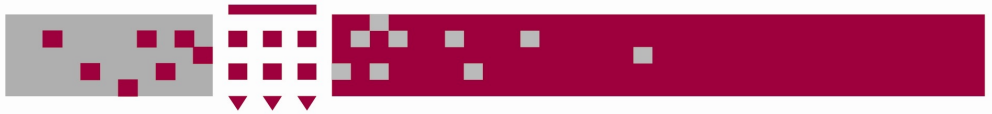
5COSC019W – Object Oriented Programming
Week 3

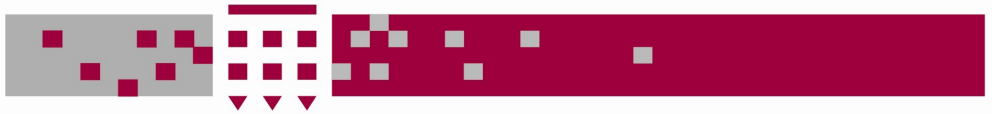
Dr. Barbara Villarini

b.villarini@westminster.ac.uk



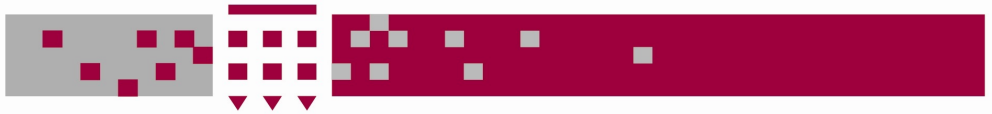
Summary

- Inheritance
- Create subclasses
- Override inherited methods
- Substitution principle
- Dynamic binding
- Polymorphism



What you learnt so far

- Classes are blueprints that we can use to create objects
- Objects have state and behaviour:
  - State: instance variables (e.g. balance)
  - Behavior: instance methods (e.g. withdraw)
- We create new objects using constructors
- We also have class methods and class variables
  - these belong to the blueprint (class) but not to any specific object

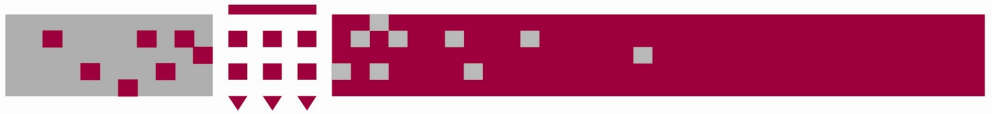


Make sure you understood:

- The concept of class / object
- How to create objects using constructors and the new keyword
- Static vs. instance contexts
- Access modifiers and how to call methods from other classes

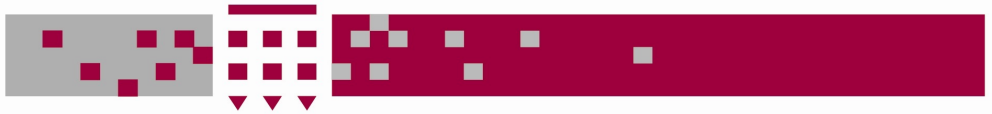
Subclasses

- In the real world, objects have similarities to other, seemingly different, objects
- Example: car **is-a** vehicle, bicycle **is-a** vehicle, truck **is-a** vehicle, Ferrari **is-a** car (and also a vehicle)
- Cars and trucks are both vehicles that have engines and that can carry a specific number of other properties
- Cars and trucks are special types of vehicles
  A vehicle is a more general type of a car or a truck



