Inheritance and Polymorphism

CS 180
Sunil Prabhakar
Department of Computer Science
Purdue University

Objectives

- Understand Inheritance
  - expressing inheritance: **extends**
  - visibility and inheritance: **protected**
  - overriding, **final**
  - constructors and inheritance: **super**
- Polymorphism
  - polymorphic messages
  - **instanceof** operator
  - abstract classes & methods: **abstract**
  -

PURDUE UNIVERSITY

Introduction

- Inheritance and polymorphism are key concepts of Object Oriented Programming.
- Inheritance facilitates the reuse of code.
- A subclass inherits members (data and methods) from all its ancestor classes.
- The subclass can add more functionality to the class or replace some functionality that it inherits.
- Polymorphism simplifies code by automatically using the appropriate method for a given object.
- Polymorphism also makes it easy to extend code.

Inheritance

Sample application

- Banking Example:
  - There are two types of accounts: checking and savings.
  - All accounts have a number, and an owner (with name, and a Social Security number), and balance.
  - There are different rules for interest and minimum balance for checking accounts and savings accounts.
- How should we model this application?
  - Two classes, one for each type of account?
  - Have to repeat code for common parts.
    - can lead to inconsistencies, harder to maintain.
  - Create three classes: Account; SavingsAccount, and CheckingAccount

Inheritance

- A superclass corresponds to a general class, and a subclass is a specialization of the superclass.
  - E.g. Account, Checking, Savings.
- Behavior and data common to the subclasses is often available in the superclass.
  - E.g. Account number, owner name, data opened.
- Each subclass provides behavior and data that is relevant only to the subclass.
  - E.g. Minimum balance for checking a/c, interest rate and computation for savings account.
- The common behavior is implemented once in the superclass and automatically inherited by the subclasses.

Inheritance

- In order to inherit the data and code from a class , we have to create a subclass of that class using the **extends** keyword.

```
public class SavingsAccount extends Account {
```

- SavingsAccount will inherit the data members and methods of Account.
- SavingsAccount is a sub (child, or derived) class; Account is a super (parent or base) class.
  - A parent (of a parent ...) is an ancestor class.
  - A child (of a child ...) is a descendant class.

The Account class

```
class Account {
   protected   String      ownerName;
   protected   int         socialSecNum;
   protected   float       balance;
 public Account() {
        this(“Unknown”, 0, 0.0);
    }
 public Account(String name, int ssn) {
        this(name, ssn, 0.0);
    }
 public Account(String name, int ssn, float bal) {
        ownerName = name;
        socialSecNum = ssn;
        balance = bal;
    }
 public String getName( ) {
      return ownerName;
    }
 public String getSsn( ) {
      return socialSecNum;
    }
 public float getBalance() {
      return balance;
    }
 public void setName(String newName) {
      ownerName = newName;
    }
 public void accrueInterest() {
      System.out.println(“No interest”);
    }
public void deposit(float amount) {
      balance += amount;
    }
}
```

PURDUE UNIVERSITY

Savings Account

```
class SavingsAccount extends Account{

    protected static final float MIN_BALANCE=100.0;
    protected static final float OVERDRAW_LIMIT=-1000.0;
    protected static final float INT_RATE=5.0;

    public void accrueInterest() {
        balance *= 1 + INT_RATE/100.0;
    }

    public void withdraw(float amount) {
        float temp;
        temp = balance - amount;
        if (temp >= OVERDRAW_LIMIT)
            balance = temp;
        else
            System.out.println(“Insufficient funds”);
    }
}
```

PURDUE UNIVERSITY

Checking Account

```
class CheckingAccount extends Account{

    protected static final float MIN_INT_BALANCE=100.0;
    protected static final float INT_RATE=1.0;

    public void accrueInterest() {
        if (balance > MIN_INT_BALANCE)
           balance *= 1 + INT_RATE/100.0;
    }

    public void withdraw(float amount) {
        float temp;
        temp = balance - amount;
        if (temp >= 0)
           balance = temp;
        else
           System.out.println("Insufficient funds");
    }
}
```

