CSC 222: Object-Oriented Programming

Fall 2017

Inheritance & polymorphism

- inheritance, derived classes
- inheriting fields & methods, overriding fields and methods
- IS-A relationship, polymorphism
- super methods, super constructor
- instanceof, downcasting
- examples: ColoredDie, SortedArrayList

1

Interfaces & polymorphism

recall that interfaces

- define a set of methods that a class must implement in order to be "certified"
- any class that implements those methods and declares itself is "certified"

```
public class myList<E> implements List<E> {
    . . .
}
```

- can use the interface name in place of a class name when declaring variables or passing values to methods

```
List<String> words = new MyList<String>();

public int sum(List<Integer> data) {
    int sum = 0;
    for (int i = 0; i < data.size(); i++) {
        sum += data.get(i);
    }
    return sum;
}
```

- polymorphism refers to the fact that the same method call (e.g., `size` or `get`) can refer to different pieces of code when called on different objects
- enables the creation of generic libraries (e.g., `Collections`)

2

Inheritance

a closely related mechanism for polymorphic behavior is *inheritance*

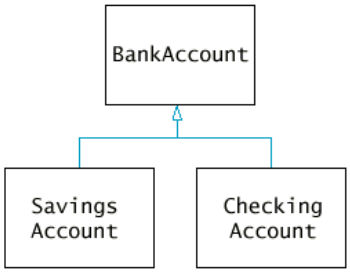
- one of the most powerful techniques of object-oriented programming
- allows for large-scale code reuse

with inheritance, you can derive a new class from an existing one

- automatically inherit all of the fields and methods of the existing class
- only need to add fields and/or methods for new functionality

example:

- *savings account* is a bank account with interest
- *checking account* is a bank account with transaction fees



3

BankAccount class

here is an implementation of a basic BankAccount class

- stores account number and current balance
- uses static field to assign each account a unique number
- accessor methods provide access to account number and balance
- deposit and withdraw methods allow user to update the balance
- toString method for printing

```
public class BankAccount {
  private double balance;
  private int accountNumber;
  private static int nextNumber = 1;

  public BankAccount() {
    this.balance = 0;
    this.accountNumber =
        BankAccount.nextNumber;
    BankAccount.nextNumber++;
  }

  public int getAccountNumber() {
    return this.accountNumber;
  }

  public double getBalance() {
    return this.balance;
  }

  public void deposit(double amount) {
    this.balance += amount;
  }

  public void withdraw(double amount) {
    if (amount >= this.balance) {
      this.balance -= amount;
    }
  }

  public String toString() {
    return this.getAccountNumber() +
           ": $" + this.getBalance();
  }
}
```

4

BankAccount example

like any other class

- can declare variables of type BankAccount
- can call methods on those objects

```
BankAccount acc1 = new BankAccount();
acc1.deposit(50.00);
System.out.println(acc1);

BankAccount acc2 = new BankAccount();
double amount = acc1.withdraw(12.50);
acc2.deposit(amount);
```

```
public class BankAccount {
  private double balance;
  private int accountNumber;
  private static int nextNumber = 1;

  public BankAccount() {
    this.balance = 0;
    this.accountNumber =
        BankAccount.nextNumber;
    BankAccount.nextNumber++;
  }

  public int getAccountNumber() {
    return this.accountNumber;
  }

  public double getBalance() {
    return this.balance;
  }

  public void deposit(double amount) {
    this.balance += amount;
  }

  public void withdraw(double amount) {
    if (amount >= this.balance) {
      this.balance -= amount;
    }
  }

  public String toString() {
    return this.getAccountNumber() +
           ": $" + this.getBalance();
  }
}
```

5

Specialty bank accounts

now we want to implement SavingsAccount and CheckingAccount

- a savings account is a bank account with an associated interest rate, interest is calculated and added to the balance periodically
- could copy-and-paste the code for BankAccount, then add a field for interest rate and a method for adding interest

- a checking account is a bank account with some number of free transactions, with a fee charged for subsequent transactions
- could copy-and-paste the code for BankAccount, then add a field to keep track of the number of transactions and a method for deducting fees

disadvantages of the copy-and-paste approach

- tedious work
- lots of duplicate code – *code drift* is a distinct possibility
  if you change the code in one place, you have to change it everywhere or else lose consistency (e.g., add customer name to the bank account info)
- limits polymorphism (will demonstrate later)

