Inheritance

CS 18000
Sunil Prabhakar
Department of Computer Science
Purdue University

Objectives

- Understand Inheritance
  - expressing inheritance: `extends`
  - visibility and inheritance: `protected`
  - overriding, `final`
  - constructors and inheritance: `super`
- Understand Interfaces
  - defining an interface: `interface`
  - implementing an interface: `implements`

Inheritance

Introduction

- Inheritance is a key feature 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.

Sample Application: Banking

- There are two types of accounts: **`Checking`** and **`Savings`**.
  - All accounts have an owner (with name, and a Social Security Number), and a balance.
- There are different rules for *interest* and *minimum balance* for checking accounts and savings accounts.
  - **`Checking`**: no overdrawing allowed, interest paid only if balance exceeds a minimum amount
  - **`Savings`**: some overdrawing allowed, interest paid irrespective of the balance

What Classes Do We Need?

- Option1: Two classes: **`SavingsAccount`**, and **`CheckingAccount`**
  - Have to repeat code for common parts.
  - Can lead to inconsistencies, harder to maintain.
  - Have to repeat for each new type of account.
- Option 2: Three classes: **`Account`**; **`SavingsAccount`**, and **`CheckingAccount`**
  - The common data and behavior of every account are defined in Account
  - The other two **inherit** common data and behavior, and then define specific data and behavior

Inheritance

- A **superclass** corresponds to a general class, and a **subclass** is a specialization of the superclass.
  - E.g., Account, CheckingAccount, SavingsAccount.
- **Behavior** and **data common** to the subclasses is often available in the **superclass**.
  - E.g., Account balance, owner name, owner SSN, getOwnerName(), setBalance()
- Each **subclass adds / modifies behavior** and **data** 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

```
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);
    }
}
```

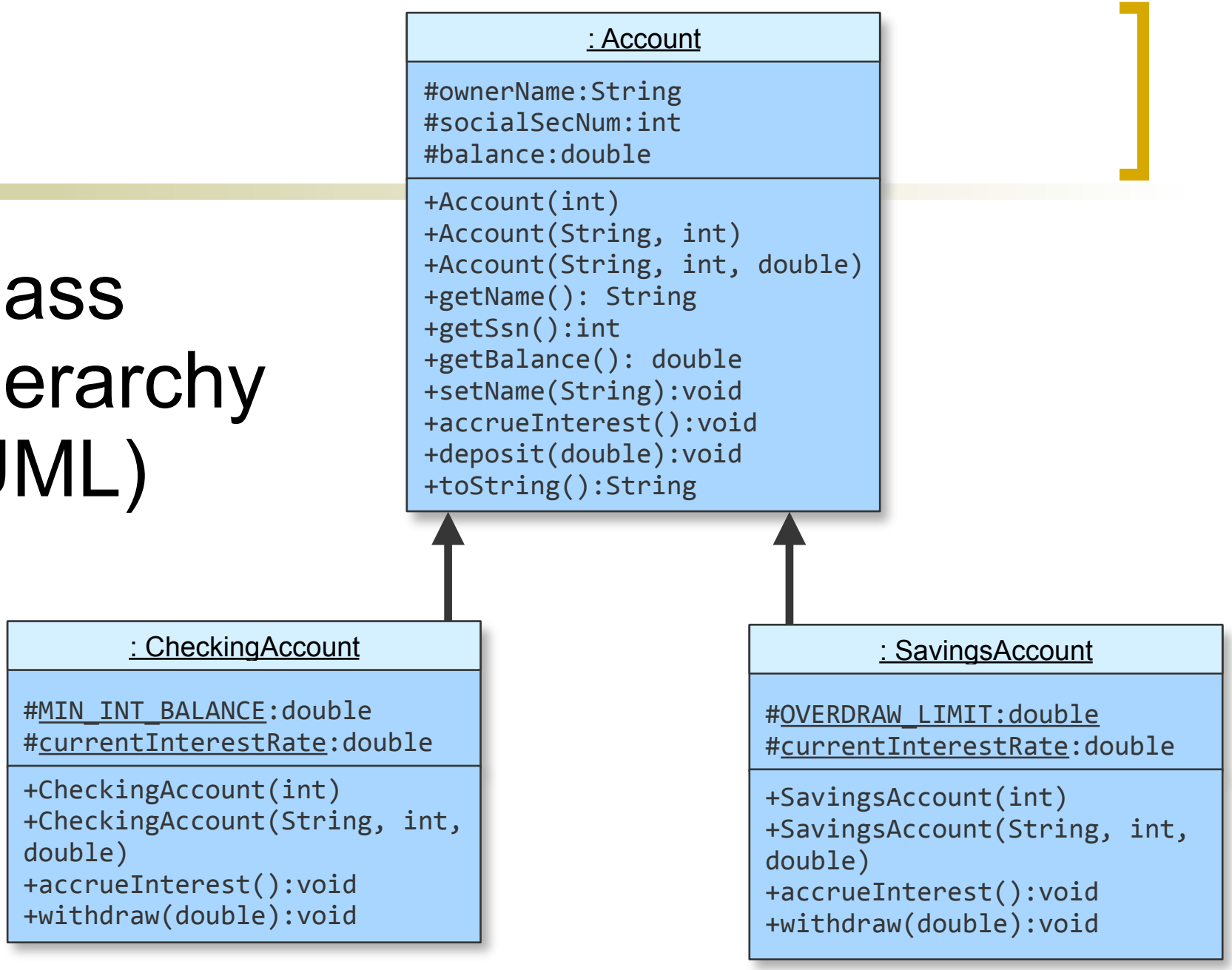
Savings Account

```
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");
    }
}
```