Visibility

PURDUE UNIVERSITY

The visibility modifiers

- **public** data members and methods are accessible to everyone.
- **private** data members and methods are accessible only to instances of the class.
- **protected** data members and methods are accessible only to instances of the class and descendant classes
- **protected** is similar to:
  - **public** for descendant classes
  - **private** for any other class

Visibility (unrelated class)

```
class Sup {
   public int a;
   protected int b;
   private int c;
}
```

```
class Sub extends Sup {
   public int d;
   protected int e;
   private int f;
}
```

```
class Test {
   Sup sup = new Sup();
   Sub sub = new Sub();

   sup.a = 5;
   sup.b = 5;   // struck out
   sup.c = 5;   // struck out

   sub.a = 5;
   sub.b = 5;   // struck out
   sub.c = 5;   // struck out
   sub.d = 5;
   sub.e = 5;   // struck out
   sub.f = 5;   // struck out

}
```

From an unrelated class, only public members are visible.

PURDUE UNIVERSITY

Visibility (related class)

```
class Sup {
   public int a;
   protected int b;
   private int c;
}
```

```
class Sub extends Sup {
   public int d;
   protected int e;
   private int f;

   public void methodA(){
      a=5;
      b=5;
      c=5;
      d=5;
      e=5;
      f=5;
   }
}
```

From a descendant class, only private members of ancestors are hidden.

PURDUE UNIVERSITY

Visibility (static members)

```
class Sup {
   public static int a;
   protected static int b;
   private static int c;
}
```

```
class Sub extends Sup {
   public static int d;
   protected static int e;
   private static int f;
}
```

```
class Test {
   Sup sup = new Sup();
   Sub sub = new Sub();

   sup.a = 5;
   sup.b = 5;
   sup.c = 5;

   sub.a = 5;
   sub.b = 5;
   sub.c = 5;
   sub.d = 5;
   sub.e = 5;
   sub.f = 5;

}
```

(The lines sup.b, sup.c, sub.b, sub.c, sub.e and sub.f are struck through on the slide.)

Same rules for class (static) members.

Visibility (static members)

```java
class Sup {
   public static int a;
   protected static int b;
   private static int c;
}
```

```java
class Sub extends Sup {
   public int d;
   protected int e;
   private int f;

   public void methodA(){
      a=5;
      b=5;
      c=5;
      d=5;
      e=5;
      f=5;
   }
}
```

Same rules for class (static) members.

PURDUE UNIVERSITY

Visibility (across instances)

```java
class Sup {
    public int a;
    protected int b;
    private int c;
}
```

```java
class Sub extends Sup {
    public int d;
    protected int e;
    private int f;

    public void methodA(Sub s){
        s.a=5;
        s.b=5;
        s.c=5;   // struck out
        s.d=5;
        s.e=5;
        s.f=5;
    }
}
```

An instance method has the same access to data members of any object of that class.

PURDUE UNIVERSITY

Overriding

Overriding

- All non-private members of a class are inherited by derived classes
  - This includes instance and class members
- A derived class may however, **override** an inherited method
  - Data members can also be overridden but should be avoided since it only creates confusion.
- To override a method, the derived class simply defines a method with the same signature (same name, number and types of parameters)
  - An overridden method cannot change the return type!
- A subclass may also **overload** any method (inherited or otherwise) by using the same name, but different signature.

PURDUE UNIVERSITY

The Account class

```
class Account {
   protected   String      ownerName;
   protected   int         socialSecNum;
   protected   float       balance;
 public Account() {
        this(“Unknown”, 0, 0.0);
    }
 public Account(String name, int ssn) {
        this(name, ssn, 0.0);
    }
 public Account(String name, int ssn, float bal) {
        ownerName = name;
        socialSecNum = ssn;
        balance = bal;
    }
 public String getName( ) {
      return ownerName;
    }
 public String getSsn( ) {
      return socialSecNum;
    }
 public float getBalance() {
      return balance;
    }
 public void setName(String newName) {
      ownerName = newName;
    }
 public void accrueInterest() {
      System.out.println(“No interest”);
    }
public void deposit(float amount) {
      balance += amount;
    }
}
```

