Hierarchies

ACS-1904 LECTURE 5

Chapter 5 - hierarchies

- IS-A associations
- superclasses
- subclasses
- inheritance
- abstract classes

Inheritance: What is it?

- The basic idea behind inheritance is that you can create a class built from/based on an existing class.
- The original class and the class derived from it share many of the same characteristics.
- The derived class adds new fields and/or new behaviours (methods)
  - The derived class has everything that the original class has and more

Inheritance: What is it?

- This allows us to change or extend the behaviours of our classes without having to re-write them.
  - Add new methods
  - Override existing methods
- We aim to ‘factor out’ common characteristics and move them ‘up’ the class hierarchy.
- Inheritance also promotes code-reuse and encapsulation.
  - i.e. the derived class can use the behaviours of the original class without knowing anything about the implementation details.

Inheritance: What is it?

- When defining a subclass
  - Extend the superclass
  - Indicate the differences between the sub-class and superclass
  - Place the most general methods and common fields in the superclass
  - Factoring out common fields and functionality means moving them to the superclass
  - This is, as you might imagine, very common in object-oriented programming.

Inheritance: What is it?

- There are many scenarios that are naturally hierarchical.
- We will look at a few examples in this discussion but before we do, can you think of a scenario that is naturally hierarchical?

Inheritance: Definition

- **inheritance**: a parent-child relationship between classes,
  - also called Superclass/Subclass, and Base class/Derived class

- allows sharing of the properties and behavior of the parent class into its child classes
  - one of the major benefits of object-oriented programming (OOP) is this code sharing between classes through inheritance

- child class adds new properties and/or behaviors or overrides existing behavior from parent

Inheritance terms

- **superclass**, **base class**, **parent class**: terms to describe the parent in the relationship, which shares its functionality

- **subclass**, **derived class**, **child class**: terms to describe the child in the relationship, which accepts functionality from its parent

- **extend**, **inherit**, **derive**: become a subclass of another class

Inheritance in Java

- in Java, you specify another class as your parent by using the keyword `extends`
  - `public class CheckingAccount extends BankAccount {`
  - the instances of your class will now receive all of the state (fields) and behavior (methods) of the parent class
  - constructors and static methods/fields are not inherited
  - by default, a class's parent is `Object`
- Java forces a class to have exactly one parent ("single inheritance")
  - other languages (C++) allow multiple inheritance

Inheritance in Java

- Note that it is possible for a class in Java to inherit from multiple classes as in the following example

- Here the class Cat inherits the state from Mammal
- Mammal's state includes that of animal
- So, Cat gets everything from both Animal and Mammal

"Has-a" Relationships

- "Has-a" relationship: when one object contains another as a field

```
public class BankAccountManager {
  private List myAccounts;
  // ...
}
```

- a `BankAccountManager` object "has-a" `List` inside it, and therefore can use it

"Is-a" relationships

- "Is-a" relationships represent sets of abilities; implemented through interfaces and inheritance

```
public class CheckingAccount
  extends BankAccount {
  // ...
}
```

- a `CheckingAccount` object "is-a" `BankAccount`
  - therefore, it can do anything an `BankAccount` can do
  - it can be substituted wherever a `BankAccount` is needed
  - a variable of type `BankAccount` may refer to a `CheckingAccount` object

Hierarchies - practitioner example

Example. Practitioner hierarchy

- Consider that we have an application dealing with doctors and pharmacists
- As doctors and pharmacists have a lot in common we generalize

  → introduce practitioner

- We can state:
  - A doctor **is a** practitioner
  - A pharmacist **is a** practitioner

Hierarchies - practitioner example

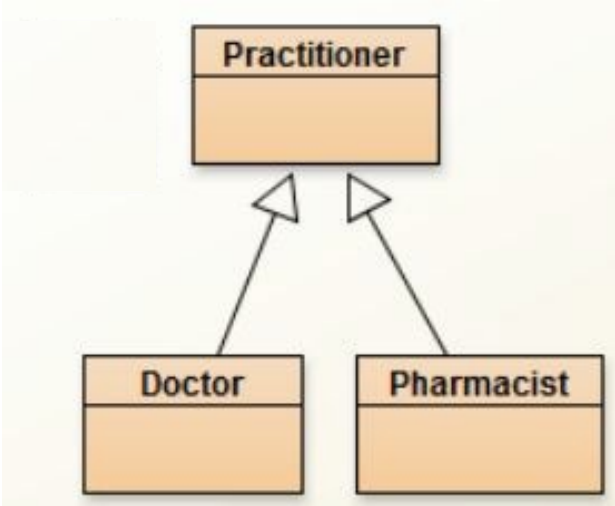
- Practitioner is a generalization of Doctor and Pharmacist
- Doctors and Pharmacists are specializations of Practitioner

