Java Inheritance

Rui S. Moreira

Class – ADT (*Abstract Data Type*)

Classes implement the concept of ADT:

- Provide a coherent representation for declaring structured data types together with code for manipulating those data types, i.e.
  - associate in the same concept (cf. class) the member variables (attributes) and member functions (methods)
- Data and code are no longer separated concepts, i.e.
  - objects have inherent behaviour allowing them to change themselves
  - there no separation between data structures and functions for manipulating those structures (as with C or Pascal)

Interface

Interfaces define interaction contracts:

- Interfaces are **mechanisms/devices** used to **interact** with objects, e.g.
  - Interact with a TV through its public interface controls
- Interface is a **collection of methods** which define the **public behaviour** exposed by several object, e.g.
  - get(), set(), move(), area(), etc.
- Interfaces **expose** classes/objects **behaviour** without revealing its internal implementation/details – **encapsulation**
- A class that implements an interface agrees to implement all the behaviour defined by the interface methods – **contract** (otherwise it is abstract)

Interface Definition

Interface

- defines the **signature** of a **set of public methods**
- method defintion has **no body/implementation**

```
/** Defines methods for managing Account Money movements */
public interface AccountMoneyI {
   public double whidraw(double amount);
   public double deposit(double amount);
   public double transfer(AccountMoneyI destiny, double amount);
   public double balance();
}

/** Defines methods for Account Ownership association */
public interface AccountOwnershipI {
   public Client getAccountOwner() throws AccountOwnerNotDefinedException;
   public void setAccountOwner(Client owner);
}
```

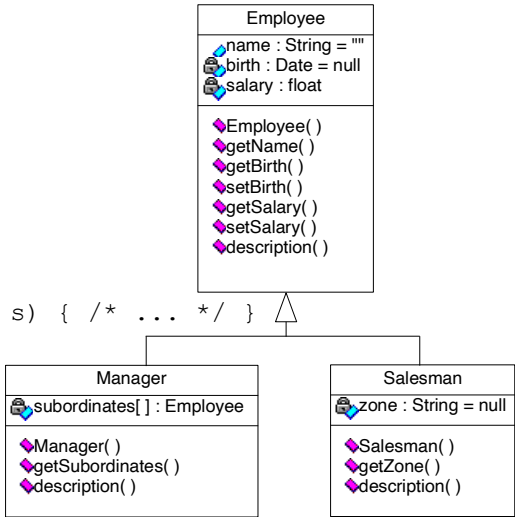
Multiple Interface Inheritance (Realization)

```
// Class implements several interfaces
public abstract class Account implements AccountMoneyI, AccountOwnershipI {
    // Attributes here...
    private final String accountNumber;
    private double balance;
    private Client owner;

    // Implement some get/set methods
    public String getAccountNumber() { return accountNumber; }
    protected double getBalance() { return balance; }
    protected void setBalance(double balance) { this.balance=balance; }
    protected Client getOwner() { return owner; }
    protected void setOwner(Client owner) { this.owner=owner; }

    // Implement only AccountOwnershipI (NOT AccountMoneyI - abstract)
    public Client getAccountOwner()throws ... { /* Impl inside... */ }
    public void setAccountOwner(Client owner) ){ /* Impl inside... */ }

    //...
}
```


Single Class Inheritance

- A Java class can inherit another class (`extends`)

```
/** Super-class Employee
    represents a generic employee */
public class Employee {
    public  String name;
    private Date birth;
    private float salary;

    /** Constructor */
    public Employee(String n, Date b, float s) { /* ... */ }
}

/** Sub-class Manager:
    represents a specific employee */
public class Manager extends Employee {
    private Employee subordinates[];

     /** Constructor */
    public Manager(String n, Date b, float s, Employee[] sub) { /* ... */ }
}
```