Class Hierarchy (UML)



Visibility

Visibility modifiers for a class

- A class can be declared to be `public` or have no modifier.
- If it is declared `public`, then it is accessible to all other classes
- If it has no modifier, then it is accessible only within its package
  - If no package is explicitly specified, then it is accessible in the default package (unnamed package)
    - cannot be imported (but accessible in same directory)

Visibility modifiers for class members

- `public` data members and methods are accessible to everyone.
- `private` data members and methods are accessible only within the class itself.
- `protected` data members and methods are accessible within the class, descendant classes, and classes in the same package
- `protected` is similar to:
  - `public` for descendant and package classes
  - `private` for any other class

Visibility (unrelated class)

```
package CS180;
public class Sup {
    public int a;
    protected int b;
    private int c;
}
```

```
package CS180;
public class Sub extends Sup {
    public int d;
    protected int e;
    private int f;
}
```

Note

```
import CS180.*;
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;

}
```

From an unrelated class, only public members are visible.

Visibility (same package)

```
package CS180;
public class Sup {
    public int a;
    protected int b;
    private int c;
}
```

```
package CS180;
public class Sub extends Sup {
    public int d;
    protected int e;
    private int f;
}
```

Note

```
package CS180;
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;

}
```

From a class in the same package, only private members are hidden.

Visibility (related classes)

```
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.

Visibility (static members)

```
package CS180;
public class Sup {
    public static int a;
    protected static int b;
    private static int c;
}
```

```
package CS180;
public class Sub extends Sup {
    public static int d;
    protected static int e;
    private static int f;
}
```

```
import CS180.*;
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;

}
```

Same rules for class (static) members.

Visibility (across instances)

```
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(Sub s){
        s.a=5;
        s.b=5;
        s.c=5;
        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.

Visibility (static members)

```
package CS180;
public class Sup {
    public static int a;
    protected static int b;
    private static int c;
}
```

```
package CS180;
public class Sub extends Sup {
    public static int d;
    protected static int e;
    private static int f;
}
```

```
package CS180;
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;

}
```

Same rules for class (static) members.

Visibility (static members)

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

Same rules for class (static) members.

Visibility for members

- In summary (from Java Tutorial):

Modifier	Class	Package	Subclass	World
`public`	Y	Y	Y	Y
`protected`	Y	Y	Y	N
`(none)`	Y	Y	N	N
`private`	Y	N	N	N

Overriding

Overriding

- All non-private members (data and methods) 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.
- A subclass may also overload any method (inherited or otherwise)

Overriding

- To override a method, the derived class defines a method with the same signature as the inherited method (same name, types of parameters - in order)
  - An overridden method cannot change the return type!