- Now we have 3 classes
  - Superclass: Practitioner
  - Subclasses: Doctor & Pharmacist

Hierarchies - practitioner example

BlueJ class diagram:

- When we create a Doctor object, that object is also an instance of Practitioner
  - Similarly: Pharmacist
- If we explicitly create a Practitioner object then that object is not an instance of Doctor nor Pharmacist.



Hierarchies - practitioner example

Practitioner

- All the fields and methods common to both Doctor & Pharmacist

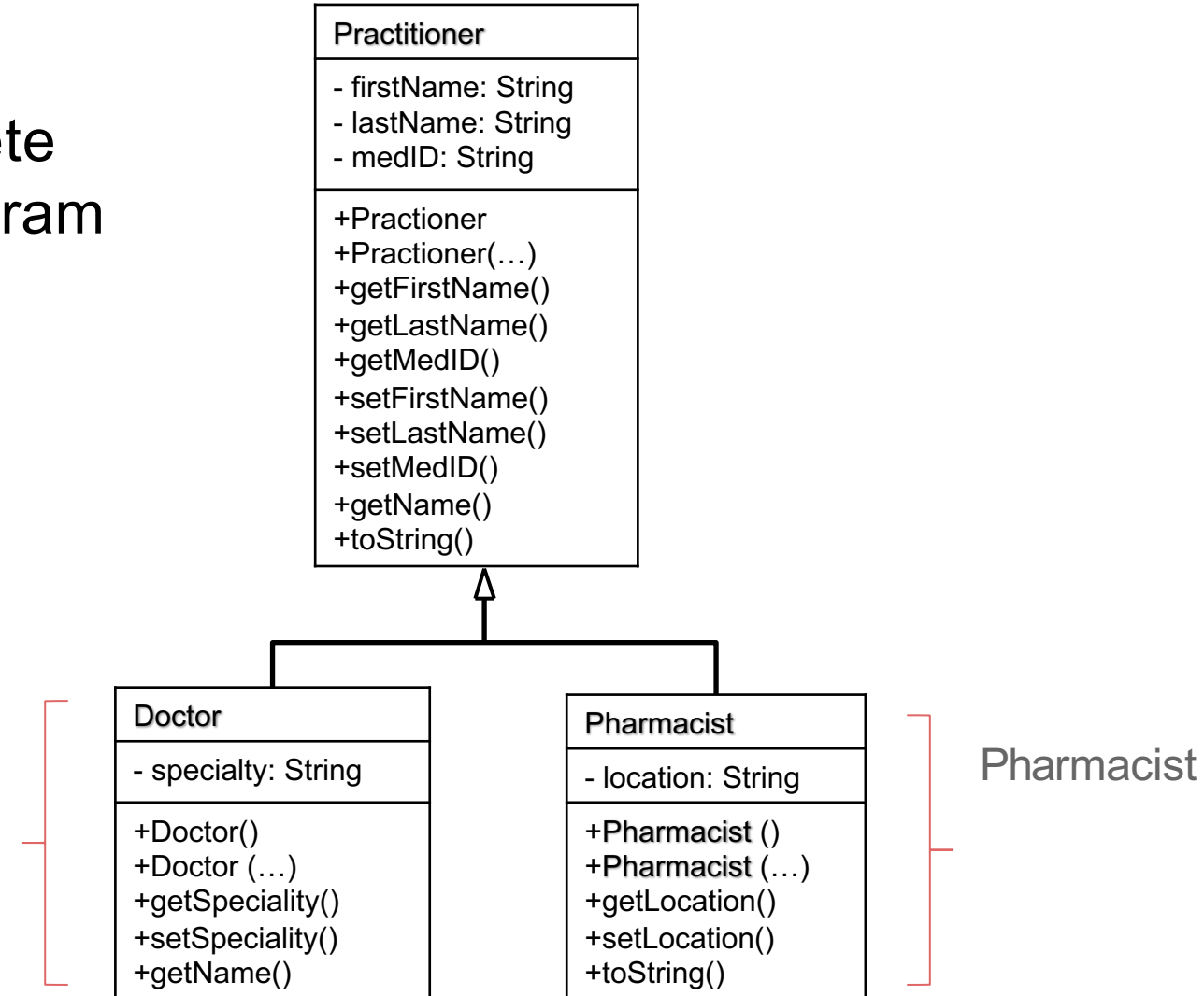
Doctor

- fields and methods pertinent to Doctor but not Pharmacist

Pharmacist

- fields and methods pertinent to Pharmacist but not Doctor

Practitioner: a more complete UML class diagram



A doctor has these fields/methods **plus** all those of Practitioner too

Pharmacist

Practitioner class(Practitioner.java)

```
public class Practitioner {
    private String firstName;
    private String lastName;
```

… normal class with constructors, getters, setters and:

```
    public String toString(){
        return getName() + " ” + medID;
    }
    public String getName(){
        return firstName + " " + lastName;
    }
}
```

Creating practitioners(CreatePractitioners.java)

```
/**
 * Demonstration class to create practitioners
 */
public class CreatePractitioners
{
    public static void main(String[] args){
        Practitioner john = new Practitioner();
        Practitioner tom  =
            new Practitioner("Tom","Smith","Prac123");

       System.out.println("Practitioners:\n"+john+"\n"+tom);
    }
}
```

Pharmacist class

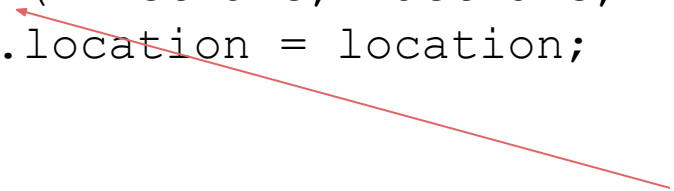
```
/**
 * The Pharmacist class
 * - a subclass of Practitioner
 * - a pharmacist "is a" practitioner
 */
public class Pharmacist extends Practitioner{
    private String location;
    /**
     * by default, the no-arg constructor calls
     * the no-arg constructor in Practitioner
     */
    public Pharmacist(){
        location = "unknown";
    }
```

Specifies this is a subclass of Practitioner