Inheritance

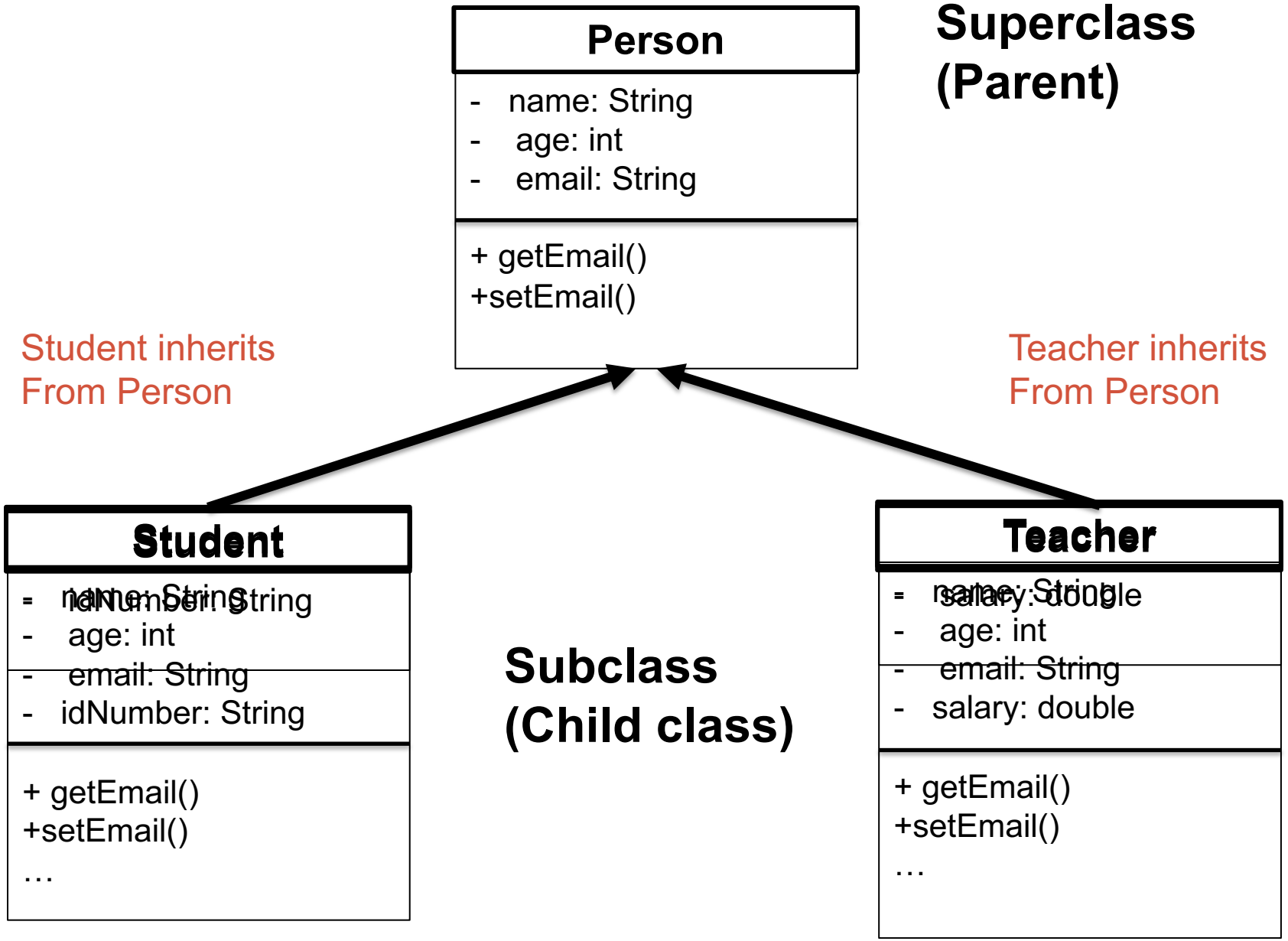
- In OOP these kind of relationships are modeled using **inheritance**
- car is a **subclass** of vehicle, vehicle is a **superclass** of a car
- **Inheritance is a basic concept of object-oriented programming**
- Inheritance allows a class to have a parent class from which it can inherit variables and methods



