**Contents** **Topic 05 -Interfaces and Inner Classes**



---

## I. Review

Review some key terms:

**Visibility (public / protected / private)**:

- When we implement or redefine a method in the subclass, it must be at least as "visible" as one in the superclass.
- A subclass cannot access the private members in the superclass.

**The static keyword**

Used to denote fields and methods that belong to a class (but not to any particular object).

**The abstract keyword**

The **abstract** keyword is applied for **classes** and **nonstatic methods**:

- When applied for a nonstatic **method**: means that we intend to provide **no implementation**; and the implementation will be provided in concrete subclasses.
- When applied for a **class**: means that the class may or may not include abstract methods. Abstract classes **cannot be instantiated** (ie. cannot be used to instantiate any object), but they **can be subclassed**.

- **abstract** is NOT for fields (no matter static or nonstatic)
- **abstract** is NOT for constructors or static methods

**Polymorphism** – An object variable (Superclass) can refer to different actual types (Superclass and subclass which are concrete). [compile time checking]

E.g., An object variable (of type A) can refer to objects of various actual types, including type A and its sub-types.

**Dynamic Binding** – Automatically select the appropriate non-static method (Not field!). [runtime checking]

# II. Interface

## Java Interface

- **Interface** is a way to describe what classes should do (not how)
  - ❑ Only headings are given to methods [1] (ie. no implementation)
  - ❑ Syntax:

```
interface Interface_Name
{
        /*nonstatic methods, static final fields*/
}
```

- A class can implement an interface (or more than one interfaces)
  - ❑ An implementing class satisfies an interface by implementing the methods given in the interface.
  - ❑ Syntax:

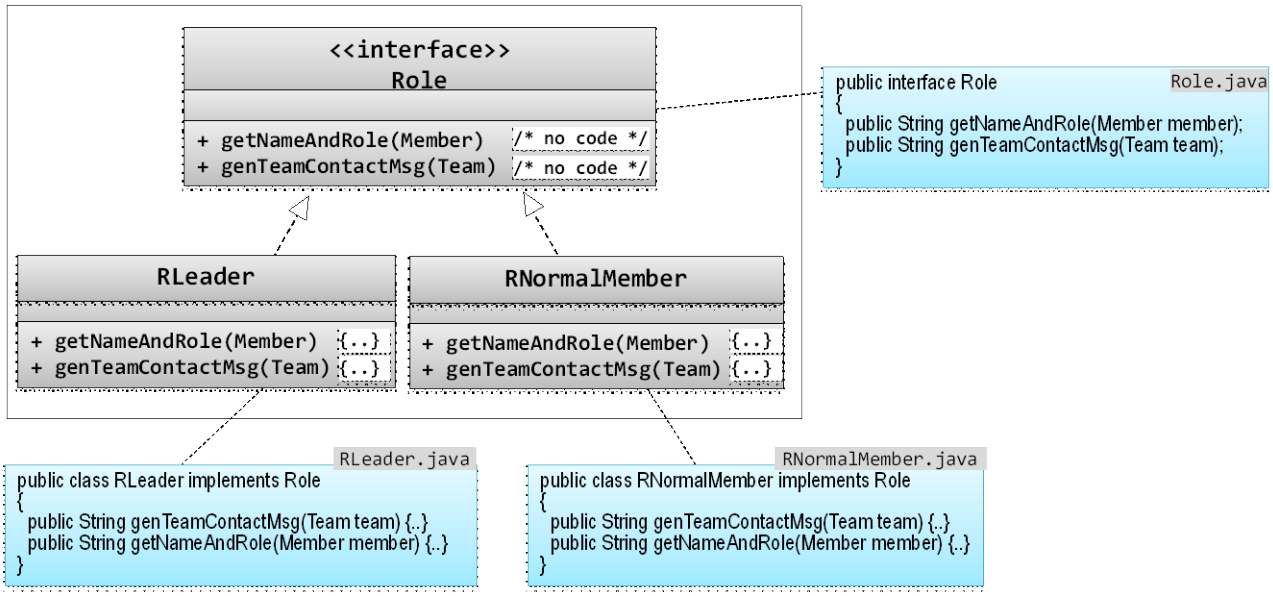
```
class Class_Name implements Interface_Name [, Interface_Name ..]
{
        ..
}
```

[1] For simplicity, here we talk about general non-static methods only.
Java 8 and onwards allow default and static methods which should come with implementation
(http://docs.oracle.com/javase/tutorial/java/IandI/defaultmethods.html)

