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
  -

Introduction

- Inheritance is a key concept 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.

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.

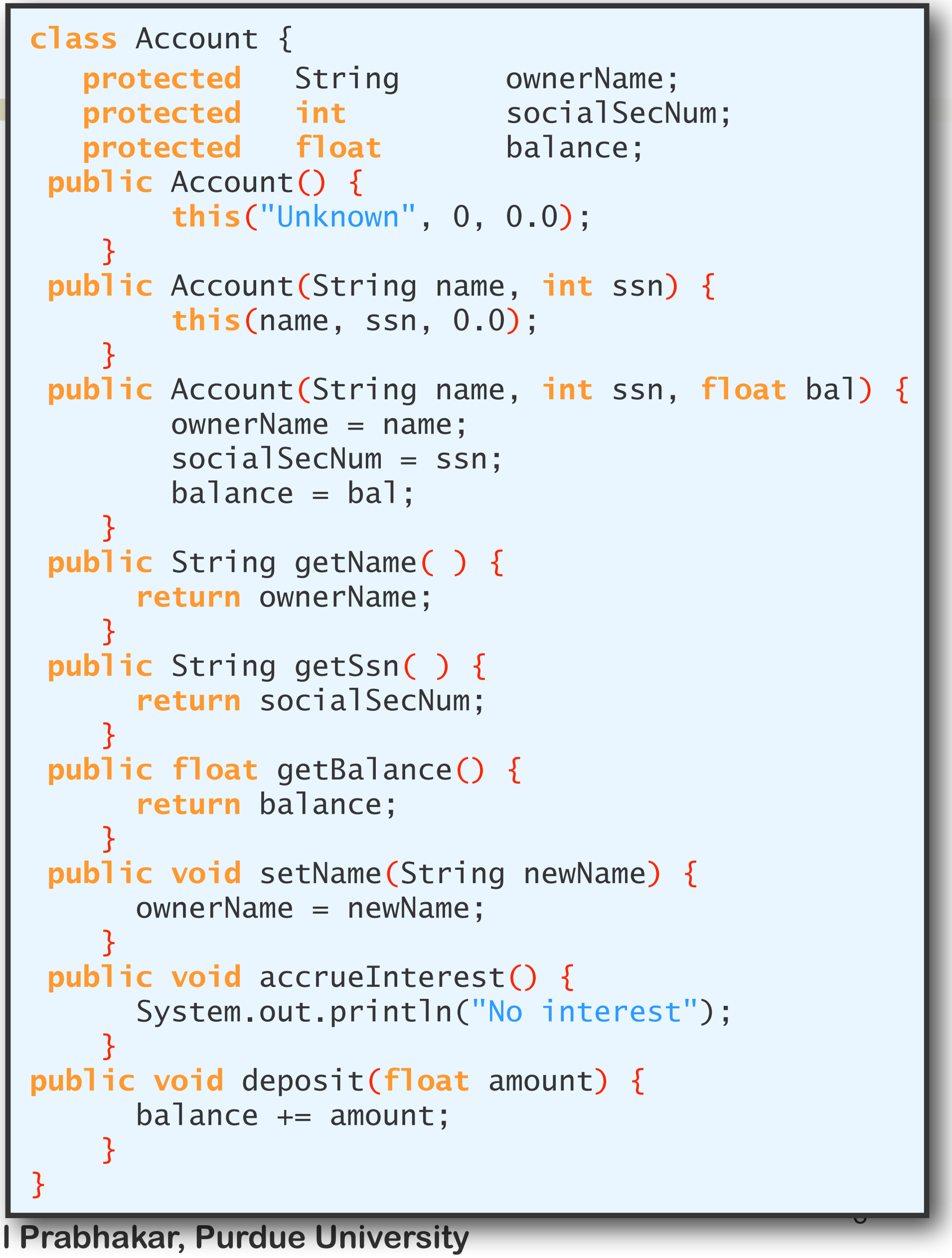
Inheritance

- 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;
    }
}
```

© Sunil Prabhakar, Purdue University

Savings Account

```java
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");
     }
}
```

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

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;~~
   ~~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 (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.

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;~~

}
```

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 (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.

