Object-oriented programming and data-structures

CS/ENGRD 2110 SUMMER 2018

Lecture 3: OO Principles - Modularity, Encapsulation

http://courses.cs.cornell.edu/cs2110/2018su

Lecture 2 Recap

- Objects: Classes, References, Instances
- Null and static keywords.
- Constructors
- Pass-by-value vs Pass-by-reference

Lecture 3

- Object-oriented programming introduces a number of important concepts
  - **Modularity**
  - **Encapsulation**
  - **Inheritance**
  - **Abstraction**
  - **Polymorphism**

- This lecture: Modularity & Encapsulation & Inheritance

- Next lecture: abstraction and polymorphism

Modularity

- Classes represent grouping of **related state and behaviour**

- Goal of OOP is to break down program into small, well-defined components with clear functionality.
  - Each class represents a sub-unit of code that can be **developed, tested and updated independently**

- Identifying classes comes with experience. Rule of thumb:
  - Nouns = Classes
  - Verbs = methods
  - A **student** registers for a **course**.

Code Reuse

- Modularity encourages code-reuse

- Group all related state/methods in a **class** (ex: Date) that can simply drop in to other classes when need that functionality
    - Ex: Defined a class **Person** with a date of birth **Date.**

- Define helper functions once, as part of the class.
    - Ex: Parameter checking can be written once in constructor, not every place create object

        -
          ```
          Date(int day, int month, int year) {
                  if (day > 31)  …
                  if (month > 12) ..
          }
          ```

Encapsulation

- **Encapsulation:** the ability of a class to hide its data and methods from other entities.
  - Variables of a class will be hidden from other classes, and can be accessed only through the methods of their current class.

- Classes should expose **functionality/services** not **implementation**
- Good practice to **hide the internals** of a class
  - Implementation hiding

- Encapsulation maximises **cohesion** and minimises **coupling**
  - **Coupling:** how much one class depends on another
  - **Cohesion:** how related everything in a class is

Access Modifiers

- Java uses **access modifiers** to **encapsulate** fields and methods

- **Definition** Access modifiers restrict the scope of a class, constructor , variable , method or data member
  - **private** int day;
  - **private** static **convertToString(int month)**

- 4 access modifiers:
  - **Public**: can be accessed by everyone
  - **Private:** can only be accessed in this class
  - **Protected:** can be accessed by this class and subclasses (def later)
  - **Default:** can be accessed by thisclass, and classes in package (def later)

Getters and Setters

- Hide fields from external classes by declaring them **private** (or **protected**)

- Use getters and setters instead
  - **Getter**: method that returns the contents of a field
  - **Setter:** method that updates a field

- Benefits of getters/setters
  - Can change/remove fields without modifying other objects
  - Can write parameter-checking code in one place

Why is it useful? Refactoring Date

Why is it useful? Refactoring Date

```
class Date {
    int day;
    String  month;
    int year;
    static  String  usStringFormat;

    String getMonth() {
        return month;
    }
}
```

Encapsulation allows us to change internals of class without changing external methods

```
class Date {
    int day;
    int  month;
    int year;
    static String usStringFormat;

    String getMonth() {
        return convertToString(month);
    }
}
```

Programming Tips

- Use private unless there is a really good reason not to

- Classes should be immutable unless good reason to make them mutable

- Comment of method should refer to functionality, not to the internal fields.

```
/** Returns the string field month **/
public String getMonth() {
        return month ;
}
```

Bad! If change inside implementation, also need to change the comments.

```
/** Returns month of the year **/
public String getMonth() {
        return month ;
}
/** Returns month of the year **/
public String getMonth() {
        return convertToString(month);
}
```

Good! Implementation can change

Inheritance - Why?

- Introducing perhaps the most important OO concept: **inheritance**

Inheritance - Why?

- □ Introducing perhaps the most important OO concept: **inheritance**

- □ Consider the following classes:

```
class Instructor {
	private String name;
	private Date dob;
	private int salary;

}
```

```
class Student {
	private String name;
	private Date dob;
	private int grade;

}
```

Inheritance - Why?

