COMP6700/2140 Inheritance

Alexei B Khorev and Josh Milthorpe

Research School of Computer Science, ANU

March 2017

Topics

1. Class declaration code reuse
2. Constructor revisited, `super`
3. Abstract classes
4. Polymorphism

Inheritance: code re-use

The ability to *re-use* existing code (in the form of class definitions), or *inherit* previous definitions in Java is realised via the mechanism of inheritance. A new *subclass* (*child, derived*) can be defined to inherit the definition of an existing class (*superclass, parent*) and also to add new class members and change the inherited ones.

```
class A {
   protected int oldField;
   public void oldMethod();
}
class B extends A {
   ... A's public/protected members are included
   ... their access modifiers can be only changed to
   ... raise their visibility, eg protected -> public
   public B(..) { // this is B's constructor
      super(..); // call to parent constructor
      newField = some_value; }
   private int newField;
   public void newMethod();
}
```

Everything (fields, methods, constructors) from the parent class can be accessed in the child class using the parent reference `super` (an example of constructor's re-use is Manager.java. Call to `super()` is always a good practice in every child constructor.

A PIE of OO

The Object-Orientation paradigm is like a pie (using Peter van der Linden's simile):

The most important aspects of the OO paradigm:

- **A**bstraction — deal with multitude of object types as if they were one

A PIE of OO

The Object-Orientation paradigm is like a pie (using Peter van der Linden's simile):

The most important aspects of the OO paradigm:

- **A**bstraction — deal with multitude of object types as if they were one
- **P**olymorphism —use objects of various type as if they were same

A PIE of OO

The Object-Orientation paradigm is like a pie (using Peter van der Linden's simile):

The most important aspects of the OO paradigm:

- **A**bstraction — deal with multitude of object types as if they were one
- **P**olymorphism —use objects of various type as if they were same
- **I**nheritance — reuse and change a type and still retaining one

A PIE of OO

The Object-Orientation paradigm is like a pie (using Peter van der Linden's simile):

The most important aspects of the OO paradigm:

- **A**bstraction — deal with multitude of object types as if they were one
- **P**olymorphism —use objects of various type as if they were same
- **I**nheritance — reuse and change a type and still retaining one
- **E**ncupsulation — hide behind an interface allowing changes in object's internals

Inheritance: Constructor's work

What happens when the constructor is called?

When an object is created, memory is allocated for all its fields, including those which are inherited from superclasses (parent and all the ancestors all the way up to the *Object* class — *Abraham* of Java's class hierarchy). The class fields are set to their respective default values *before* the constructor's phases begin:

1. The superclass constructor — either default `super()` or one with parameters `super(...)` — is invoked;
   - phases 1,2,3 of the previous level are executed based on the code of the superclass;
2. The class fields are initialised using their initialisers and *initialisation blocks* (the blocks of code *outside* the constrictors or class methods which initialise the fields of the object; see the example Body.java) in the same order in which they are declared in the class
3. The rest of the constructor body — everything **after** `super()` — is executed (that's why `super()` should be the first statement inside the constructor body, or be absent altogether);

Explicit constructor invocation: One class constructor can invoke another constructor of the *same class* by calling `this(...)`. `super()`, or `this()` must be the first constructor statement. `this()` helps to write a reusable constructor code only once, and then invoke it by calling `this(arg1,arg2,…)` in other constructors when same initialisation is required.

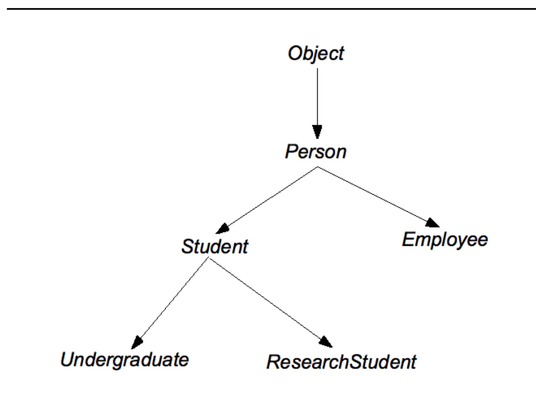
Important: if a class can be extended, do not call any non-final non-private methods inside its constructor!

Constructor Phases

This table shows the values which the Y subclass (of the class X) fields have at different stages of the `Y()` constructor executions. The source code for the parent class X.java and its child Y.java is good for understanding what is gong on:

Step	What Happens	xMask	yMask	fullMask
0	Fields set to defaults	0	0	0
1	Y constructor invoked	0	0	0
2	X constructor invoked	0	0	0
3	*Object* constr. invoked	0	0	0
4	X field initialised	0x00ff	0	0
5	X constructor executed	0x00ff	0	0x00ff
6	Y field initialised	0x00ff	0xff00	0x00ff
7	Y constructor executed	0x00ff	0xff00	0xffff

Inheritance Hierarchy



In an inheritance hierarchy, every class from a sub-tree represents a *subtype* of the type represented by the root of sub-tree. Type and its subtype are in **"IsA"** relationship.