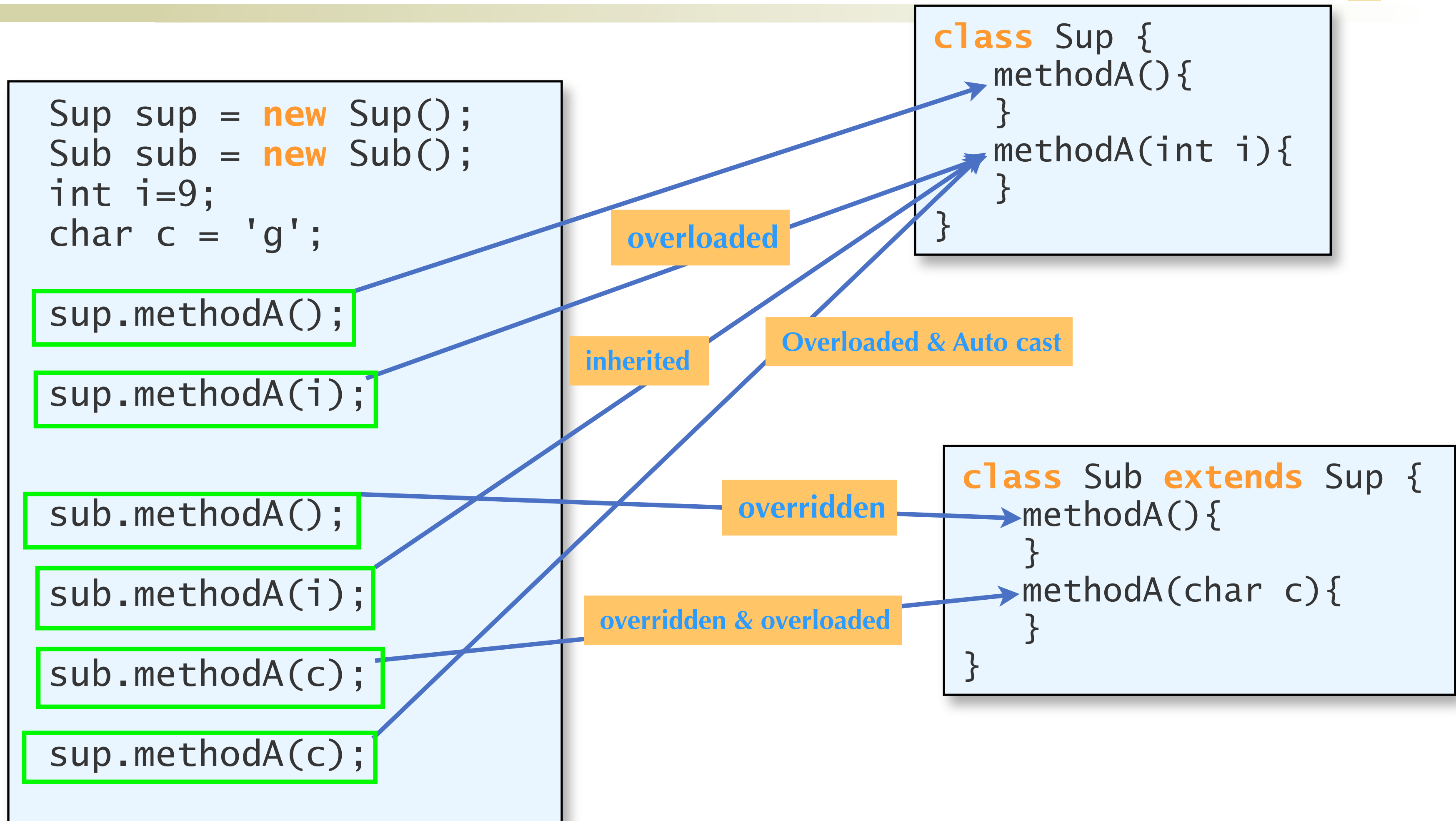
PURDUE
UNIVERSITY

Savings Account

```
class SavingsAccount extends Account{

    protected static final float MIN_BALANCE=100.0;
    protected static final float OVERDRAW_LIMIT=-1000.0;
    protected static final float INT_RATE=5.0;

    public void accrueInterest() {
        balance *= 1 + INT_RATE/100.0;
    }

    public void withdraw(float amount) {
        float temp;
        temp = balance - amount;
         if (temp >= OVERDRAW_LIMIT)
           balance = temp;
         else
           System.out.println("Insufficient funds");
    }
}
```