## Example 1 - Working with Member Roles (Lab Exercise Reviewed)

Team.java

```
Role r;
if (roleType=='l') r = new RLeader();
else /*roleType=='n'*/ r = new RNormalMember();
```



**Example 2** **An interface can be implemented by multiple classes**
**A class can implement multiple interfaces**

```
interface A {void f1(); void f2();}

interface B {void f3(); void f4();}

class C implements A
{
   public void f1() {..}
   public void f2() {..}
}

class D implements A,B
{
   public void f1() {..}
   public void f2() {..}
   public void f3() {..}
   public void f4() {..}
}

abstract class C2 implements A
{
    public void f1() {..}
}
```

```
public static void main(String[] args)
{
   A a;
   B b;
   C c;
   D d;
   C2 c2;

   //a = new A(); // Cannot instantiate the type A
   //b = new B(); // Cannot instantiate the type B
   c = new C();
   d = new D();
   //x5 = new C2(); // Cannot instantiate the type C2

   a = new C(); // Upcasting is no problem
   b = new D();
   a = c;       // Upcasting is no problem
   b = d;
   c = (C)a; // Downcasting requires explicit cast
   d = (D)b;
}
```

We can use A, B, C, D, C2 as data types. However, only C and D can be instantiated (create object instances)

More considerations:

(1) Two interfaces can contain the same method and implemented in one class

```
interface A2 {void f1(); void f5();}
class X implements A, A2
{
    public void f1() {..}
    public void f2() {..}
    public void f5() {..}
}
```

Both A and A2 have the same void f1()

(2) An abstract class implements an interface

```
abstract class C2 implements A
{
    public void f1() {..}
}
```

Not implement all methods for A, so marked "abstract".

(3) An interface extends one or more interfaces

```
interface T extends A,B { void f5();}
```

- A class can only extend one class.
- But an interface can extend more interfaces.

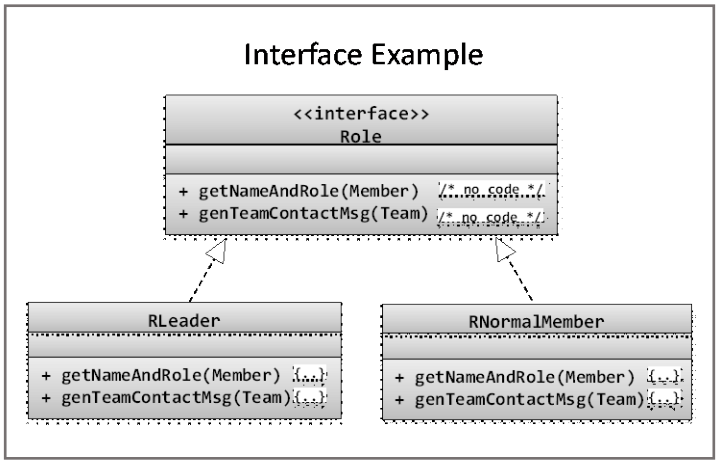
Here interface T has f1(),f2(),f3(),f4(),f5().
T is a sub-interface of both A and B.

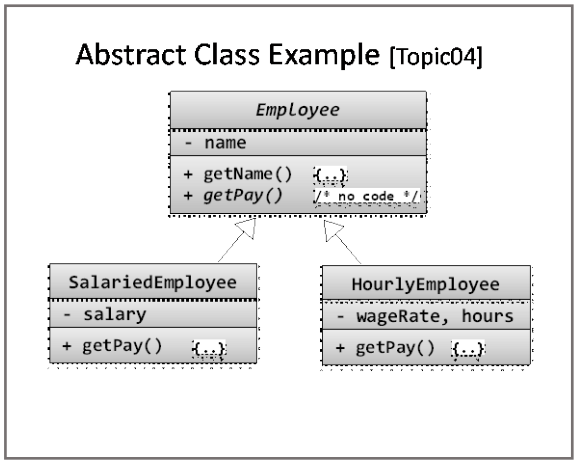
(4) A class can extend a superclass + implement interface(s)

```
class D2 extends D implements T
{
    public void f5() {..}
}
```

```
class Emloyee extends Person
implements I1, I2
{
    ..
}
```

## Interface vs Abstract class




Similarities

- Cannot instantiate them (ie. cannot create object instances)
- Contain methods which are to be implemented by other classes.