No-arg constructor -automatic call to superclass no-arg constructor

Pharmacist class

```
/**
 * constructor for when information is available
 */
public Pharmacist(String firstName, String lastName,
                  String medID, String location){
    // note the explicit call to a Practitioner constructor
    super(firstName, lastName, medID);
    this.location = location;
}
```



Invoke the superclass constructor by using super

Pharmacist class(Pharmacist.java)

```
// getters
    public String getLocation(){
        return location;
    }
// setters
    public void setLocation(String location){
        this.location = location;
    }
}
```

Normal getter/setter for specific fields added for pharmacists

Doctor class

```
/**
 * The Doctor class
 * - a subclass of Practitioner
 * - an instructor "is a" practitioner
 */
public class Doctor extends Practitioner {
    private String specialty;
    /**
     * no-arg constructor, recall default call
     * to Practitioner no-arg constructor
     */
    public Doctor(){
        specialty = "unknown";
    }
```

Specifies this is a subclass of Practitioner

No-arg constructor -automatic call to superclass no-arg constructor

Doctor class

```
/**
  * constructor with firstname etc
  */
 public Doctor(String firstName, String lastName,
                String medID, String specialty)       {

     // note call to superclass constructor
     super(firstName, lastName, medID);
     this.specialty = specialty;
 }
```

Invoke the superclass constructor by using super

Doctor class(Doctor.java)

```
    public String getSpecialty(){
         return specialty;
    }
    public void setSpecialty(String specialty){
        this.specialty = specialty;
    }

     @Override
     public String getName(){
        return "Dr. "+getFirstName()+" "+getLastName();
    }
}
```

Getters/setters for specialty

*Same **signature** as method in superclass - overrides getName() in superclass*

Creating practitioners, doctors, and pharmacists

The class Practitioners (next slide)

