(remedial) Java

anastasia.bezerianos@lri.fr

Packages

Packages (1)

```
package path.to.package.foo;
```

- Each class belongs to a package
- Packages groups classes that serve a similar purpose.
- Packages:
  - Help avoid classes with the same name
  - Classes in other packages need to be imported

Packages (2)

```
package path.to.package.foo;
```

- Are just directories (in the disk or jar)
- For example
  - class3.inheritanceRPG is located in
  - <workspace>\RemedialJava\src \class3\inheritenceRPG
  - The compiled classes under …\RemedialJava\bin\...

Packages (3)

- Defining packages

```
package path.to.package.foo;
class Foo {
}
```

the class full name is **package.ClassName**

Packages (2)

- Defining packages

```
package path.to.package.foo;
class Foo {
}
```

the class full name is **package.ClassName**

- Using packages

```
import path.to.package.foo.*;
import path.to.package.Foo;
```

Packages example

```
package neighborhood;

public class Car {
}
```

```
package neighborhood;

public class Road {
}
```

Packages example

```
package citytools;

import neighborhood.Car;
import neighborhood.Road;

public class City {
   public static void main (String[] args){
      Car car = new Car();
      car.move(30);
   }
}
```

Packages example

```
package citytools;

// import neighborhood.Car;
// import neighborhood.Road;

public class City {
   public static void main (String[] args){
      neighborhood.Car car = new neighborhood.Car();
      car.move(30);
   }
}
```

Why packages?

- Combine similar functionality

```
fr.upsud.libraries.Library
fr.upsud.libraries.Book
```

- Separate similar names

```
shopping.List
packing.List
```

Convention (domain.society.project.modules)

```
fr.upsud.remedialjava.class3
```

Why packages?

- All classes "see" classes in the same package (no import needed)
- All classes "see" classes in `java.lang`
  - Such as: `java.lang.String; java.lang.System`
- Java has a LOT of packages/classes. Reuse them to avoid extra work (use your java version)
  - http://docs.oracle.com/javase/8/docs/
  - http://docs.oracle.com/javase/8/docs/api

Collections

Remember our menu orders

```
public class Menu {

    private Dish[] _dishes = new Dish[10];

    …
}
```

- What would happen if we wanted a menu of more than 10 items (even better, a menu that we can periodically add new dishes to)?

Object collections: ArrayLists

- Java comes with several collections, i.e., objects that represent a grouping of other objects. Examples include ArrayLists, Sets, Maps, etc.
  (see Java API on collections for more info)

ArrayList:
Modifiable list (internally implemented with Array)
- Get/put item by index
- Add/delete items
- Loop over items

Object collections: ArrayLists

- Java comes with several collections, i.e., objects that represent a grouping of other objects. Examples include ArrayLists, Sets, Maps, etc.
  (see Java API on collections for more info)

ArrayList: Modifiable list (internally implemented with Array)

- get(index) to access item at index (not array[i])
- Add/delete items (no size restriction)
- Loop over items

ArrayList example

```
public class ArrayListExample {
  public static void main(String[] arguments) {

    ArrayList<String> someStrings = new ArrayList<String>(); // creation

    // adding as many items as we want
    someStrings.add("Alice");
    someStrings.add("Bob");
    someStrings.add("Steve");

    System.out.println( "Size is " + someStrings.size() ); // can get its size

    System.out.println(someStrings.get(0)); // can get items in index position 0
    System.out.println(someStrings.get(5) ); // error when outside

    for (int i = 0; i < dishes.size(); ++i ) // loop using index
        System.out.println(someStrings.get(i));

    dishes.set(2, "Peter");// replace item at index

    for (String s: someStrings) // another loop over collection
        System.out.println( s ); // that gives us references
  }
}
```

Exercise 1 (part one & two)

www.lri.fr/~anab/teaching/remedialJava/ex-class3.pdf

Reminder:

- Import classes from other packages using **import package.classname;**
- ArrayList:
  - `ArrayList<Class> someName = new ArrayList<Class>();`
  - `get(index)`: returns object in index pos
  - `set(index,object)`: sets object in index pos
  - `add(object)`: adds an object at the end of the ArrayList
  - `remove(?)`: removes object at either index, or object itself
  - Loop using index, or over elements `for(Class c: someCollection)`

