Polymorphism

CS 18000
Sunil Prabhakar
Department of Computer Science
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.

Polymorphism

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

```
Account account;

account = new CheckingAccount();
. . .

account = new SavingsAccount();
```

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.

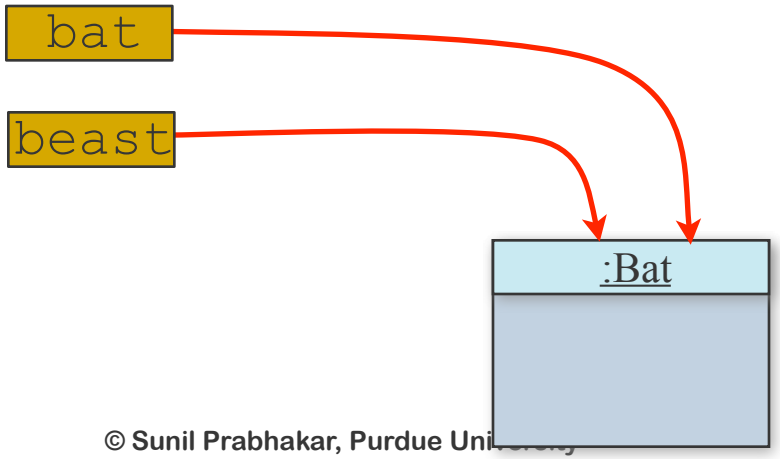
Polymorphism example

```
Bat bat = new Bat();

Animal beast;

beast = bat;

beast.eat();

((Bat) beast).fly();

Bat bat1 = (Bat) beast;

bat1.eat();

bat1.fly();
```

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

Note: beast.fly()
will not compile.

```
public 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.

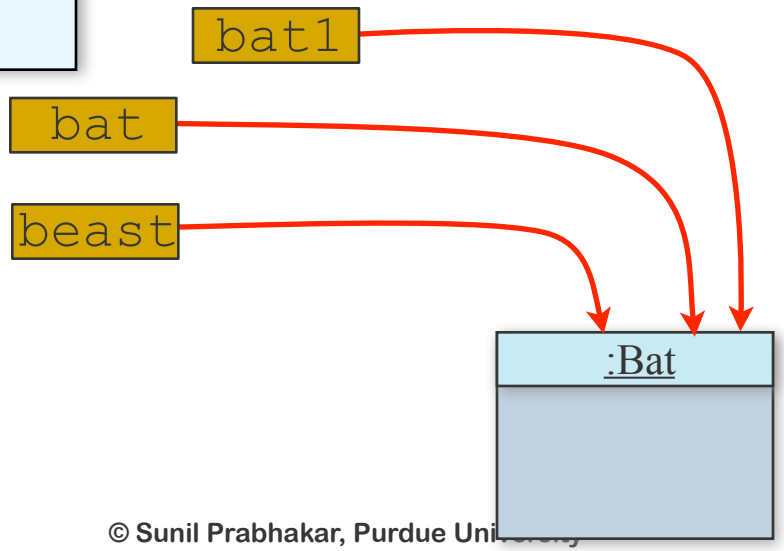
Polymorphism example

```
Bat bat = new Bat();

Animal beast;

beast = bat;

beast.eat();

((Bat) beast).fly();

Bat bat1 = (Bat) beast;

bat1.eat();

