Inheritance

Learning Objectives

Learn how to use **inheritance** to create **subclasses** that extend **superclasses**.

Agenda

- Warm-up
- Learn concepts: inheritance, superclass, subclass
- Learn inheritance Java syntax:
    - Classes
    - Constructors
- Overriding methods

Warm-up

Column A	**Column B**
person	triangle
	teacher
shape	square
	student

1) Choose one of the items from Column A and say the “IS-A” relationship with the items you connected it with from Column B
2) What are some attributes (instance variables) you could use to describe the Column A item you chose which the Column B items could share.
3) What are some actions (methods) your Column A item does or things that could be computed about your Column B?

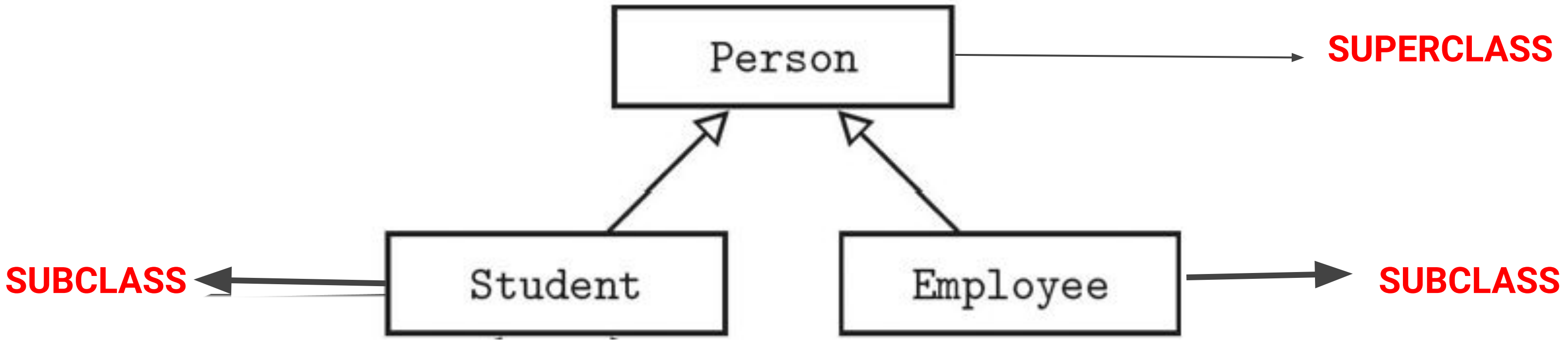
Inheritance

Inheritance defines a relationship between objects that share characteristics.

How this works?

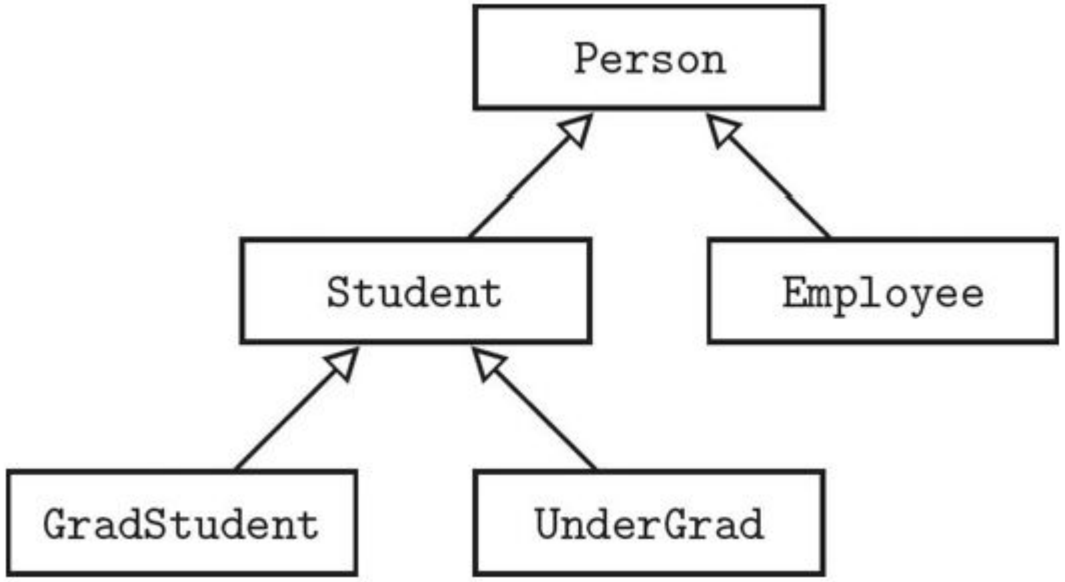
A new class, called a **subclass**, is created from an existing class, called a **superclass**, by absorbing its state and behavior and adding these with features unique to the new class.