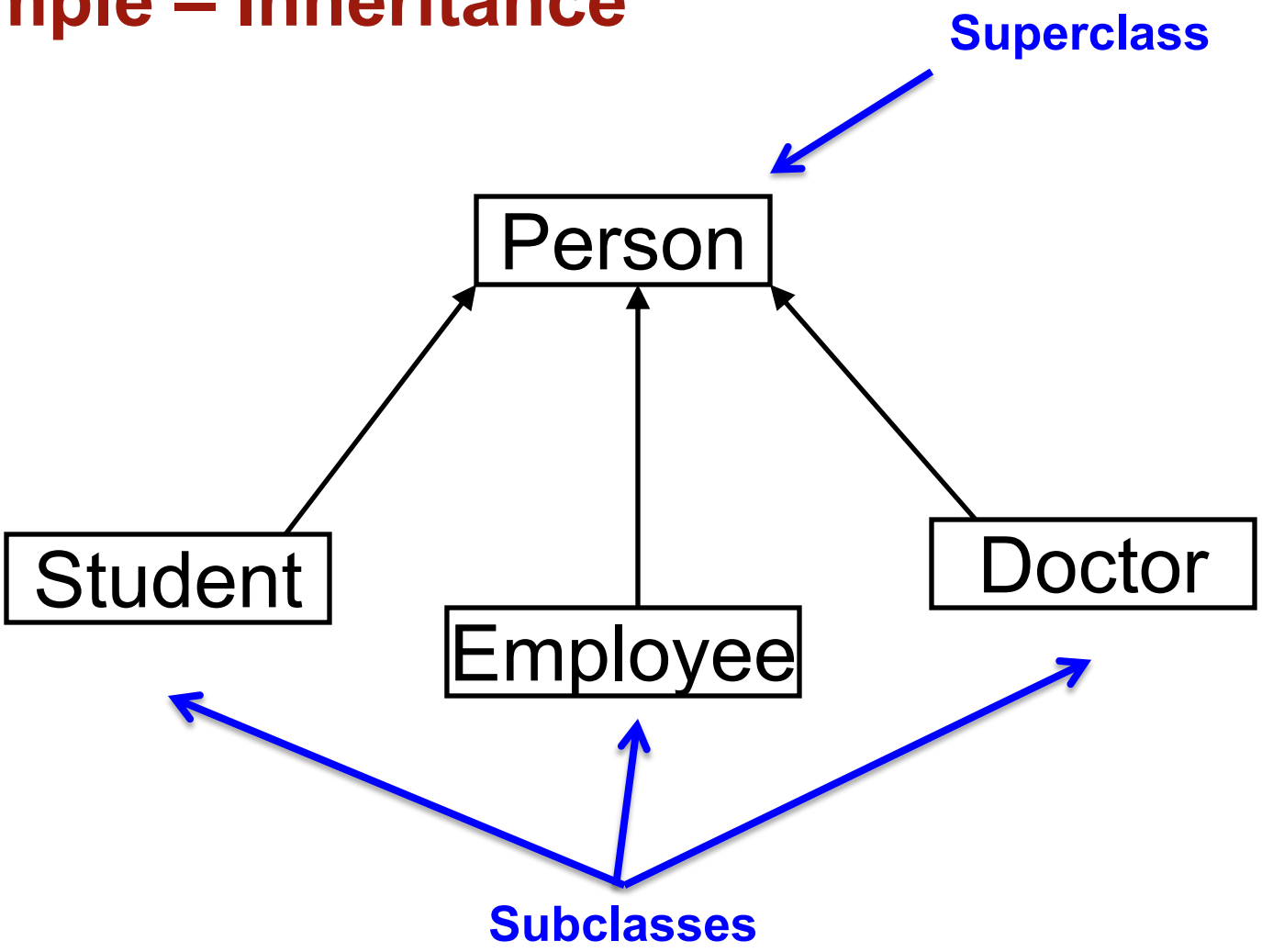
Inheritance in Classes

- If a class B inherits from class A then it contains all the characteristics (information structure and behavior) of class A
- The parent class is called *base* class and the child class is called *derived* class
- Besides inherited characteristics, derived class may have its own unique characteristics