Single Class Inheritance (Generalization)

```
// Class extends a single class
public class AccountUnsafe extends Account {

    // Implement methods from AccounMoneyI without safety checks
    public double whidraw(double amount){
          super.setBalance(super.getBalance() - amount);
          return getBalance();
    }
    public double deposit(double amount){
          this.setBalance(this.getBalance() + amount);
         return super.getBalance();
    }
    public double transfer(AccountMoneyI destiny, double amount){
          double b=this.withdraw(amount);
         //To avoid catching the exception we use a cast
         ((AccountUnsafe)destiny).deposit(amount);
         return b;
    }
    public double balance(){ return getBalance(); }
}
```


Class Inheritance

- Java allows only **single class inheritance**, i.e.
  - extend only 1 class at a time (diamond problem)
- Subclasses inherits only **attributes** and **methods** from super-classes, however...
  - **private** attributes and methods of *super-class* are NOT accessible to *sub-class (must use gets/sets instead);*
- **Constructors** are not inherited, therefore sub-classes…
  - must implement its own constructors
  - must explicitly call *super-class* constructors

Operators: `this` and `super`

- Java provides 2 operators:
  - **this**: for referencing the class/object itself
  - **super**: for referencing its super-class/object

```
/** Sub-class Employee: represents a generic employee */
public class Employee {
   /* Atributes */
   public  String name;
   private Date birth;
   private float salary;

   /** Constructor */
   public Employee(String n, Date b, float s) {
        this.name = n;    // SET OBJECT ATRIBUTE name to n
        this.birth = b;   // SET OBJECT ATRIBUTE birth to b
        this.salary = s;  // SET OBJECT ATRIBUTE salary to s
   }
}
```


Operators: `this` and `super`

- Java provides 2 operators:
  - **this**: for referencing the class/object itself
  - **super**: for referencing its super-class/object

```
public class Manager {
   //Use ArrayList instead of basic array []
   private ArrayList<Employee> subordinates = new ArrayList();
   /** Constructor */
   public Manager(String n, Date b, float s, Employee[] subs) {
        super(n, b, s); // EXPLICITLY CALL SUPER-CLASS CONSTRUCTOR
        for(Employee e : subs) this.subordinates.add(e);
   }
   /** Another constructor */
   public Manager(String n, Date b, float s) {
        super.name = n;     // WE MAY SET PUBLIC ATRIBUTES
        //super.birth = b;     NOT ALLOWED TO SET PRIVATE ATTRIBUTES!!!!
        super.setBirth(b); // ALTERNATIVELLY USE PUBLIC METHODS
        super.setSalary(s);// SET PRIVATE ATTRIBUTE VIA PUBLIC METHOD
   }
}
```

Object Polymorphism

- An object can take several shapes (due to inheritance)

```
//Variable emp is of type Employee
Employee emp = null;

//Set reference to a new Manager (Manager is also Employee)
emp = new Manager("John", new Date(2, 08, 1980), 1000, arrayEmps);

//CAN ACCESS all attributes/methods of Employee
s = emp.getSalary(); // THIS IS ALLOWED!!!!

// CANNOT ACCESS attributes/methods since emp is declared Employee
arraySubord = emp.getSubordinates(); // NOT ALLOWED!!!!

//Reference var emp can point to a different object
emp = new Salesman("Mary", new Date(2, 08, 1980), 1000, "Porto");
```


Cast

- **Explicit cast**: force an object to be of a given type

```
//CAN ACCESS attributes/methods of Manager
//by casting/masking object emp to Manager
arraySubord = ((Manager)emp).getSubordinates(); // ALLOWED!!!!

//CAN ALSO ACCESS attributes/methods of Salesman
//by casting/masking the object emp to Salesman
zone = ((Salesman)emp).getZone(); // ALSO ALLOWED!!!!
```

NB:

- We **CAN ONLY CAST** objects that have/inherit the same base-type, i.e., share a super-class!
- We **CANNOT CAST** between different types of objects, i.e., do not share a super-class!