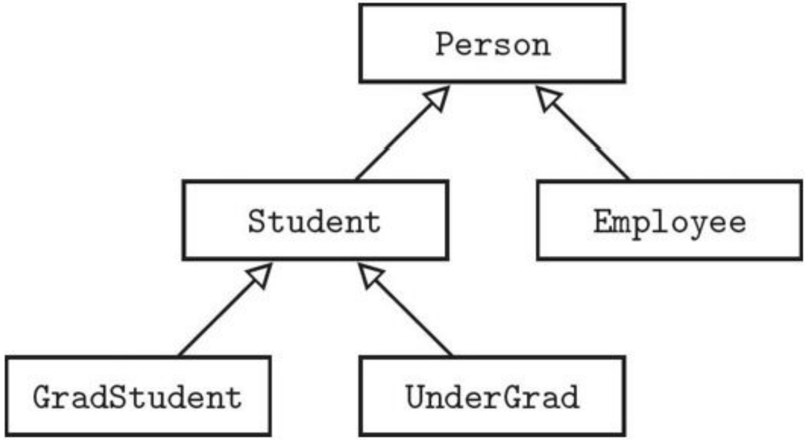
The subclass inherits characteristics of its superclass.



Inheritance Hierarchy

A subclass can itself be a superclass for another subclass, leading to an inheritance hierarchy of classes.





Employee **is-a** Person
Student **is-a** Person
GradStudent is-a Student
UnderGrad is-a Student

So, if a UnderGrad **is-a** Student and a Student **is-a** Person, then a UnderStudent **is-a** Person.

Note - The opposite is not necessarily true: A Person may not be a Student, nor is a Student necessarily an UnderGrad.

Superclass and Subclass Characteristics

- Constructor cannot be inherited (subclasses have a different name)
- Every subclass inherits the public or protected variables and methods of its superclass.
- Subclasses may have additional methods and instance variables that are not in the superclass.
- A subclass may redefine a method it inherits. This is called method **overriding**

Inheritance

Inheritance defines a relationship between objects that share characteristics.

Why we use Inheritance?

- Code reusability
- Prevents repeating code
- Readability and organization
- Ease of maintenance

Java Syntax: Superclass and Subclass

```
public class Superclass
{
        //private instance variables
        //constructors
        //public methods
        //private methods
}
```

```
public class Subclass extends Superclass
{
        //additional private instance variables
        //constructors (Not inherited)
        //additional public methods
        //inherited public methods whose implementation is overridden
        //additional private methods
}
```

Constructors Superclass

```
public class Student {
    // data members
    public final static int NUM_TESTS = 3;
    private String name;
    private int[] tests;
    private String grade;

    // constructor
    public Student() {
        name = "";
        tests = new int[NUM_TESTS];
        grade = "";
    }

    // constructor
    public Student(String studName, int[] studTests,
String studGrade) {
        name = studName;
        tests = studTests;
        grade = studGrade;
    }
}
```

Constructors Subclass

```
public class UnderGrad extends Student {
    // contructor
    public UnderGrad() {
        super();
    }

    // constructor
    public UnderGrad(String studName, int[]
studTests, String studGrade) {
        super(studName, studTests, studGrade);
    }
}
```