Object collections: Maps

- Stores a (*key, value*) pair of objects
- Look up the *key*, get back the *value*
- *Key* needs to be unique

Example:

- Address book: map name to email address
- Menu: map dish number to Dish object

- TreeMap (sorted), HashMap (pseudorandom)

Map example

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

    HashMap<String,String> emails = new HashMap<String, String>(); // creation

    emails.put("John", "john@doe.com");    // adding entry of type String:String
    emails.put("Cat", "cat@withemail.org");

    System.out.println( emails.size());    // gets size

    for (String k : emails.keySet())       // gets all keys
        System.out.println(k);

    if ( emails.containsKey("John") )      // checks if key exists
        emails.replace("John", "johnnie@gmail.com");   // replaces value

    for (String v : emails.values())       // gets all values
        System.out.println(v);

    if (!emails.isEmpty())                 // checks if empty
        System.out.println("I have stuff!");

    emails.remove("Cat");     // removes key

    for (HashMap.Entry<String,String> pairs: emails.entrySet())
        System.out.println(pairs);    // iterates over all entries
    }
}
```

A bit more on methods

A bit more on methods

- Methods (including constructors) can be overloaded (multiple methods, same name)

  (one way of ensuring polymorphism in Java)

```
public Car {
    this.Car("unknown", "white"); // I want a default value
}

public Car(String myname, String mycolor){
    name = myname;   // or this.name = myname;
    color = mycolor; // or this.color = mycolor;
}
```

Overloading

- The compiler searches for the best method, based on the **TYPE** of the **arguments passed**

```
public class PrintStream {
    public void println(String text){
    ...
    }
    public void println(double value){
    ...
    }

    public static void main (String args[]){
        PrintStream out = System.out;
        out.println("toto");
        out.println(3.0);
        out.println(2);
    }
}
```

Overloading

- The return TYPE of the method is not considered, so it is impossible to differentiate methods just by their return TYPE

```
public class BadOverloading {
    public int f(){
    ...
    }
    public double f(){
    ...
    } // f is already defined in BadOverloading

    public static void main (String args[]){
        BadOverloading bo = new BadOverloading();
        bo.f();
    }
}
```

Overloading, when to use?

- We use overloading when methods have the same semantic (purpose), but different arguments

```
public class Math {
    public float sqrt(float value){…}
    public double sqrt (double value){…}
} // this is ok

public class List {
    public void remove (Object value) {…}
    public void remove (int index) {…}
} // if we want to be picky, this is NOT ok, why?
```

Varargs

- It is possible to define methods with variable number of arguments using the notation "…"
- Arguments are put in a table

```
public class PrintStream {
   public void printf (String text, Object... args){
      ...
   }
}
```

```
   public static void main (String[] args) {
      PrintStream out = System.out;
      out.printf("%d\n",2);
   }
```

Varargs

- Varargs are considered as tables, so we cannot overload varargs and tables

```
public class VarargsOverloading {
   private static int min(int... array){ }

   private static int min(double... array){ }

   private static int min(int[] array){
   } // min(int...) already defined in VarargsOverloading
}
```

Exercise 1 (part 3)

www.lri.fr/~anab/teaching/remedialJava/ex-class3.pdf

Reminder:

- Syntex for varargs

  method **TYPE** (other input params, **TYPE**… name)

Inheritance

A basic inheritance example

- Imagine a very simple RPG game

```
public class Person {
    public String _name;
    public int _health = 100;
    public int _mana = 0;

    public void sayName (){
        System.out.println(_name);
    }

    public void attackPerson(Person target){
        System.out.println(_name + " attacking " + target._name);
        target._health -= 10;
    }
}
```

A basic inheritance example

- We now want to create a Wizard

```
public class Wizard {
    // need to copy all of Person staff
    // booooring
}
```

- Why go into all the trouble? Wizard has and does everything that a simple Person does …

A basic inheritance example

- Wizard is a subclass of Person