Differences

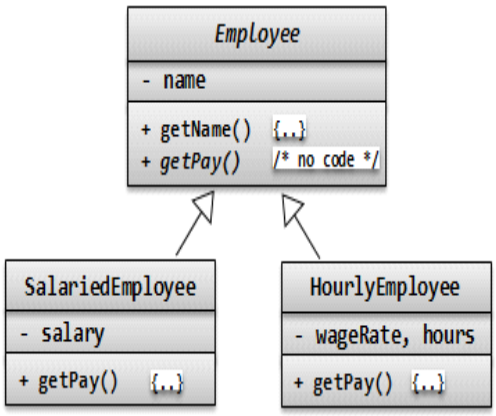
| Abstract class | Interface |
|---|---|
| A subclass can only inherit one abstract class<br>Abstract class does not support multiple inheritance.<br>`abstract class A {}`<br>`abstract class B {}`<br>`class C extends A, B {}` ✘<br>We cannot have multiple superclasses | A class can implement 1 or more interfaces.<br>Interface is a way to approximate multiple inheritance.<br>`interface A {..}`<br>`interface B {..}`<br>`class C implements A,B{..}` ✓ |
| Allow access modifiers (private / protected / public) | All methods are public<br>We do not need to write the keywords *public* , *abstract*. They are implicit for Interfaces. |
| Can provide shared method code (default behavior)<br>Nonstatic methods can be abstract or non-abstract | No shared method code:<br>Nonstatic methods are abstract *Generally speaking1*<br>- We write headers only |
| Has constructors | No constructors |
| Allow various kinds of fields: static or not, final or not | No object fields<br>- Any field defined in an interface is actually treated as static and final |

[1] For simplicity, here we talk about general non-static methods only.
Java 8 and onwards allow default and static methods which should come with implementation
(http://docs.oracle.com/javase/tutorial/java/IandI/defaultmethods.html)

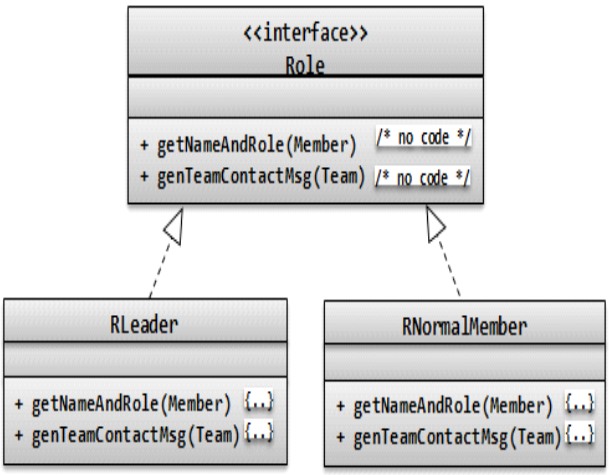
## Which should you use, abstract classes or interfaces?

[http://docs.oracle.com/javase/tutorial/java/IandI/abstract.html]

- Consider using **abstract classes** for any point below:
  - Want to **share code** among several closely related classes.
  - Expect that subclasses have **many common methods or fields**, or require non-public access modifiers such as **protected and private**.
  - Want to declare useful **object fields**. So that methods can access and modify the state of the object to which they belong.