Types and Classes from Inheritance Hierarchy

Type is determined by declaration, while the class is set by instantiation. The two need not be the same, but they cannot be arbitrary:

```
class Person { void justLive(); }
class Student extends Person {
    void study();
}
class Undergraduate extends Student {
    void study() { /* just study */ };
}
class ResearchStudent extends Student {
    void study() { /* do research */ }
}
class Employee extends Person {
    void work();
}
```

```
Person p = new Student("Jack Sparrow"); // upcasting, OK: Student IsA Person
Student st = new Person("Bill Turner"); // illegal, class cast exception
Student st = p; // unsafe, p may not be Student type
p.study(); // illegal, downcasting, needs explicit cast
((Student)p).study(); // OK if p IsA Student, or class cast exception is thrown
Employee e = new Student("Barbossa"); // illegal, incompatible types
```

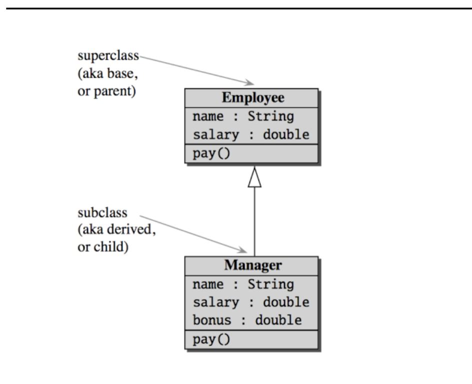
Inheritance: Overriding

- Every class in Java inherits
  - either explicitly via `extends`
  - or implicitly from the *Object* class
- Hence, all public and protected methods from the *Object* class are present in *all* classes
- Two such methods — `equals()` and `hashCode()` play important role, they often must be overridden (will be discussed later)
- The inheritance does not only allow to re-use existing code — it also let you modify an inherited definition to meet new requirements
- This is achieved through *overriding* of inherited methods — providing a new implementation without changing the signature and the contract (to achieve the polymorphism, see next slide) of the inherited method

```java
class Employee {
    ...
    double pay() {
       return salary;
    }
}
```

```java
class Manager
  extends Employee {
      ...
    double pay() {
      return super.pay()
            + bonus;
    }
}
```

Inheritance class diagram



Parent's attributes are accessed via `super` reference anywhere inside the child class. Multiple parent constructors differ by their parameters, as ordinary methods. *Eg*, the *Manager* constructor which calls `super(String,String,double)` refers to `Employee(String,String,double)`.

Polymorphism and dynamic binding

The mechanism of extending an old type isn't just good in terms of code re-use. It allows to write shorter, yet more flexible and expressive programs. Namely, if the client of your code can use an object of some class, it can also use objects of its subclasses. This feature is called *polymorphism*, meaning that an object of a given class can, in fact, have multiple forms — of its own class or of any class it extends. Put it another way: polymorphism is the ability for *different objects* to respond to *identical messages*.

All fields and methods of a given class, which are visible outside (1), together with the conditions when they are used (2) and the effects they produced (3), are collectively known as the *class contract* — it represents the declared *raison d'etre* of the class. Class extension provides two kinds of inheritance:

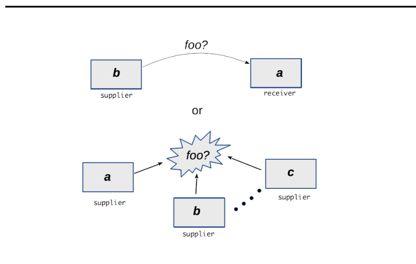
- inheritance of *contract* or *type* — here the subclass is endowed with the type of the superclass and can be used where the superclass can be used;
- inheritance of *implementation* — here the subclass acquires the implementation of the superclass in the form of its (non-private) fields and methods.

Example of the `payroll()` method in the EmployeeTest.java program. Depending on the class of the Employee object which is used in the `payroll()` method, *different* `getSalary()` methods are used. This binding of the call (*ie*, method invocation) to a particular method is known as *dynamic binding*. Another set of examples are shapes and their area.

Dynamic binding and multiple dispatch

When classes of `a` and `b` are known only at run-time:

- `a.foo(b)` — resolving the method based on the argument, *overloading*
- `a.foo(b)` — resolving the method based on the object, *polymorphism*, *dynamic binding*
- `a.foo(b)` — resolving the method based on both, *double dispatch*
- `foo(a,b,c,…)` — in the paradigm-neutral form, *multiple dispatch*



In OO languages, the multiple dispatch implementation is possible but "smells" (fragile code) — so called *Visitor Pattern* (lab 5, extra). There are languages where the multiple dispatch is natural (Lisps, Julia). There are languages where a reasonable solution is possible (Python), but not encouraged (rather complex).