```
public class Wizard extends Person {

}
```

A basic inheritance example

- Wizard is a subclass of Person
  - Wizard can use everything* that Person has

    e.g., `wizard1._health +=1;`

*except for private fields and methods (keyword protected may be used)

A basic inheritance example

- Wizard is a subclass of Person
  - Wizard can use everything* that Person has
    e.g.,`wizard1._health +=1;`
  - Wizard can do everything* that Person can do
    e.g.,`wizard1.attackPerson(person1);`

*except for private fields and methods (keyword protected may be used)

A basic inheritance example

- Wizard is a subclass of Person
  - Wizard can use everything* that Person has
    e.g.,`wizard1._health +=1;`
  - Wizard can do everything* that Person can do
    e.g.,`wizard1.attackPerson(person1);`
  - You can use Wizard as a Person too
    e.g., `person1.attackPerson(wizard1);`

*except for private fields and methods (keyword protected may be used)

A basic inheritance example

- Let's improve Wizard

```
public class Wizard extends Person {

    int _mana = 100;
    ArrayList<String> _spells;

    public void cast (String spell){
        // do cool stuff here
        _mana -= 10;
    }
}
```

A basic inheritance example

- Can we inherit from inherited classes?

```
public class GrandWizard extends Wizard {



}

grandWizard1.name = "Gandalf";
grandWizard1.attackPerson (person1);
grandWizard1.sayName();
```

How does Java do that?

- What Java does when it sees `grandWizard1.sayName();`
  - Look for `sayName()` in GrandWizard
  - It's not there! Does GrandWizard have a parent?
  - Look for `sayName()` in Wizard
  - It's not there! Does Wizard have a parent?
  - Look for `sayName()` in Person
  - Got it! Call `sayName()` and print the field `_name`

Inheritance structure

Person:
Parent of Wizard, Elf, …

Subclasses of Person

Subclass of Wizard

Can only inherit from one class

Can only inherit from one parent class

Multiple inheritance not possible …

Why?

A basic inheritance example

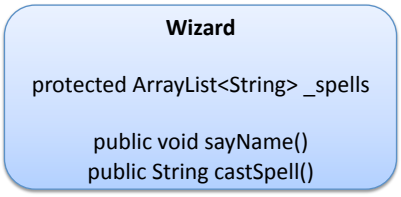
- Adapting methods from inherited classes

```
public class GrandWizard extends Wizard {

    public void sayName (){
        System.out.println("Grand Wizard"  + name);
    } //overriding (ancestor) parent method
}

grandWizard1.name = "Gandalf";
grandWizard1.attackPerson (person1);
grandWizard1.sayName();
```

Exercise 2 - part one & two

www.lri.fr/~anab/teaching/remedialJava/ex-class3.pdf

Reminder:



- Part one

  example of a single class diagram

- Part two

  inheritance between classes

```
class CHILDCLASS extends PARENTCLASS {

}
```

  a child class can override (i.e., redefine) a method from a parent/ancestor class by redefining it in its body

Access to parent information

- A class inherits all the members of the parent (super) class, i.e., all fields and methods and can use them without having to do anything more.

Access to parent information

- A class inherits all the members of the parent (super) class, i.e., all fields and methods and can use them without having to do anything more.
- All except Constructors, that are not inherited
  - Unless the default constructor is used, constructors need to be redefined for each child (more later).

Access to parent information

- Methods are inherited.
- Adding a *method of the same name* as one in a super-class redefines (overrides) the method. This way we have implementations adapted to the semantics of the method.

```
class Person {
  public void sayName (){
    System.out.println(_name);
  }
}
```

```
class GrandWizard extends Wizard {
  public void sayName (){
    System.out.println("GW" + _name);
  }
}
```

Access to parent information

- Fields are inherited
- When we add a *field of the same name* as one in a parent class, the field of the super class is no longer visible to us (we say it is hidden or masked)

```
class Person {
  public int _name;
  public int sayName (){
    System.out.println(_name);
  }
}
```

```
class Wizard extends Person {
  public String _name;
  public String sayName (){
    System.out.println("GW" + _name);
  }
}
```

Access to parent information

- What if we want to access methods or fields from the super class, when we have overridden the methods or masked the fields (e.g., use the `sayName()` of Person for the GrandWizard Zed?

Access to parent information

- We use the keyword **super**