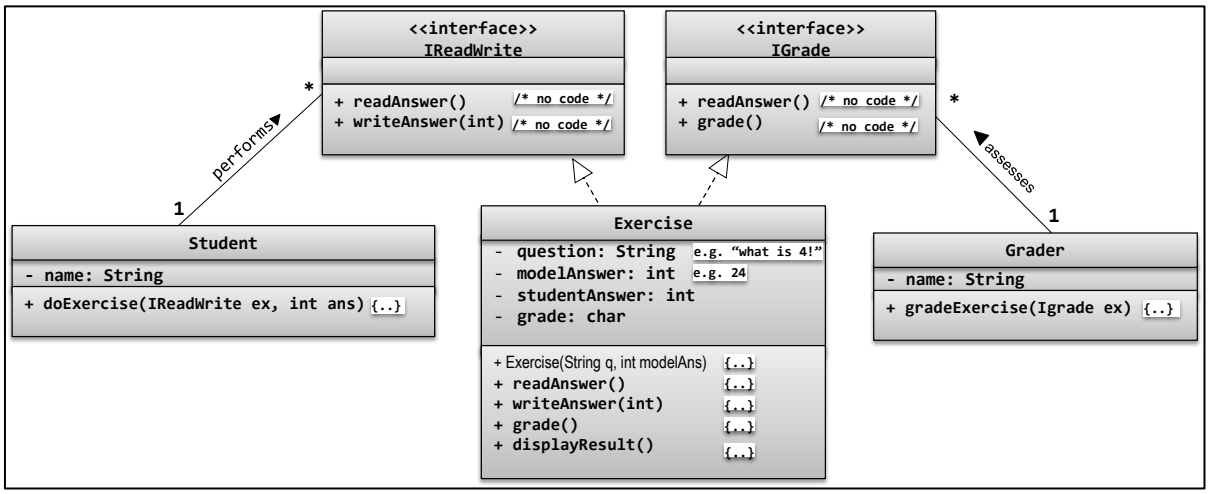

- Consider using **interfaces** for any point below:
  - Expect that **unrelated classes would implement** your interface. For example, the interfaces `Comparable` and `Cloneable` are implemented by many unrelated classes.
  - Want to specify the behavior of a particular data type, but **not concerned about who implements** its behavior.
  - Want to take advantage of **multiple inheritance of type** (See Example 3 in next page).



## Example 3 - Grader, Student, Exercise

- This example illustrates the **Practical use of "one class with multiple interfaces"**
- *Storyboard: A grader can only read/grade students' exercises, while a student can only read/write the exercise.*
- You will see that the `Exercise` class implements both the `IGrade` and `IReadWrite` interfaces.
- Goal: To **filter functionalities** so that

different users (here graders and students) who receive the same object (here the exercise object) can use the dedicated functions only.




```
class Student
{
    private String name;
    public Student(String n) {name=n;}
    public void doExercise(IReadWrite x, int ans)
    {
        x.writeAnswer(ans);
    }
}
```

```
class Grader
{
    private String name;
    public Grader(String n) {name=n;}
    public void gradeExercise(IGrade x)
    {
        x.grade();
    }
}
```

```
interface IReadWrite
{
    void readAnswer();
    void writeAnswer(int anAnswer);
}
```

```
interface IGrade
{
    void readAnswer();
    void grade();
}
```

```
class Exercise implements IGrade, IReadWrite
{
    private int studentAnswer;
    private char grade;
    private final String question;
    private final int modelAnswer;
    public Exercise(String q, int a)
    {
        question = q; modelAnswer=a;
    }
    public void writeAnswer(int anAnswer)
    {
        studentAnswer=anAnswer;
    }
    public void readAnswer()
    {
        System.out.println(
            "Student's answer is "+ studentAnswer);
    }
    public void grade()
    {
        if (studentAnswer==modelAnswer) grade='A';
        else grade='F';
    }
    public void displayResult()
    {
        System.out.println(
            "Student's answer is "+studentAnswer+
            ", grade is: "+grade);
    }
}
```

```
public static void main(String[] args)
{
    Exercise ex = new Exercise("What is 4!", 24);
    Student m = new Student("Mary");
    Grader h = new Grader("Helena");
    m.doExercise(ex,24);
    h.gradeExercise(ex);
    ex.displayResult();
}
```

**Output:**

```
Student's answer is 24,
grade is: A
```

## The Java Comparable Interface (sorting)

JAVA provides **sorting** methods for **comparable objects**

1) Arrays : **`Arrays.sort(array);`**
2) Collections (e.g. `ArrayList`) : **`Collections.sort(array_list);`**

Before using the above, we have to solve for some issues:

***"Nobody knows that trees and plants are to be sorted and, even if we are told, but how to sort them? ... 😐 "***

Correspondingly, in JAVA we have to

①tell that the objects are to be sorted, and ②decide how to *compare* them in sorting.

