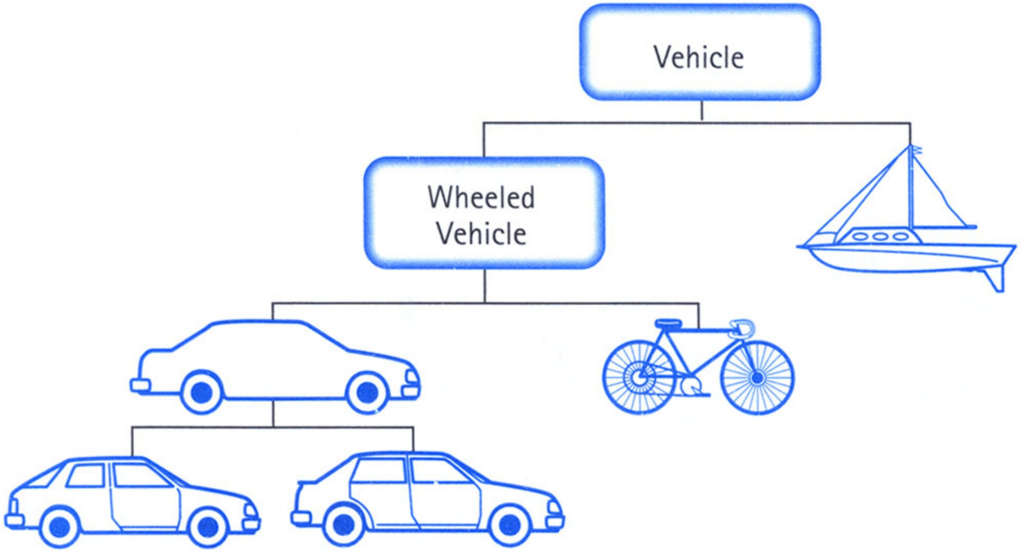
COMP 249: Object Oriented Programming II

Chapter 7 - Inheritance



Class SavingsAccount

```
public class SavingsAccount
{

  private int acctNum;
  private int balance;
  private String name;
  private double interestRate;

  public SavingsAccount() {

     acctNum = 0;
     balance = 0;
     name = "";
     interestRate = 0;

  }

  public double getBalance()

   { … }

  public void deposit(double amount)

   { … }

…
```

Class CheckingAccount

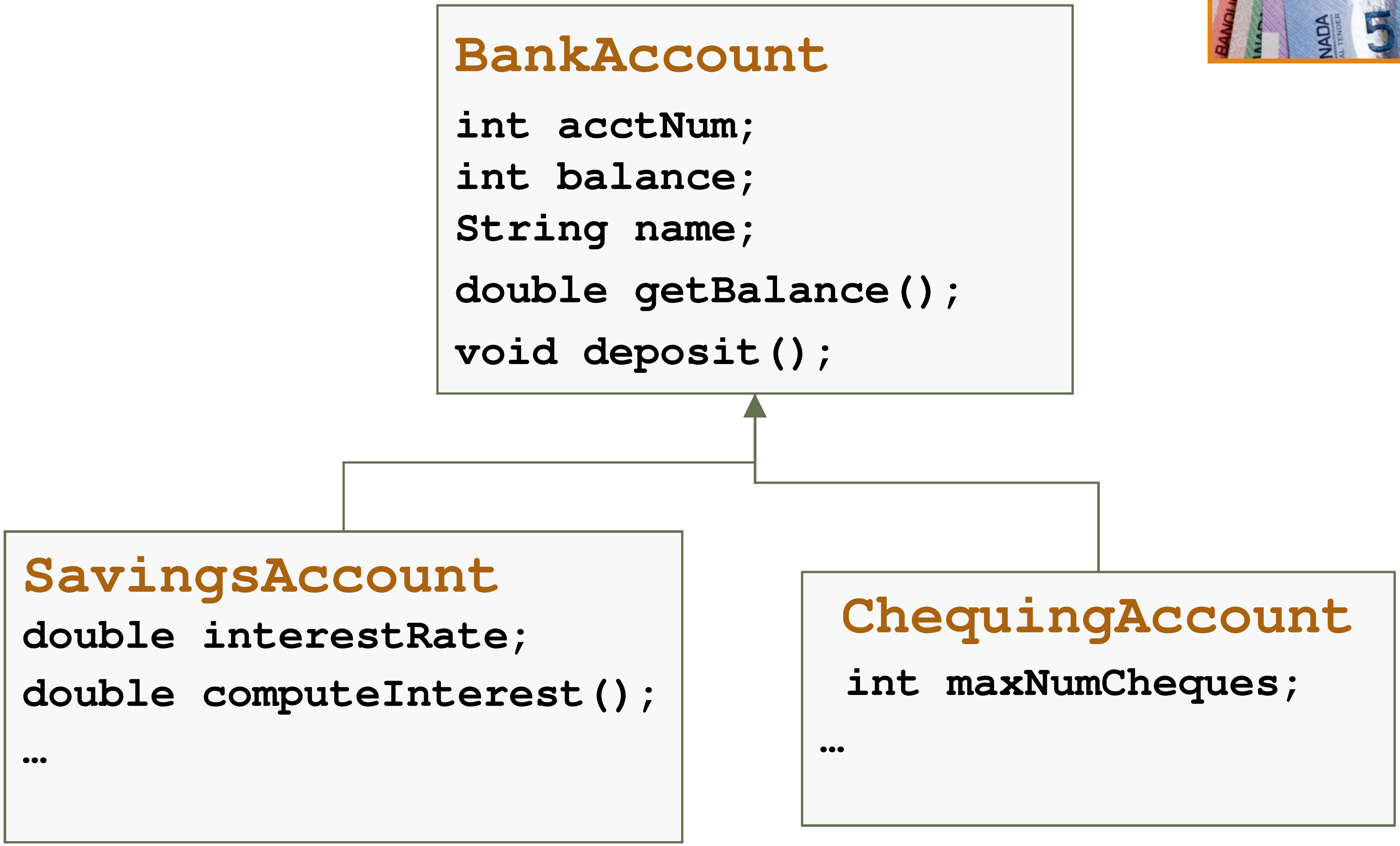
```
public class CheckingAccount
{
   private int acctNum;
   private int balance;
   private String name;
   private int maxNumCheques;

   public CheckingAccount() {
       acctNum = 0;
       balance = 0;
       name = "";
       maxNumCheques = 0;
   }
   public double getBalance ()
       { … }
   public void deposit (double amount)
       { … }
 …
}
```