Let’s create another Subclass adding an instance variable

```
public class GradStudent extends Student {
    // additional data member
    private int gradID;

    // contructor
    public GradStudent() {
        super();
        gradID = 0;
    }

    // constructor
    public GradStudent(String studName, int[]
studTests, String studGrade, int gradStudID) {
        super(studName, studTests, studGrade);
        gradID = gradStudID;
    }
}
```

The **super** Keyword

The keyword **`super`** is very useful in allowing us to first execute the superclass method and then add on to it in the subclass.

Superclass references

A **superclass reference variable** can hold an object of that superclass or of any of its subclasses. This is a type of **polymorphism** which will be defined further in the next lesson.

For example, a `Student` reference variable can hold a `UnderGrad` or `GradStudent` object.

```
Student StudentOne = new Student();
Student StudentTwo = new UnderGrad();
Student StudentThree = new GradStudent();
```

Superclass references

The opposite is NOT true. **You CANNOT declare a variable of the subclass to reference a superclass object.**

For example, a **`UnderGrad`** reference variable cannot hold a **`Student`** object because **not all `Students` are `UnderGrads`**.

```
// A subclass variable cannot hold the superclass object!
// A UnderGrad is-a Student, but not all Students are
UnderGrads.
UnderGrad q = new Student();  // ERROR
```

Inherited get/set methods

Subclasses automatically inherit the instance variables and methods from the superclass.

But, if the inherited instance variables are private, which they should be, the child class cannot directly access the them using dot notation.

How then does the subclass access them?

The answer - by using the appropriate inherited `get` or `set` method.

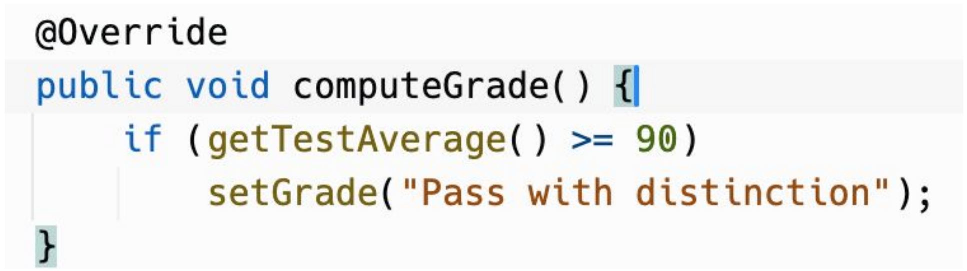
The child class can use public **accessors** (also called **getters** or **get methods**) which are methods that get instance variable values. The child class also can use public **mutators** (also called **modifier methods** or **setters** or **set methods**) which set their values.

Let’s write a **get** and **set** methods in our superclass Student

What if we want to modify the inherited method? How do we do that?

- We can modify inherited methods to add additional behaviors. This is called **overriding.**
- When we override a method, it *must* have the **identical method signature of the method in the superclass - same method name, parameter type list and return type.**
- The overridden method in the subclass will be called *instead* of the method in the superclass.
- Note that any method that is called must be defined within its own class or its superclass.

You may see the **`@Override`** annotation above a method. This is optional but it provides an extra compiler check that you have matched the method signature exactly.



```java
@Override
public void computeGrade() {
    if (getTestAverage() >= 90)
        setGrade("Pass with distinction");
}
```

Overriding versus Overloading

OVERLOADED METHODS:

Overloaded methods have the **same method name but a different number or type of parameters.**

Overloaded methods **appear in the same class.**

To overload a method the method must have the same name, but the parameter list must be different in some way. It can have a different number of parameters, different types of parameters, and/or a different order for the parameter types. The return type can also be different.

OVERRIDDEN METHODS:

Overriding applies only to inheritance when we are **using the exact same method signature in a subclass to modify the method's behavior.**