G51PRG:
Introduction to Programming
Second semester

Lecture 4

Natasha Alechina
School of Computer Science & IT
nza@cs.nott.ac.uk

Previous lecture

- Static modifier
- extending classes
- superconstructing

Plan of the lecture

- A bit on the current exercise: class definitions, file names, testing code
- overriding methods
- **`super`** keyword
- polymorphism

Files and classes

- You can define several Java classes in one file
- For example, in a file called File.java, you can have

```
class A {
  …
}
class B {
   …
}
```

- proviso: at most one class can be declared public. If for example B is declared public, the file should be called B.java

Book class and Textbook class

- For the current exercise, classes should be defined in separate files.
- This is usually a good practice anyway.
- So you need to have Book.java file with the definition of the Book class and a Textbook.java file with the definition of the Textbook class.

Testing

- You are asked to write a class definition (fields, constructor and some methods).
- You need to check if it is correct: constructor sets the fields, methods work as they should.
- How do you go about writing the testing code and where do you put it?

Testing code

- You need to run the tests, so you need a main() method.
- In general, it will call the methods you are testing and perhaps some other methods you need for systematic testing (for example, to read test data from files, generate testing data…)
- For very simple programs like Book and Textbook, the main can do all the work.
- Where do you put the main method? In the class you are testing or in a separate class?

Where to put the main() for testing

- Occasionally you need to access private fields of your class when testing. For example, if you want to test the constructor which sets private fields. Or when you are trying to understand how Java works and just experiment with inheritance and private fields. In this case the only place to put the main() is the class itself.
- Normally what you want to test is the class behaviour as seen by other programs, that is, non-private members of the class. Then it is better to write a separate class for testing and put the main() there.

How to test Book and Textbook (1)

- You need to test if your constructor works: whether it sets the class fields to correct values and whether the inventory number is assigned correctly.
- Obviously you need to create several objects (otherwise no way to test the inventory numbers).
- You need to create several books and textbooks, and you need to check that they get numbers in sequence: book1 gets number 1, textbook1 gets number 2, book2 gets number 3 etc. If you ask for book1's number after you created more books and textbooks, it should still have inventory number 1.

How to test Book and Textbook (2)

- The easiest way to achieve this is to hardwire object creation and testing in the main. You can call print() method to display the details of the book/textbook and you can also put in print statements which tell you what the expected outcome should be:

```
Book book3 = new Book("A","T3","P","Y");
System.out.println("Test 3: expected
  outcome A,T3,P,Y, number 3");
book3.print();
```

How to test Book and Textbook (3)

```
Book book3 = new Book("A","T3","P","Y");
System.out.println("Test 3: expected
  outcome A,T3,P,Y, number 3");
book3.print();
Textbook t = new
  Textbook("AA","TT","PP","YY", "CC");
System.out.println("Test 4: expected
  outcome AA,TT,PP,YY,CC number 4");
t.print();
System.out.println("Test 5: expected
  outcome A,T3,P,Y number 3");
book3.print();
```


How to test in general

- Think of all behaviours which you need to test.
- Construct test cases (after I call this method - the outcome should be like that).
- Put test cases in the main (print statement with expected outcome - then display the actual result).
- When there are too many test cases to produce and check manually, generate a test file with data and get the program to save all failed tests to another file instead of matching outputs on the screen yourself as above.

Using the parent's constructor

- Usually, a constructor in the child class will need to set more values than in the parent class
- Can call super() to do the work which the constructor in superclass does
- Using a constructor from a superclass is called superconstructing.

Using the parent's constructor

- For example, if Point has the following constructor:

```
Point(int x, int y){
    this.x = x;
    this.y = y;
 }
```

- Then we could add the following constructor to Pixel

```
Pixel(int x, int y, Color color){
    super(x,y);
    this.color = color;
 }
```

Changing parent's methods and fields

- Extending classes would have been too restrictive if you could only inherit a method exactly as it is and could not change or expand it.
- We can override the parent's method completely or use it as part of the implementation of the child's method.

