Inf1B

Inheritance A

Fiona McNeill
adapting earlier versions by Perdita Stevens, Ewan Klein, Volker Seeker, et al.

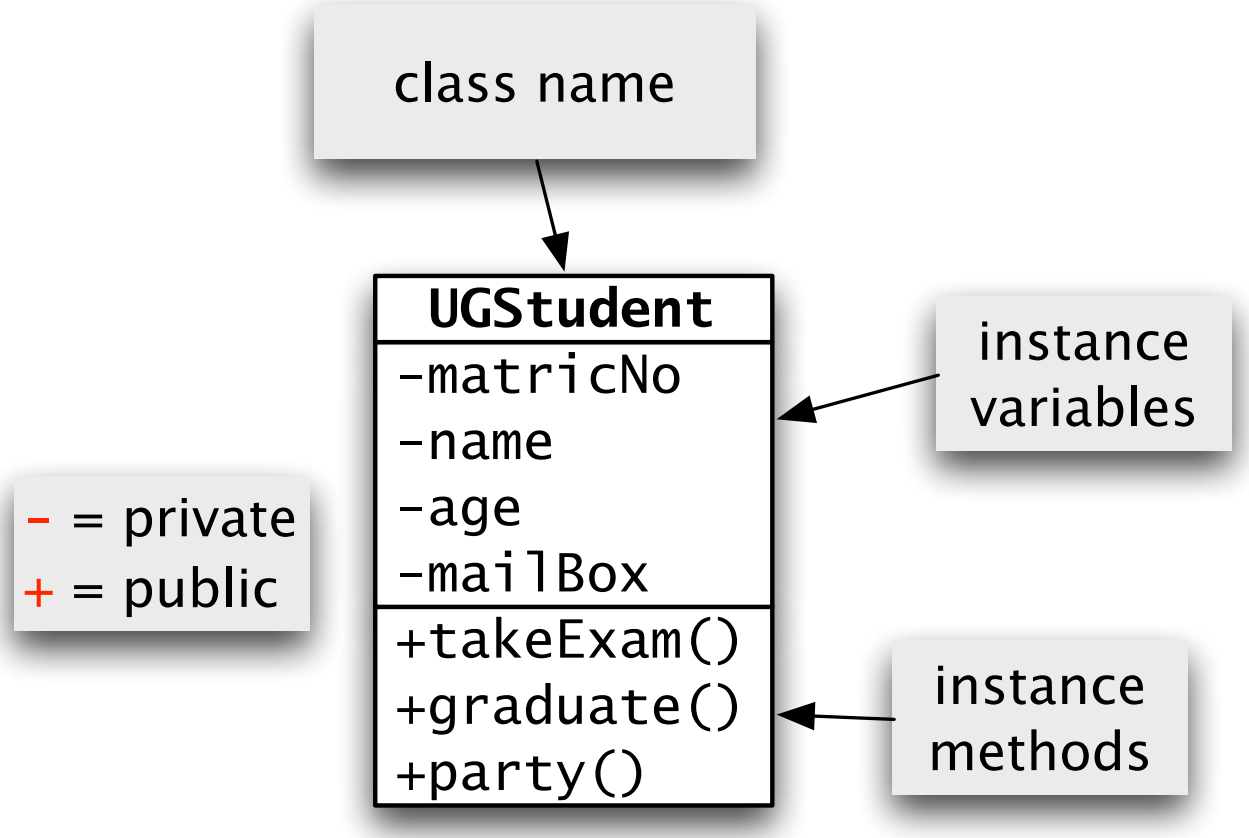
School of Informatics

UML Class Diagrams

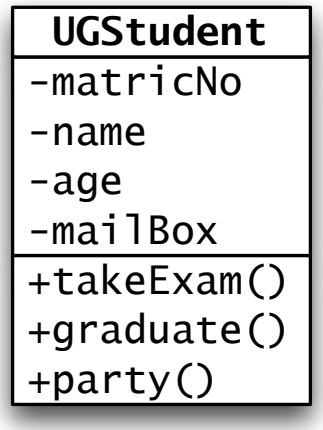
UML: language for specifying and visualizing OOP software systems

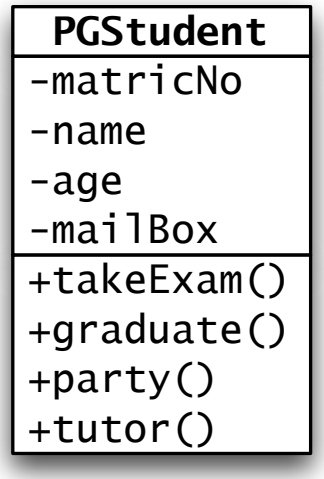
UML class diagram:

- specifies class name, instance variables, methods, ...