- Do not confuse this with *overloading*
  - Overloading of a method (inherited or not) is done by defining a method with a different signature than the method being overridden
  - The signature consists of the name and types (in order) of all the arguments.
  - The return type is not part of the signature and can be changed when overriding (not when overloading)

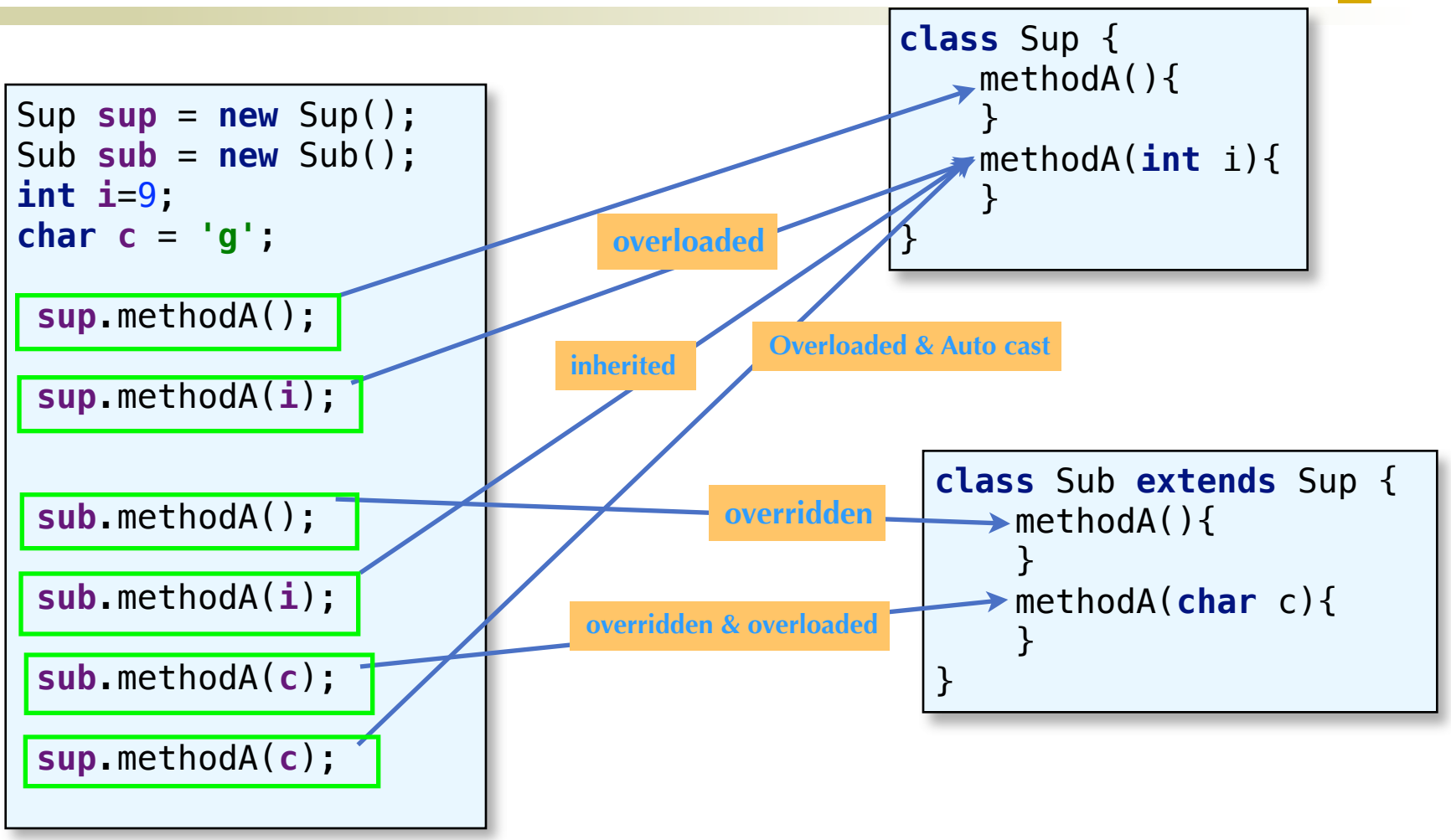
The Account class

```
public class Account {
    protected String ownerName;
    protected int socialSecNum;
    protected double 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, 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");
    }
}
```