Comparing Accounts ….

```
public class SavingsAccount {

  private int acctNum;
  private int balance;
  private String name;

  private double interestRate;


  public SavingsAccount() {

       acctNum = 0;
       balance = 0;
       name = "";

       interestRate = 0;

  }

  public double getBalance()

     { … }

  public void deposit(double
  amount)

     { … }
```

```
public class CheckingAccount {
   private int acctNum;
   private int balance;
   private String name;
   private int maxNumCheques;

   public CheckingAccount() {
        acctNum = 0;
        balance = 0;
        name = "";
        maxNumCheques = 0;
   }
   public double getBalance()
       { … }
   public void deposit(double amount)
      { … }
  …
}
```

Solution …



BankAccount

```
int acctNum;
int balance;
String name;
double getBalance();
void deposit();
```

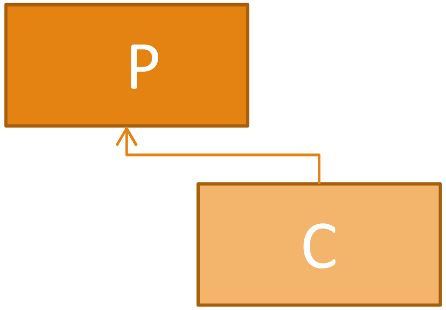
SavingsAccount

```
double interestRate;
double computeInterest();
…
```

ChequingAccount

```
int maxNumCheques;
…
```

Inheritance



When a class C acquires the properties of another class P, then class C is said to have **inherited** class P

- class P is called **base**, **parent** or **super** class
- class C is called a **child** or **sub** class.

A subclass inherits the members defined by the super class and adds its own, unique elements.

The keyword **`extends`** is used to inherit a class.

Ex: `class subclass extends superclass {…}`

Class BankAccount

```
public class BankAccount {
   int acctNum;
   int balance
   String name;

   public BankAccount() {
       acctNum = 0;
       balance = 0;
       name = "";
   }
   public double getBalance () { … }
   public void deposit (double money) { … }
 …
}
```

```
BankAccount
int acctNum;
int balance;
String name;
double getBalance();
void deposit();
```

New SavingsAccount Class

```
public class SavingsAccount ___________ ___________
{

  private double interestRate; // specific to savings accounts

  public SavingsAccount() {
       interestRate = 0;
  }

  // specific methods of a savings account
  public double computeInterest() {…}
}
```

```
SavingsAccount
double interestRate;
double computeInterest();
…
```

New ChequingAccount Class:

```
public class __________ _________ ___________
{
  ____________________

  public CheckingAccount(____________) {

      ___________________;
  }
}
```

```
ChequingAccount
 int maxNumCheques;
…
```

Inheritance

More sophisticated form of code re-use

Requires significant similarities between objects

Represents the **IS-A** relation

Examples:

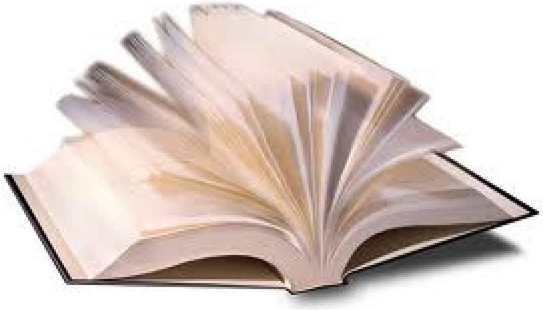
- a (any) Monkey **is-a** Primate
- a (any) Dictionary **is-a** Book

Not to confuse with instances of classes

- Koko is a **specific** monkey
- Webster is a **specific** dictionary

Class Book

```
class Book {
  private int pages = 1500;

  public void setPages(int numPages) {
      pages = numPages;
  }

  public int getPages() {
      return pages;
  }
}
```



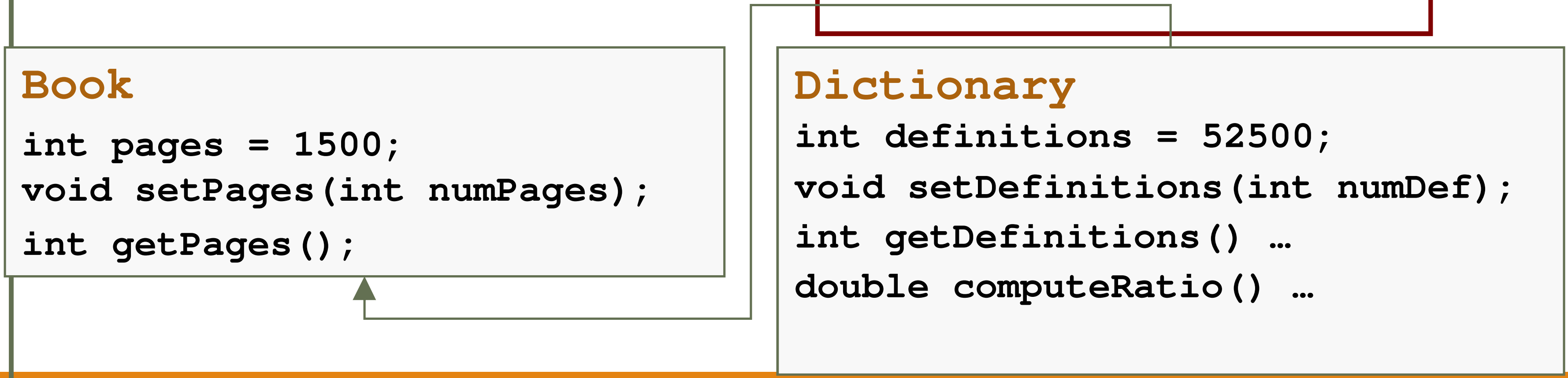
Class Dictionary

```
class Dictionary extends Book {

   private int definitions = 52500;


   public void setDefinitions(int numDef) {
        definitions = numDef;

   }

   public int getDefinitions() {
        return definitions;

   }

  public double computeRatio() {
        return
        A.   definitions/pages;
        B.   definitions/getPages();
        C.   getDefinitions()/pages;
        D.   getDefinitions()/getPages();
        E.   Only 2 are valid

  } //computeration

}
```

Example: Words.java

```
public class Words {
  public static void main (String[] args) {
    Dictionary webster = new Dictionary();
    System.out.println("Nb of pages:" + webster.getPages());
    System.out.println("Nb of defs:"+webster.getDefinitions());
    System.out.println("Defs per page:"+webster.computeRatio());
  }
}
```

Output



Book

```
int pages = 1500;
void setPages(int numPages);
int getPages();
```

Dictionary

```
int definitions = 52500;
void setDefinitions(int numDef);
int getDefinitions() …
double computeRatio() …
```

