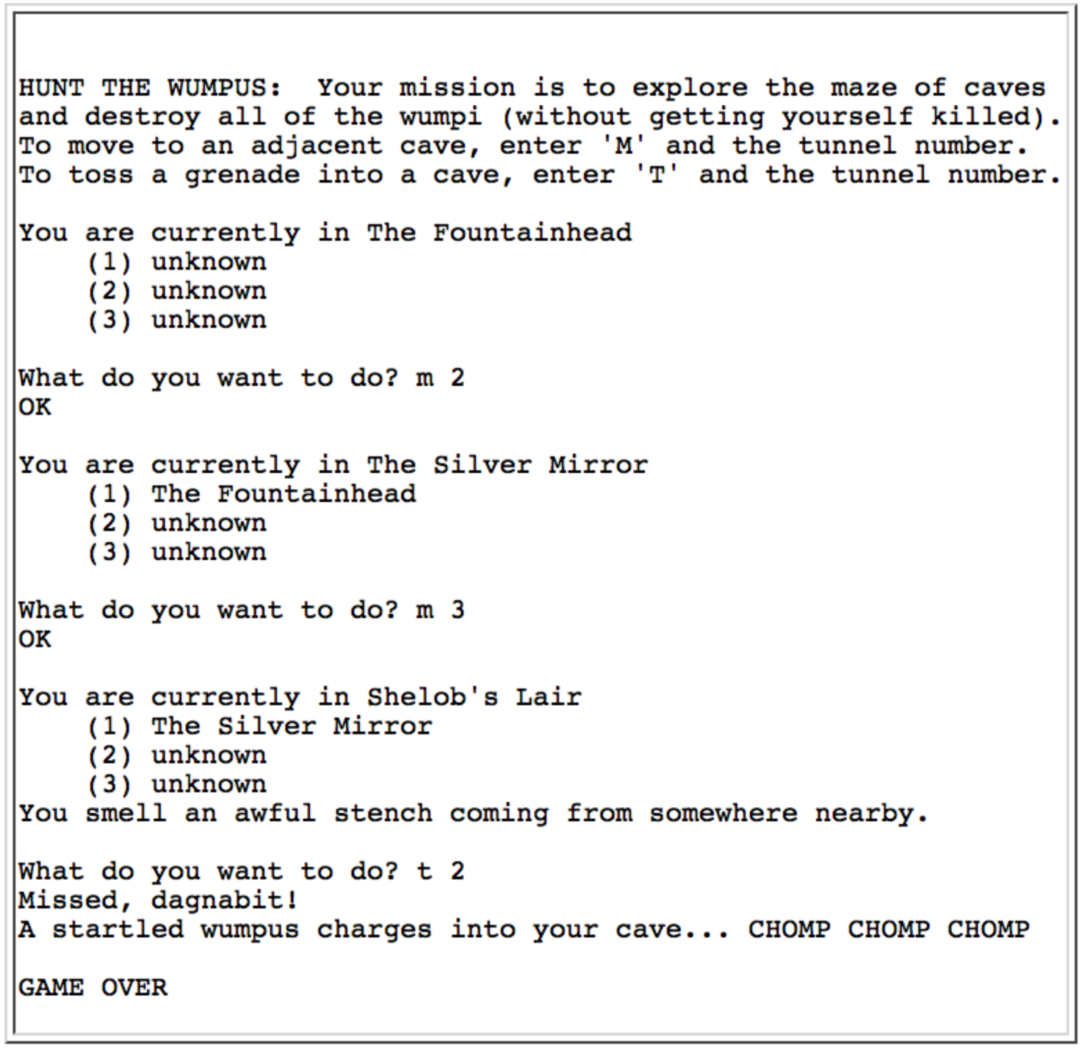
CSC 222: Object-Oriented Programming

Fall 2015

Inheritance & polymorphism

- Hunt the Wumpus, enumerated types
- inheritance, derived classes
- inheriting fields & methods, overriding fields and methods
- IS-A relationship, polymorphism
- super methods, super constructor
- instanceof, downcasting