Overriding and overloading



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()
```

Inheritance

- A class that does not explicitly extend any other class implicitly extends the **`Object`** class.
- Thus all classes are descendants of the **`Object`** class.
  - They all inherit methods `toString()`, `equals()`, `clone()`, `getClass()`, `wait()`, `notify()`, `notifyAll()`, `finalize()` and `hashcode()`
- If the only common ancestor of two classes is the **`Object`** class, they are said to be unrelated classes.

Overriding `Object` Class Methods

- Overriding some of the inherited methods can be useful to add functionality
  - `toString()`— called implicitly when an object is concatenated with a string. Modify to produce an informative string.
  - `equals()` — modify to test meaningful equality of objects depending upon the application
  - `finalize()` — called when the object is destroyed. Modify to do cleanup.
  - `hashcode()` — modify to reflect content

Inheritance and Constructors

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

Example: Account

```
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;
    }
    . . .
}
```

Savings Account

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

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

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

Checking Account

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

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

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

    }
    . . .
}
```

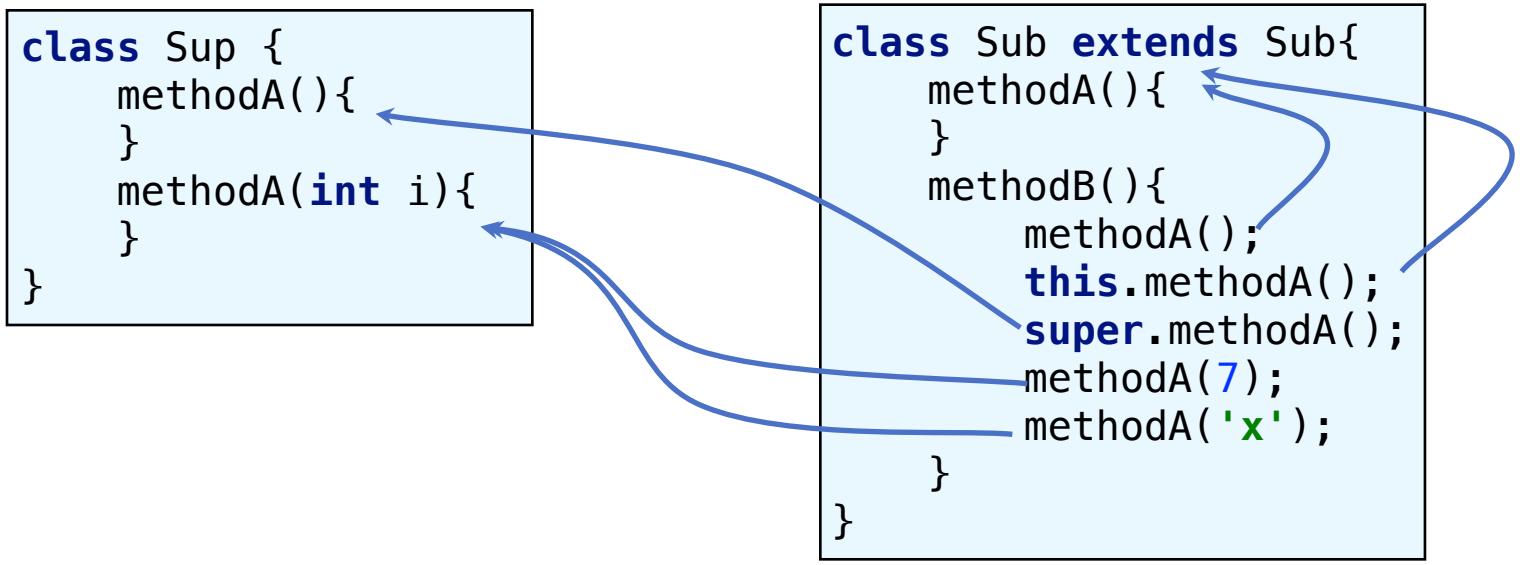
The `super` keyword

- 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 explicitly.

