Polymorphism

CS 18000
Sunil Prabhakar
Department of Computer Science
Purdue University

Objectives

- Polymorphism
  - polymorphic messages
  - **instanceof** operator
  - abstract classes & methods: **abstract**

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.

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.

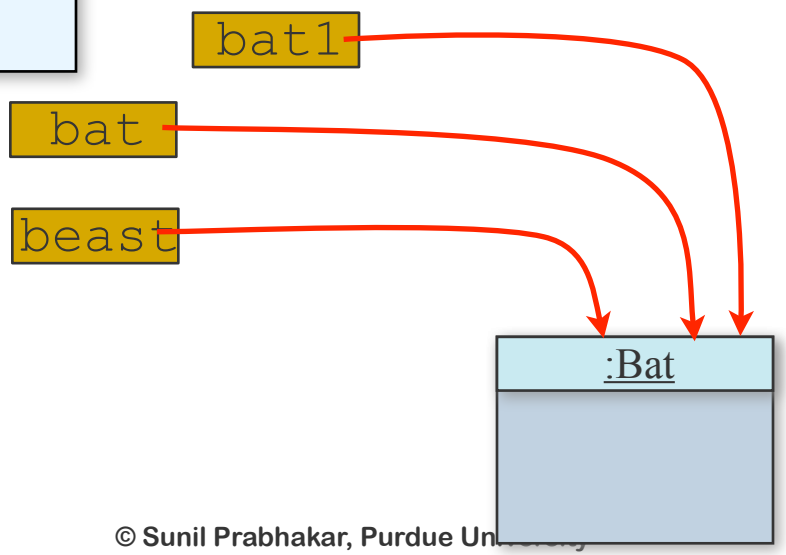
Polymorphism example

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

beast = bat;

beast.eat();

((Bat) beast).fly();

Bat bat1 = (Bat) beast;

bat1.eat();

bat1.fly();
```

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

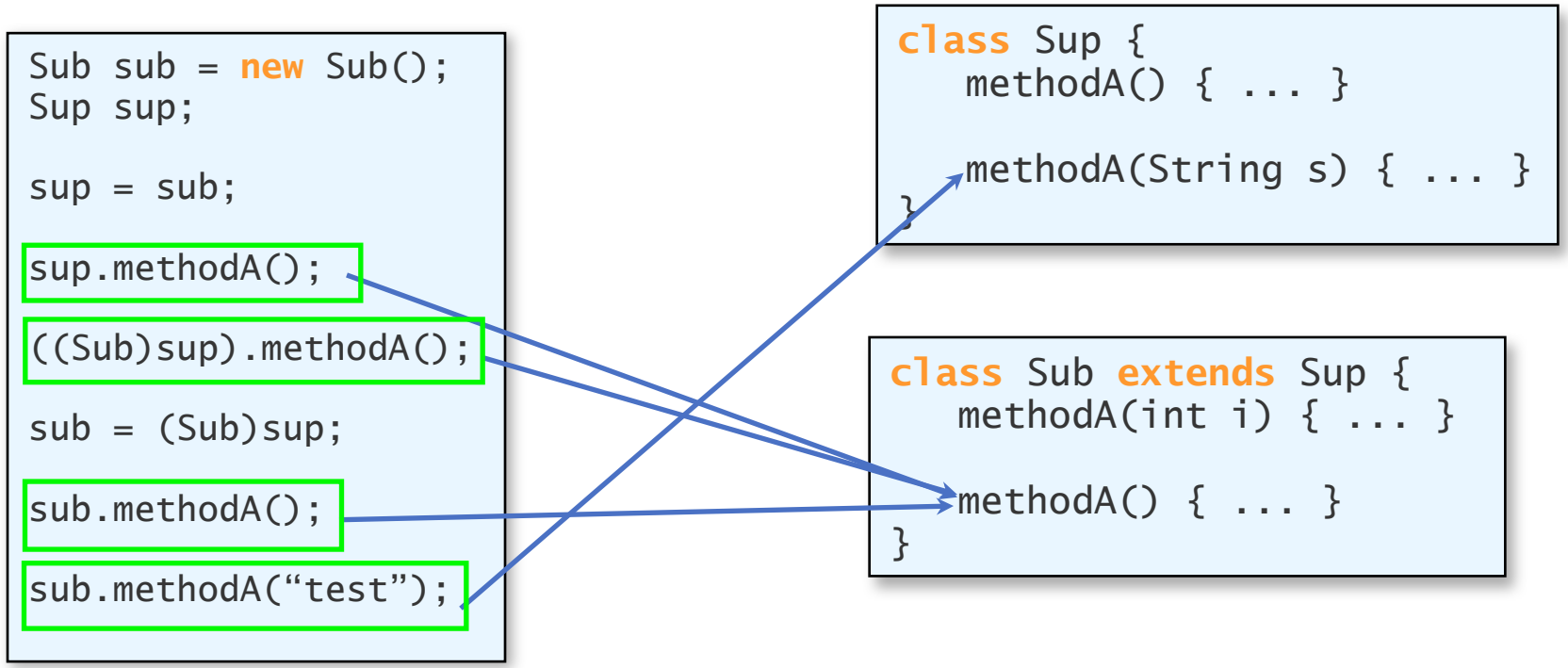
```
class Bat extends Animal {
   eat() { ... }

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

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



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

Example



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 ++;
    }
}
```

Abstract Classes & Methods

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.

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.

Interfaces

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.