Inheritance vs Composition

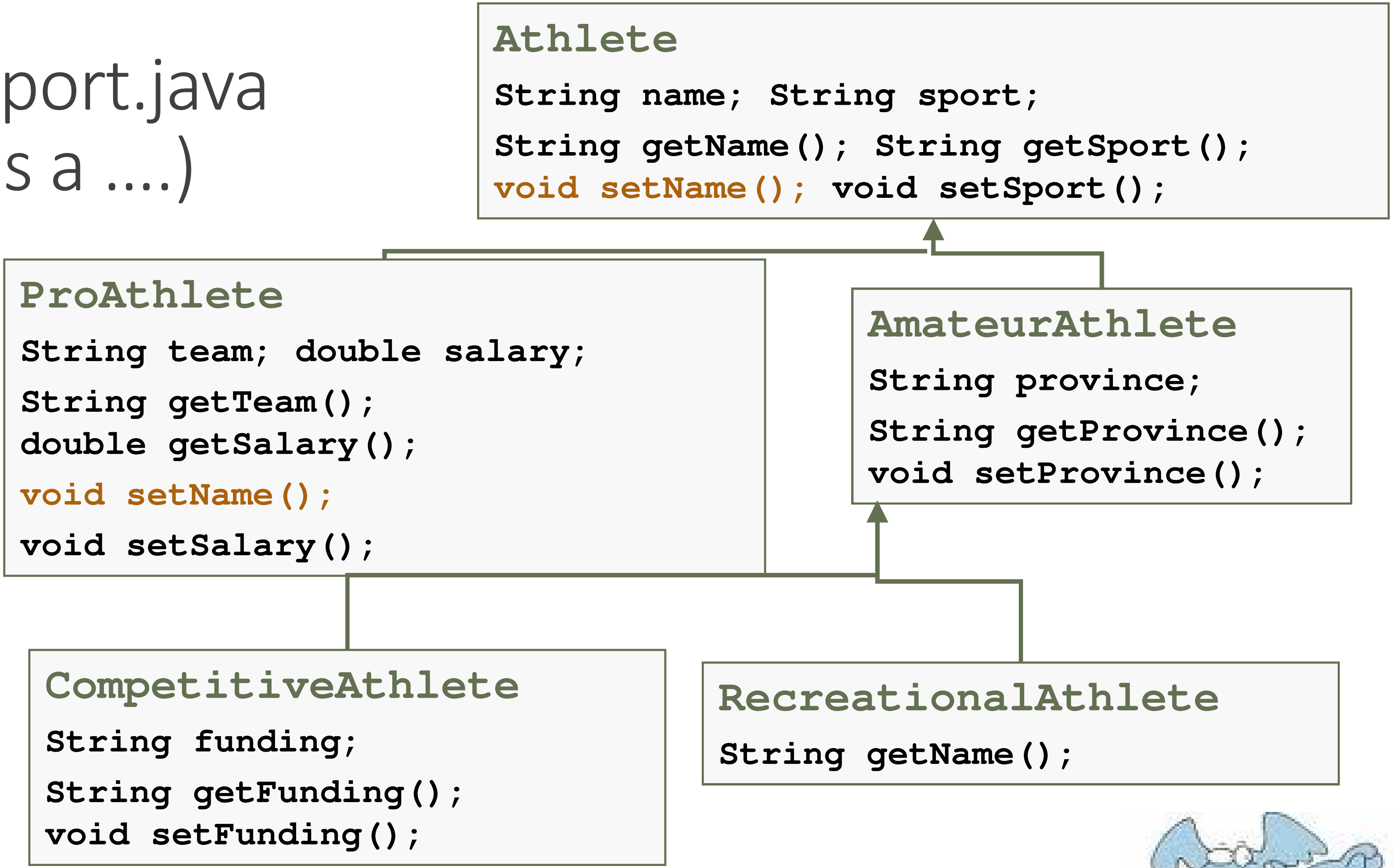
inheritance is ___________ relation

- ex: **Bank.java**, **Sport.java**

composition is _________ relation

- ex: **Car.java**

Sport.java (is a ....)



Athlete

```
String name; String sport;
String getName(); String getSport();
void setName(); void setSport();
```

ProAthlete

```
String team; double salary;
String getTeam();
double getSalary();
void setName();
void setSalary();
```

AmateurAthlete

```
String province;
String getProvince();
void setProvince();
```

CompetitiveAthlete

```
String funding;
String getFunding();
void setFunding();
```

RecreationalAthlete

```
String getName();
```

Car.java (has a ….)

Engine

```
void start();
void stop();
void rev();
```

Door

```
void open();
void close();
```

Window

```
Void rollDown();
void rollUp();
```

Wheel

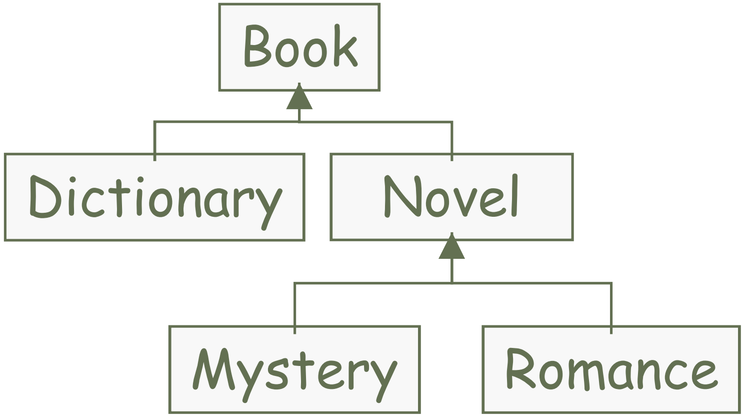
```
void inflate();
```

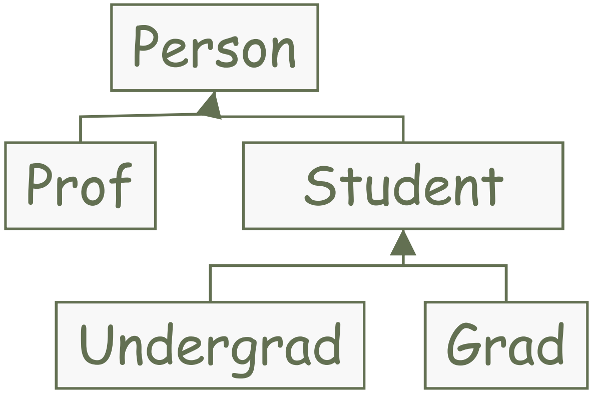
Car

```
Engine engine;
Door left;
Door right;
Wheel[] wheel;
```

Class Hierarchies

A child class can itself be the parent of another class, forming a *class hierarchy*





Good class design: put all common features as high in the hierarchy as is reasonable

An inherited member is continually passed down the line

Just Checking

What does a derived class automatically inherit from the base class?

A. instance variables

B. static variables

C. public methods

D. all of the above

Method Overriding

Typically, we add functionality to the derived class

Two ways:

- Write completely new methods
- Re-define inherited ones

Method overriding

- Derived class re-defines an inherited method
- Can we still access the parent method? ________

Method Overriding

The new method must have:

- the same signature and return type as the parent's method,
- but can have different code (obviously!)