`super` keyword example.



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

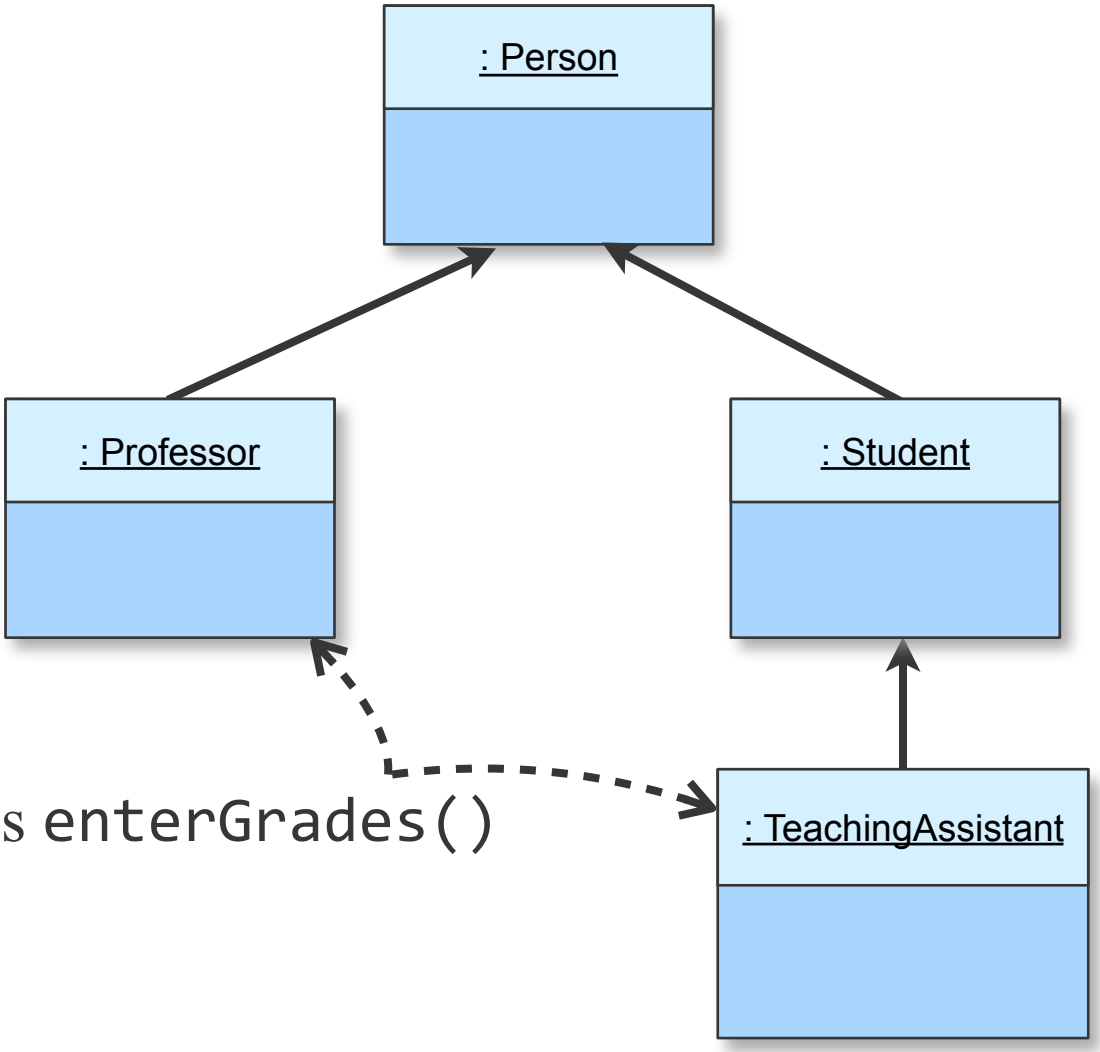
```
class Sub extends Sub{
    methodA(){
    }
    methodB(){
        methodA();
        this.methodA();
        super.methodA();
        methodA(7);
        methodA('x');
    }
}
```