```
class Wizard extends Person {

  public String sayName (boolean modest){
   if (modest)
      super.sayName();
   else
      System.out.println("GW" + _name);
  }
}
```

Access to parent information

- We use the keyword **super**

```
class Wizard extends Person {

  public String sayName (boolean modest){
   if (modest)
      super.sayName();
   else
      System.out.println("GW" + _name);
  }
}
```

- Only works for immediate parent and non-static methods (they belong to their class)

Access to parent information

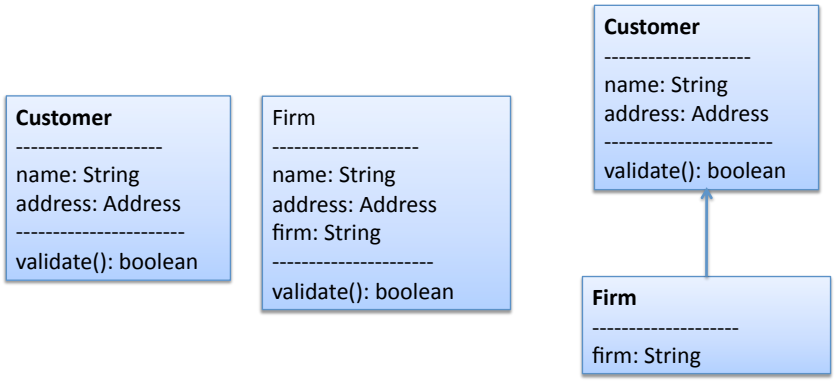
- We use the keyword **super**
- Constructors are not inherited.
  - Unless the default constructor is used, constructors need to be redefined for each child.
  - First line in the constructor needs to be a call to the constructor of the parent class, using **super**

```
class Person {
 public Person(String n){
  _name = n;
 }
}
```

```
class Wizard {
 public Wizard(String name, int m) {
   super(name);
    _mana = m;
 }
}
```

Inheritance

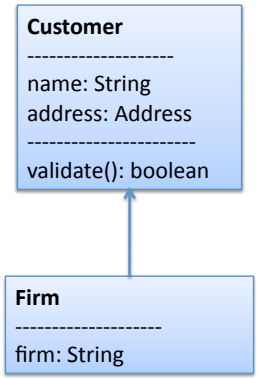
Another example:



In reality, true inheritance is rare …

Inheritance, reasons

- We use inheritance to:
  - Reuse members (structural), e.g., Customer & Firm
  - Redefine some methods, e.g., change the code to validate()
  - Express sub-types, e.g., Wizard is a Person



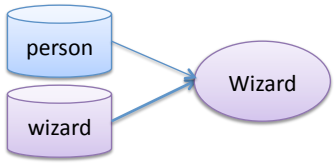
Upcasting, downcasting

- Upcasting: the ability to consider a reference of a child (sub-class) as if it is a reference of a parent-(super-class).
- In Java done automatically.

Upcasting, downcasting

- Upcasting: the ability to consider a reference of a child (sub-class) as if it is a reference of a parent- (super-class).
- In Java done automatically.

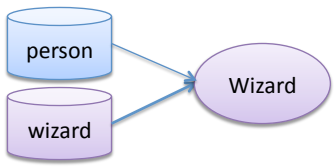
```
public static main (String[] args){

    Wizard wizard = new Wizard();
    System.out.println(wizard);

    Person person = wizard;
    System.out.println(person);
}
```



Upcasting, downcasting

- Upcasting: the ability to consider a reference of a child (sub-class) as if it is a reference of a parent- (super-class).
- In Java done automatically.

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

    Wizard wizard = new Wizard();
    System.out.println(wizard);

    Person person = wizard;
    System.out.println(person);
}
```



- Note: the object is still a Wizard, but we can treat it as a Person (e.g., add it in an `ArrayList<Person>`)

Upcasting, downcasting

- Downcasting: the ability to consider a reference to a parent (super-class) as if it is a reference of to a child (sub-class)

Upcasting, downcasting

- Downcasting: the ability to consider a reference to a parent (super-class) as if it is a reference of to a child (sub-class)