Hunt the Wumpus



```
HUNT THE WUMPUS:  Your mission is to explore the maze of caves
and destroy all of the wumpi (without getting yourself killed).
To move to an adjacent cave, enter 'M' and the tunnel number.
To toss a grenade into a cave, enter 'T' and the tunnel number.

You are currently in The Fountainhead
    (1) unknown
    (2) unknown
    (3) unknown

What do you want to do? m 2
OK

You are currently in The Silver Mirror
    (1) The Fountainhead
    (2) unknown
    (3) unknown

What do you want to do? m 3
OK

You are currently in Shelob's Lair
    (1) The Silver Mirror
    (2) unknown
    (3) unknown
You smell an awful stench coming from somewhere nearby.

What do you want to do? t 2
Missed, dagnabit!
A startled wumpus charges into your cave... CHOMP CHOMP CHOMP

GAME OVER
```

for HW6, you will implement a text-based adventure game from the 1970's

- you will wander through a series of caves, with each cave connected to 3 others via tunnels
- there are dangers in the caves
  - you can be eaten by a Wumpus
  - you can fall into a bottomless pit
  - you can be picked up and moved by bats
- you can sense dangers in adjacent caves
- you have a limited number of grenades you can throw when you smell a Wumpus

Cave class

you must implement a class that models a single cave

- each cave has a name & number, and is connected to three other caves via tunnels
- by default, caves are empty & unvisited (although these can be updated)

how do we represent the cave contents?

- we could store the contents as a string: "EMPTY", "WUMPUS", "BATS", "PIT"

```
Cave c = new Cave("Cavern of Doom", 0, 1, 2, 3);
c.setContents("WUMPUS");
```

- potential problems?

there are only 4 possible values for cave contents

- the trouble with using a String to represent these is the lack of error checking

```
c.setContents("WUMPAS");     // perfectly legal, but ???
```

Enumerated types

there is a better alternative for when there is a small, fixed number of values

- an enumerated type is a new type (class) whose value are explicitly enumerated

```
public enum CaveContents {
    EMPTY, WUMPUS, PIT, BATS
}
```

- note that these values are NOT Strings – they do not have quotes

- you specify a enumerated type value by ENUMTYPE.VALUE

```
    c.setContents(CaveContents.WUMPUS);
```

since an enumerated type has a fixed number of values, any invalid input would be caught by the compiler

CaveMaze

the CaveMaze class reads in & stores a maze of caves

- provided version uses an ArrayList (but could have used an array)
- currently, only allows for moving between caves
- you must add functionality

```
20
 0  1  4  9 The Fountainhead
 1  0  2  5 The Rumpus Room
 2  1  3  6 Buford's Folly
 3  2  4  7 The Hall of Kings
 4  0  3 14 The Silver Mirror
 5  1  9 11 The Gallimaufry
 6  2  7 12 The Den of Iniquity
 7  3  6  8 The Findledelve
 8  7  3 13 The Page of the Deniers
 9  0  5 10 The Final Tally
10  9 11 14 Ess four
11  5 10 12 The Trillion
12  6 11 13 The Scrofula
13  8 12 18 Ephemeron
14  4 10 15 Shelob's Lair
15 15 16 19 The Lost Caverns of the Wyrm
16 15 17 19 The Lost Caverns of the Wyrm
17 16 17 18 The Lost Caverns of the Wyrm
18 13 17 19 The Lost Caverns of the Wyrm
19 15 16 17 The Lost Caverns of the Wyrm
```

```
public class CaveMaze {
  private ArrayList<Cave> caves;
  private Cave currentCave;

  public CaveMaze(String filename) throws java.io.FileNotFoundException {
      Scanner infile = new Scanner(new File(filename));

      int numCaves = infile.nextInt();
      this.caves = new ArrayList<Cave>();
      for (int i = 0; i < numCaves; i++) {
          this.caves.add(null);
      }

      for (int i = 0; i < numCaves; i++) {
          int num1 = infile.nextInt();
          int num2 = infile.nextInt();
          int num3 = infile.nextInt();
          int num4 = infile.nextInt();
          String name = infile.nextLine().trim();
          this.caves.set(num1, new Cave(name, num1, num2, num3, num4));
      }

      this.currentCave = this.caves.get(0);
      this.currentCave.markAsVisited();
  }
  . . .
```