- Creates objects
- Adds each object to a list
- Iterates through the list
  - displays the type of object
    - `instanceof` is used to determine the type of an object
  - displays the object's first name
    - getFirstName is defined in Practitioner

Practitioners.java

```
public class Practitioners {
   public static void main(String[] args){
      // List of practitioners
      ArrayList<Practitioner> practitioners =new ArrayList();
      // Create some practitioners
      Practitioner pr = new
               Practitioner("Sam","Smith","Prac222")
      ;  Doctor dr = new
               Doctor("Jill","Jones","Doc111","Dermatology");
      Pharmacist ph = new
               Pharmacist("Eddy","Edwards","Pha222","Drugco"
      );  practitioners.add(pr);
      practitioners.add(dr);
      practitioners.add(ph);
      for (Practitioner p: practitioners) {
         String type="practitioner";
         if (p instanceof Doctor) type="doctor";
         if (p instanceof Pharmacist) type="pharmacist";
         System.out.println(type+" "+p.getFirstName());
   }}}
```

Create objects

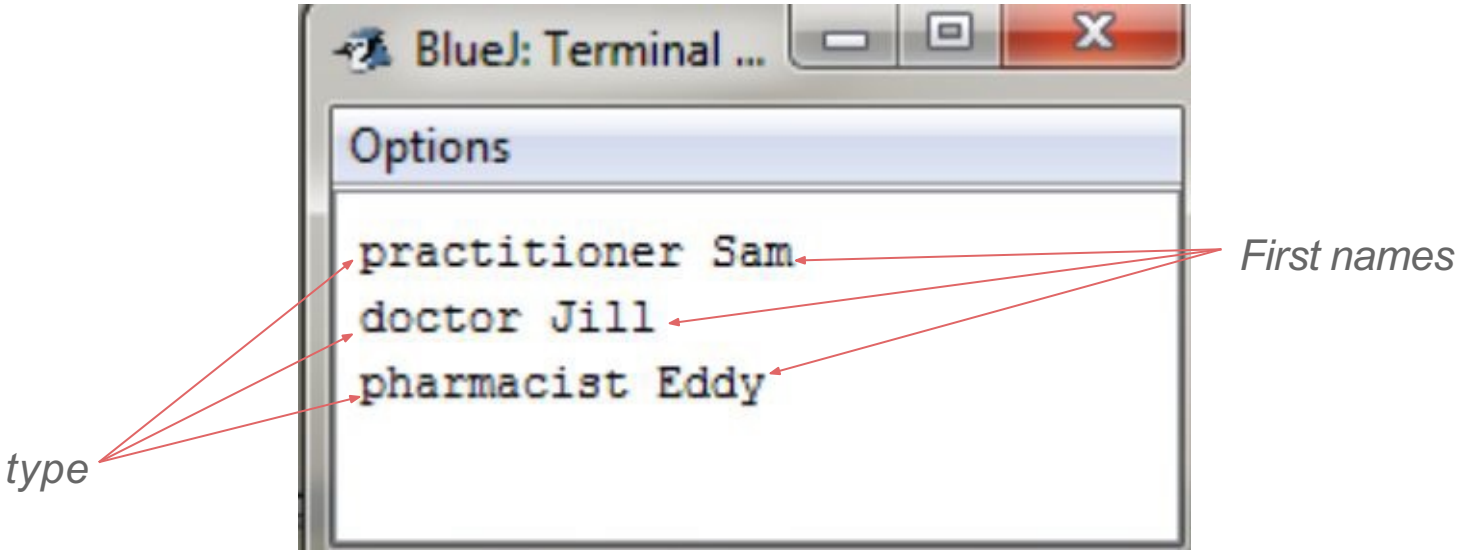
Add to list

Display type and firstname

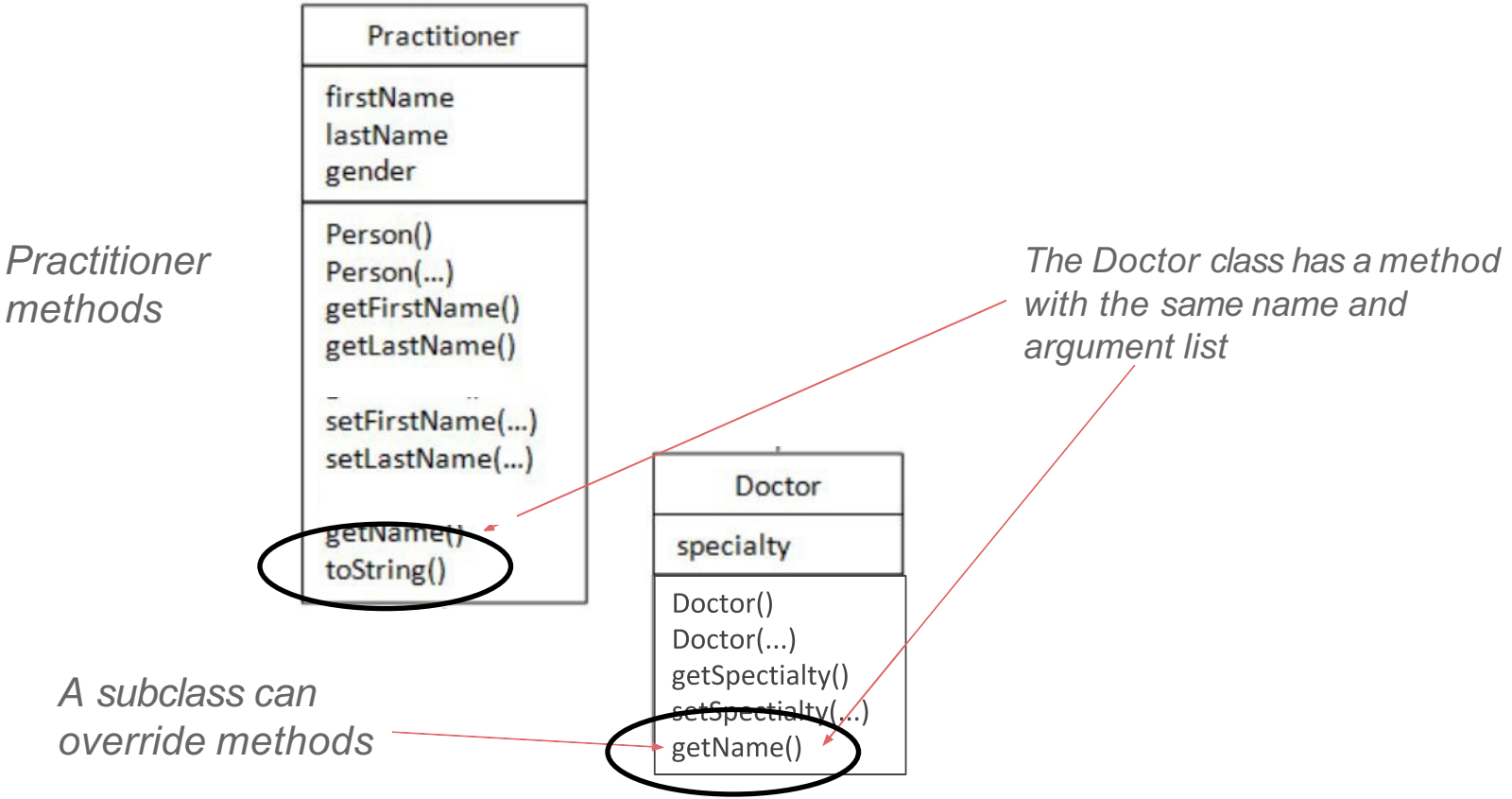
Instanceof operator

getFirstName() is defined in Practitioner

Output


BlueJ: Terminal ...
Options
practitioner Sam
doctor Jill
pharmacist Eddy
type
First names

Overriding methods - getName()



Overriding methods - getName()

If not a Doctor object JVM uses:

In Practitioner:

```
public String getName(){
    return firstName+" "
        +lastName;
}
```

If a Doctor object JVM uses:

In Doctor:

```
public String getName(){
    return "Dr."
        +getFirstName()+" "
        +getLastName()+","
        +getSpecialty();
}
```

