Comparable
{
  private double batAvg;
  public boolean is_greater_than(Batter x)
  {
    return (bat_avg > x.get_bat_avg());
  }
}
```

- This shows how the *Comparable* interface expresses the commonality among classes like *Batter*, *Integer*, *String*, etc. – all can be compared

Interface types

- *Interfaces* specify methods for classes but not implementations (data members or method definitions)
- *Example:* an interface for collections of measurable objects may have *sort* and *average* methods
- All methods are public by definition; *public* is not specified; no constructor
- An interface is implemented by a class but may not be *instantiated*

Interface example

- ```
  Public interface Quantifiable
  { double getQuantity(); }
  ```
- Implementation must define *getQuantity*
- Example that uses *Quantifiable*:

```
public static double
sum(Quantifiable[] A)
{
   double sum = 0;
   for(int i = 0; i < A.length; i++)
        sum += A[i].getQuantity();
   return sum;
}
```
```

A *Shape* interface

- An interface type, rather than inheritance, may be appropriate because shape-related methods don't share code
- *Methods: getArea, getPerimeter*
- *Example code:*

```
Shape[] s = new Shape[5];
s[0] = new Circle(5);
s[0] = new Triangle(10,10,10);
s[0] = new Square(16);
for (int i = 0; i < s.length; i++) …
```


Java class library interfaces

- *ActionListener* (java.awt)
- *Serializable* (java.io)
- *Comparable* (>, <, ==)
- *Formattable* (printf)
- *Runnable* (multithreading)
- Collection interfaces: *List*, *Set*, *Map*, *Iterator* (*java.util*)

Extending an interface

- Interfaces can be reused, like base classes, by using the keyword *extends*
- Example: a *List<E>* interface may extend the *Iterable<E>* one

4. Polymorphism

- How does an outdoor painter decide what to do when asked to "paint 15 Oak St.", which might be a home or a commercial building?
- How does a GUI know what to do when *paint()* is called by a *Window* object?

Subtopic objectives

5.4a Explain polymorphism

5.4b Implement polymorphism†

Overloaded methods and operators are polymorphic

- Meaning of a method name like *sum may* depend on data types of the operands used
- Compiler chooses which overloaded function to call, based on parameter types
- This is a weak form of polymorphism
- With *late binding*, the compiler does not know the parameter types

Polymorphism

- When different classes implement the same interface, the different implementations reflect *polymorphism*
- *Example:* Different graphical objects are drawn in different ways, but all may use the same name *paint* to be drawn, where *paint* is a method of an interface class
- Polymorphism allows us to declare a collection of objects of heterogeneous types and draw all of them using the same method name *paint*

Polymorphism:

- The feature by which objects of different classes in an inheritance hierarchy may respond differently to calls to methods of the same name
- It includes the ability of a base-class method to call a derived-class method

Polymorphism uses inheritance

- Application programmer who uses application-class library writes a derived class that redefines the base-class event handler
- Elements of a list of base-class references may point to object of different derived classes
- *Example:* processing a mixed payroll of hourly and salaried employees

When does the compiler not know the class of an object?

- When base-class pointer is used: *p_shape->draw();*
- When a base-class function calls a derived-class function

Polymorphism is useful with:

- Heterogeneous collections of objects of different derived classes
- Application frameworks, in which base-class *run* calls derived-class event handlers

Derived-class methods may override base-class ones

```
class Animal
{
   public Animal() { };
   public void tellMotion()
    {out.println("Animals move around.");};
}
class Bird extends Animal
{
   public Bird() { super(); };
   public void tellMotion()
     { out.println("Birds move on two legs and wings"); };
};

public static void main(String[] args)
{
   Bird b = new Bird();
   b.tellMotion();
}
```

