# CSC 222: Object-Oriented Programming

# Spring 2012

Inheritance & polymorphism

- derived class, parent class
- inheriting fields & methods, overriding fields and methods
- bank account example
- IS-A relationship, polymorphism
- super methods, super constructor
- colored dice example
- instanceof, downcasting



## Inheritance

inheritance is a mechanism for enhancing existing classes

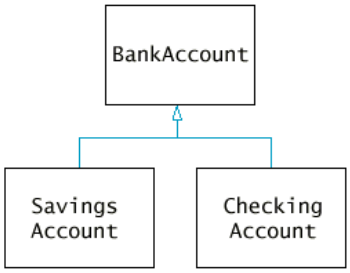
- one of the most powerful techniques of object-oriented programming
- allows for large-scale code reuse

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;
      }
  }
}
```

3

## 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)

4

# 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);
  }
}
```

5

# 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
```

6

## 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;
  }
}
```

7

## 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

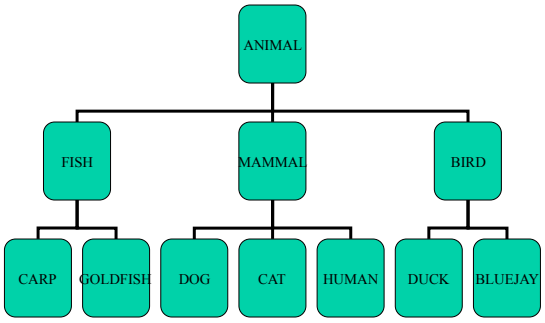
8

## 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




9

## 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

10

## Example use

```
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;
  }
}
```

13

## 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 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;
  }
}
```

14

# 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;
  }
}
```

15