bat1.fly();
```

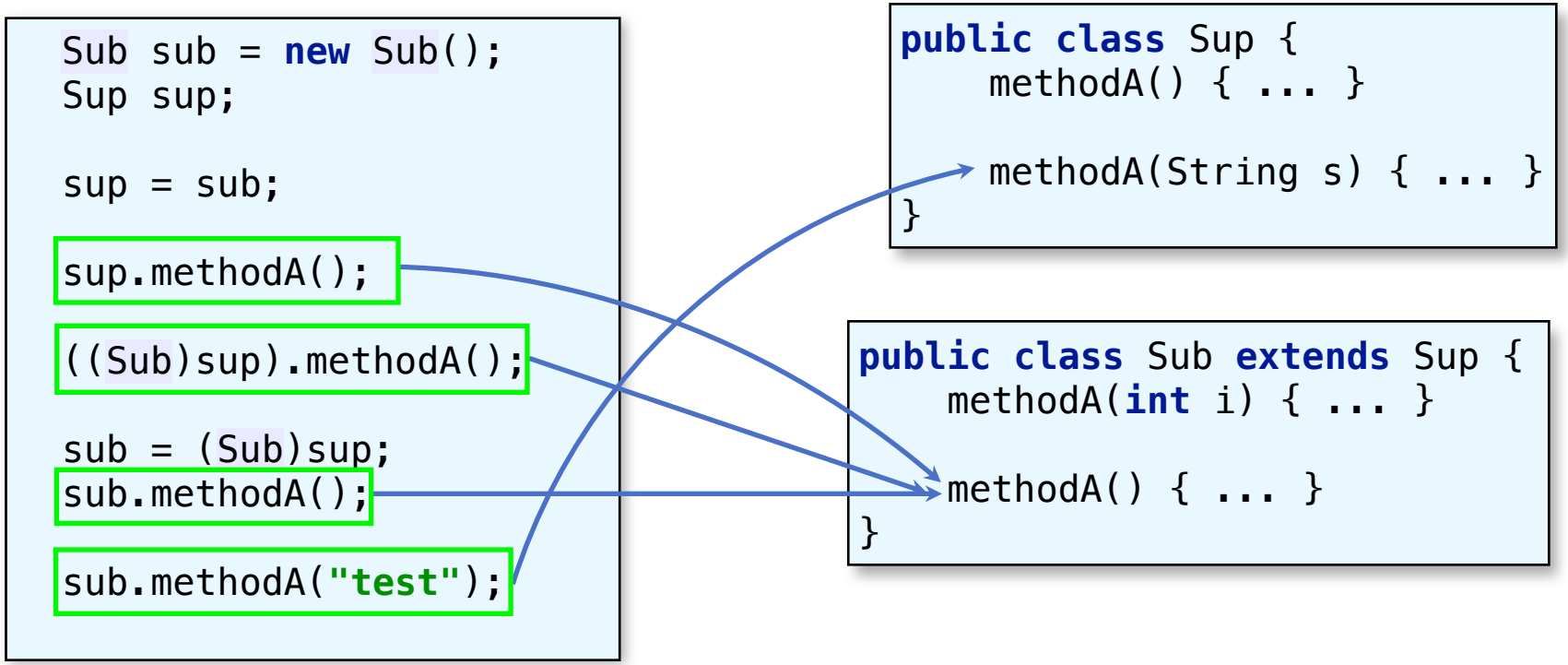
```
public class Animal {
    eat() { ... }
}
```

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

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

Note: beast.fly() will not compile.



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 ++;
    }
}
```

object reference

type name

The `instanceof` Operator

```
obj instanceof TypeName
```

- tests whether `obj` belongs to
  - a class names `TypeName` , or
  - a class that is a descendant of `TypeName`, or
  - a class that implements the `TypeName` interface, or
  - a class whose ancestor implements the `TypeName` interface

Finding the Class of an Object

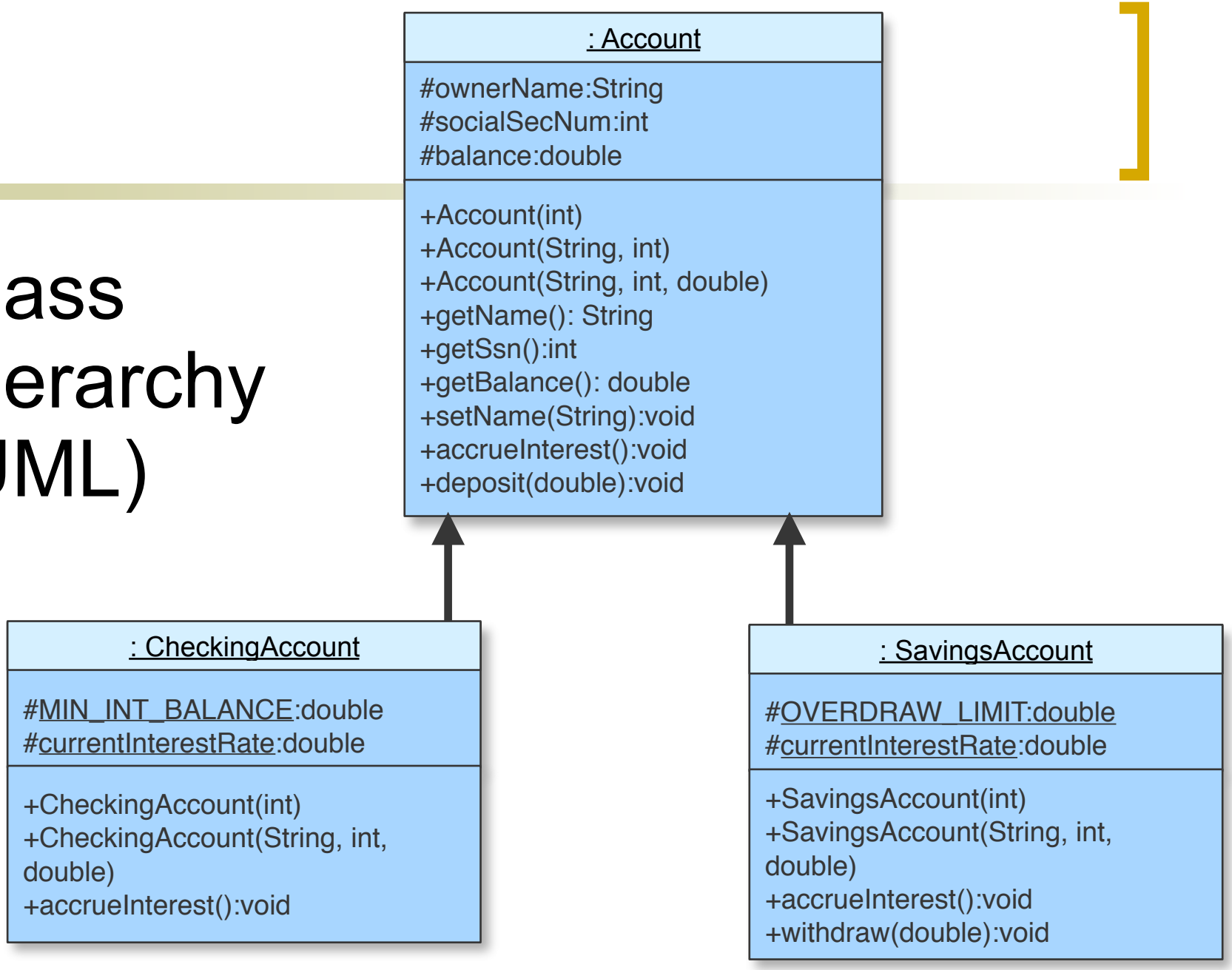
- The **`instanceof`** operator can test if an object belongs to a given class
  - but what if we just want to know the name of the class of an object?
  - we can use the `getClass()` method to get a Class object representing the class of the object
  - the `getName()` method of the Class class gives the name of the class.