Covariant overriding

A method can also return a reference type. For instance, a method for calculating a *Number* type value which is what the method returns:

```
// in class MyNumber
double real;
public MyNumber reciprocal() {
   return new MyNumber(1 / real);
}

// in class MyComplexNumber
@Override
//without covariant overriding the return type should be as in parent class
public MyComplexNumber reciprocal() {
    double abs = real*real + imag*imag;
    return new MyComplexNumber(real / abs, - imag / abs);
}
```

If the class *MyNumber* is extended to *MyComplexNumber* and you want to override `resiprocal()` to return, quite appropriately, the *MyComplexNumber* value, this is possible. This technique, called *covariant return type*, means that when overriding, the return type is allowed to vary in the *same direction* as the subclass (see covariant overriding example).

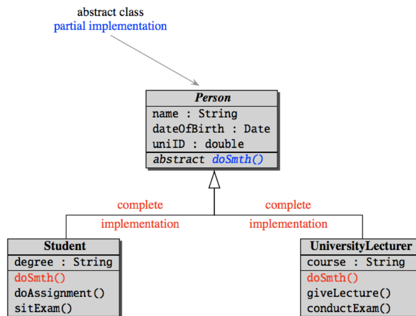
Abstract classes: Partial implementation

Abstract classes allow to capture an *abstraction* — a state or behavioural feature which is common in a family of types, which they *all* have, but in somewhat *different* (particular) way. The whole family is characterised by declaring that something *is* done, but not *how* it is done. The "how", *ie*, the concrete implementation, differs from one type to another. In Java, an abstract class *usually* has one or more declared, but not implemented methods which are marked with the keyword `abstract` (one can declare a class `abstract` without having a single `abstract` method in it). Abstract classes *cannot be instantiated*. The classes which provide implementation of abstract methods (concrete subclasses) must extend the abstract class. The opposite is also possible: a class can extend a concrete superclass and "override" a normal (implemented) method to make it abstract (why would it be useful?).

```
public abstract class Person {
   ... class fields as in an ordinary class
  public abstract void doSmth();
   ... plus more abstract and/or normal methods
}

public class Student extends Person {
    public void doSmth() {
        study();
        studyMore();
        partyAtLast();
    }
}
```

Completing (Abstract) Class Definition



Multiple Inheritance

Multiple inheritance — an ability to define a type which combines properties of more than one existing type — is a double-edge sword: powerful and dangerous.

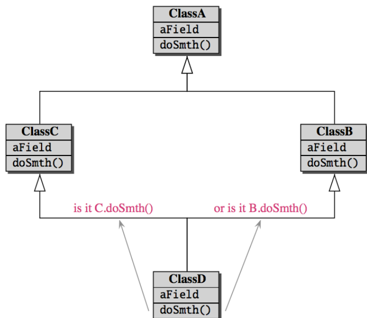
It would be often beneficial for a new class to inherit at once from more than one parent, *eg*, a *HollywoodActor* object can go on trial for a (real life) crime, like tax evasion or murder. In such case, a class *AccusedActor* can be created which needs to inherit from both the *HollywoodActor* and *Civilian* (to account for attributes like SS number, or `payTax()` method). (Less queer example is the *SwissArmyKnife* class.)

The diamond of death problem

The problem with multiple inheritance occur when two (or more) parents of the same class have *common ancestry*. Who is `super`? *ClassB* or *ClassC*? If each of them had modified `doSmth()`, or had hidden `aField`, which copy of them `ClassD` inherited?

Some OO languages (`C++`, `Eiffel`, `Python`) allow multiple inheritance, but resolving DDP adds complexity and makes design and use of class hierarchy more complicated. Java constrains the multiple inheritance by allowing only *single inheritance of implementation* and *multiple inheritance of contract*.

Diamond of Death



With pros and cons of the multiple (by implementation) inheritance, the choice (allowed or not) is less significant today, mainly due to lesser importance (and use) of the inheritance itself. "The new generation chooses composition over inheritance."

OO without classes — Prototype Programming

What is more important for OOP — classes or objects? A class is an accumulation of properties which are common to instances. When a computation is conceived "from the top", defining classes is an adequate approach to capture abstraction, and class based implementation for creating objects is appropriate. Since Java is a statically typed compiled language, once an object is created, its structure and behaviour will not change (unless you are using the reflections or some other "black art", that is). This is good since it guarantees that object's contract will not change (which gives a layer of security to be expected in statically-typed languages), but sometimes one can benefit from the ability to change object's properties while it is alive. Dynamic OO languages (like Python) allow just that, but they still begin with classes (in Python, there are ways to restrain object's variability via the *slot*-mechanism).

What if the problem in question does not yield to easy classification: multiple objects do exist, but finding common features to define their class is not possible. (In philosophy, this problem was emphasised by Ludwig Wittgenstein). There is an alternative paradigm which is known as *prototype based* (OO) programming.

In languages like *Self, Lua, Smalltalk* (which allows classes, too) and (most popular) *JavaScript*, objects are created by cloning from a set of predefined object literals which are called **prototypes**, with the ability to add new attributes and behaviours during the object lifetime. In the prototype-based languages, the notion of type hierarchy does not exist.

Where to look for this topic in the textbook?

- Hortsmann's *Core Java for the Impatient*, Ch. 4.1
- Oracle's Java Tutorial Inheritance