Points and Pixels again

```
class Point {
   public int x,y;
   public void clear() {
      this.x = 0;
      this.y = 0;
   }
}
class Pixel extends Point {
   Color color;
   public void clear() {
      super.clear();
      color = null;
   }
}
```

Keyword **`super`**

- **`super`** references fields and methods from the superclass just as **`this`** references fields and methods from the current object.
- **`super.clear()`** means the **`clear()`** method from the superclass.
- When **`super.something`** is invoked, the runtime system looks back up the inheritance hierarchy to the first superclass which contains the required **`something`** .

Overriding

- We can totally change the implementation of some method in the subclass (don't have to call **`super`** first!).

Points and Pixels (3)

```
class Point {
…
   public boolean hasColor() {
      return false;
   }
}
class Pixel extends Point {
   Color color;
…
   public boolean hasColor() {
      return true;
   }
}
```


Changing fields in the subclass

- Just as methods inherited from the superclass can change, fields can be modified as well.
- For example, we could change the type of x and y in Pixel from int to short:

```
class Pixel extends Point{
      short x,y;
}
```

- or even just declare them again as ints: this will produce new fields also called x and y:

```
class Pixel extends Point{
      int x,y;
}
```

Changing fields in the subclass

- In general, subclass and superclass can have a field with the same name but different type or holding different values. In this case the field of superclass is hidden .
- You can access the parent's value of **x** with **`super.x`** (if it's not private) as opposed to **`this.x.`**

Changing fields in the subclass

- For example:

```
class A {
  int x = 0;
}
class B extends A {
  int x;
  public B(int a) {this.x = a;}
  public int getX() {return this.x;}
  public int getSuperX(){return super.x;}
}
```

Changing fields in the subclass

- Then if we do

```
  B obj = new B(5);
  System.out.println(obj.getX());
// will get 5
  System.out.println(obj.getSuperX());
// will get 0
```


Polymorphism

- So far it was all about code re-use in class definitions.
- Another great advantage of inheritance is code re-use due to ***polymorphism.***
- Polymorphism means that the type system is flexible. If a method expects an argument of type A, for example

```
distance(A obj)
```

- then sometimes you can pass it arguments of another type:

```
B obj = new B(5);
distance(obj)
```

and it will still compile and run.

Polymorphism in Java

- Different languages have different degrees of flexibility in type systems.
- Prolog, LISP, some scripting languages are very relaxed.
- Haskell is pretty strict.
- Java has ***inheritance polymorphism:*** you can use objects of subclasses where the expected type of argument is an object of a superclass.
- Rule of thumb: if Pixel extends Point then ***it is a special kind of*** Point. So where Points are required, Pixel will do just as well (not vice versa: where Pixels are required, Points may not have enough information to work).

Example

- We have a **`distance (Point p)`** method in the Point class. Pixel extends Point.
- So we if we have an object of type Pixel, we can pass it to the **`distance`** method:

```
Point point = new Point(4,5);
Pixel pix = new Pixel(5,6,Color.red);
System.out.println(point.distance(pix));
```

- so we do not need to write four separate methods to find distance between two points, two pixels, a point and a pixel, a pixel and a point. One method works!

Another example

- If we have an array designed to hold references to Points, we can put a Pixel in that array.

```
Point[] points = new Point[3];
Pixel pix = new Pixel(5,6,Color.red);
points[0] = pix;
```

- Note that if you try to stick a String or a Person into an array of Points, the compiler would not let you.
- Also, if you have an array of Pixels, you cannot put a Point there.

General rule

- Class B extends A
- all Bs are a special kind of As
- you can use Bs where As are expected.

Further reading

- Sun Java tutorial:

http://java.sun.com/docs/books/tutorial/java/javaOO/index.html