Different behaviour according to the type of object

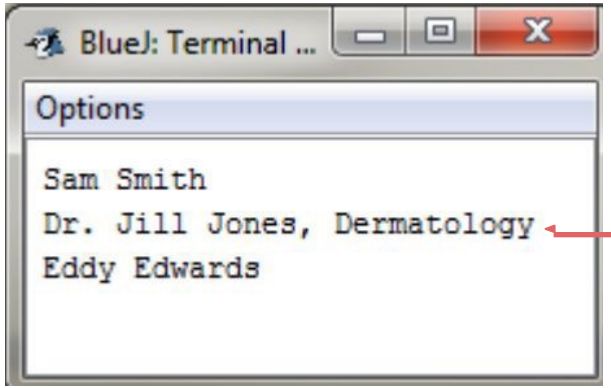
The method used is determined at runtime

- polymorphically
- happens automatically

Overriding methods - getName()(Practitioners1.java)

Consider:

```
for (Practitioner p: practitioners) {
    // display name
    // getName() in Doctor overrides
    // getName() in Practitioner
    System.out.println( p.getName() );
}
```



A Doctor object
→ different behaviour

`super` prefix

Methods in a subclass *automatically* override methods in a superclass

- You can specify a particular superclass method to execute if you use the *super* prefix, e.g.:

```
super.getName()
```

- Need to be careful… suppose getName in Doctor was coded as:

```
public String getName() {
    return "Dr. "+ getName()+", "+getSpecialty();
}
```

*No prefix, hence this method is referring to itself and this is now a **recursive** call (a later topic)*

`protected` access control modifier

`protected` designates that access is only granted to classes that are designated as subclasses of the given class through inheritance

```
public class Practitioner {
    protected String firstName;
    protected String lastName;
    protected String sign;

...
```

```
public class Doctor extends Practitioner{

...

    public String getName(){
       return "Dr. " +firstName+ " " +lastName;
    }
}
```

Abstract classes and methods

An abstract class cannot be instantiated.

Abstract classes and methods

Suppose we never instantiate a Practitioner object.

- We can make Practitioner abstract to prevent that from accidentally happening:

```
public abstract class Practitioner
```

- And now you get a compile time error if you code:

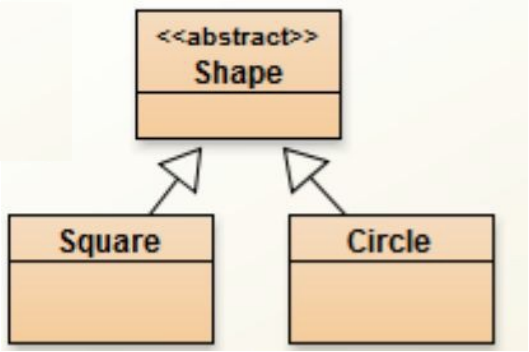
```
Practitioner p = new Practitioner();
```

Shapes example

Shape is abstract and so cannot be instantiated.

Shape also has an abstract method, area

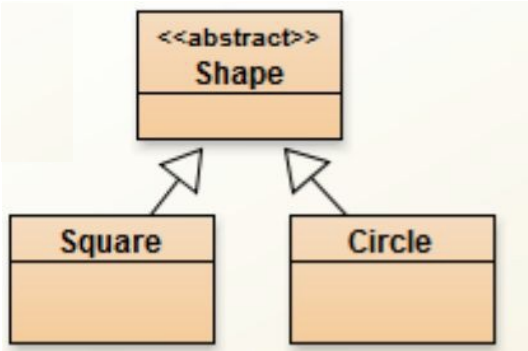
- no code, just a header
- subclasses must have code



Square and Circle are not abstract

- we say they are concrete subclasses
- these can be instantiated

Shapes example




```
public abstract class Shape {

    protected String id;

    public Shape (String id){
        this.id = id;
    }

    public abstract double area();

    public String toString(){
        return id;
    }
}
```