Interfaces

Problem

- Consider the Person class, with descendant classes: Student, TA (subclass of Student), Professor
- Only TAs and Professors should (implement) enter grades.
- How should we set up our code so that we only try to call the enterGrades() method on objects that implement this method?
  - There is no class Graders of which TAs and Professors are descendants;
  - TAs and Professors are already part of our hierarchy

Class Hierarchy



Where does `enterGrades()` belong?

Solution

- We want to ensure a certain behavior (enterGrades) that does not naturally belong to a particular class
- Instead of a class, we use a Java Interface to declare this behavior (as method headers with no bodies)
- Each class that should have this behavior defines the body for these methods
  - The class declares that it is providing this behavior explicitly

The Java Interface

- An interface is like a class, except it has only abstract methods and constants.
  - 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.

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
- *Java 7 & earlier: Interfaces do not provide shared code, they only require certain behavior.*
  - *Interfaces may define constants*

Using the Instructor Interface

```
public interface Instructor {
    public void enterGrades();
}
```

```
public class Professor extends Person implements Instructor {

    . . .

    public void enterGrades(){
        //some code that enters student grades;
        System.out.println("Professor " + lastName + " has entered grades for " + course);
    }
}
```

```
public class TeachingAssistant extends Person implements Instructor {

    . . .

    public void enterGrades(){
        //some code that enters student grades;
        System.out.println("TA " + lastName + " has entered grades for " + course);
    }
}
```

Testing the Instructor Interface

```
public class InterfaceExample {
    public static void main(String[] args) {
        Professor prof;
        TeachingAssistant ta;
        Student[] students = new Student[2];

        prof = new Professor("Sunil", "Prabhakar", "sunil@purdue.edu",
                             "3144F", "CS18000");
        ta = new TeachingAssistant("Jane", "Java", 7, 4.00, prof,
                             "Gold", "CS18000", "JJ@p.e", "LWSN");
        students[0] = new Student("Alice", "Java", 7343, 4.00, null);
        students[1] = new Student("Jason", "Smith", 23423, 4.00, null);

        handleSession(ta, "Lab");
        handleSession(prof, "Lecture");
        // handleSession(students[0], "Lab"); //not allowed by the compiler.

    }

    private static void handleSession(Instructor instructor, String session){
        //do other stuff for session
        instructor.enterGrades();
    }
}
```

Recall: ActionListener interface

- Consider the addActionListener() method

```
void addActionListener(??? listener)
```

- What should be the type of its argument?
- Objects from many different classes could listen. E.g,. a TicTacToeView object or a WordProcessor object could be listeners.
- We need to call the `actionPerformed` method on each listener when something happens. Where is this method defined?
  - In TicTacToeView and WordProcessor
  - So what should the type of listener be above?

Solution: ActionListener Interface

- An interface is the solution to this problem
- The ActionListener interface defines the necessary method
- The data type of listener is `ActionListener`:

```
void addActionListener(ActionListener listener)
```

- Thus the listener object must belong to a class that implements this interface

Using the ActionListener Interface

```
package java.awt.event;
public interface ActionListener extends EventListener {
    public void actionListener(ActionEvent e);
}
```

```
public class TicTacToeView implements ActionListener {

    . . .

    public void actionPerformed(ActionEvent e){
        //code to handle actions

    }
}
```

```
public class TeachingAssistant extends Person implements Instructor {

    . . .

    public void actionPerformed(ActionEvent e){
        //code to handle actions

    }
}
```

Notes

- We cannot instantiate an interface. We can only have references of this type, but the objects referenced belong to classes that implement the interface
- Interfaces can define constants
- Java 8: can also define default code for the method
- An interface can also be a descendant of another interface