Derived-class method is called

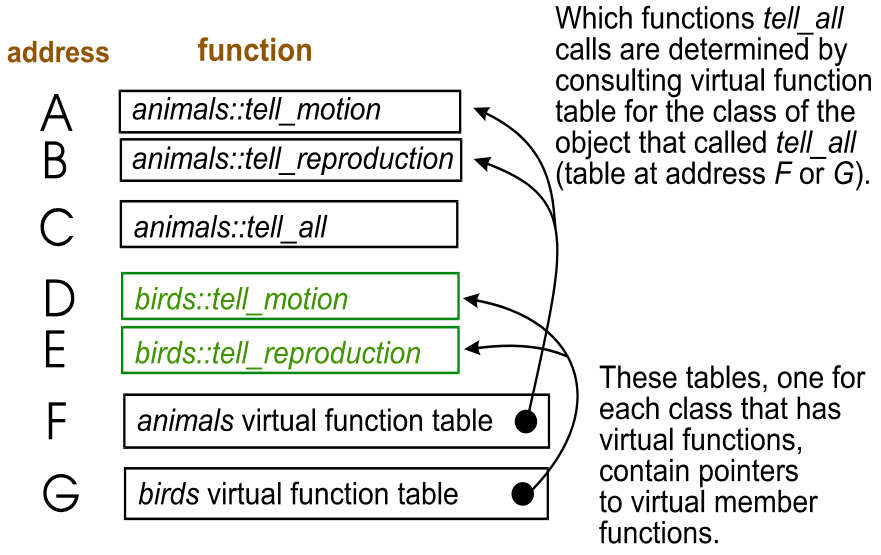
Output:

```
Birds move on two legs and wings
```


switch logic vs. polymorphism

- *Workaday example:* A lawnmower shop takes different fix-it jobs. What to do depends on nature of the problem.
- Two approaches to a solution:
  1. Write one repair plan using logic like *switch*
  2. For each category of repair job, write a repair plan
- Advantage of #2: Shop may change its capabilities without altering main loop -- just define new classes and repair plans

Base-class methods may call methods of an *abstract* derived class



Late binding

- *Definition:* Determination of the call address of a method call by lookup at run time, rather than at compile time.
- *Advantage*: Allows polymorphic behavior because which method is called will depend on the class of the calling object.

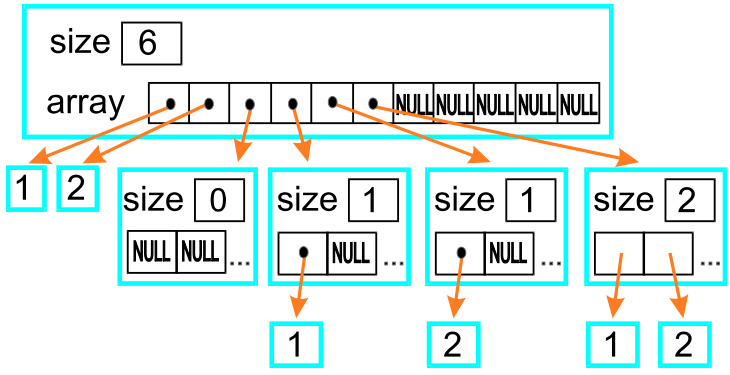
Abstract base classes

- An *abstract base class* is intended for use with polymorphism
- An abstract class cannot be instantiated
- An abstract base class are designated *abstract*
- Any declared but undefined methods are designated *abstract*
- Base-class methods may be called using "super."

A mixed collection of shapes

```
class Shape
{
    private extent;
    public void draw() {}
}
class Line extends Shape
{ … }
class Square extends Shape
{ … }
[main:]
Shape[] A = {new Line(), new Square()};
```

A set of integers and sets of integers



{ 1, 2, { }, {1}, {2}, {1,2} }

References

Cay Horstmann. *Big Java*, 3rd ed. Wiley, 2008. Ch. 9, 18.

S. Reges and M. Stepp. *Building Java Programs*. Pearson, 2014. Ch. 9