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.

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;
    }
}
```

Savings Account

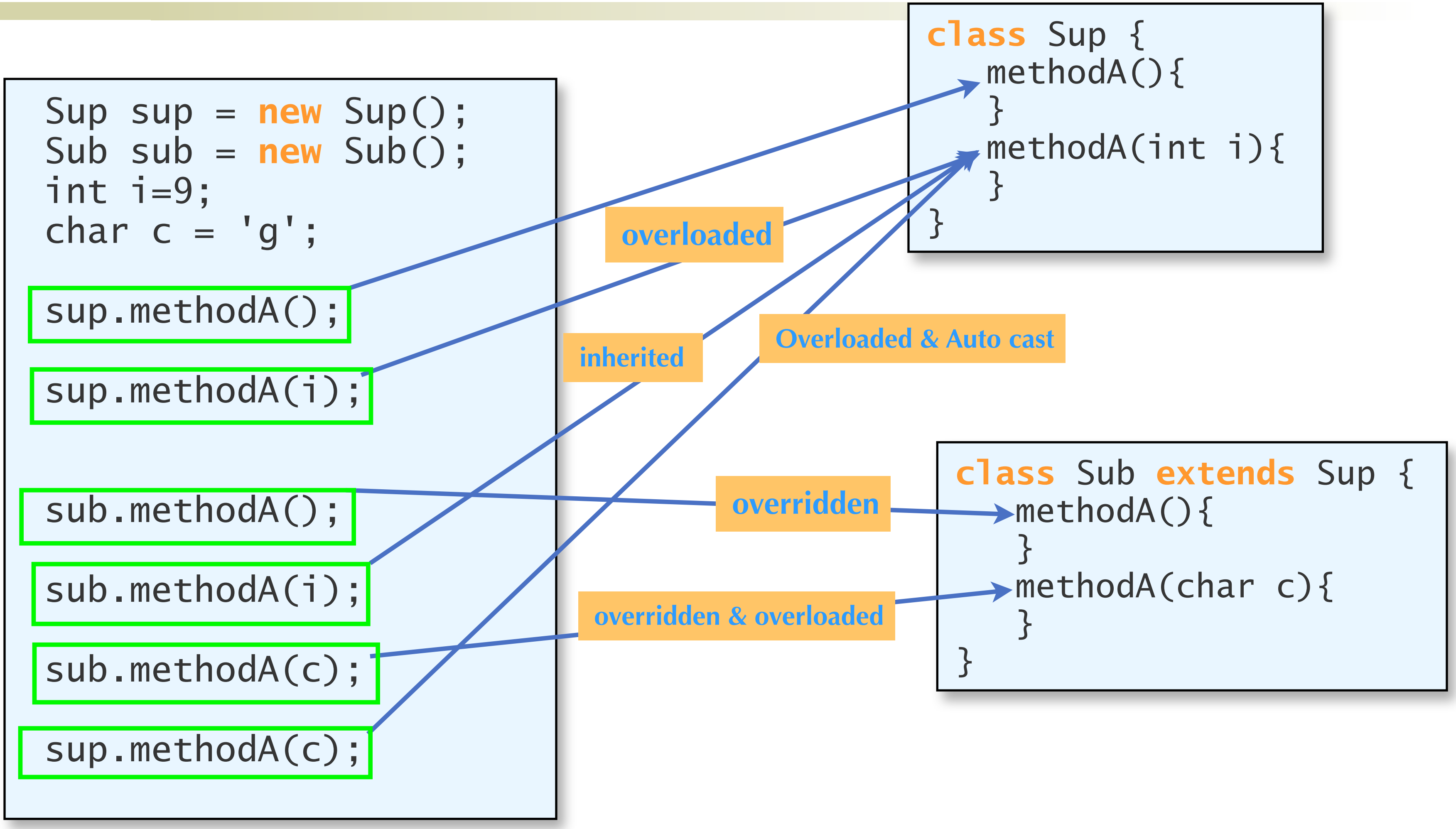
```
class SavingsAccount extends Account{