6

SavingsAccount class

inheritance provides a better solution

- can define a SavingsAccount to be a special kind of BankAccount
  inherits common features (balance, account #, deposit, withdraw, toString)
- simply add the new features specific to a savings account
  need to store interest rate, provide method for adding interest to the balance

- general form for inheritance:

```
public class DERIVED_CLASS extends EXISTING_CLASS {
  ADDITIONAL_FIELDS

  ADDITIONAL_METHODS
}
```

note: the derived class does not explicitly list fields/methods from the existing class (a.k.a. parent class) – they are inherited and automatically accessible

```
public class SavingsAccount extends BankAccount {
  private double interestRate;

  public SavingsAccount(double rate) {
    this.interestRate = rate;
  }

  public void addInterest() {
    double interest =
        this.getBalance()*this.interestRate/100;
    this.deposit(interest);
  }
}
```

7

Using inheritance

```
BankAccount generic = new BankAccount();          // creates bank account with 0.0 balance

generic.deposit(120.0);                           // adds 120.0 to balance

generic.withdraw(20.0);                           // deducts 20.0 from balance

System.out.println(generic.getBalance());         // displays current balance: 100.0
```

```
SavingsAccount passbook = new SavingsAccount(3.5);// creates savings account, 3.5% interest

passbook.deposit(120.0);                          // calls inherited deposit method

passbook.withdraw(20.0);                          // calls inherited withdraw method

System.out.println(passbook.getBalance());        // calls inherited getBalance method

passbook.addInterest();                           // calls new addInterest method

System.out.println(passbook.getBalance());        // displays 103.5
```

8

CheckingAccount class

can also define a class that models a checking account

- again, inherits basic features of a bank account
- assume some number of free transactions
- after that, each transaction entails a fee
- must *override* the deposit and withdraw methods to also keep track of transactions
- can call the versions from the parent class using super

```
super.PARENT_METHOD();
```

```
public class CheckingAccount extends BankAccount {
  private int transactionCount;
  private static final int NUM_FREE = 3;
  private static final double TRANS_FEE = 2.0;

  public CheckingAccount() {
    this.transactionCount = 0;
  }

  public void deposit(double amount) {
    super.deposit(amount);
    this.transactionCount++;
  }

  public void withdraw(double amount) {
    super.withdraw(amount);
    this.transactionCount++;
  }

  public void deductFees() {
    if (this.transactionCount > CheckingAccount.NUM_FREE) {
      double fees =
          CheckingAccount.TRANS_FEE *
           (this.transactionCount-CheckingAccount.NUM_FREE);
      super.withdraw(fees);
    }
    this.transactionCount = 0;
  }
}
```

9

Interfaces & inheritance

recall that with interfaces

- can have multiple classes that implement the same interface
- can use a variable of the interface type to refer to any object that implements it

```
List<String> words1 = new ArrayList<String>();
List<String> words2 = new LinkedList<String>();
```

- can use the interface type for a parameter, pass any object that implements it

```
public void DoSomething(List<String> words) {
  . . .
}

-------------------------------------------

DoSomething(words1);

DoSomething(words2);
```

the same capability holds with inheritance

- could assign a SavingsAccount object to a variable of type BankAccount
- could pass a CheckingAccount object to a method with a BankAccount parameter

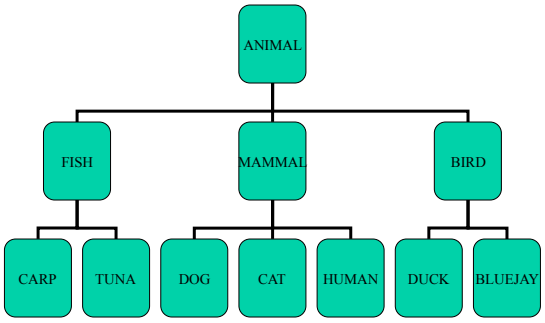
10

IS-A relationship

the IS-A relationship holds when inheriting

- an object of the derived class is still an object of the parent class
- anywhere an object of the parent class is expected, can provide a derived object