- Introducing perhaps the most important OO concept: **inheritance**

- Consider the following classes:

```
class Instructor {
        private String name;
        private Date dob;
        private int salary;

}
```

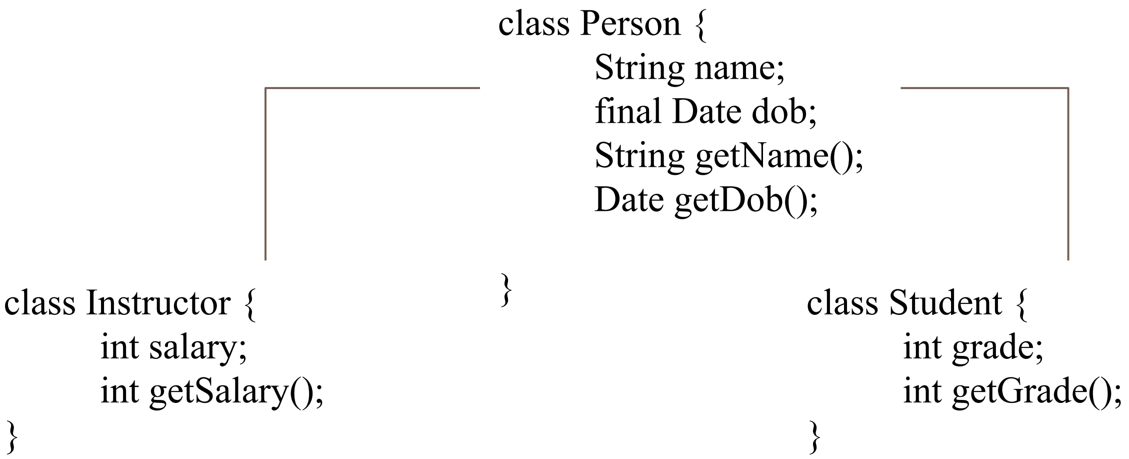
```
class Student {
        private String name;
        private Date dob;
        private int grade;

}
```

Lots of code duplication

Inheritance - Why?

- Instructor and Student share features, differ in others
  - Implicitly, both are a **specialisation** of a type **Person**

- Inheritance allows developers to express these relationships



```
class Person {
        String name;
        final Date dob;
        String getName();
        Date getDob();

}
```

```
class Instructor {
        int salary;
        int getSalary();
}
```

```
class Student {
        int grade;
        int getGrade();
}
```

Inheritance Tree

- **Definition** Inheritance allows a class to be derived from another class to create a hierarchy of classes that share a set of attributes and methods.

- Inheritance introduces an **is-a** relationship: class **B is-a** instance of class **C**
  - The inheritance hierarchy should reflect modeling semantics, not implementation shortcuts

Inheritance Tree

- **Definition** Inheritance allows a class to be derived from another class to create a hierarchy of classes that share a set of attributes and methods.

- Inheritance introduces an **is-a** relationship: class **B is-an** instance of class **C**
  - The inheritance hierarchy should reflect modeling semantics, not implementation shortcuts

- Examples
  - **Instructor** is a **Person, Student** is a **Person**
  - **Triangle** is a **Shape**?
  - **BankAccount** is a **CheckingAccount**?
  - **Animal** is a **Person?**

Inheritance - Terminology

- □ Person is a **base class**
- □ Instructor is a **derived class.** It inherits both state and functionality from the base class.

- □ Person is a **superclass** of Instructor. Instructor is a **subclass** of Person.
- □ SummerInstructor is a **subclass** of Instructor. Instructor a **superclass** of Summer Instructor