PURDUE UNIVERSITY

Overriding and overloading



PURDUE UNIVERSITY

Limiting inheritance and overriding

- If a class is declared to be final, then no other classes can derive from it.

```
public final class ClassA
```

- If a method is declared to be final, then no derived class can override this method.
  - A final method can be overloaded in a derived class though.

```
public final void methodA()
```

PURDUE UNIVERSITY

The Object class

- If a class does not (explicitly) extend another class then it implicitly extends the Object class.
- This class is the parent of all classes.
- Methods:
  - equals(), toString(), clone(), finalize(), …
- Overriding some of these methods can be useful to add functionality
  - equals() -- actually test meaningful equality

PURDUE
UNIVERSITY

Inheritance and Constructors

- Constructors of a class are *not* inherited by its descendants.
- In each constructor of a derived class, we must make a call to the constructor of the base class by calling: `super();`
  - This must be the first statement in the constructor.
- If this statement is not present, the compiler automatically adds it as the first statement.
- You may optionally call some other constructor of the base class, e.g.: `super( "some string" );`
- As always, if we do not define any constructor, we get a default constructor.

Constructors and inheritance

- For all classes, calls to the constructors are chained all the way back to the constructor for the `Object` class.
- Recall that it is also possible to call another constructor of the same class using the **`this`** keyword.
- However, this must also be the first statement of the constructor!
- A constructor cannot call another constructor of the same class and the base class.

Constructors

```
class Sup(){
    public Sup(){
    }
    public Sup(int i){
    }
}
```

```
class Sup(){
    public Sup(){
        super();
    }
    public Sup(int i){
        super();
    }
}
```

```
Sup sup1, sup2;
Sub sub1, sub2, sub3;

sup1 = new Sup();
sup2 = new Sup(7);

sub1 = new Sub();
sub2 = new Sub('y');
sub3 = new Sub(5);
```

```
class Sub extends Sup{
    public Sub(){
        this('x');
    }
    public Sub(char c){
        …
    }
    public Sub(int i){
        super(i);
        …
    }
}
```

```
class Sub extends Sup{
    public Sub(){
        this('x');
    }
    public Sub(char c){
        super();
        …
    }
    public Sub(int i){
        super(i);
        …
    }
}
```

Added by the compiler

PURDUE UNIVERSITY

Example: Account

```
class Account {
    protected    String      ownerName;
    protected    int         socialSecNum;
    protected    float       balance;




  public Account(String name, int ssn) {
         this(name, ssn, 0.0);
     }
  public Account(String name, int ssn, float bal) {
         ownerName = name;
         socialSecNum = ssn;
         balance = bal;
     }
   . . .

}
```

Savings Account

```
class SavingsAccount extends Account{
   protected static final float MIN_BALANCE=100.0;
   protected static final float OVERDRAW_LIMIT=-1000.0;
   protected static final float INT_RATE=5.0;

   public SavingsAccount (String name, int ssn) {
     this(name, ssn, 0.0);
   }
   public SavingsAccount (String name, int ssn, float bal) {
     super(name, ssn, bal);
     if (bal < MIN_BALANCE)
        System.out.println(“Insufficient starting funds”);
   }

   . . .
}
```

PURDUE UNIVERSITY

Checking Account

```
class CheckingAccount extends Account{
   protected static final float MIN_INT_BALANCE=100.0;
   protected static final float INT_RATE=1.0;

   public CheckingAccount (String name, int ssn) {
     this(name, ssn, 0.0);
   }
   public CheckingAccount (String name, int ssn, float bal) {
     super(name, ssn, bal);
     if (bal < 0)
        System.out.println("Insufficient starting funds");

   }


   . . .
}
```