The type of the object executing the method determines which version of the method is invoked (the derived class or the parent's)

Example1: Method Overriding

```
public class Parent {
  public void hello() {
       System.out.println(“Hello from parent”);
  }
}
public class Child extends Parent {
  public void hello() {
       System.out.println(“Hello from Child”);
  }
}
```

```
…
Child baby = new Child();
baby.hello();
…
```

output?

Method Overriding

`final` methods cannot be overridden (by definition)

`static` methods cannot be overridden

variables can be overridden

- called *shadowing variables*
- but don't do it… it's confusing

Overriding vs overloading

Overriding and overloading are 2 different things!

Overriding =

- a child overrides its parent's method
- i.e. provides the same signature and return type, but different code

Overloading =

- not necessarily related to inheritance
- 2 methods have the same name but different argument list

Example: Overload vs Override

```
class OverLoad {
    public int xyz = 9;
    public void stuff(int x){
                System.out.println("x is an int in OverLoad");
    }
    public void stuff(double x){             // overloading? overriding?
                System.out.println("x is a double in OverLoad");
    }
}
class OverRide extends OverLoad {
  public int xyz = 99; // don't do this! creates confusion
  public void stuff (char x){ // overloading? overriding?
                System.out.println("x is a char in overRide");
    }
  public void stuff (int x) {                // overloading? overriding?
    System.out.println("x is an int in overRide");
    System.out.println("call parent stuff(int x)");
    super.stuff(x);
    }
}
```

see Over.java

So output is?

```
public static void main(String[] args) {

  OverLoad testA = new OverLoad();
  testA.stuff(4);
  testA.stuff(4.0);
  System.out.println("xyz in parent = " + testA.xyz);

  System.out.println();
   …
```

```
sclass OverLoad {

  public int xyz = 9;

  public void stuff(int x){

       System.out.println("x is an int in OverLoad"); }


  public void stuff(double x){

       System.out.println("x is a double in OverLoad");}    }
```

So output is …?

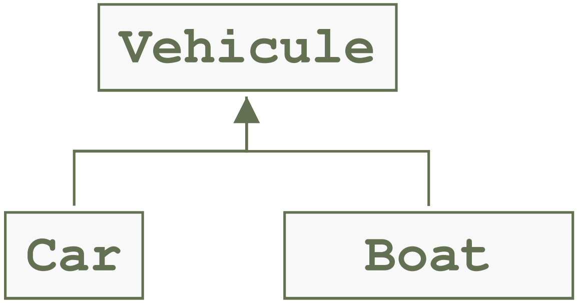
```
  …

  OverRide testB = new OverRide();
  testB.stuff('4');
  testB.stuff(4.0);
  testB.stuff(4);
  System.out.println("xyz in child = " + testB.xyz);

  }
```

```
class OverRide extends OverLoad {

  public int xyz = 99; // don't do this! creates confusion

  public void stuff (char x){

       System.out.println("x is a char in overRide");}

  public void stuff (int x) {

    System.out.println("x is an int in overRide");

    System.out.println("call parent stuff(int x)");
    super.stuff(x);}

}
```

The **super** constructor

A child class inherits its parent's members

Sometimes we need to access directly the parent's members (rather than inherited version)



Vehicule

Car

Boat

The **super** keyword

- **super. member** --> always uses the superclass's member
  - ex: **super.aMethod()**
  - ex: **super.anAttribute**
- **super()** --> calls the constructor of its immediate superclass *(more details in 2 min.)*

Can we do **super.super.aMethod()?**

Example: **super**

```
public class BankAccount {
  protected double getBalance() {return 1000.0;}
}
----------------------------------------
public class SavingsAccount extends BankAccount{
  protected double getBalance() {return 1010.0;}

  protected void printBalance() {
     System.out.println(super.getBalance());
     System.out.println(getBalance());
     System.out.println(this.getBalance());
   }
}
```

Output?

The **`this`** Constructor

In a constructor, **`this`** can be used as a name for invoking another constructor in the same class

- same restrictions on **`super`** apply to **`this`**

If you have to call **`super`** and **`this`**

- call **`this`** first,
- then the constructor that is called must call **`super`** as its first action
- can’t have both in a same method

The **this** Constructor

```
class A {
  public A() {
    this( false );
  }
  public A(boolean
someFlag) {
  }
}
```

```
class B extends A {
  public B() {
    super();
  }
  public B( boolean someFlag ) {
    super( someFlag );
  }
  public B ( int someNumber ) {
    this(); //
  }
}
```

```
 this( false);
A()----------------> A(false)
^
|
| super();
|
|     this();
B() <--------------- B(5)  <--- you start here
```

The `this` Constructor

- Often, a no-argument constructor uses **this** to invoke an explicit-value constructor
- Example:

```
public HourlyEmployee() {
  this("No name", new Date(), 0, 0);
}

public HourlyEmployee(String theName,
                      Date theDate,
                      double theWageRate,
                      double theHours)
{…}
```

Inheritance and Constructors

Constructors are **not** inherited, even though they have public visibility

But we often want to use the parent's constructor to initialize the "parent's part" of the object

Initialize from parent class down to current derived class

Inheritance and Constructors

If you don't call the parent's constructor yourself:

- Java will automatically call **`super()`** (parent's constructor) as 1st line of every constructor

So if you provide your own constructors with arguments

- You must explicitly call the parent's constructor (**`super(argument list)`**) since no default constructor will be provided

Inheritance and Constructors

```
public class BankAccount {
   int acctNum; int balance;
   String name;

   public BankAccount(int aNum, int aBal;
   String aName) {
        acctNum = aNum; balance = aBal;
   name = aName;}
   public double getBalance () { … }
    …  }
```

```
public class SavingsAccount extends Account {

  double interest;

  public SavingsAccount(int aNum, int aBal, String aName,
  double aRate) {

       super(aNum, aBal,aName); //explicit call to parent's
                                 // constructor

       interest = aRate;

  } }
```

What if you forget the call…

```
public class BankAccount {
   int acctNum; int balance;
   String name;
   public BankAccount(int aNum, int aBal; String aName) {
     acctNum = aNum;
     balance = aBal;
     name = aName;
   }
   public double getBalance () { … }
   …
}
```

```
public class SavingsAccount extends Account {

  double interest;

  public SavingsAccount(int aNum, int aBalString aName, double aRate) {

        //super(aNum, aBal,aName); //explicit call to parent's
                                 // constructor

       interest = aRate;

  }

}
```