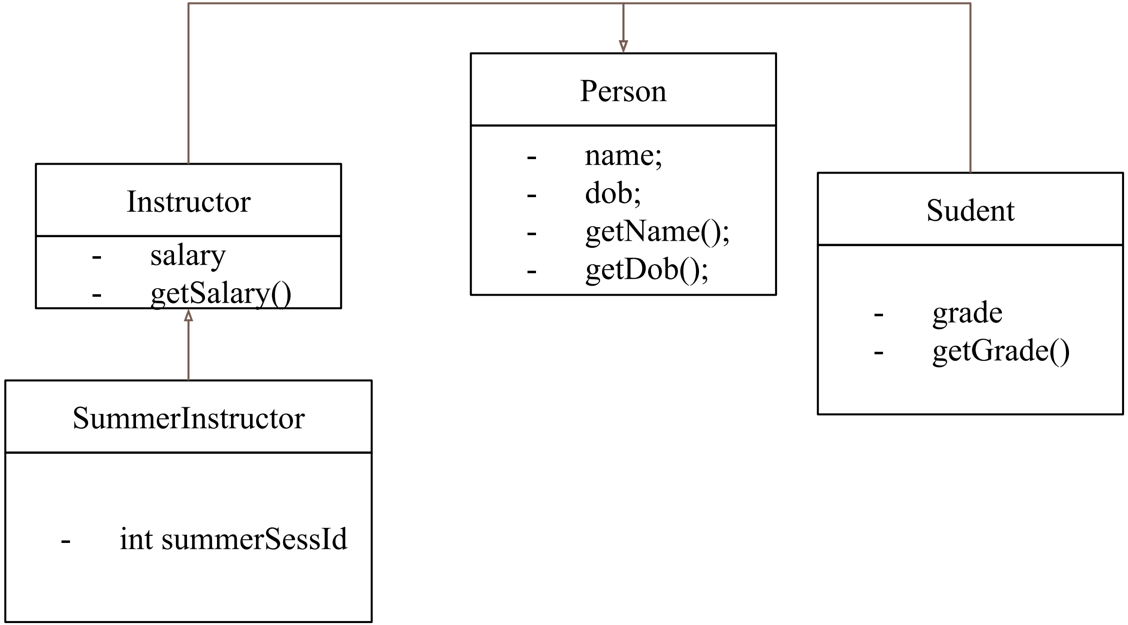
- □ Other phrasing
  - □ Instructor inherits/derives/extends Person

```
class Person {
        String name;
        final Date dob;
        String getName();
        Date getDob();

}

 class Instructor {
         int salary;
         int getSalary();
 }

   class SummerInstructor {
           int summerSessId;
   }
```

Inheritance - Graphically

□ Class hierarchies and dependencies are often represented using UML diagrams

□ Won’t go through it in detail, but you should look it up



Person

- name;
- dob;
- getName();
- getDob();

Instructor

- salary
- getSalary()

Sudent

- grade
- getGrade()

SummerInstructor

- int summerSessId

Defining a subclass in Java

- Derived classes in Java use the **extends.**
  - class Instructor **extends** Person { … }

- Inherit all fields from the base class, except **fields marked as private**
  - No need to redeclare them in the derived class!

- To allow **subclasses** to access fields, but prevent all other classes from accessing them, must mark them as **protected**

Casting

- Possible to **type cast** between numeric types
  - int i = 5 ; float f = (float) i;

- Inheritance tree allows us to typecast objects to any of the types **above it** in the inheritance tree

- Two types of casts
  - **Widening conversions**
  - **Narrowing conversions**

Widening conversions

- **Definition**: cast an object to its parent in the inheritance tree

  - Person p = (Instructor) natacha;
  - Person p = (Student) jack;

- It is **always** possible to **upcast** an object
  - an Instructor instance is always a Person instance
  - But, when cast to a superclass, cannot access methods of the subclass

- Allows you to use an Instructor/Student instance every time you want a Person object.

Narrowing conversions

- **Definition:** cast an object to a child in the inheritance tree

  - Person natacha = new … ;
  - Instructor i = (Instructor) natacha;

- Narrowing conversions are **dangerous.** It is **not always** possible to downcast an object
  - a Person instance is not always an Instructor
  - Remember the typing error in Python? Downcasting in Java may generate a **runtime** exception.

Shadowing (Also called Hiding)

- Where did we see this term before?