- consider a real-world example of inheritance: animal classification



11

Polymorphism

in our example

- a SavingsAccount is-a BankAccount (with some extra functionality)
- a CheckingAccount is-a BankAccount (with some extra functionality)

- whatever you can do to a BankAccount (e.g., deposit, withdraw), you can do with a SavingsAccount or Checking account
  - derived classes can certainly do more (e.g., addInterest for SavingsAccount)
  - derived classes may do things differently (e.g., deposit for CheckingAccount)

polymorphism: the same method call can refer to different methods when called on different objects

- the compiler is smart enough to call the appropriate method for the object

```
BankAccount acc1 = new SavingsAccount(4.0);
BankAccount acc2 = new CheckingAccount();

acc1.deposit(100.0);          // calls the method defined in BankAccount
acc2.deposit(100.0);          // calls the method defined in CheckingAccount
```

- allows for general-purpose code that works on a class hierarchy

12

Example use

```
import java.util.ArrayList;

public class Portfolio {
  private ArrayList<BankAccount> accounts;

  public Portfolio() {
    this.accounts = new ArrayList<BankAccount>();
  }

  public void addAccount(BankAccount acc) {
    this.accounts.add(acc);
  }

  public double getTotalBalance() {
    double total = 0.0;
    for (BankAccount acc : this.accounts) {
      total += acc.getBalance();
    }
    return total;
  }

  public boolean addToAccount(int accountNumber, double amount) {
    for (BankAccount acc : this.accounts) {
      if (acc.getAccountNumber() == accountNumber) {
        acc.deposit(amount);
        return true;
      }
    }
    return false;
  }

  public String toString() {
    String text = "";
    for (BankAccount acc : this.accounts) {
      text += acc + "\n";
    }
    return text;
  }
}
```

note: the inputs to addAccount can be BankAccounts, SavingsAccounts, or CheckingAccounts

- the methods only assume the common BankAccount methods
- each object contains references to its own methods, so the correct version is called each time

13

In-class exercise

define the BankAccount, SavingsAccount, and CheckingAccount classes

create objects of each class and verify their behaviors

are account numbers consecutive regardless of account type?

- should they be?

what happens if you attempt to withdraw more than the account holds?

- is it ever possible to have a negative balance?

14

Another example: colored dice

```
public class Die {
  private int numSides;
  private int numRolls;

  public Die(int sides) {
    this.numSides = sides;
    this.numRolls = 0;
  }

  public int roll() {
    this.numRolls++;
    return (int)(Math.random()*this.numSides)+1;
  }

  public int getNumSides() {
    return this.numSides;
  }

  public int getNumRolls() {
    return this.numRolls;
  }
}
```

we already have a class that models a simple (non-colored) die

- can extend that class by adding a color field and an accessor method
- need to call the constructor for the Die class to initialize the numSides and numRolls fields

```
super(ARGS);
```

```
public enum DieColor {
    RED, WHITE
}

public class ColoredDie extends Die {
  private DieColor dieColor;

  public ColoredDie(int sides, DieColor c){
    super(sides);
    this.dieColor = c;
  }

  public DieColor getColor() {
    return this.dieColor;
  }
}
```

15

ColoredDie example

consider a game in which you roll a collection of dice and sum their values

- there is one "bonus" red die that counts double

```
import java.util.ArrayList;
import java.util.Collections;

public class RollGame1 {
  private static final int NUM_DICE = 5;

  private ArrayList<ColoredDie> dice;

  public RollGame1() {
    this.dice = new ArrayList<ColoredDie>();

    this.dice.add(new ColoredDie(6, DieColor.RED));
    for (int i = 1; i < RollGame1.NUM_DICE; i++) {
      this.dice.add(new ColoredDie(6, DieColor.WHITE));
    }
    Collections.shuffle(dice);
  }

  public int rollPoints() {
    int total = 0;
    for (ColoredDie d : this.dice) {
      int roll = d.roll();
      if (d.getColor() == DieColor.RED) {
        total += 2*roll;
      }
      else {
        total += roll;
      }
    }
    return total;
  }
}
```