Classes with Stuff in Common





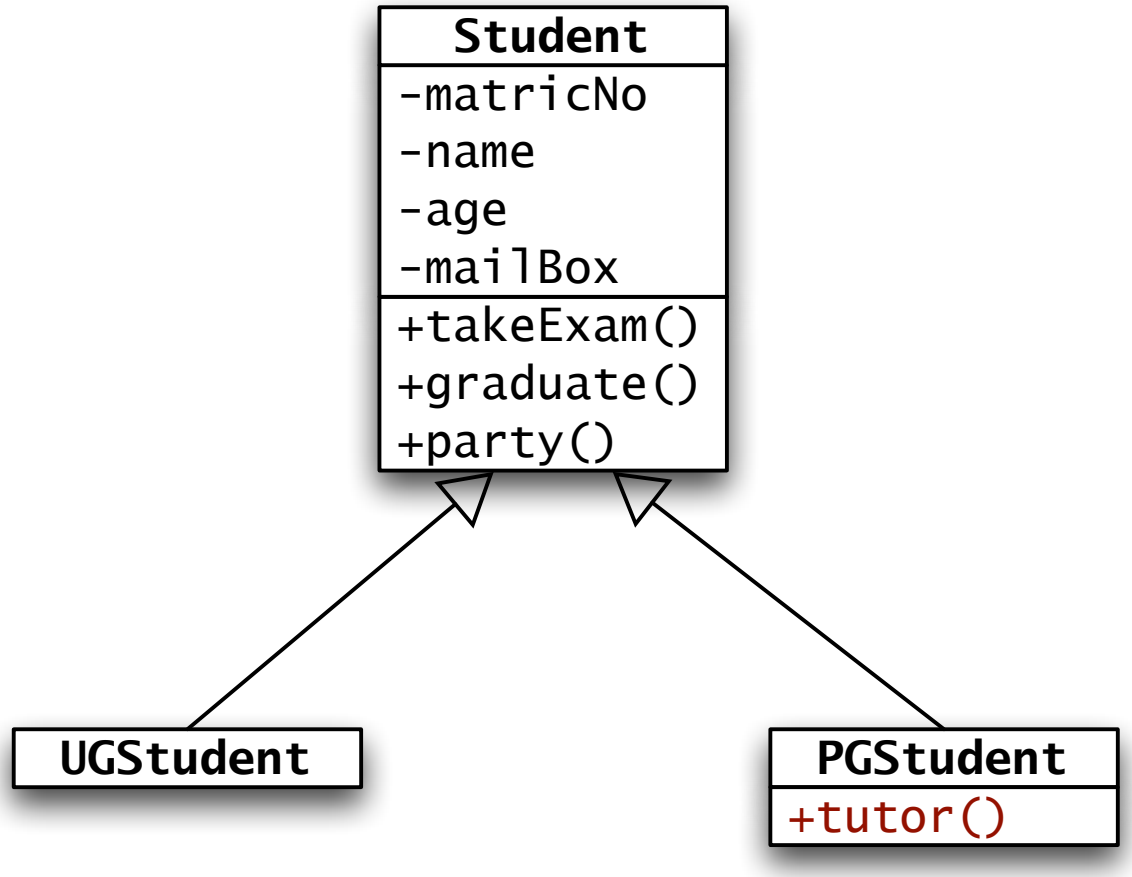
- Lots of duplication across the two classes
- More importantly, many clients should be able to work with both: don't want to duplicate *their* code.
- How do we eliminate the duplication?

Classes with Stuff in Common

- Lots of duplication across the two classes
- More importantly, many clients should be able to work with both: don't want to duplicate *their* code.
- How do we eliminate the duplication?