CaveMaze (cont.)

currently,

- can move between caves
- only see the names of caves you have already visited

- you must add the full functionality of the game (incl. adding & reacting to dangers, winning/losing)

```
  . . .

  public String move(int tunnel) {
     this.currentCave =
          this.caves.get(this.currentCave.getAdjNumber(tunnel));
     this.currentCave.markAsVisited();

     return "OK";
  }

  public String showLocation() {
    String message = "You are currently in " +
                     this.currentCave.getCaveName();
    for (int i = 1; i <= 3; i++) {
        Cave adjCave = this.caves.get(this.currentCave.getAdjNumber(i));
        if (adjCave.hasBeenVisited()) {
            message += "\n    (" + i + ") " + adjCave.getCaveName();
        }
        else {
            message += "\n    (" + i + ") unknown";
        }
    }
    return message;
  }

  . . .
}
```

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`)

Inheritance

a closely related mechanism for polymorphic behavior is *inheritance*

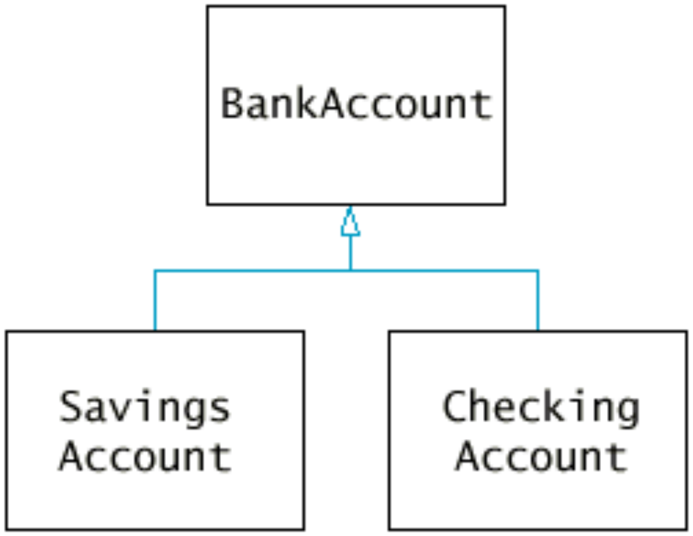
- one of the most powerful techniques of object-oriented programming
- allows for large-scale code reuse

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



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

```
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;
      }
  }
}
```

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 explain later)

SavingsAccount class

inheritance provides a better solution

- can define a SavingsAccount to be a special kind of BankAccount
  automatically inherit common features (balance, account #, deposit, withdraw)
- 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);
  }
}
```

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
```