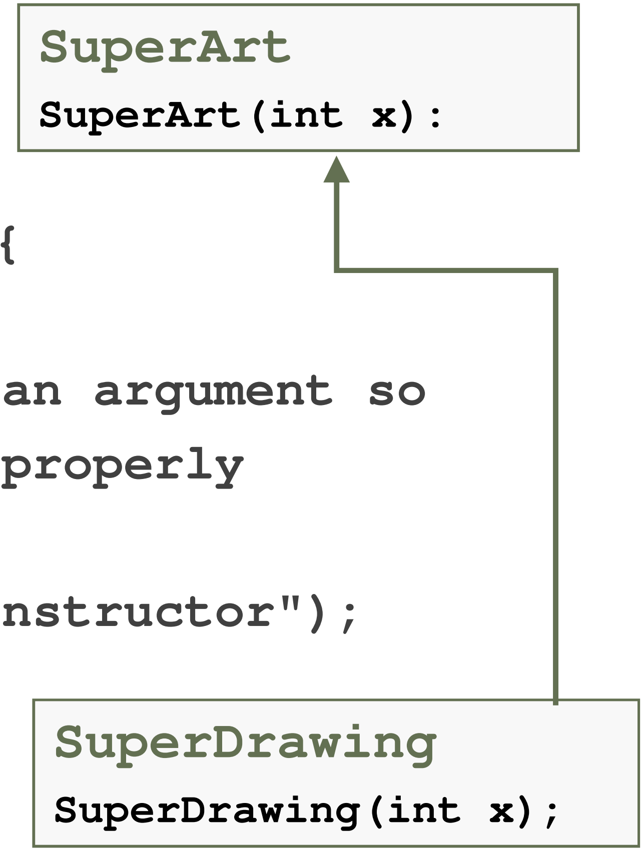
Simple Examples:

SuperCartoon.java

SuperWords.java

SuperCartoon.java

```
class SuperArt {
  public SuperArt(int x) {
      System.out.println("Art constructor");
    }
}

class SuperDrawing extends SuperArt {
  public SuperDrawing(int x) {
      // the parent constructor has an argument so
      // you must invoke the parent properly
      super(x);
      System.out.println("Drawing constructor");
    }
}
```



SuperArt

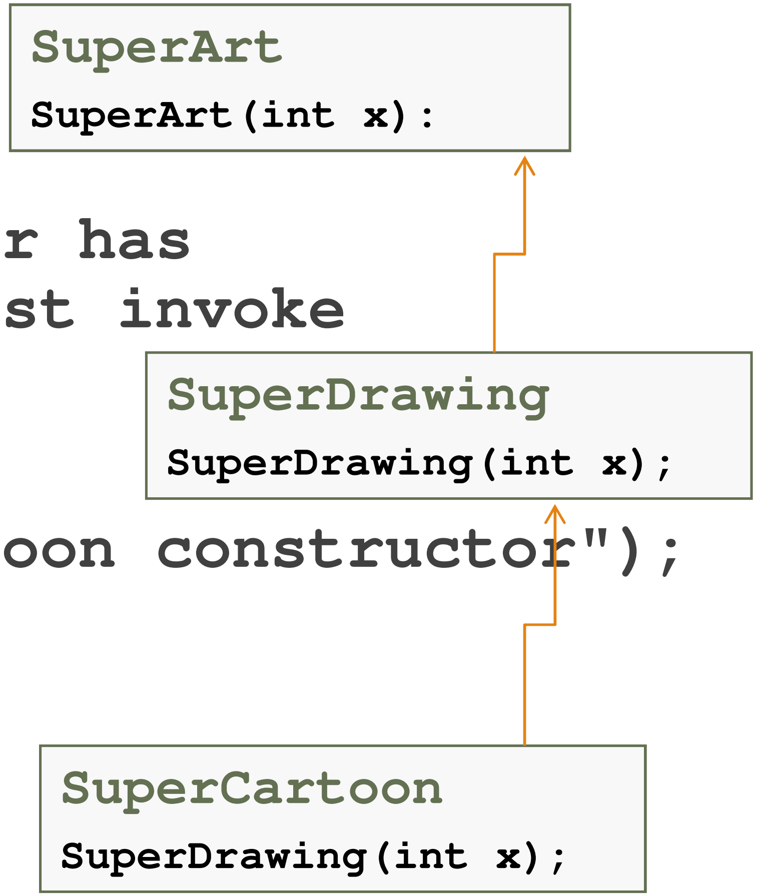
SuperArt(int x):

SuperDrawing

SuperDrawing(int x);

SuperCartoon.java …

```
public class SuperCartoon extends SuperDrawing {


  public SuperCartoon()
  {
    // the parent constructor has
    // an argument so you must invoke
    // the parent properly
    super(55);
    System.out.println("Cartoon constructor");
  }
```



SuperArt
SuperArt(int x):

SuperDrawing
SuperDrawing(int x);

SuperCartoon
SuperDrawing(int x);

SuperCartoon.java …

```
  public static void main(String[] args)
  {
      SuperCartoon sc = new SuperCartoon();
      // no constructor arguments required
  }

}
```

Output?

Member Visibility

Visibility modifiers determine which class members get inherited and which do not

Members declared with **`public`** visibility are inherited

Members declared with **`private`** visibility are not inherited

- well… private variables are… but not by "name"… you can only access them through a public accessor method from the parent

but **`public`** members violate encapsulation

so we have the **`protected`** visibility

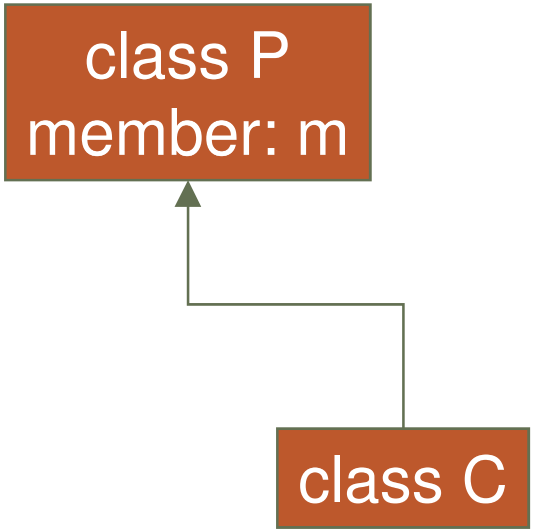
Member Visibility – public to default

if m is public:

- anyone has access to m (P.m)
- C inherits m (C.m)

if m is private:

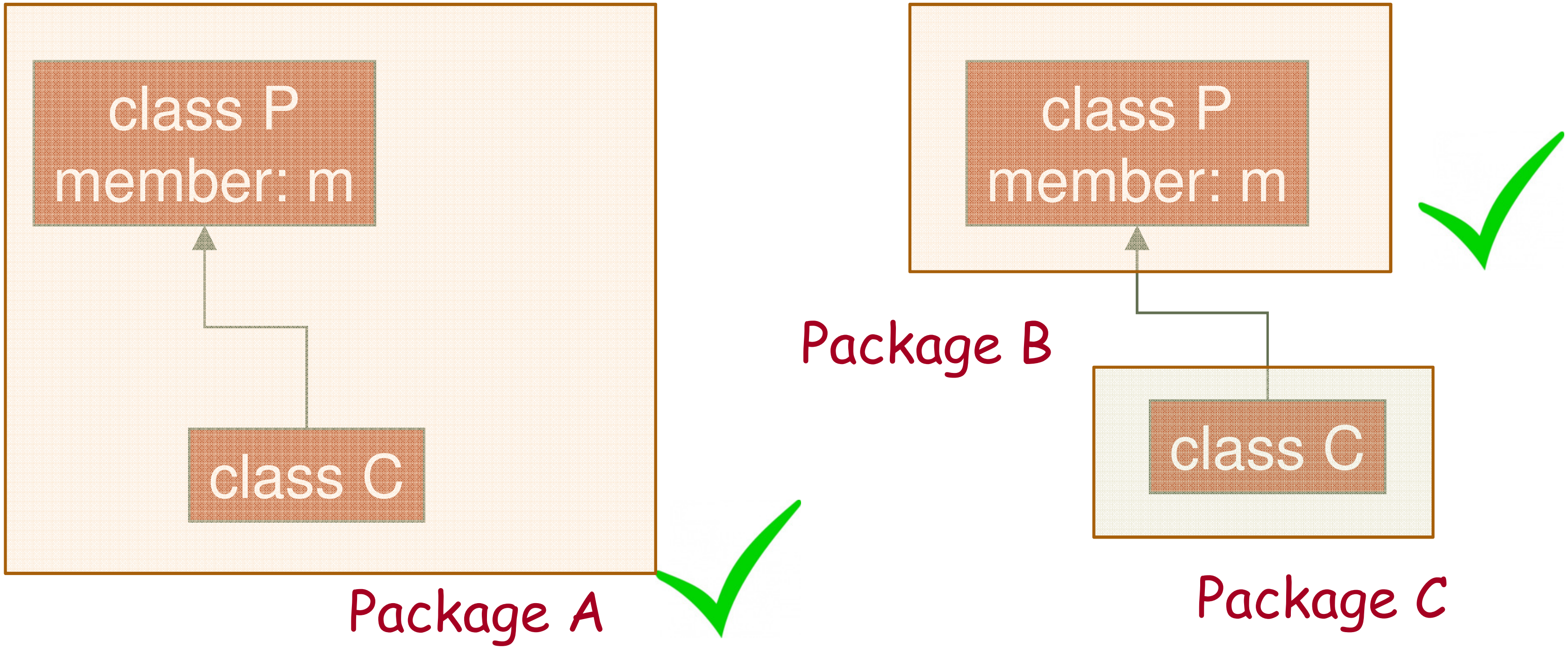
- only class P has access to m
- C does not inherit m



Member Visibility – public to default …

if m is protected:

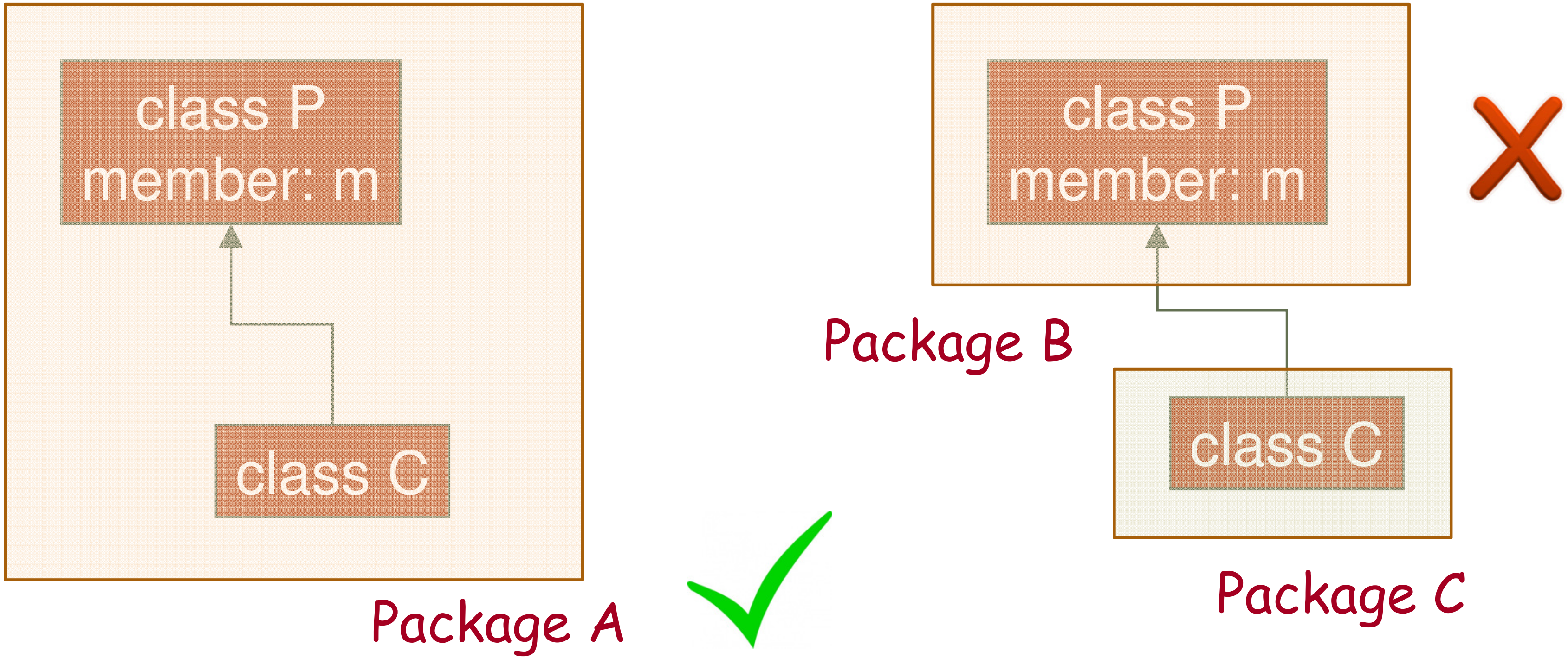
- all classes in the same package have access to m
- C inherits m (even if not in same package as P)