Abstracting Common Stuff

Inheritance hierarchy:
Subclass (`UG`, `PG`) inherit from superclass (`Student`)



Arrow with open head indicates generalization in UML class diagram.

Subclasses and superclasses

- Subclass (e.g. `UG`) inherits* the members of superclass (e.g. `Student`)
- Subclass is a specialization of superclass — superclass is generalization of subclass

*[details to be further specified...]

X IS-A Y?

The IS-A test

- Is `ClassX` a subclass of `ClassY`?
- Test: can we say that `ClassX` `IS-A` ('is a kind of') `ClassY`?
- Does an instance of `ClassX` have all the properties (and maybe more) that an instance of `ClassY` has?

IS-A Candidates

1. `Kitchen subclass-of Room`
2. `Room subclass-of House`
3. `Violinist subclass-of Musician`
4. `Sink subclass-of Kitchen`
5. `Musician subclass-of Person`
6. `Lady Gaga subclass-of Singer`
7. `Student subclass-of Musician`

Inheritance

Subclass inherits all the members (instance variables and methods) of the superclass.

In Java: subclass extends superclass.

```
public class PGStudent extends Student {
    ...
}
```

Inheritance

Subclass inherits all the public and protected members (instance variables and methods) of the superclass.

In Java: subclass extends superclass.

```
public class PGStudent extends Student {
    ...
}
```

The protected Access Modifier

```
public class Student {
    ...
    private String name;
    ...
}

public class PGStudent extends Student {
    public void tutor() {
        System.out.println("Hello, I am " + name);
        ...
    }
}
```

This will cause a compiler error because `name` is not visible in the `PGStudent` subclass.

The protected Access Modifier

```
public class Student {
    ...
    protected String name;
    ...
}

public class PGStudent extends Student {
    public void tutor() {
        System.out.println("Hello, I am " + name);
        ...
    }
}
```

Now `name` is visible to all subclasses of `Student` and the code will compile.

Access Modifiers Summary

Modifier	Class	Package	Subclass	Global
Public	Yes	Yes	Yes	Yes
Protected	Yes	Yes	Yes	No
Default	Yes	Yes	No	No
Private	Yes	No	No	No

Inheritance

Subclass inherits all the public and protected members (instance variables and methods) of the superclass.

Inheritance

Subclass inherits all the public and protected members (instance variables and methods) of the superclass.

A subclass can add new members of its own.

Inheritance

Subclass inherits all the public and protected members (instance variables and methods) of the superclass.

A subclass can add new members of its own.

By default, methods that are inherited from superclass have same implementation in subclass.