     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");
     }
}
```

Overriding and overloading



```
Sup sup = new Sup();
Sub sub = new Sub();
int i=9;
char c = 'g';

sup.methodA();

sup.methodA(i);


sub.methodA();

sub.methodA(i);

sub.methodA(c);

sup.methodA(c);
```

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

```
class Sub extends Sup {
    methodA(){
    }
    methodA(char c){
    }
}
```

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()
```

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

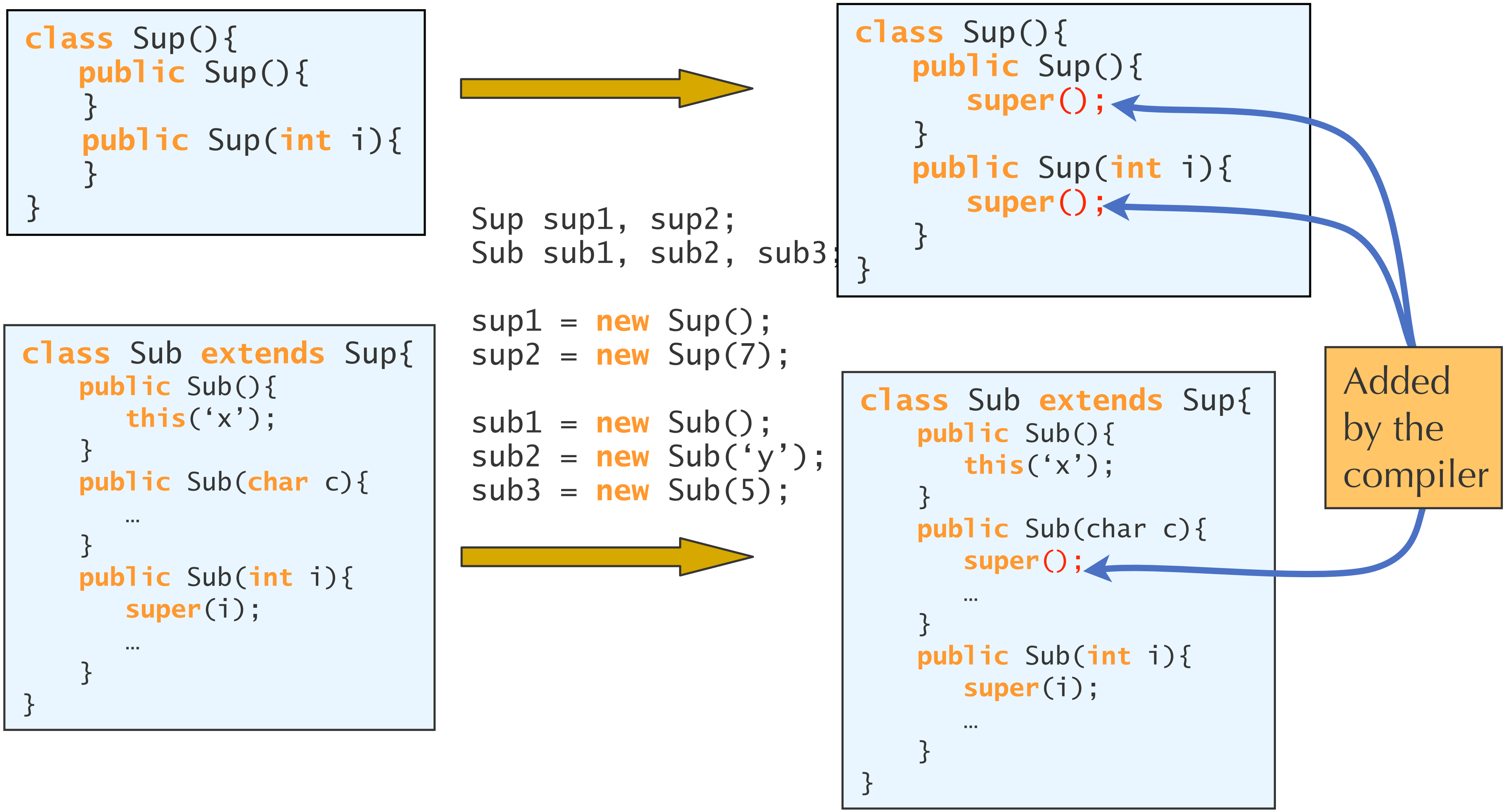
Inheritance and Constructors