These are what we do in JAVA:

- the class should implement the interface: `java.lang.Comparable<type>`
- this `Comparable` interface has a method to be implemented: `int compareTo(type another)`

**Return value:**
**0 if equal**
**1 if *this* is larger than *another***
**-1 if *this* is smaller than *another***

Example: Employees ordered by salaries

```
class Employee implements Comparable<Employee>
{
    private final String name;
    private double salary;
    private final Day hireDay;
    ..
    @Override
    public int compareTo(Employee another)
    {
        if (this.salary==another.salary) return 0;
        else if (this.salary>another.salary) return 1;
        else return -1;
    }
}
```

```
class Employee implements Comparable<Employee>
{
    ..
    @Override
    public int compareTo(Employee e2) {
        .. //check this.salary and e2.salary
    }
}
```

```
public static void main(String[] args)
{
    /* sort an array of employees */
    Employee[] arr = new Employee[3];
    arr[0] = new Employee("Carl Cracker", 75000, 1987, 12, 15);
    arr[1] = new Employee("Harry Hacker", 50000, 1989, 10, 1);
    arr[2] = new Employee("Tony Tester", 40000, 1990, 3, 15);
    Arrays.sort(arr);
    for (Employee e : arr)
        System.out.println(e);

    /* sort an arraylist of employees */
    ArrayList<Employee> arrlist = new ArrayList<>();
    arrlist.add(arr[2]);arrlist.add(arr[0]);arrlist.add(arr[1]);
    Collections.sort(arrlist);
    for (Employee e : arrlist)
        System.out.println(e);
}
```

Output:

```
name=Tony Tester,salary=40000.0,hireDay=15 Mar 1990
name=Harry Hacker,salary=50000.0,hireDay=1 Oct 1989
name=Carl Cracker,salary=75000.0,hireDay=15 Dec 1987
name=Tony Tester,salary=40000.0,hireDay=15 Mar 1990
name=Harry Hacker,salary=50000.0,hireDay=1 Oct 1989
name=Carl Cracker,salary=75000.0,hireDay=15 Dec 1987
```

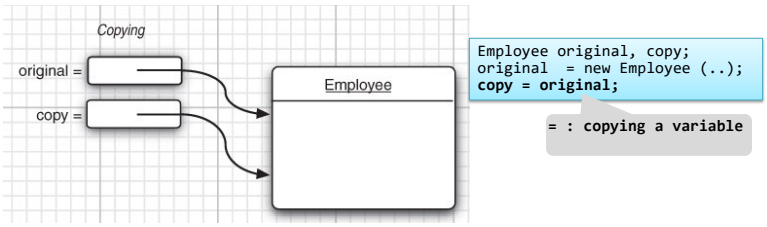
## The Java Cloneable Interface (copying)

- **Introduction to Cloning:**

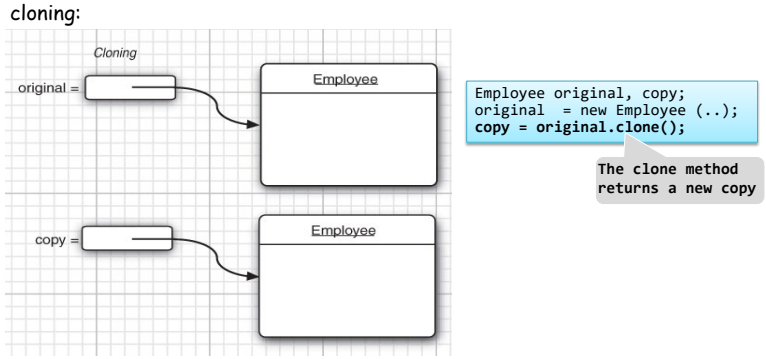
  To **clone** an object, it means to make a new copy of the object. ← **Different from copying**!!!

**Copying** an object variable vs **Cloning** an object

copying :

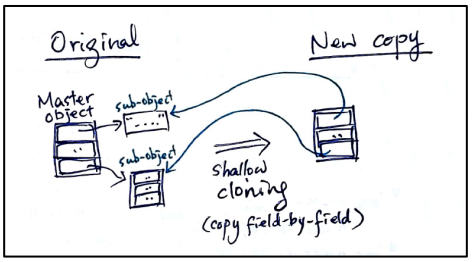



cloning:

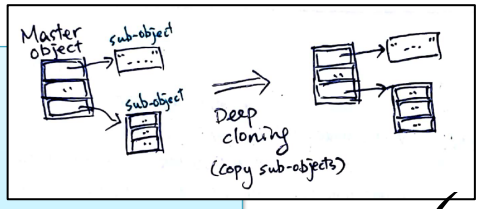



- **To make an object cloneable, we need to**

  - Make the class **implements `java.lang.Cloneable`**
  - Redefine the method : **`public type clone()`**
    - The **`Object` class** provides **`protected Object clone()`** which **copies field-by-field** (**Shallow-cloning**)
      If a field is a reference, it only copies the reference, that **refers to the same subobject**