- Shadowing in subclasses follows similar rules
    - Can redefine variables in child classes
    - Use **bottom-up** rule to figure out which variable will be accessing

```
class Instructor {
        int salary = 500;
        int getSalary();
}
```

```
class SummerInstructor {
        int salary = 700;
}
```

Variable salary is **shadowed**

What will print?

```
        SummerInstructor si = new SummerInstructor()
        System.out.println(si.salary);
```

Shadowing (Also called Hiding)

- □ Where did we see this term before?

- □ Shadowing in subclasses follows similar rules
  - □ Can redefine variables in child classes
  - □ Use **bottom-up** rule to figure out which variable will be accessing

```
class Instructor {
        int salary = 500;
        int getSalary();
}
```

```
class SummerInstructor {
        int salary = 700;
}
```

I personally dislike shadowing. Risks causing errors and confusion, and can (should) usually be implemented differently.

Overriding

- **Definition** A method that is inherited from the superclass can be overridden by redeclaring it in the subclass.

- Java makes overriding explicit by using the **@Override** annotation
  - Use it like your life depends on it!

```
class Instructor {
    int salary = 500;
    int getSalary() {
        return salary;
    }
}
```

```
class SummerInstructor {
    int summerBonus = 700;

    @Override
    int getSalary() {
        return salary + summerBonus;
    }
}
```

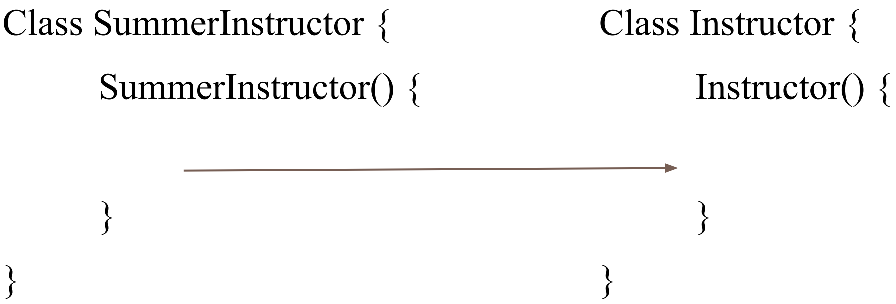
Moving up and down the tree

- Java provides two **keywords** to move up and down the tree hierarchy
  - **this** keyword returns a reference to the current instance of the object
  - **super** keyword enables direct access to the parent of the object
- Homework will let you play with those in more detail.

Constructor Chaining

- Recall that every class has either:
  - an implicit default constructor that is called during initialisation.
  - one or more constructors

- A subclass implicitly (or explicitly) calls the constructors of all its Constructors are **chained** in an inheritance tree



```
Class SummerInstructor {                Class Instructor {

      SummerInstructor() {                    Instructor() {



      }                                       }

}                                       }
```

Constructor Chaining

- Constructor chaining can be used to minimise code duplication
  - No need to rewrite initialisation logic of base class in every derived class
  - In Java, can use **super** keyword to call the **inherited** constructor

```
class Person {
        Person(String name, Date dob) {
                this.name = name;
                this.dob = dob;
        }
}
```

```
class Instructor {
        Instructor(String name, Date dob, int salary) {
                super(name, dob, salary)
                this.salary = salary;
        }
}
```

```
class Student {
        Student(String name, Date dob, int grade) {
                super(name, dob);
                this.grade = grade;
        }
}
```

Java Inheritance

- Class Object is the root of the class hierarchy.
  - Every class has Object as a superclass.
    - All objects implement the methods of this class

- Class provides a number of interesting methods that every class inherits and can override
  - equals(), toString(), clone() and hashCode()
  - We'll see these later.

- Look up the Javadoc!
  - https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html

Immutability

- **Definition** An object or field is considered *immutable* if its state cannot change after it is constructed

- To make a field immutable, use access modifier **final**
  - static final ukDateFormat;
  - Why is it not enough to mark field **private** and not provide a **setter** method?

- Benefits of immutability
  - Easier to write clean, reliable code
  - Easier to maintain invariants in the presence of concurrent modifications

- A class is immutable if its marked as **final** and all its fields are also **final**

References in JavaHyperText

immutable

final

access modifier

modularity

encapsulation

inheritance

constructor

shadowing

overriding

casting