Inheritance

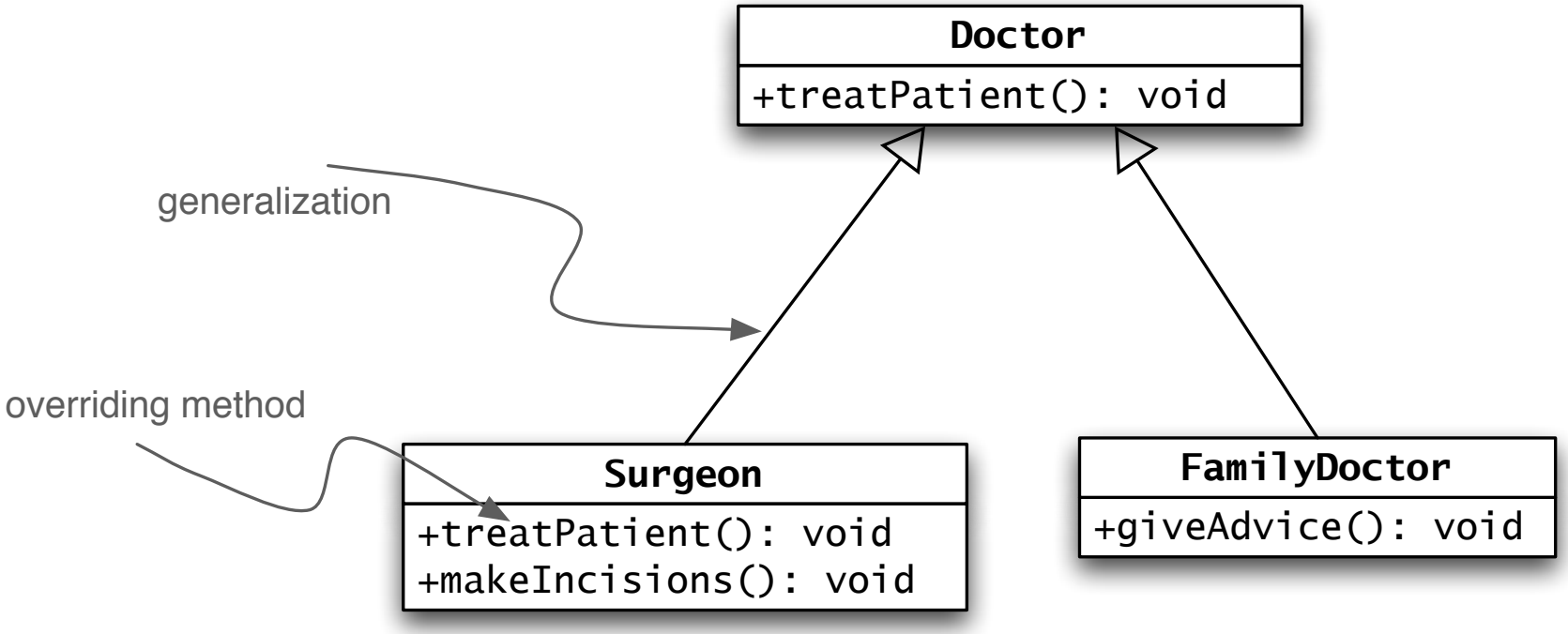
Subclass inherits all the public and protected members (instance variables and methods) of the superclass.

A subclass can add new members of its own.

By default, methods that are inherited from superclass have same implementation in subclass.

Except if the subclass overrides the inherited method.

Doctor Example, 1



For handy guide to UML, see
`http://www.loufranco.com/blog/assets/cheatsheet.pdf`

Doctor Example, 2

Doctor

```
public class Doctor {
        public void treatPatient() {
                // perform a checkup
         }
}
```

Doctor Example, 3

FamilyDoctor

```
public class FamilyDoctor extends Doctor {
        public void giveAdvice() {
                // tells you to wrap up warmly
         }
}
```

NB We put this class into a new file `FamilyDoctor.java`

Doctor Example, 4

Surgeon

```
public class Surgeon extends Doctor {
        public void treatPatient() {
                // perform surgery
                // overrides inherited method
                // Can call Doctor's version:
                super.treatPatient();
        }
        public void makeIncisions() {
                // use a scalpel
                // a new method
        }
}
```

NB We put this class into a new file `Surgeon.java`

Method Overriding

- Method *m* in subclass `B` overrides method *m'* in superclass `A` if *m* has exactly the same signature (i.e. name and parameters) as *m'*. (Return type? Later...)
- Normally, *m* replaces the implementation of *m'*.

Doctor

```
Doctor d = new Doctor();
d.treatPatient(); //  Use implementation in Doctor class
```

Surgeon

```
Surgeon s = new Surgeon();
s.treatPatient(); //  Use implementation in Surgeon class
```

What does it print?

```
public class Vehicle {
  public void drive() {
    System.out.println("drivedrive");
  }
}

public class Car extends Vehicle { }
public class Bike extends Vehicle { }

public class Main {
  public static void main(String[] args) {
    Car c = new Car();
    c.drive();
    Bike b = new Bike();
    b.drive();
  }
}
```

What does it print?

```
public class Vehicle {
  public void drive() {
    System.out.println("drivedrive");
  }
}

public class Car extends Vehicle { }
public class Bike extends Vehicle { }

public class Main {
  public static void main(String[] args) {
    Car c = new Car();
    c.drive();
    Bike b = new Bike();
    b.drive();
  }
}
```

Prints **drivedrive** twice because Bike and Car inherit drive implementation from Vehicle.