  - We redefine the clone() method to handle **cloning of mutable subobjects**




```
class Employee implements Comparable<Employee>, Cloneable
{
   ..
   @Override
   public Employee clone() throws CloneNotSupportedException
   {
      Employee copy = (Employee) super.clone();

      copy.hireDay = new Day(
         this.hireDay.getYear(),
         this.hireDay.getMonth(),
         this.hireDay.getDay());

      copy.name = new String(this.name);
      return copy;
   }
}
```

Call the Object superclass's clone() method

Construct a copy for this.hireDay,
or call .clone of this.hireDay (if Day.clone is available)

Actually can be omitted.
Reason: Since strings are immutable, it is Okay to let both original and copy refer to the same string.

---

Why okay? Well, if one changes the name, the *change* is actually to create a new string object. [Ref. Lec05_Ex.pdf]

## Illustration: Correct code

- Super.clone() only performs Shallow-cloning
- So we need to perform deep-cloning for mutable subobjects

```
class Employee implements Comparable<Employee>, Cloneable
{
   ..
   @Override
   public Employee clone() throws CloneNotSupportedException
   {
      Employee copy = (Employee) super.clone();

      copy.hireDay = new Day(
         this.hireDay.getYear(),
         this.hireDay.getMonth(),
         this.hireDay.getDay());

      copy.name = new String(this.name);
      return copy;
   }
}
```

Call the Object superclass's clone() method

Actually can be omitted (see last slide)

Use Cloning

```
public static void main(String[] args) {
   Employee e = new Employee("Carl Cracker", 75000, 1987, 12, 15);
   Employee e2 = e;
   Employee e3 = e.clone();
   e.getHireDay().setDay(1988,1,1);
   e.setName("Helena"); //"Helena" is a new string object
   e.setSalary(88000);

   System.out.println(e);
   System.out.println(e2);
   System.out.println(e3);
}
```

**Output:**

```
name=Helena,salary=88000.0,hireDay=1 Jan 1988
name=Helena,salary=88000.0,hireDay=1 Jan 1988
name=Carl Cracker,salary=75000.0,hireDay=15 Dec 1987
```

## Illustration: Incorrect code

- Super.clone() only performs Shallow-cloning
- So we need to perform deep-cloning for mutable subobjects

**if not done, then the new object copy will refer to subobjects in the original one.**

```
class Employee implements Comparable<Employee>, Cloneable
{
   ..
   @Override
   public Employee clone() throws CloneNotSupportedException
   {
      Employee copy = (Employee) super.clone();

      /* copy.hireDay = new Day(
            this.hireDay.getYear(),
            this.hireDay.getMonth(),
            this.hireDay.getDay());

      copy.name = new String(this.name); */
      return copy;
   }
}
```

Call the Object superclass's clone() method

Actually can be omitted (see last slide)

Use Cloning

```
public static void main(String[] args) {
   Employee e = new Employee("Carl Cracker", 75000, 1987, 12, 15);
   Employee e2 = e;
   Employee e3 = e.clone();
   e.getHireDay().setDay(1988,1,1);
   e.setName("Helena"); //"Helena" is a new string object
   e.setSalary(88000);

   System.out.println(e);
   System.out.println(e2);
   System.out.println(e3);
}
```

**Output:**

```
name=Helena,salary=88000.0,hireDay=1 Jan 1988
name=Helena,salary=88000.0,hireDay=1 Jan 1988
name=Carl Cracker,salary=75000.0,hireDay=1 Jan 1988
```

☹

# III Inner Classes

```
public class OuterClass
{
    private class InnerClass
    {
        .. Fields and methods of InnerClass
    }
    .. Fields and methods of OuterClass
}
```

## Inner Class - Introduction

- An *inner class* is a class defined within its *outer class*.
- Often used as *helping classes*
- Advantage of using Inner class:

  1) Better organization of code:

     The helper class is contained inside the outer class, rather than written separately.

  2) Access across inner/outer classes:
     (a) The outer class have access to the inner class's methods and nonstatic fields[1] (even if they are private).
     (b) The inner class has access to the outer class's methods and fields (even if they are private).

