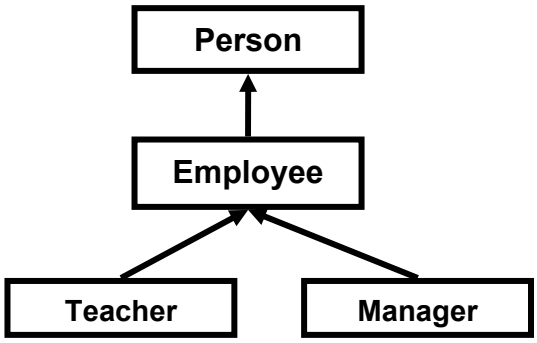
Polymorphism

What?

Polymorphism literally means "of multiple shapes/forms" and in the context of OOP, polymorphism means "having multiple behaviors".

How?

Consider the following class hierarchy. We have already discussed this hierarchy in the handout on "Inheritance".



- Normally, when we want to create an object of a class we call its constructor. To create an object of class Person we write

```
Person per = new Person ( );
```

  There is nothing strange in the above line of code as we create objects of a class by calling its any of the available constructor.

- BUT, polymorphism allows us also to write the following line of code

```
Person per = new Employee ( );
```

  Don't be surprised; just revisit the concept of inheritance i.e. *is a* relationship. Since we can say that *Employee is a Person.* It means that a variable of type Person can point to an object of type Employee. That's what we have translated above in the form of code. What do you think about the following line?

```
Person ahmad = new Teacher ( );
```

```
Employee usman = new Manager ( );
```

These are 100 percent correct and rightly so, because Teacher *is a* Person as well as Manager *is an* Employee.

Note: The reversal of the above is not necessarily to be true. For example, Person is not necessarily a Teacher. As according to the above given hierarchy, Person can be either an Employee or Person can be a Teacher or a Manger.

Rule of thumb

- Since a child class object is also an object of its parent class therefore it can be pointed by a reference of parent class. Or in other words "A parent class reference variable can be used to point any objects of its descendants (subclasses)".

Understanding Polymorphism

- Just cast a glance over the hierarchy given above so that we can completely understand the concept of polymorphism.
- From the hierarchy, we came to know that Person is a (direct) superclass of Employee and also Person is a (indirect) superclass of Teacher as well as of Manager.
- Hence, by bearing in mind *is a* relationship, we can say that Employee is a Person or Teacher is a Person or Manager is a Person.
- Or we can say that Person can take the form of Employee, Person can take the form of Teacher and Person can take the form of Manager. (Remember that polymorphism means "of multiple forms").

Polymorphic Methods

Happily, there is no new concept under this title. In fact, overridden methods are so called polymorphic ("having different behavior").

Just think a while, why we need method overriding? Of course, overriding allows the subclass to modify the behavior (method) of the superclass as needed. This indicates that a method in subclass will have a different behavior from the one in the superclass.

Point to Remember

- A polymorphic method results in different actions (behaviors) depending on the object being referenced. This phenomenon often referred as *late binding* or *run-time binding*.

  This has been explained in the following example:

Polymorphism Example

The `Test.java` class uses the instances of class `Employee` and `Teacher`. We have already defined these classes in the handout on "Inheritance". These classes are used in this example without any modification.

Instructions

- Modify the file `Test.java` of inheritance example, one we defined in the handout on "Inheritance".
- Write the following lines of code inside `Test.java`.
- Make sure that `Employee` and `Teacher` classes are available in the current directory you are working in to run the `Test` class.

Code

The Test class acts as a driver class as it contains the main method. Objects of Employee & Teacher class are created inside main and calls are made to display and toString method using these objects.

```
class Test{

  public static void main (String args[]){

    // Make employee references
    Employee ref1, ref2;

    ref1 = new Employee(89, "khurram ahmad");

    // is-a relationship, polymorphism
    ref2 = new Teacher (91, "ali raza", "phd");

    ref1.display(); //call to Employee class display method
    ref2.display(); //call to Teacher class display method
```

```
    System.out.println("Employee: " +ref1.toString());

    System.out.println("Teacher: " + ref2.toString());


 } //end of main
}//end class
```

Understanding Run-Time Binding

In the above code example, two Employee class references namely *ref1* & *ref2* are created. An *Employee* object is assigned to *ref1* & Teacher object is assigned to *ref2* (polymorphism). This is shown in the example code as

```
ref2 = new Teacher (91, "ali raza", "phd" );
```

Now, take next line of code, i.e.

```
ref1.display ( );
```

It will make call to display method of Employee class. But the next line of code contains the real interesting part.

```
ref2.display ( );
```

Apparently, it seems that it will also make a call to the display method of Employee class object, but this does not happen. Actually, it makes call to display method of Teacher class. Let's closely examine what is happening behind the scenes?