If it compiles, what does it print?

```
public class Vehicle {
  public void drive() {
    System.out.println("drivedrive");
  }
}
public class Car extends Vehicle {
  public void drive() {
    System.out.println("rollroll");
  }
}
public class Bike extends Vehicle {
  public void drive() {
    System.out.println("pedalpedal");
  }
}
public class Main {
  public static void main(String[] args) {
    Car c = new Car(); c.drive();
    Bike b = new Bike(); b.drive();
  }
}
```

If it compiles, what does it print?

```
public class Vehicle {
  public void drive() {
    System.out.println("drivedrive");
  }
}
public class Car extends Vehicle {
  public void drive() {
    System.out.println("rollroll");
  }
}
public class Bike extends Vehicle {
  public void drive() {
    System.out.println("pedalpedal");
  }
}
public class Main {
  public static void main(String[] args) {
    Car c = new Car(); c.drive();
    Bike b = new Bike(); b.drive();
  }
}
```

Prints **rollroll** and **pedalpedal** because Bike and Car override Vehicle's drive implementation with their own.

If it compiles, what does it print?

```
public class Vehicle {
  public void drive() {
    System.out.println("drivedrive");
  }
}
public class Car extends Vehicle {
  public void drive() {
    super.drive();
  }
}
public class Bike extends Vehicle {
  public void drive() {
    System.out.println("pedalpedal");
  }
}
public class Main {
  public static void main(String[] args) {
    Car c = new Car(); c.drive();
    Bike b = new Bike(); b.drive();
  }
}
```

If it compiles, what does it print?

```
public class Vehicle {
  public void drive() {
    System.out.println("drivedrive");
  }
}
public class Car extends Vehicle {
  public void drive() {
    super.drive();
  }
}
public class Bike extends Vehicle {
  public void drive() {
    System.out.println("pedalpedal");
  }
}
public class Main {
  public static void main(String[] args) {
    Car c = new Car(); c.drive();
    Bike b = new Bike(); b.drive();
  }
}
```

Prints **drivedrive** and **pedalpedal** because Car's drive method calls the super class's drive method.

The Design Process

1. Look for objects that have common attributes and behaviours.
2. Design a class that represents the common state and behaviour.
3. Decide if a subclass needs method implementations that are specific to that particular subclass type.
4. Carry out further abstraction by looking for groups of subclasses that might have common behaviours.

Encapsulation and Inheritance

Student

```
public class Student {
    private final String firstName;
    private final String lastName;
    private final String matric;

    public Student(String fn, String ln, String m) { ... }

    public String getFirstName() { ... }
    public String getLastName() { ... }
    public String getMatric() { ... }
}
```

Encapsulation and Inheritance

UG

```
public class UG extends Student {
    private String tutGroup = "";

    public void setTutGroup(String s) {
        tutGroup = s;
    }
    public String getTutGroup() {
        return tutGroup;
    }
    public String toString() {
        return "UG [firstName=" + firstName + ",
                    lastName=" +  lastName +
                ", matric=" +    matric +
                ", tutGroup=" +  tutGroup + "]";
    }
}
```

Encapsulation and Inheritance

UG

```
public class UG extends Student {
    private String tutGroup = "";

  ...

    public String toString() {
        return "UG [firstName=" + firstName + ",
                    lastName=" +  lastName +
                ", matric=" +     matric +
                ", tutGroup=" +  tutGroup + "]";
    }

}
```

Won't work!

Encapsulation and Inheritance

UG

```
public class UG extends Student {
    private String tutGroup = "";

  ...

      public String toString()
        return "UG [firstName=" + getFirstName() + ", " +
                  "  lastName=" +  getLastName() +
                  ", matric=" +    getMatric() +
                  ", tutGroup=" +  tutGroup + "]";

    }

}
```

Encapsulation and Inheritance

Student

```
public class Student {
    protected final String firstName;
    protected final String lastName;
    protected final String matric;

    public Student(String fn, String ln, String m) { ... }

    public String getFirstName() { ... }
    public String getLastName() { ... }
    public String getMatric() { ... }
}
```