---

([1]) static field cannot exist in a nonstatic inner class, unless initialized with a constant expression

```
private final static int testing=3; //ok
private static int testing=3; //not allowed
```

**Inner Class Example 1.**

Recall:

"The outer class have access to the inner class's methods and nonstatic fields (even if they are private)"

```
class BankAccount                                   Outer class
{
  private class Money                               Inner class
  {
    private String currency;  //e.g. "HKD", "RMB", "NTD", "JPY", "KRW", "USD", "GBP"
    private double value;

    public Money(String c, double b) {currency=c; value=b;}

    @Override
    public String toString() {return currency+" "+value;}
  }

  private Money balance;

  public BankAccount(String currency)
  {
    balance = new Money(currency, 0.00);
  }

  public String getBalance()
  {
    return balance.toString();
  }

  public void addMoney(double incr)
  {
    balance.value += incr;
  }
}
```

The outer class uses the inner class to define an object field

- Use a constructor of the inner class
- Use a method of the inner class
- Access a field of the inner class

An object of the inner class

```
public static void main(String[] args)
{
  BankAccount account =
    new BankAccount("HKD");

  account.addMoney(300);

  System.out.println(
    "Account balance = "
    + account.getBalance());
}
```

Output:

```
Account balance = HKD 300.0
```

## Inner Class Example 2.

Recall:

"The inner class has access to the outer class's methods and fields (even if they are private)."

```
class BankAccount
{
  private class Money
  {
    private String currency; //e.g. "HKD", "RMB", "NTD", "JPY", "KRW", "USD", "GBP"
    private double value;

    public Money(String c, double b) {currency=c; value=b;}

    @Override
    public String toString() {return currency+" "+value+" owned by "+owner;}
  }

  private Money balance; private String owner;

  public BankAccount(String currency, String ow)
  {
    balance = new Money(currency, 0.00); owner = ow;
  }

  public String getBalance()
  {
    return balance.toString();
  }

  public void addMoney(double incr)
  {
    balance.value += incr;
  }
}
```

**A field of the outer class**

**Note: we don't write the outer object like:**

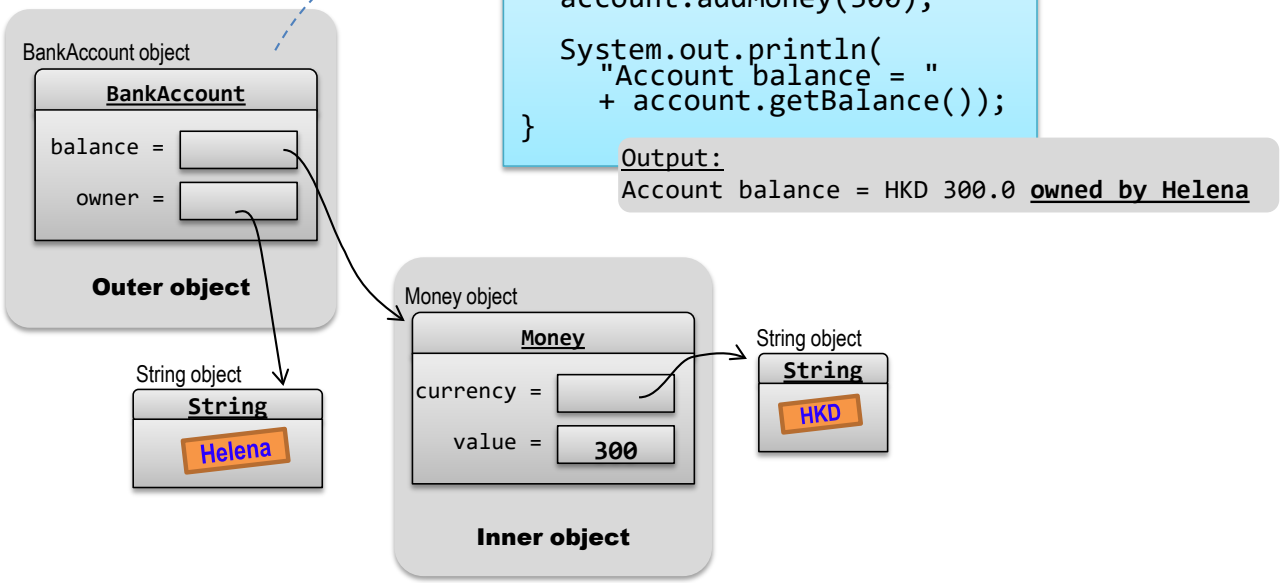
*outer*.**owner**

No need

See the **drawing** in next slide for illustration

```
public static void main(String[] args)
{
   BankAccount account =
     new BankAccount("HKD",
                     "Helena");

   account.addMoney(300);

   System.out.println(
     "Account balance = "
     + account.getBalance());
}
```