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

    // assuming I know about Zed
    for (Person p: persons){
        if ( p.getName().equals( "Zed" ) ){
            Wizard zed = (Wizard)zed;
        }
    }
}
```

Upcasting, downcasting

- Downcasting: the ability to consider a reference to a parent (super-class) as if it is a reference of to a child (sub-class)

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

    // assuming I know about Zed
    for (Person p: persons){
        if ( p.getName().equals( "Zed" ) ){
            Wizard zed = (Wizard)zed;
        }
    }
}
```

- If you want to check if an object belongs to a class can use **instanceof,** e.g., `(p.instanceof Wizard)`

(aside) OOP Polymorphism

- Polymorphism is the ability for something ☺ to take many forms. In Java it can be seen in:
  - Objects: because we can upcast, an object can be seen as being of the class of (one of) it's ancestors
  - Methods: because we can override a method that we inherit to adapt its behavior
  - Methods: because we can overload a method (i.e., have a method with same name but different input parameters) to adapt its implementation to the context of the input parameters

Exercise 2 – part three

www.lri.fr/~anab/teaching/remedialJava/ex-class3.pdf

Reminder:

- Constructors are not inherited
- You can access the parent's constructor with **super**
- In the constructor, super needs to be the first command

Inheritance

- In Java all class are inheriting directly or indirectly from the class `java.lang.Object`
  - Directly: when we declare a new class without inheritance, the compiler adds `extends Object`
  - Indirectly: when we declare a new class that inherits from another one, the parent class inherits (directly or indirectly) from the class Object

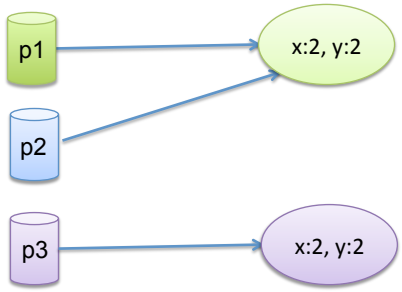
Object class

- Parent class of all Java classes
- Has basic methods (we can redefine/override)
  - toString() & hashCode()
  - equals()
  - getClass()
  - clone()
  - finalize()

Equality tests

- The operators == and != test
  - Values for primitive/basic types (e.g., int, double)
  - Values of references for Objects

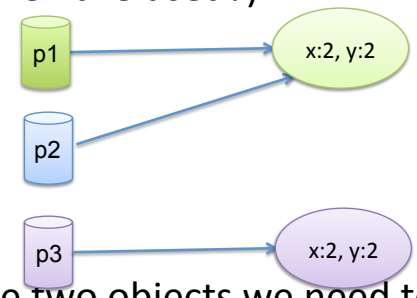
```
Point p1;
p1=new Point(2,2);
Point p2;
p2=p1;
Point p3;
p3=new Point(2,2);
p1==p2; //true
p1==p3; //false
p2==p3; //false
```



equals()

- There is a method equals(Object) in Object
- But generally its implementation tests references (what is an exception we have used?)

```
Point p1=new Point(2,2);
Point p3=new Point(2,2);
p1==p3;         //false
p1.equals(p3); //false
```



- To structurally compare two objects we need to redefine (override) the method equals().

Advanced Inheritance

Can only inherit from one class

Can only inherit from one parent class

(not entirely true …)

If both Wizard and Elf implement (differently) `sayName()`, which one does WizardElf call?

Interfaces (not UI)

But what if the method was not implemented?

Interfaces (not UI)

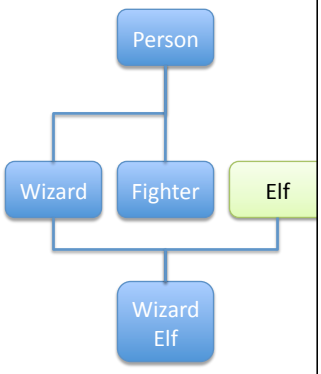
- Interfaces define a type (as do classes) but without code. We use the keyword **interface**

```
public interface Elf {
    public void seeInTheDark(); // simple declaration, no code
    public String otherCoolStuffICanDo ( int coolness );
}
```

- In the interface you can declare methods that subclasses will then have to implement
  - We call these abstract methods
  - They are always declared public

Interfaces

Then we need to define a class that inherits the declarations of the abstract methods in the interface, and provides code for them