PURDUE UNIVERSITY

Super keyword

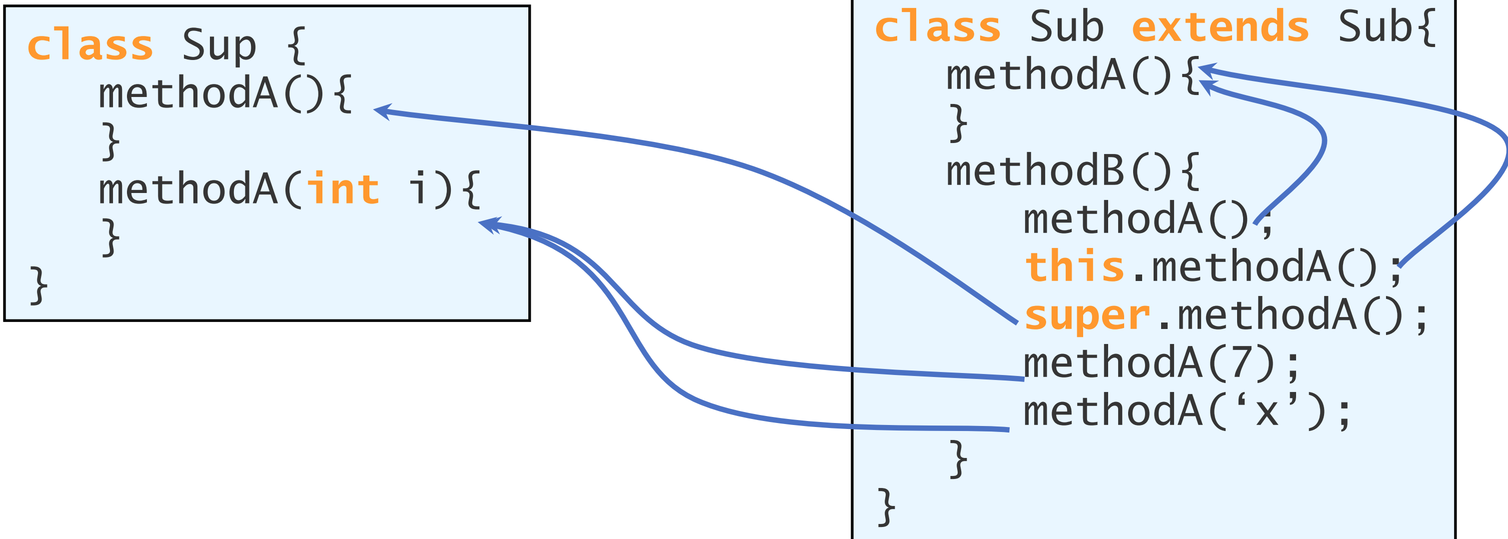
- The super keyword is a call to the constructor of the parent class.
- It can also be used to call a method of the parent class:

```
super.methodA();
```

- This can be useful to call an overridden method.
- Similarly, it can be used to access data members of the parent.

PURDUE UNIVERSITY

super keyword example.



Quiz

- Which method is executed for the method call?

```
Sup sup = new Sup();
Sub sub = new Sub();

sub.methodA('s');
```

A

B

```
class Sup {
   methodA(){
   }
   methodA(int i){
   }
}
```

C

D

```
class Sub extends Sup{
   methodA(){
   }
   methodA(String s){
   }
}
```

Polymorphism

Polymorphism

- Polymorphism allows a variable of a given class to refer to objects from any of its descendant classes
- For example, if Elephant and Tiger are descendant classes of Mammal, then we can:

```
Mammal someMammal;

someMammal = new Elephant();
. . .
someMammal = new Tiger();
```

Bank Account Collection

- Polymorphism naturally allows us to manage all accounts using a single collection:

```
Account localAccounts = new Account[100];
. . .
localAccounts[0] = new SavingsAccount("Jane", 77788777, 1000);
localAccounts[1] = new CheckingAccount("John", 32432523, 100);
localAccounts[2] = new SavingsAccount("Kim", 78687655, 2000);
. . .
```

Polymorphic method

- Polymorphism also makes it easy to execute the correct method.
- E.g, to compute the interest for all accounts:

```
for (int i = 0; i < 100; i++) {
    localAccounts[i].accrueInterest();
}
```

- If localAccounts[i] refers to a SavingsAccount object, then the accrueInterest() method of the SavingsAccount class is executed.
- If localAccounts[i] refers to a CheckingAccount object, then the accrueInterest() method of the CheckingAccount class is executed.

