10A. Polymorphism

- Objectives - when we have completed this set of notes, you should be familiar with:
  - defining polymorphism and its benefits
  - using inheritance to create polymorphic references
  - using interfaces to create polymorphic references

Polymorphism

- The term *polymorphism* literally means "having many forms"
- A *polymorphic reference* is a variable that can refer to objects of various types
- A method invoked through a polymorphic reference can change from one invocation to the next

Polymorphism

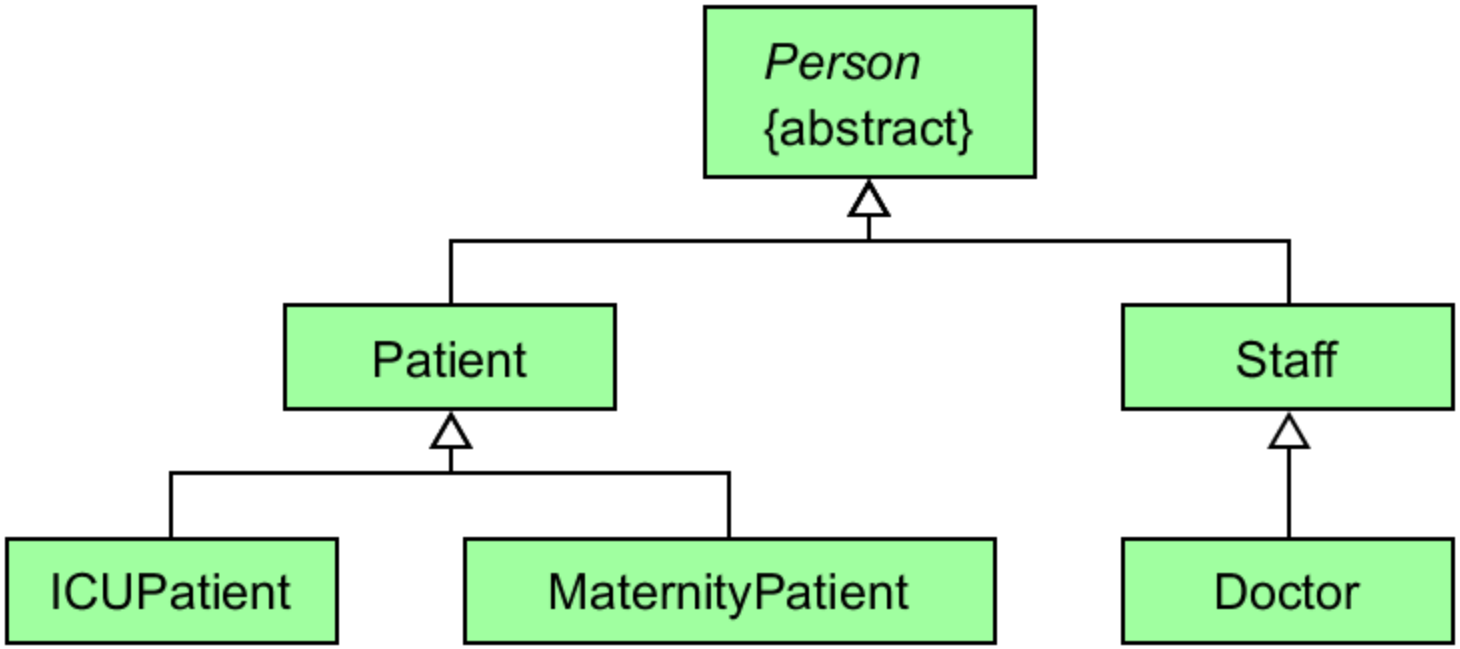
- Suppose we create the following reference variable:

```
Occupation job;
```

- Java allows the variable job to reference an `Occupation` object or any other object of a compatible type
- This compatibility can be established using **inheritance** or using **interfaces**

References and Inheritance

- Consider the following inheritance hierarchy:



References and Inheritance

- An object reference can refer to an object of its class, **or to an object of any subclass**
- For example, the following code is valid:

```
Person p = new Patient("Jane", "Lane", "US", 1970);
p = new ICUPatient("Jake", "Lane", 19, 1980);
Staff s = new Doctor("Joan", "Lane", "Lab", "");
```

- Assigning a child object to a parent reference is considered to be a *widening conversion*

References and Inheritance

- Assigning an parent reference to a child reference can be done also, but it is considered a narrowing assignment and must be done with a cast. In addition, the object referenced by the parent type must be an instance of the child class or one of its subclasses. Example:

```
Staff s = new Doctor("Joan", "Lane", "Lab", "");
Doctor d = (Doctor) s;
```

- The widening conversion is the most useful

Polymorphism via Inheritance

- When an object reference is declared as a parent type, you only have access to the methods in the parent class.
  - The methods specific to the child class can only be accessed using casting.
- For example, you cannot invoke s.setSpecialty() on the s object below even though setSpecialty is defined in Doctor.java. A cast is required since it is unknown to Staff.

```
Staff s = new Doctor("Joan", "Lane", "Lab", "");
((doctor)s).setSpeciality("Surgeon");
```

Polymorphism via Inheritance

- When a method is invoked on object referenced by a superclass variable, although the method must be present in the superclass, the version of the method associated with the subclass is the one that is actually invoked.
- Consider PrintHospital.java
- Note that different versions of getId() are invoked even though all objects in personArray are of type Person.

```
for (Person pObj : personArray) {
   System.out.println(pObj.getId());
}
```

Binding

- Consider the following method invocation:

```
obj.toString();
```

- At some point, this invocation is *bound* to the definition of the method that it invokes
- If this binding occurred at compile time, then that line of code would call the same method every time
- However, Java defers method binding until run time -- this is called *dynamic binding* or *late binding*
- Thus, in our PrintHospital example, the appropriate getID() method was bound as the program was running

Polymorphism via Interfaces

- An interface name can be used as the type of an object reference variable

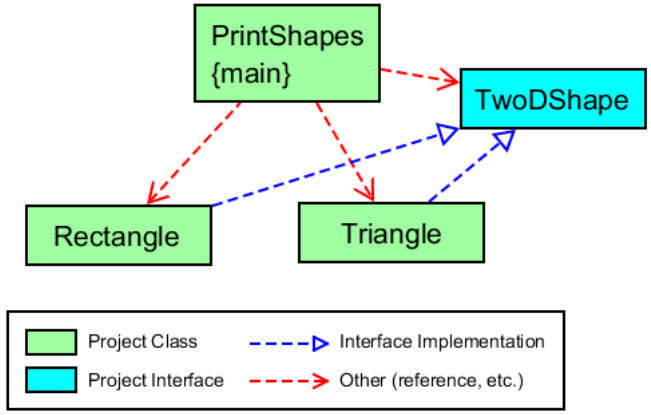
```
Comparable current;
```

- The current reference can be used to point to any object of any class that implements the Comparable interface
- The version of compareTo that the following line invokes depends on the type of object that current is referencing

```
current.compareTo(cObj);
```

Polymorphism via Interfaces

- Recall that the Rectangle and Triangle objects both implement the TwoDShape interface. PrintShapes.java provides an example of polymorphism via inheritance.



- Obviously the shape1 reference can access both getNumberSides and getPerimeter, but note that it can also access the toString method for printing.

Polymorphism

- All object references in Java are potentially polymorphic, because all classes inherit from the Object class and all interfaces have implicit abstract methods that match the public methods in the Object class unless these methods are explicitly declared [JLS 9.2]).
- The object class has the following methods (as well as others)
  - clone(), equals(), toString()
- Thus any object reference, regardless of its declaration type, can access the above methods

Polymorphism

- The lecture on Wednesday will continue with polymorphism and provide an introduction to ordering and sorting objects.
- Make sure that you study both sets of notes for the quiz on Wednesday.