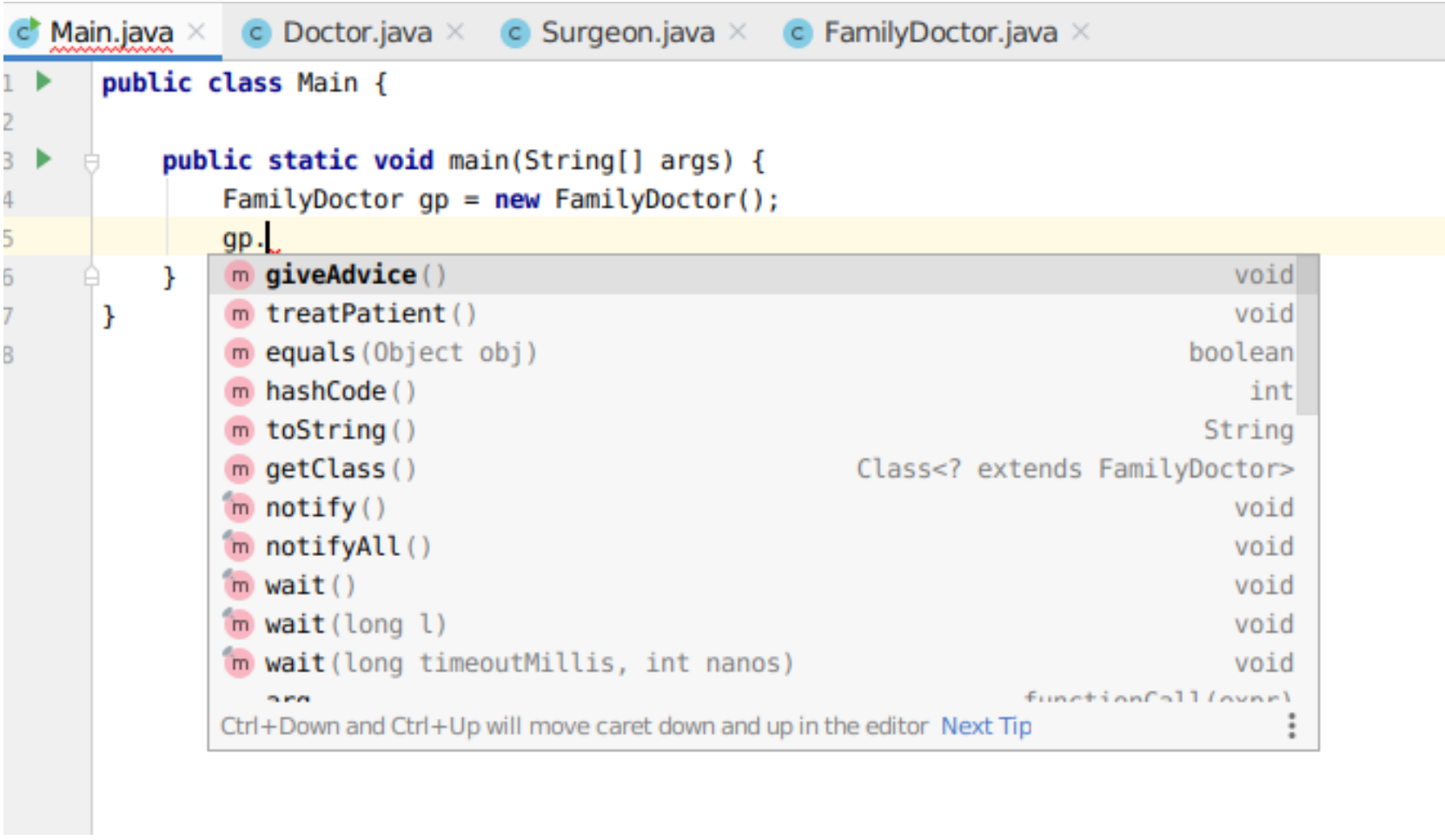
Encapsulation and Inheritance

- private instance variables (fields) cannot be directly accessed by subclass.
- Can only be accessed via setter and getter methods (which are inherited from superclass).
- protected instance variables are still hidden from other classes but accessible by subclasses.
  $\rightarrow$ *However, you might not want to allow this for users of your library.*

The Object Superclass

FamilyDoctor Members

Inherited and non-inherited members



This can also be seen in the Java API

`https://docs.oracle.com/en/java/javase/11/docs/api/index.html`

The class Object

Doctor's Superclass

```
public class Doctor {
        void treatPatient() {
                ...
         }
}
```

The class Object

Doctor's Superclass

```
public class Doctor extends Object {
        void treatPatient() {
                ...
         }
}
```

- Object is the superclass of every class in Java!
- If a class doesn't explicitly extend some superclass, then it implicitly extends Object.
- That is, we don't need to add extends Object.