Dynamic Binding

- At compile time, it is not known which version of a polymorphic method will get executed
  - This is determined at run-time depending upon the class of the object
- This is called dynamic (late) binding
- Each object of a subclass is also an object of the superclass. But not vice versa!
- Do not confuse dynamic binding with overloaded methods.

Object Type

- Consider the inheritance hierarchy:

  Object ← Mammal ← Bear
- An object of class Bear is also an object of classes Mammal and Object.
- Thus we can use objects of class Bear wherever we can use objects of class Mammal.
- The reverse is not true.
- A reference of type Mammal can refer to an object of type Bear. However if we want to access the functionality of Bear on that object, we have to type cast to type Bear before doing that.

Polymorphism benefits

- Consider a student class which requires the student to have an account.
- Can use polymorphism to easily achieve this.
  - e.g., Account acct;
- Account can be the type for method parameters and also return types.
- Examples

PURDUE
UNIVERSITY

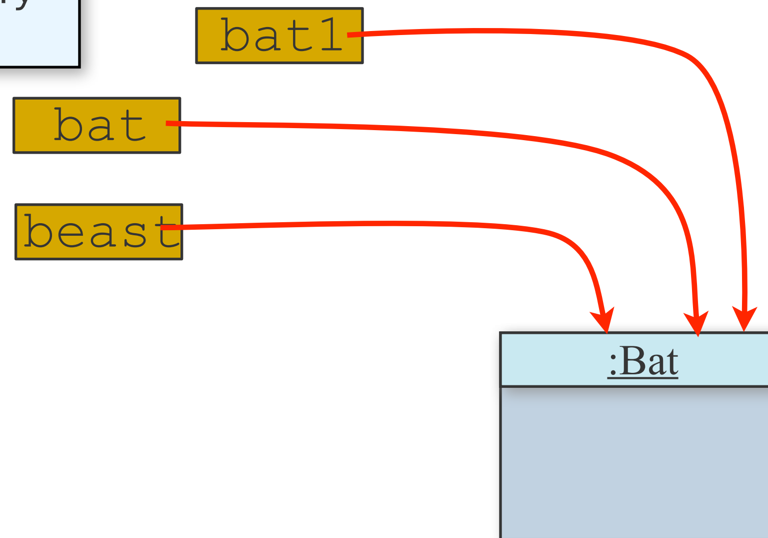
Polymorphism example

```
Bat bat = new Bat();
Animal beast;

beast = bat;

beast.eat();

((Bat) beast).fly();

Bat bat1 = (Bat) beast;
bat1.eat
();
bat1.fly
();
```

```
class Animal {
    eat() { ... }
}
```

Note: `beast.fly()` will not compile.

```
class Bat extends Animal {
    eat() { ... }

    fly() { ... }
}
```



Casting to Bat will work, but a runtime exception (ClassCastException) will be thrown if the object is not really a Bat object.

PURDUE UNIVERSITY

Example

```
Sub sub = new Sub();
Sup sup;

sup = sub;

sup.methodA();
((Sub)sup).methodA();
sub = (Sub)sup;
sub.methodA();
sub.methodA("test");
```

```
class Sup {
    methodA() { ... }

    methodA(String s) { ... }
}
```

```
class Sub extends Sup {
    methodA(int i) { ... }

    methodA() { ... }
}
```

PURDUE UNIVERSITY

The instanceof Operator

- The instanceof operator can help us discover the class of an object at runtime.
- The following code counts the number of Savings accounts.

```
new savingsAccCount = 0;
for (int i = 0; i < numActs; i++) {
    if ( localAccounts[i] instanceof SavingsAccount ) {
        savingsAccCount ++;
    }
}
```

PURDUE UNIVERSITY

Abstract Classes & Methods

PURDUE
UNIVERSITY

Abstract Superclasses and Methods

- Super classes are useful for grouping together common data and code.
- In some cases, we can have objects of a superclass.
  - e.g., Account -- generic type of account.
- In other cases, superclass objects are not needed.
  - e.g., Mammal -- all objects must have some more details (Dog, Cat, ...).
  - to disallow object of a class, we can make it abstract.