- 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   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

```java
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");
   }

   . . .
}
```

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");

   }


   . . .
}
```

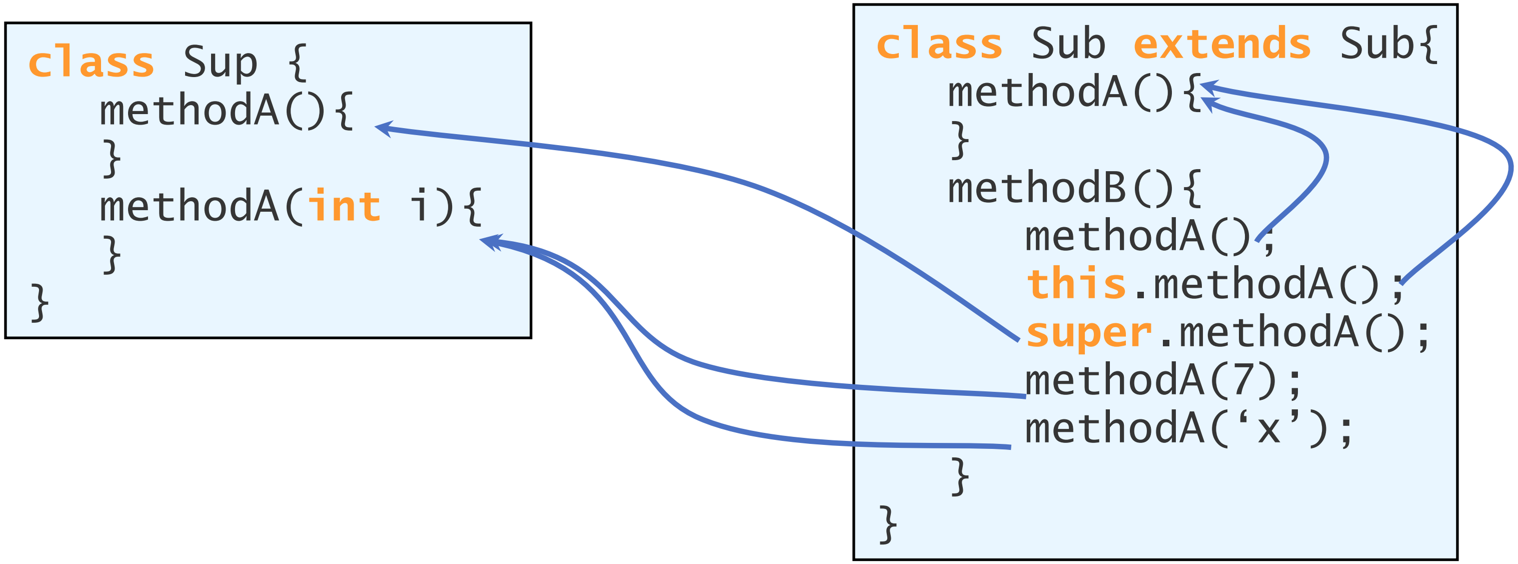
Super keyword

- 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.

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

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?

PURDUE UNIVERSITY

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.