We use the keyword **implements**

Interface implementation

```
public interface Elf {
    public void seeInTheDark(); // simple declaration, no code
    public String otherCoolStuffICanDo ( int coolness );
}


public class WizardElf extends Wizard implements Elf{

    public void seeInTheDark(){
        System.out.println("I see as well as a cat" );
    }

    public String otherCoolStuffICanDo ( int coolness ){
        if ( coolness < 5 )
            System.out.println("Not so cool after all" );
    }
}
```

Interfaces

- The compiler verifies that once a class implements an interface, it implements **all** its methods
- A class that implements an interface, is a sub-type of the interface (and all the cool upcasting works here as well)

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

    ElfWizard elw = new ElfWizard();

    Wizard wizard_alias = elw;
    Elf elf_alias = elw;
}
```

Interfaces

- An interface has no code (i.e., is not implemented) and so cannot be instantiated (i.e., we cannot create objects of their type using **new**)
- The compiler verifies that once a class implements an interface, it implements **all** its methods
- A class can implement more than one interfaces, and thus inherits their combined declarations
  - This is the closest Java comes to multiple inheritance

Interfaces

```
public interface Elf {
    public void seeInTheDark(); // simple declaration, no code
}
public interface Running {
    public int run( int time ); // simple declaration, no code
}


public class WizardElf extends Wizard implements Elf, Running{
    private int _distance = 0;

    @Override
    public void seeInTheDark(){ // inherited from Elf
        System.out.println("I see as well as a cat" );
    }
    @Override
    public int run ( int time ){// inherited from Running
        _distance += time*10;
    }
}
```

Interface or Class Inheritance

- Inherit from a class:
  - The subclass is a type of the parent class
- Implement an interface:
  - Enforce the implementation of specific methods (i.e., shared functionality)
  - Objects that share different groups of functionalities

(a call to interface methods is a bit slower)

Exercise 3

www.lri.fr/~anab/teaching/remedialJava/ex-class3.pdf

Reminder:

- Interface creation: `public interface INTERFACENAME{}`
- Interface only has (abstract) method declarations, all public
- A class implements one or more interfaces

  `class CLASSNAME implements INTERFACENAME{}`
- A class that implements an interface needs to **provide code for all** the abstract methods defined in the interface (the keyword @Override proceeds their code)

Abstract classes

- It is possible to create in Java classes with abstract methods, i.e., methods that child classes need to implement
  - Abstract classes can also have fields and methods with implementation as well, so it is a partially implemented class
  - They can inherit normally from other classes or implement interfaces
- BUT they cannot have instances (i.e., we cannot create an object from them using **new**)

Abstract classes

```
public interface Moving {
    public int run( int time );
    public int walk( int time );
}
public interface Resting {
    public void stand( int time );
    public void sleep( int time );
}

public abstract class Person implements Moving,Resting {
    String _name;
    int _distance;
    public int run( int time ){
        distance += time*10;       // implemented method
    }
    public abstract void sayName(); // declaration only
}

// What does Wizard have to do?
public class Wizard extends Wizard {
}
```

Type and Class: reminder

Aside note:

- Variables have a Type
- References have a Class

```
public static void main (String args[]){
   String s = args[0];
   Car k = new Car (“peugeot”, “pink”);
}
```

Type Class

Types vs classes

```
public interface Moving { … }
public interface Resting {… }
public abstract class Person implements Moving,Resting {…}
public class Wizard extends Person { … }
public class Fighter extends Person { … }
```

- Which of the following are correct?

```
Person p = new Person("John");
Wizard z = new Wizard("Zed");
Fighter f = new Fighter("Fred");
Resting r = new Resting("Resting");

Person p2 = z;

ArrayList<Person> people = new ArrayList<Person>();
people.add(z);
people.add(f);
for (Person ip: people){
   if (ip instanceof Wizard) {…}
}
```

Object collections: ArrayLists

(Aside)

- ArrayLists are one type of List (that is an interface, more on this later). Often it is best to declare the type as List and instantiate (i.e., implement) using ArrayList.
- This can make easier in the future to change the implementation of your List (to use for example LinkedList)

  (see Java API on collections for more info)