PURDUE UNIVERSITY

Abstract class

- A class is an abstract class if
  - it has the abstract modifier,
  - one or more of its methods have the abstract modifier (and no body), or
  - it inherits an abstract method for which it does not provide an implementation (body).

```
public abstract class Mammal {
   ...
}
```

```
public class Polygon {
   public abstract float computeArea();
}
```

- No instances of an abstract class can be created.
- private and static methods cannot be abstract methods.

Abstract class example

```
abstract class Account {
   protected   String      ownerName;
   protected   int         socialSecNum;
   protected   float       balance;


 public Account(String name, int ssn) {
       this(name, ssn, 0.0);
    }
 public Account(String name, int ssn, float bal) {
       ownerName = name;
       socialSecNum = ssn;
       balance = bal;
    }
 public String getName( ) {
      return ownerName;
    }
 public String getSsn( ) {
      return socialSecNum;
    }
 public float getBalance() {
      return balance;
    }
 public void setName(String newName) {
      ownerName = newName;
    }
 public abstract void accrueInterest();
 public abstract void withdraw(float amount);
 public void deposit(float amount) {
      balance += amount;
    }
}
```

```
class SavingsAccount extends Account{
   protected static final float
        MIN_BALANCE=100.0;
   protected static final float
        OVERDRAW_LIMIT=-1000.0;
   protected static final float
        INT_RATE=5.0;
   public void accrueInterest() {. . .}
   public void withdraw(float amount) {. . . }
}
```

```
class CheckingAccount extends Account{
   protected static final float
        MIN_INT_BALANCE=100.0;
   protected static final float
        INT_RATE=1.0;
   public void accrueInterest() {. . .  }
    public void withdraw(float amount) {. . .}
}
```

Abstract example (contd.)

- Non-private members of the abstract parent class are inherited.
- Note: constructors are not inherited! Default constructor calls super!

```
public class Test {

    public static void main(String[ ] args){
        Account a;
        SavingsAccount s;
        CheckingAccount c;

        a = new Student();
        s = new SavingsAccount("John", 78787887);
        s = new SavingsAccount();
        c = new CheckingAccount();

        System.out.println(s.getName());
        System.out.println(c.getName());
    }
}
```

Cannot instantiate abstract class.

Error: constructor not inherited!

Inherited from abstract parent class.

PURDUE UNIVERSITY

Interfaces

PURDUE
UNIVERSITY

Interfaces in Java

- Interfaces are Java's solution to multiple inheritance.
- In some languages (e.g., C++) a class can inherit from multiple classes
  - causes complications
- Java classes can only inherit from one other class
- Interfaces do not provide shared code, they only require certain behavior.

Recall: ActionListener interface

- Consider the addActionListener() method
- What is the type of its argument?
- Any object could be a listener
  - void addActionListener(Object listener)?
- E.g,. a Pet object or a Dog object could be listeners.
- We will call the actionPerformed() method on this listener, so must ensure that this method exists for the listener object.
- How?

Possible solution

- Declare the argument to be of type Object
  - Can't ensure that the method exists
- How about creating a subclass of Object, called ListenerObject with this method?
- Now, each listener object's class must extend ListenerObject
  - this could work for Pet
  - but not for Dog (since Dog extends Pet already)!

ActionListener Interface

- An interface is the ideal solution.
- The ActionListener interface defines the necessary method
- The data type of listener is ActionListener:
  - void addActionListener(ActionListener listener)
- Thus we must pass an object from a class that implements this interface
- An interface is not a class -- we cannot create instances of an interface.

The Java Interface

- An interface is like a class, except it has only constants and abstract methods.
  - An abstract method has only the method header, or prototype. No body.
- Interfaces specify behavior that must be supported by a class.
- A class implements an interface by providing the method body to the abstract methods stated in the interface.
- Any class can implement an interface.
- A class can implement multiple interfaces.

© Sunil Prabhakar Purdue University