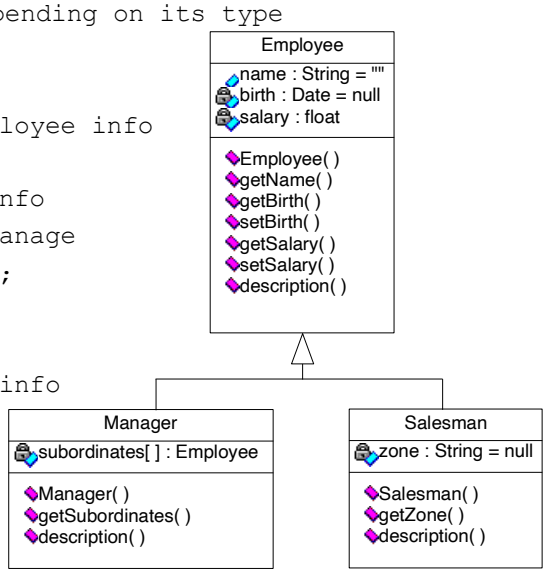
Cast (restore obj shape)

```
// Generic Object variable can point to any kind of object
Object obj = null;
obj = new Rectangulo(p1, p2);
//Or
obj = new Circle(p3);

//Check object type at runtime to cast...
if (obj instanceof Rectangulo){
   // CAST - force rec to reference the Rectangulo object
   Rectangulo rec = (Rectangulo)obj;
   // Now we can access the attributes/methods of Rectangulo
   double area = rec.area();
} else if (obj instanceof Circulo){
   // CAST - force cir to reference the Circulo object
   Circulo cir = (Circulo)obj;
   double raio = cir.raio();
}
```


Object Polymorphism

- Operator **instanceof** (check instance type)

```
// Manage info about an Employee instance depending on its type
public void manageInfo(Employee e){
   if (e instanceof Employee) {
        e.description(); // Prints basic Employee info
   } else if (e instanceof Manager) {
        e.description(); // Prints Manager info
        // Obtain array of subordinates to manage
        sub = ((Manager)e).getSubordinates();
        //...
   } else if (e instanceof Salesman) {
        e.description(); // Prints Salesman info
        // Obtain Salesman zone to manage
        zone = ((Salesman)e).getZone();
        //...
   }
}
```



Method Polymorphism - Overload

- **Overload methods**:
  - same overloaded **method name** can identify different methods
  - each overloaded method must have **different parameter types**
  - each overloaded method can have the **same or different return type**

```
public class PrintClass {
   // Same method name with different parameters/arguments
   public void print(int i){ System.out.print("[int]:i="+i); }
   public void print(float f){ System.out.print("[float]:f="+f); }
   public void print(String s){ System.out.print("[Srtring]:s="+s);
   }
}
```


Method Polymorphism – Override

- Override Methods (redefine behaviour of *super-class* methods):
  - *sub-classes* may add new attributes or methods to *super-class*
  - *sub-classes* can also redefine/override methods of *super-class* (i.e. change the behaviour of super-class methods)

```
//Employee class implements description() which may be overriden
//by sub-classes Manager and Salesman to add specific info
public class Manager {
   @Override
   public String description(){ // Add additional Manager info
        return super.description()+";"+this.subordinates.size();
   }
}

public class Salesman {
   @Override
   public String description(){ // Add additional Salesman info
        return super.description()+";"+this.zone;
   }
}
```