16

instanceof

if you need to determine the specific type of an object

- use the instanceof operator
- can then downcast from the general to the more specific type
- note: the roll method is defined for all Die types, so can be called regardless
- however, before calling getColor you must downcast to ColoredDie

```
import java.util.ArrayList;
import java.util.Collections;

public class RollGame2 {
  private ArrayList<Die> dice;
  private static final int NUM_DICE = 5;

  public RollGame2() {
    this.dice = new ArrayList<Die>();

    this.dice.add(new ColoredDie(6, DieColor.RED));
    for (int i = 1; i < RollGame2.NUM_DICE; i++) {
      this.dice.add(new Die(6));
    }
    Collections.shuffle(dice);
  }

  public int rollPoints() {
    int total = 0;
    for (Die d : this.dice) {
      int roll = d.roll();
      total += roll;
      if (d instanceof ColoredDie) {
        ColoredDie cd = (ColoredDie)d;
        if (cd.getColor() == DieColor.RED) {
          total += roll;
        }
      }
    }
    return total;
  }
}
```

17

SortedArrayList

recall our earlier discussions on searching and sorting

- if you simply add new values to the end of an ArrayList, then
  add method → O(1) indexOf method → O(N)
- if you add values in the proper, sorted order (assuming they are Comparable), then
  add method → O(N) indexOf method → O(log N)

we could define a SortedArrayList class that keeps its elements sorted

- most methods would behave exactly as they did for ArrayList
  e.g., size, get, remove, clear, toString
- some would need to be overriden to maintain & take advantage of order

  add? set? indexOf?

to avoid duplication and code drift, can utilize inheritance to build off the existing class

18

SortedArrayList

- no new fields, so just call the super constructor to initialize
- add can use binary search to find the location where the item should go, then super.add to add it there
- set has to keep the list sorted, so easiest to remove then add
- indexOf can use binary search
- note that indexOf takes an Object as parameter
- can't use instanceof on a generic type, but can get by with try/catch

```
import java.util.ArrayList;
import java.util.Collections;

public class SortedArrayList<E extends Comparable<? super E>>
extends ArrayList<E> {
  public SortedArrayList() {
    super();
  }

  public boolean add(E item) {
    int index = Collections.binarySearch(this, item);
    if (index < 0) {
      index = -index - 1;
    }
    super.add(index, item);
    return true;
  }

  public E set(int index, E item) {
    E oldItem = this.remove(index);
    this.add(item);
    return oldItem;
  }

  public int indexOf(Object item) {
    try {
      int index = Collections.binarySearch(this, (E)item);
      if (index >= 0) {
        return index;
      }
      return -1;
    }
    catch (ClassCastException e) {
      return -1;
    }
  }
}
```

19

Power of inheritance

note that you can extend a class without any knowledge of its internals

- we didn't need to know anything about ArrayList's implementation to extend to SortedArrayList
  - ✓ just needed to know which methods to override
  - ✓ even if overridden, could still call parent method using super
- don't even need to have access to the source code
  - ✓ suppose it was proprietary code – no problem!

taking advantage of polymorphism, can define methods or data structures that work on a whole family of classes

- Collections.sort will work on any List of Comparable values
  - ✓ includes predefined classes ArrayList & LinkedList
  - ✓ now includes the new SortedArrayList class
  - ✓ can be extended to classes that haven't even been invented yet!
- similarly, an ArrayList<Die> object can contain Die objects or ColoredDie objects

20

OO summary

interfaces & *inheritance* both provide mechanism for class hierarchies

- enable the grouping of multiple classes under a single name
- an interface only specifies the methods that must be provided
  each class that implements the interface must provide those methods
  a class can implement more than one interface
- a derived class can inherit fields & methods from its parent
  can have additional fields & methods for extended functionality
  can even override existing methods if more specific versions are appropriate

the IS-A relationship holds using both interfaces & inheritance

- if class C implements interface I, an instance of C "is a(n)" instance of I
- if class C extends parent class P, an instance of C "is a(n)" instance of P
- *polymorphism*: `obj.method()` can refer to different methods when called on different objects in the hierarchy

  e.g., `savAcct.withdraw(20);` `chkAcct.withdraw(20);`

21