Shapes example

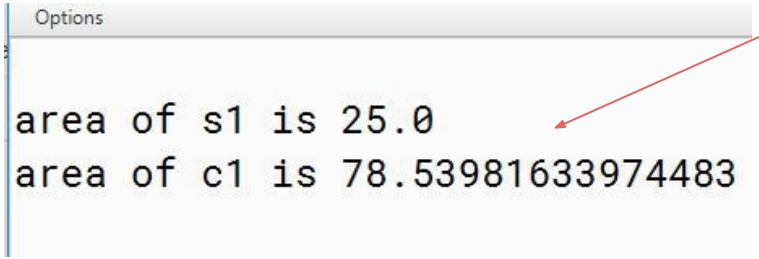
Shape.java
Circle.java
Square.java

```
public class Square extends Shape {

    private int length;

    public Square(String id, int
length){
        super(id);
        this.length = length;
    }

    public double area() {
        return length*length;
    }
}
```

```
public class Circle extends Shape {

    private int radius;

    public Circle (String id, int radius){
        super(id);
        this.radius = radius;
    }

    public double area(){
       return Math.PI*radius*radius;
    }
}
```

Shapes example

```
import java.util.ArrayList;
public class UseShapes {
public static void main(String[] args) {
    ArrayList<Shape> shapes =  new ArrayList();
    shapes.add(new Square(5));
    shapes.add(new Circle(5));
    for (Shape s: shapes)
       System.out.println( s +" "+ s.area() );
    }
}
```



different area calculations/methods used

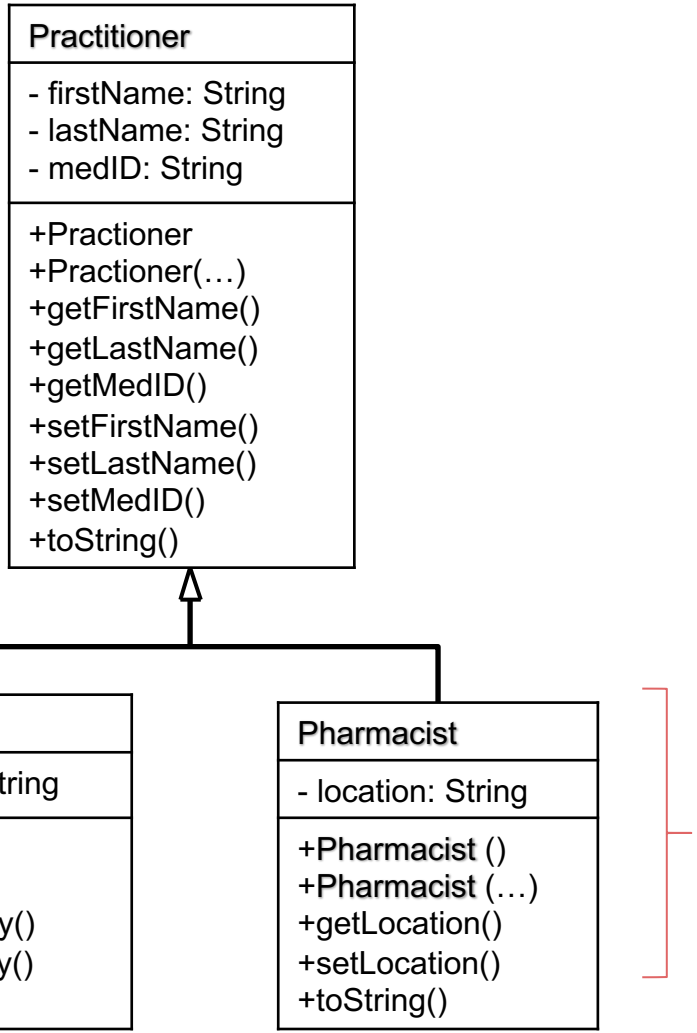
Summary

- All classes are subclasses of Object
  - Object has the default definitions for `toString`, `equals`, etc.
- In a class diagram use the Δ symbol to designate the superclass
- An object instantiated from a subclass is an instance of that subclass, and at the same time it is an instance of its superclass
- `instanceof` is a binary operator used to test an object's type
- Subclass methods override superclass methods

Summary

- At runtime the JVM determines the method to run for an object based on the object's type
  - i.e. polymorphically
- Abstract classes cannot be instantiated
- An abstract method has no implementation
  - implementation is left to the subclasses
- Subclasses can have only one superclass
  - hence, we have hierarchies
- Enums cannot be arranged in a hierarchy (there is no "is a" relationship with Enums only "has a")

Practitioner: a more complete UML class diagram



A doctor has these fields/methods **plus** all those of Practitioner too

Pharmacist