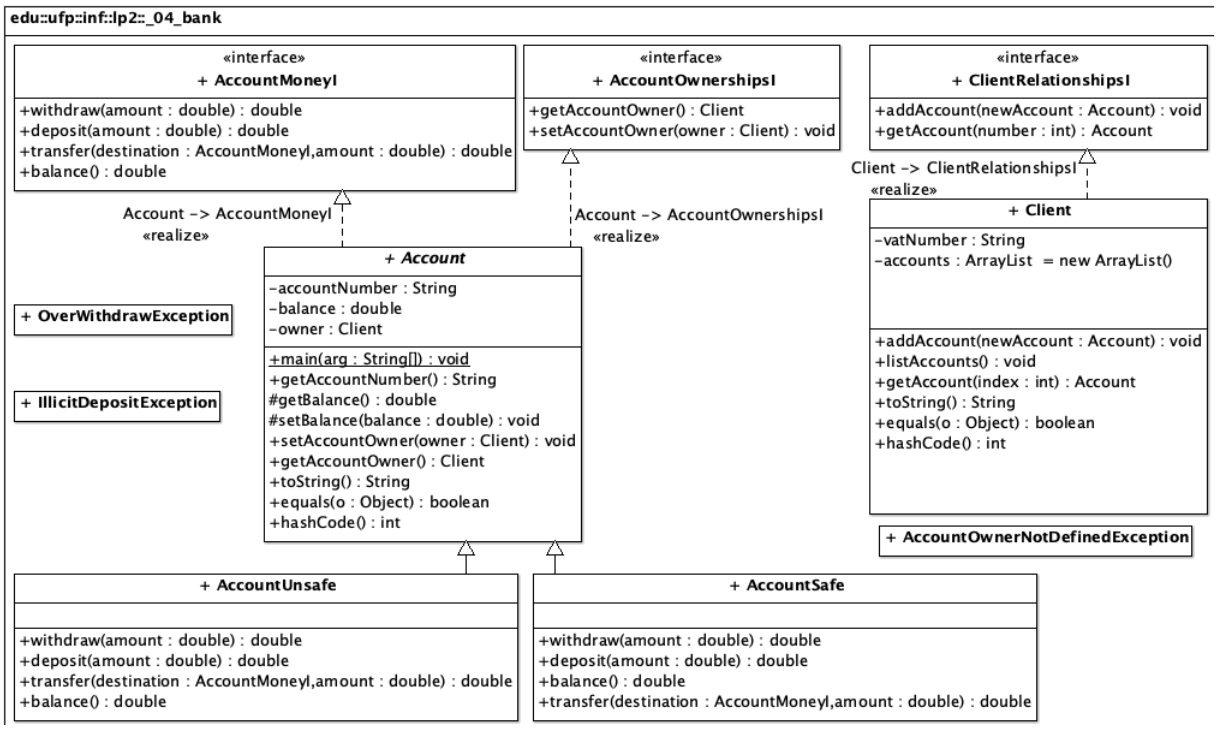
Method Polymorphism – Override

- When calling overridden methods in different sub-classes
  - Will execute different implementations according to the type of object

```
//Calling description() method on different objects...
public static void main(String args[]) {
    Employee e1 = null; //e1 may reference an Employee, Manager or Salesman

    //Instantiate an Employee
    e1 = new Employee("Peter", new Date(2, 08, 1980), 1000.0f);
    //Executes method from Employee object/class to print basic Employee info
    System.out.println(e1.description());

    //Instantiate a Manager
    e1 = new Manager("Alex", new Date(2, 08, 1980), 1000.0f, subordinates);
    //Executes method from Manager object/class to print all Manager info
    System.out.println(e1.description());

    //Instantiate a Salesman
    e1 = new Salesman("Michael", new Date(2, 08, 1980), 1000.0f, "Porto");
    //Executes method from Salesman object/class to print all Salesman info
    System.out.println(e1.description());
}
```


Exercises

- Implement `_04_bank` package to represente bank accounts and owners:
  - Create interfaces (`AccountMoneyI`, `AccountOwnershipsI`, `ClientRelationshipsI`)
  - Create classes (`Account`, `AccountUnsafe`, `AccountSafe`, `Client`)



Exercises

- Implement `_05_figgeo` package to represent geometric figures:
  - Create new interfaces (`FigGeoDimsI`, `FigGeoDrawI`, `FigGeoRelsI`) and classes (`FigGeo`)
  - Refactor some of previous classes from `_01_intro` (`Point`, `Rectangle`, `Circle`, `Triangle`)