When we compile our Test class using javac command, the compiler will ensure that display method should exist in the Employee class. On execution of Test program, java will check that ref2 is pointing to which class's object? In our case, it is pointing to Teacher class that's why it will call the display method of Teacher class.

As all this has happened at the run time or at the execution time of the program, for this reason it is called run time binding or late binding.

Type Casting

Java is a strongly typed language. You cannot declare a variable or a reference of a class without specifying its type. Sometime there is a requirement to convert (cast) one data type into another. There are two types of casting: UpCasting & DownCasting. While casting, following points should be kept in mind

UpCasting

In upcasting we convert a smaller datatype into a larger datatypes. Since converting a smaller dataytype into a larger datatype does not cause any information loss therefore upcasting is implicit (i.e. it occurs automatically and we do not have to write any extra piece of code)

DownCasting

In downcasting we convert a larger datatype into a smaller datatype. Since converting a larger dataytype into a smaller datatype may cause an information loss therefore downcasting is explicit (i.e. it does not occur automatically and we have to tell compiler that we want to downcast otherwise it won't let us compile the code)

Upcasting and Downcasting in Primitives

In primitive datatypes such as int, float, double upcasting occurs when we convert a smaller data type (in terms of size (bytes)) into a large data type. For example; converting int (4 bytes) to long (8 bytes) or float (4 bytes) to double(8 bytes). This has been shown in the following code fragment.

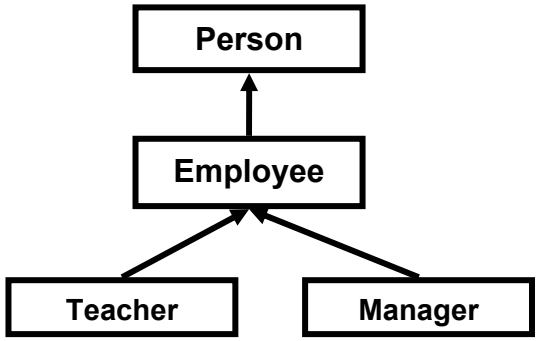
```
int i = 4;
double d ;
d = i;
```

Since double is a bigger type, conversion is automatic but when we convert from double to int there can be a loss of information and therefore explicit casting is required (i,e, we are telling the compiler that we know what we are going to do, so do it.)

```
double  d = 4;
int i ;
i =(int)d;
```

Explicit typecasting of double into an integer

Upcasting and downcasting in Objects



Often students find it hard to understand the concept of upcasting in classes. They think that since Teacher class is a child of Employee so it should contain more variables then Employee (due to inheritance, its own + that of Employee) and therefore objects of Teacher class need more memory and hence Teacher is a larger Datatype.

This concept is wrong, a simple rule of thumb is that a class that is higher in the hierarchy is a larger class so you can implicitly upcast your child objects to parent class type. Here Employee class is a bigger class then Teacher or Manager class, similarly Person is larger class then all of Employee, Teacher or Manager. To understand why, from inheritance we know that every teacher is an employee and every manager is also an employee, it means that Employee class has more objects than either of Teacher or Manager (It contains its own objects + objects of Teacher + objects of Manager), which shows that Employee is a larger class/datatype and upcasting to larger datatype is implicit. Also another view could be that since Teacher and Manager are Employee (due to inheritance) hence a variable declared of type Employee can point to objects of type Teacher or Manager

```
Employee e ;
Teacher t = new Teacher()
e = t ;
```

Here ‘e’ is of type Employee and ‘t’ is of type Teacher and ‘t’ is also pointing to a teacher object since Teacher is an Employee (due to inheritance) therefore ‘e’ can point to object pointed by ‘t’. Don’t need to do anything implicit /upcasting takes place

However converting from Teacher to Employee won’t be possible without explicit casting e.g

```
Employee e = new Teacher() ;
Teacher t;
t = (Teacher)e ;
```

Here though ‘e’ is pointing to a teacher object but still we can not assign it to ‘t’ without explicit downcast because Teacher is a smaller data type. Also since ‘e’ can point to Employee object as well as Teacher object as well as Manager object so it would not be possible for the compiler to tell in which type we want to cast ‘e’ hence we have told the compiler explicitly here that we want to convert the object pointed by ‘e’ to Teacher datatype. Had it been a wrong cast, for example suppose instead of pointing to a Teacher object ‘e’ is pointing to a Manager object then it would have caused the exception.