Some Methods of `Object`

`Object` defines methods that are available to every class. E.g.,

- `equals(Object o)` — test whether two objects are equal.
- `hashCode()` — numerical ID; equal objects must have equal hash codes.
- `toString()` — returns a textual representation of an object; automatically invoked by methods like `System.out.println()`.
- Since every class inherits `toString()` from `Object`, you have already been overriding this method!

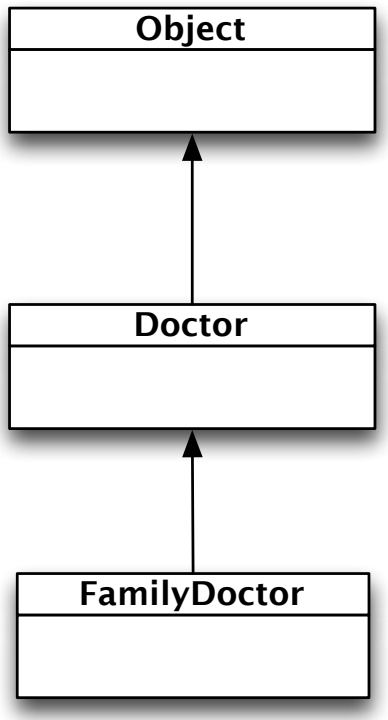
Objects ...

- have a static (compile-time) type defined inside a class
- are instances of classes created at runtime
- are created using a constructor and the new keyword
- are reference types
- reside on the heap memory rather than the stack
- are destroyed automatically by the garbage collector
- derive from the `Object` superclass
- inherit some default methods (e.g. toString, equals, hashCode, ...)

Constructor Chaining

Constructor Chaining

- All constructors in object's inheritance tree run when a new instance is created.
- `FamilyDoctor extends Doctor`



Constructor Chaining

- Each constructor, implicitly or explicitly, invokes a constructor of the direct superclass as the first thing it does...
- ...so that by the time it does anything else, all aspects of the object defined in any superclass have been properly defined.
- Only `Object` has no direct superclass, so that's where the process stops.
- Syntax for explicitly invoking a constructor: `super(arg1, arg2, ...)`.
- You can omit this call if what you want is a no-argument constructor: the compiler automatically inserts `super()`.
- But if there is no no-argument constructor, you'll get a compile-time error.
- The constructor call must always be the first instruction in the constructor's body.

Constructor Chaining

Student

```
public Student(String fn, String ln, String m) {
      firstName = fn;
      lastName = ln;
      matric = m;
}
```

UG extends Student

```
private String tutGroup

public UG(String fn, String ln, String m, String tutGroup) {
      super(fn, ln, m); // call the superclass constructor
      this.tutGroup = tutGroup;
}
```

What does it print?

```
public class Vehicle {
 protected String noise;
 public Vehicle() {
   noise = "drive";
 }
 public void drive() {
   System.out.println(noise + noise);
 }
}

public class Car extends Vehicle {
 public Car() {
   noise = "roll";
 }
}

public class Main {
 public static void main(String[] args)
       {
   Car c = new Car();
   c.drive();
 }
}
```

What does it print?

```
public class Vehicle {
 protected String noise;
 public Vehicle() {
   noise = "drive";
 }
 public void drive() {
   System.out.println(noise + noise);
 }
}

public class Car extends Vehicle {
 public Car() {
   noise = "roll";
 }
}

public class Main {
 public static void main(String[] args)
       {
   Car c = new Car();
   c.drive();
 }
}
```

Prints **rollroll** because "roll" is assigned after "drive" in constructor chain.