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 meaningful, or are incomplete.
  - e.g., Mammal -- all objects must have some more details (Dog, Cat, ...).
  - to disallow object of a class, we can make it abstract.

Class Hierarchy (UML)



The Account class

```
public class Account {
    protected String ownerName;
    protected int socialSecNum;
    protected double balance;

    public Account(int ssn) { this("Unknown", ssn, 0.0);}

    public Account(String name, int ssn) { this(name, ssn, 0.0);}

    public Account(String name, int ssn, double bal) {
        ownerName = name;
        socialSecNum = ssn;
        balance = bal;
    }

    public String getName() { return ownerName; }

    public int getSsn() {return socialSecNum;}

    public double getBalance() {return balance;}

    public void setName(String newName) {ownerName = newName;}

    public void accrueInterest() {
        System.out.println("No interest");
    }

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

    public String toString(){
        return ("Account owner: " + ownerName + ", SSN:" +
           socialSecNum + ", balance:" + balance);
    }
}
```

Savings Account

```
public class SavingsAccount extends Account {
    protected static final double OVERDRAW_LIMIT = -1000.0;
    protected static double currentInterestRate = 5.0;

    public SavingsAccount(int ssn) { super(ssn); }

    public SavingsAccount(String name, int ssn, double bal){
        super(name, ssn, bal);
    }

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

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

Checking Account

```
public class CheckingAccount extends Account{
    protected static final double MIN_INT_BALANCE = 100.0;
    protected double currentInterestRate = 1.0;

    public CheckingAccount(int ssn) { super(ssn); }

    CheckingAccount(String name, int ssn, double bal){
        super(name, ssn, bal);
    }
    public void accrueInterest() {
        if (balance > MIN_INT_BALANCE)
            balance *= 1 + currentInterestRate /100.0;
    }

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

Abstract Classes

- In our earlier solution, we can create Account objects that have no withdraw method.
  - Is this acceptable?
- If not, how do we prevent it?
- We can make the Account class abstract — i.e., one that cannot be instantiated
- We do this by:
  - Adding the **abstract** modifier to the class
  - And optionally, adding one or more abstract methods to the class.

The abstract Account class

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

    public Account(int ssn) { this("Unknown", ssn, 0.0);}

    . . .

    }
}
```

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

    public Account(int ssn) { this("Unknown", ssn, 0.0);}

    . . .

    public abstract void withdraw(double 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 Account();
        s = new SavingsAccount("John", 7878);
        s = new SavingsAccount(7878);
        c = new CheckingAccount(3454);

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

Cannot instantiate abstract class.

Error: constructor not inherited!

Inherited from abstract parent class.

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

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

Abstract classes

- No objects of an abstract class can be created.
- They are still useful to define data and methods that are shared by descendant classes.
- A non-abstract (or concrete) descendant class can create objects.
  - These classes inherit the members defined in the base abstract class.

Abstract Methods

- Why add an abstract method?
- This ensures that any concrete descendant class must define this behavior
  - Usually represents essential behavior that must be defined before we can create objects
  - E.g., we must be able to withdraw from any account.
- A descendant of an abstract class must also be abstract unless it defines all the necessary abstract methods.