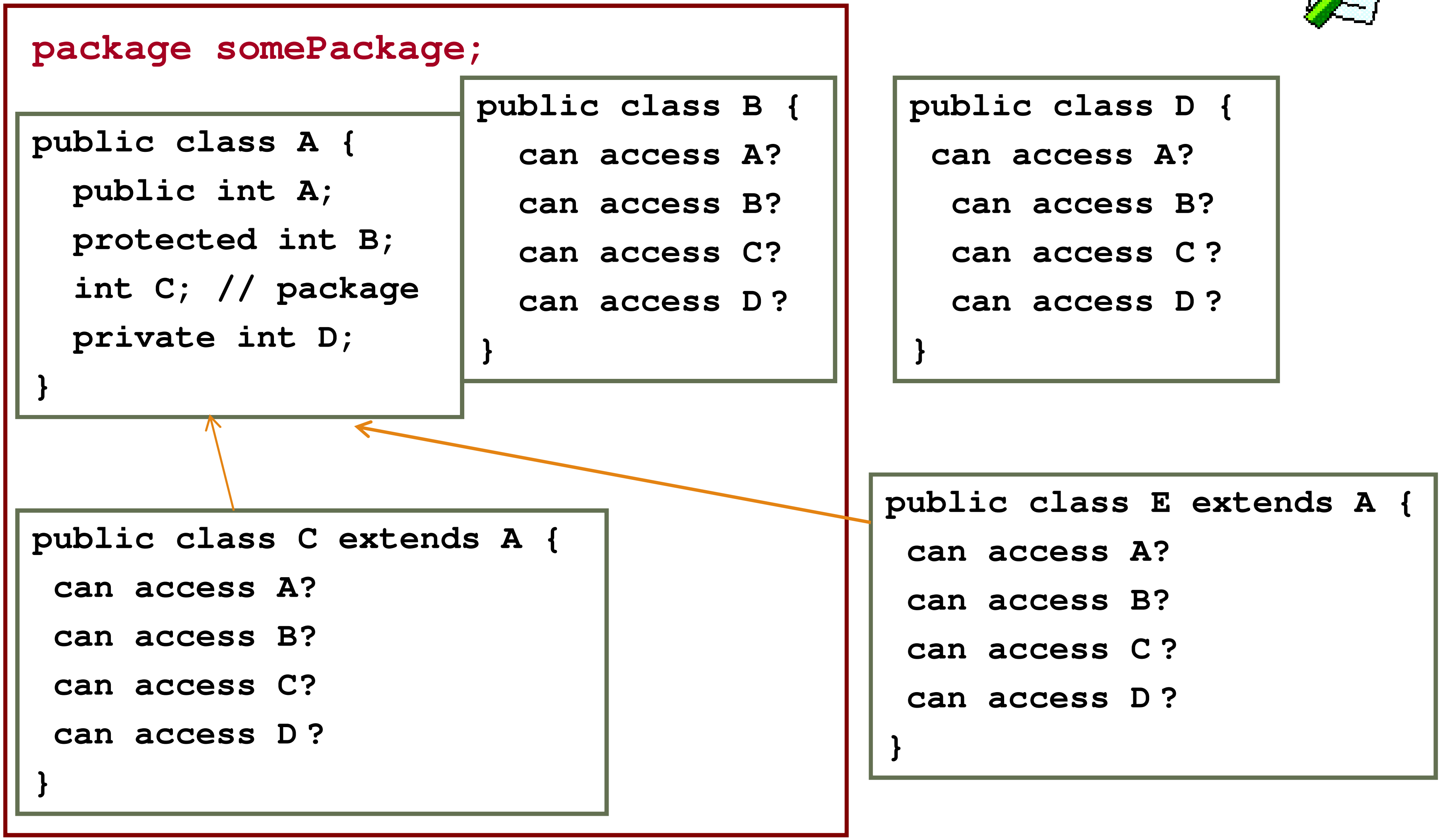
Member Visibility – public to default …

if m has default visibility (i.e. nothing is mentioned):

- only the classes in the same package have access to m
- C inherits m only if it is in the same package as P



Member Visibility Exercise



```
package somePackage;
```

```
public class A {
   public int A;
   protected int B;
   int C; // package
   private int D;
}
```

```
public class B {
   can access A?
   can access B?
   can access C?
   can access D ?
}
```

```
public class C extends A {
  can access A?
  can access B?
  can access C?
  can access D ?
}
```

```
public class D {
 can access A?
   can access B?
   can access C ?
   can access D ?
}
```

```
public class E extends A {
 can access A?
 can access B?
 can access C ?
 can access D ?
}
```

TestInherit.java

Method Overriding

A derived class can change the visibility of a superclass's methods, but only if it provides more access.

So it must:

- have the **same name as a superclass method**
- have the **same parameter list as a superclass method**
- have the **same return type as as a superclass method**

Method Overriding

So it must:

- . . . . .
- the access modifier for the overriding method may not be more restrictive than the access modifier of the superclass method
  - if the superclass method is **public**, the overriding method must be **public**
  - if the superclass method is **protected**, the overriding method may be **protected** or **public**
  - if the superclass method is **package**, the overriding method may be **package, protected, or public**
  - if the superclass methods is **private, _______________???**
- why more access?

Member Visibility

In general you should create:

- Private instance variables
- Public accessor/mutator methods (getX/setX)

WHY???

- why not just use public variables?

Protected visibility

Protected can be over-used

- Do not use when what you want is just access to variables in subclasses.
- Use public getters and setters for that.

Use protected when:

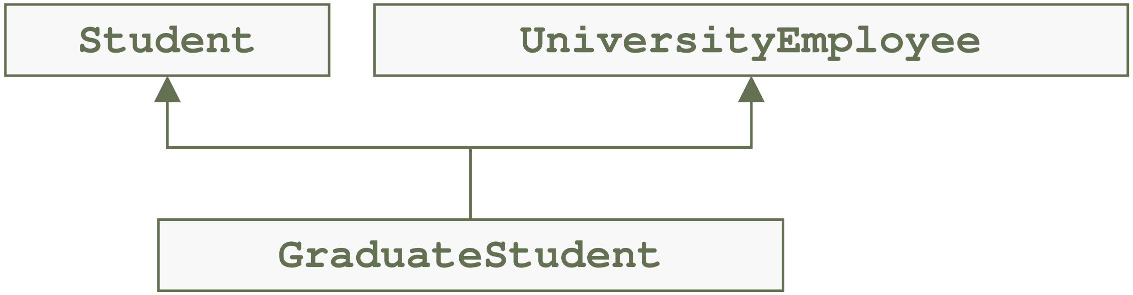
- direct access to members (usually methods) when sub-classes might be in different packages
- AND also hide these methods from the outside world

A weaker form of encapsulation since protected members are not private to sub-classes.