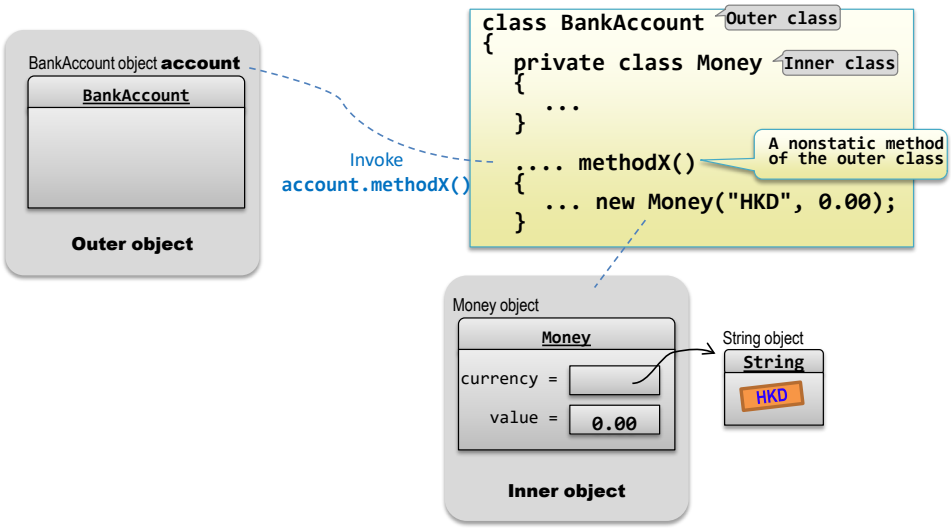
Output:
Account balance = HKD 300.0 owned by Helena



Association between inner class object and outer object

E.g. In a nonstatic method (**methodX**) of the outer class, we create an inner object.

The *implicit parameter* (`this`, or known as the *calling object*) of the call to **methodX**, is then the outer object of the created inner object.




Note: **methodX** can mean the constructor of the outer-class, i.e., the one in example 2 (last page).

**More details about Inner Class (For interested students only)**

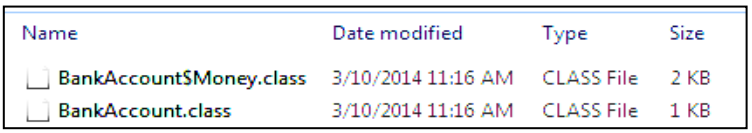
- An inner class can be
  - Nonstatic, like our example :
    - Nonstatic inner class object must arise from an outer class object.
    - Has a connection between an outer class object and the inner class object.
  - Static
    [We do not go into details. Interested students may read Core Java Chp06 / Absolute Java Chp13]
    Static: No connection between outer class object and inner class object.
    (eg. inner class object created in a static method of the outer class)

```
class BankAccount                    Outer class
{
   private class Money               Inner class
   {
      private String currency;
      private double value;

      public Money(String c, double b) {..}

      @Override
      public String toString() {return ..;}
   }

   private Money balance;

   public BankAccount(String currency)
   {
      balance = new Money(currency, 0.00);
   }

   public String getBalance()
   {
      return balance.toString();
   }

   public void addMoney(double incr)
   {
      balance.value += incr;
   }
}
```

```
public class OuterClass
{
   private static class InnerClass
   {
      .. Fields and methods of InnerClass
   }
   .. Fields and methods of OuterClass
}
```

## Other Facts

- Each inner class gets its own .class file:



| Name | Date modified | Type | Size |
|---|---|---|---|
| BankAccount$Money.class | 3/10/2014 11:16 AM | CLASS File | 2 KB |
| BankAccount.class | 3/10/2014 11:16 AM | CLASS File | 1 KB |

- Visibility of Inner Class
  - An inner class can be private, like example 2
  - An inner class can be public.
    If so, it can be used outside the outer class.
    [We do not go into details. For interested students only]

```
BankAccount.Money amount;
amount = account.new Money("USD",123);
System.out.println(amount.toString());
```

- Interesting variations:
  - Nested inner classes
  - Anonymous class
    (want one object only, lazy to give class name; created using `new` in a method, as an inner class)
  - When a class inherits an outer class, the inner class is also inherited.
    [We do not go into details. Interested students may read Core Java Chp06 / Absolute Java Chp13]

--- end ---