What does it print?

```
public class Vehicle {
  private String noise;
  public Vehicle() {
    noise = "drive";
  }
  public void drive() {
    System.out.println(noise + noise);
  }
}

public class Car extends Vehicle { }

public class Main {
  public static void main(String[] args
      ) {
    Car c = new Car();
    c.drive();
  }
}
```

What does it print?

```
public class Vehicle {
  private String noise;
  public Vehicle() {
    noise = "drive";
  }
  public void drive() {
    System.out.println(noise + noise);
  }
}

public class Car extends Vehicle { }

public class Main {
  public static void main(String[] args
      ) {
    Car c = new Car();
    c.drive();
  }
}
```

Prints **drivedrive** because default no-arg ctor is provided for `Car` by default which automatically calls no-arg ctor from `Vehicle`. `drive` method is inherited.

What does it print?

```
public class Vehicle {
  private String noise;
  public Vehicle() {
    noise = "drive";
  }
  public void drive() {
    System.out.println(noise + noise);
  }
}

public class Car extends Vehicle {
  public void drive() {
    System.out.println(noise);
  }
}

public class Main {
  public static void main(String[] args
      ) {
    Car c = new Car();
    c.drive();
  }
}
```

What does it print?

```
public class Vehicle {
  private String noise;
  public Vehicle() {
    noise = "drive";
  }
  public void drive() {
    System.out.println(noise + noise);
  }
}

public class Car extends Vehicle {
  public void drive() {
    System.out.println(noise);
  }
}

public class Main {
  public static void main(String[] args
      ) {
    Car c = new Car();
    c.drive();
  }
}
```

Does not compile because access to `noise` is private and therefore not allowed in `Car`. - `protected` would have worked.

What does it print?

```
public class Vehicle {
  protected String noise;
  public Vehicle(String noise) {
    this.noise = noise;
  }
  public void drive() {
    System.out.println(noise + noise);
  }
}

public class Car extends Vehicle {
  public void drive() {
    System.out.println(noise);
  }
}

public class Main {
  public static void main(String[] args
      ) {
    Car c = new Car("roll");
    c.drive();
  }
}
```

What does it print?

```
public class Vehicle {
  protected String noise;
  public Vehicle(String noise) {
    this.noise = noise;
  }
  public void drive() {
    System.out.println(noise + noise);
  }
}

public class Car extends Vehicle {
  public void drive() {
    System.out.println(noise);
  }
}

public class Main {
  public static void main(String[] args
      ) {
    Car c = new Car("roll");
    c.drive();
  }
}
```

Does not compile because `Vehicle`'s one argument ctor needs to be called explictly using `super` in `Car`'s ctor.

Constructors are not inherited!

What does it print?

```
public class Vehicle {
  protected String noise;
  public Vehicle(String noise) {
    this.noise = noise;
  }
  public void drive() {
    System.out.println(noise + noise);
  }
}

public class Car extends Vehicle {
  public Car(String noise) {
    super(noise);
  }
  public void drive() {
    System.out.println(noise);
  }
}

public class Main {
  public static void main(String[] args
      ) {
    Car c = new Car("roll");
    c.drive();
  }
}
```

What does it print?

```
public class Vehicle {
  protected String noise;
  public Vehicle(String noise) {
    this.noise = noise;
  }
  public void drive() {
    System.out.println(noise + noise);
  }
}

public class Car extends Vehicle {
  public Car(String noise) {
    super(noise);
  }
  public void drive() {
    System.out.println(noise);
  }
}

public class Main {
  public static void main(String[] args
      ) {
    Car c = new Car("roll");
    c.drive();
  }
}
```

This works and prints **roll**.

Summary

Inheriting from a superclass:

- the subclass gets all the public and protected members (instance variables and methods) of the superclass;
  - `public class UGStudent extends Student`
- the subclass may add members, and also override methods.
- So subclass extends (adds to) the behaviour of its superclass.
- Inheritance corresponds roughly to taxonomic relations for everyday concepts.
- In Java, you can only inherit from one superclass.

Problems with using inheritance:

- Easy to get muddled with inheritance hierarchies.
- Subclass is tightly coupled with superclass.
- Changes in superclass can break subclass — fragile base class problem.

Reading

Objects First

Chapter 10 *Improving Structure with Inheritance*

Stop at 10.7 Subtyping for now ...