- Side effects are possible (e.g., when used with an instance variable)
- “Use it when you have to”

Multiple Inheritance

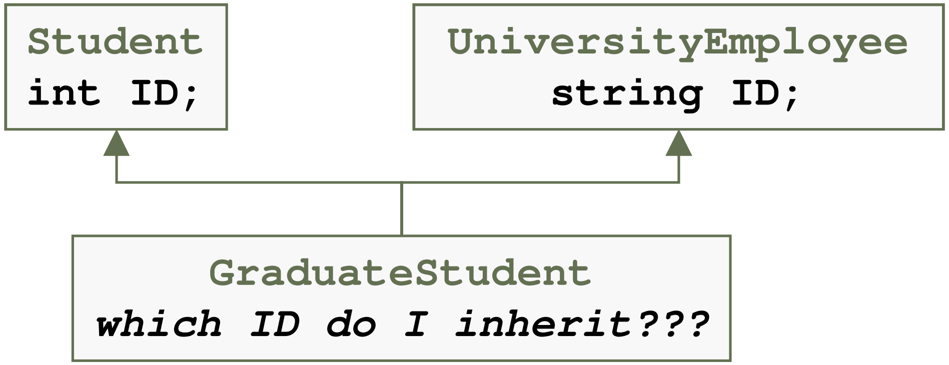
Inherit functionality from two (or more) unrelated parents



interesting…

Multiple Inheritance

but what if:



Which member does child inherit if both parents (or any ancestor from different paths) have members with same name?

so Java does not allow multiple class inheritance

But ….

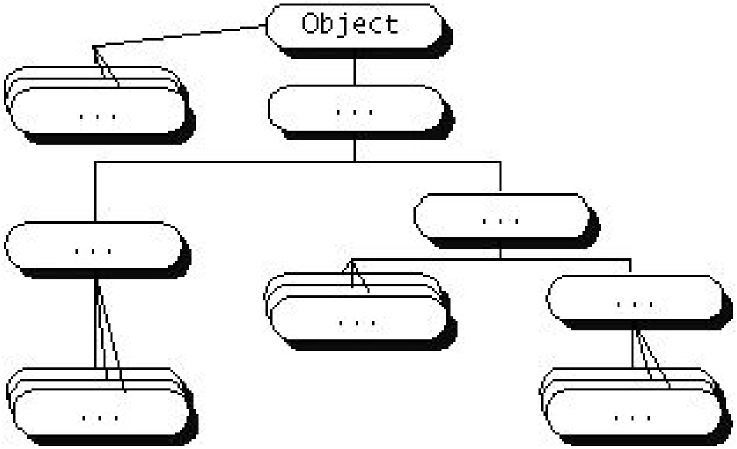
- Java allows multiple interface inheritance
- more on this later…

The `Object` Class

Every class in Java is an extended class, whether it is declared with an **`extends`** keyword or not.

If a class does not explicitly extend from another class, it implicitly extends the **`Object`** class.

```
public class myClass {…}
public class myClass extends Object {…}
```



The `Object` Class …

`Object` class provides methods for all objects:

`toString() clone()  equals()` …

Their code may not be very useful…

- So we often override them

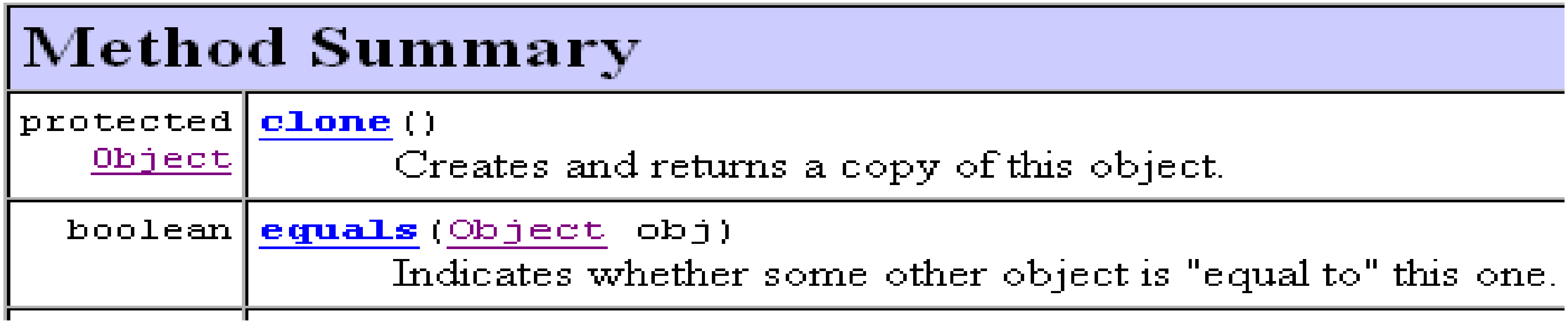
but their existence is very useful

- ex: That’s why `println` can call `toString` for any object – all objects are guaranteed to have a `toString` method via inheritance

The **equals** method

the **equals** method is inherited from **Object**

equals in Object :



Method Summary	
protected Object	clone() Creates and returns a copy of this object.
boolean	equals(Object obj) Indicates whether some other object is "equal to" this one.

The **`equals`** method

So methods like this just overloads it:

```
public boolean equals(Employee otherEmployee)
{ . . . }
```

To override **`equals`** you must have:

- type cast the parameter to the given class (e.g., **`Employee`**)
- compare each of the instance variables of both objects

Example

```
public boolean equals(Object otherObject)
{
  if(otherObject == null)
    return false;

  else if(getClass() != otherObject.getClass())
    return false;
  else
  {
    Employee otherEmployee = (Employee)otherObject;
    return (name.equals(otherEmployee.name) &&
            hireDate.equals(otherEmployee.hireDate));
  }
}
```

`instanceof` and **`getClass`**

to check the class of an object

- **`instanceof`** operator
- **`getClass()`** method
  - in **`Object`** class
  - final method (cannot be overriden)

instanceof and **getClass**

getClass() method is more exact

- The **instanceof** operator simply tests if an object is an instance of (a descendent) of a class
  - ex: fifi **instanceof** Dog,
    fifi **instanceof** Animal , …

- The **getClass()** method used in a test with
  == or **!=**
  tests if two objects *were created with* the same class
  - ex: fifi.getClass() is Dog

The `final` modifier

with **variables**

- `public static final int SIZE = 4;`
- indicates a constant

with **classes**

- `public final class xyz`
- class cannot be extended
- be careful: you are violating fundamental OOP principle
- not used that much

The `final` modifier …

with **methods**

- `public final int abc()`
- method cannot be overridden
- ex: may not want to allow a complex or sensitive method to be changed

Example: Stop.java

```
public class Stop {
  // final used with primitive values to create constants


  // Attributes

  public static final int STUFF = 20;
       // proper way - value in the class

  public final int fuzz = 5;
       // also constant but a copy in every object

  public final int val;
       // blank final - value will be set later


  // Methods

  // initialize the blank final at run-time
  //  - cannot be changed after

  public Stop(int val) { this.val = val; }
```

Example: Stop.java

```
// Methods

  // cannot over-ride this method in any sub-class

  public final int calculateFoo()
  { return fuzz + STUFF + val; }



  public static void main (String[] args) {

       Stop test = new Stop(100);

       System.out.println("val = " + test.calculateFoo()
  );

  }
}
```

STUFF = 20 (Static/final)
fuzz = 5 (Final)
val= ? (Final)

Output