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;
  }
}
```

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

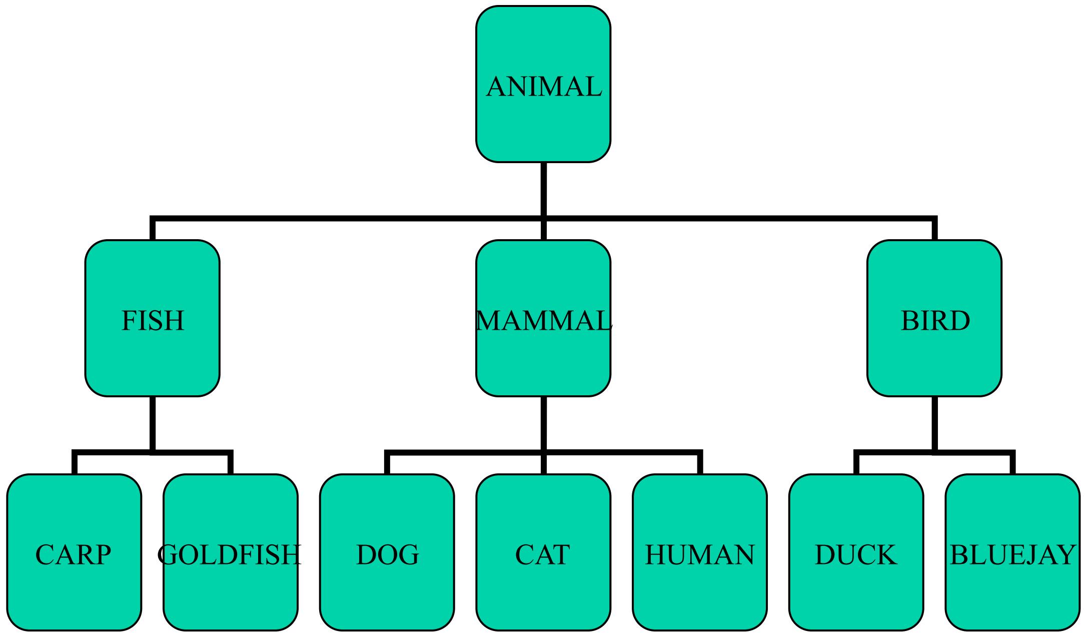
- could assign a SavingsAccount object to a variable of type BankAccount
- could pass a CheckingAccount object to a method with a BankAccount parameter

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



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

Example use

```java
import java.util.ArrayList;

public class AccountAdd {
  public static void main(String[] args) {
    SavingsAccount xmasFund = new SavingsAccount(2.67);
    xmasFund.deposit(250.0);

    SavingsAccount carMoney = new SavingsAccount(1.8);
    carMoney.deposit(100.0);

    CheckingAccount living = new CheckingAccount();
    living.deposit(400.0);
    living.withdraw(49.99);

    ArrayList<BankAccount> finances = new ArrayList<BankAccount>();
    finances.add(xmasFund);
    finances.add(carMoney);
    finances.add(living);

    addToAll(finances, 5.0);
    showAll(finances);
  }

  private static void addToAll(ArrayList<BankAccount> accounts, double amount) {
    for (int i = 0; i < accounts.size(); i++) {
      accounts.get(i).deposit(amount);
    }
  }

  private static void showAll(ArrayList<BankAccount> accounts) {
    for (int i = 0; i < accounts.size(); i++) {
      System.out.println(accounts.get(i).getAccountNumber() + ": $" +
                         accounts.get(i).getBalance());
    }
  }
}
```

note: in addToAll, the appropriate deposit method is called on each BankAccount (depending on whether it is really a SavingsAccount or CheckingAccount)

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?

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;
  }
}
```

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

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

public class RollGame {
  private ArrayList<ColoredDie> dice;
  private static final int NUM_DICE = 5;

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

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

  public int rollPoints() {
    int total = 0;
    for (int i = 0; i < NUM_DICE; i++) {
      int roll = this.dice.get(i).roll();
      if (this.dice.get(i).getColor() == DieColor.RED) {
        total += 2*roll;
      }
      else {
        total += roll;
      }
    }
    return total;
  }
}
```

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 RollGame {
  private ArrayList<Die> dice;
  private static final int NUM_DICE = 5;

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

    this.dice.add(new ColoredDie(6, DieColor.RED));
    for (int i = 1; i < RollGame.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 = this.dice.get(i).roll();
      total += roll;
      if (d instanceof ColoredDie) {
        ColoredDie cd = (ColoredDie)d;
        if (cd.getColor() == DieColor.RED) {
          total += roll;
        }
      }
    }
    return total;
  }
}
```

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);`