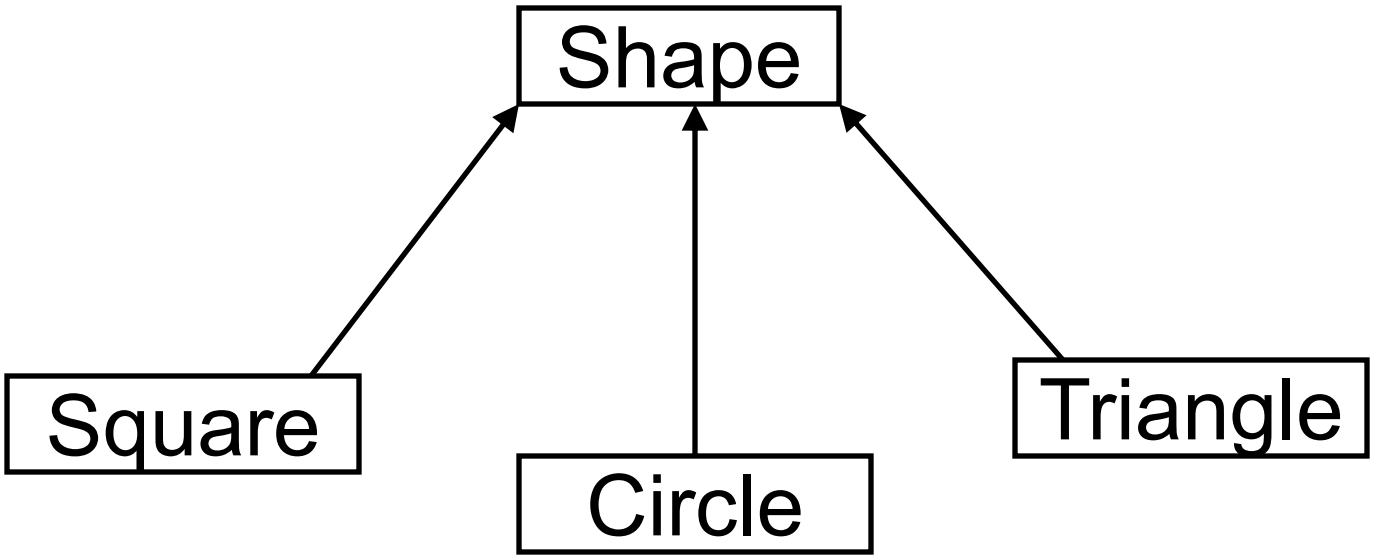
Inheritance



Example – Inheritance

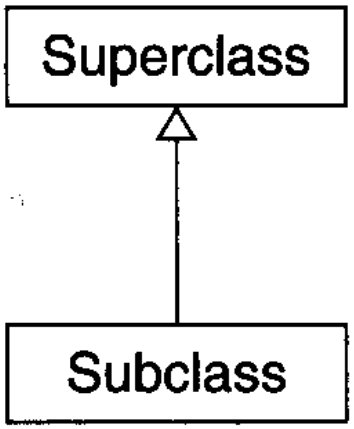


Another Example



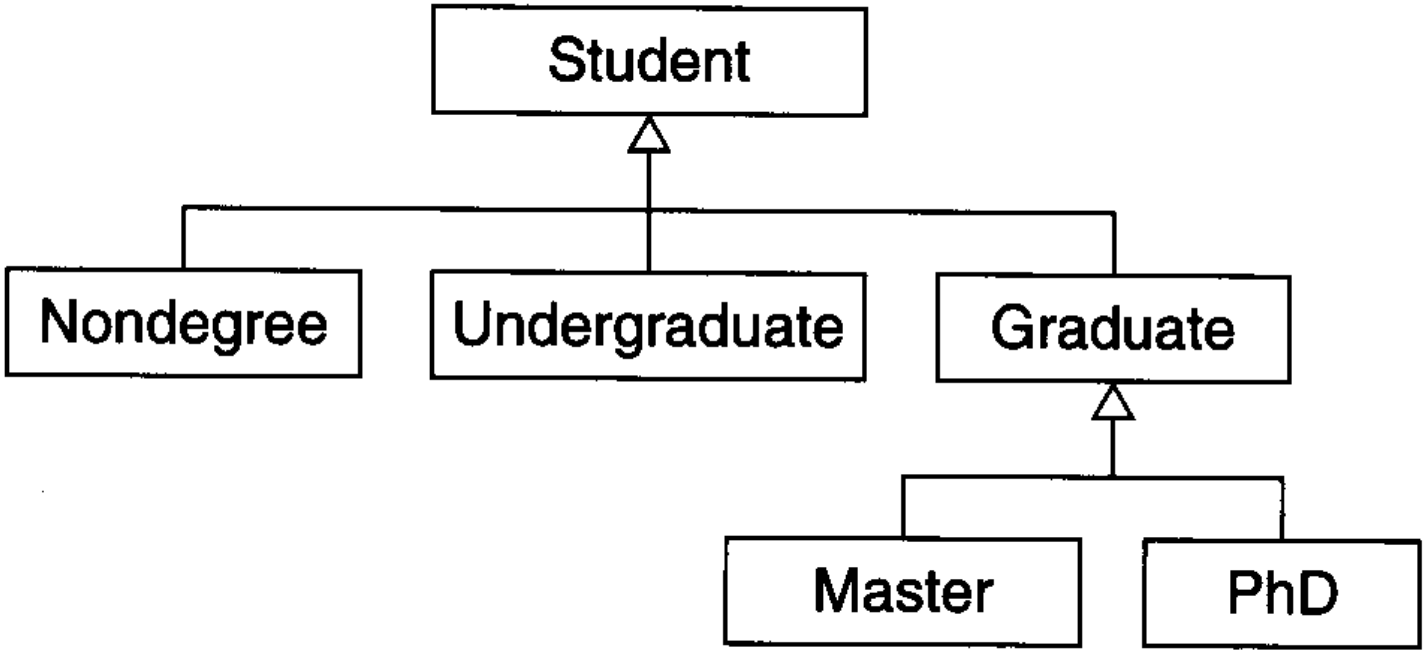
UML: Modeling Relationships

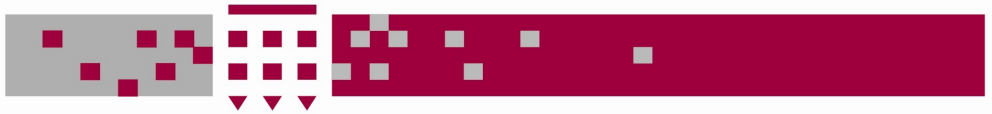
- Class diagrams can be used to model:
  - Inheritance -- extension and implementation



extension
of classes

Example





Inheritance – "IS A" or "IS A KIND OF" Relationship

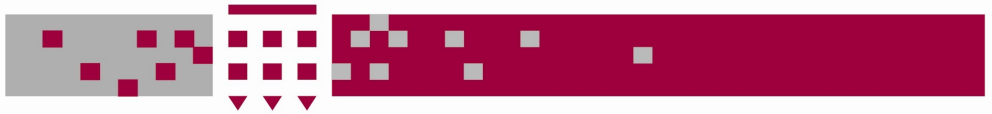
A Teacher is a Person
A Student is a Person
A Student is a Module

A Car is a Vehicle
A Motorbike is a Vehicle
A bus is a Car

A SavingAccount is kind of BankAccount

A Ferrari is a Car is a Vehicle

A Cat is a Mammal is an Animal



Inheritance – Advantages

- Reuse
- Less redundancy
- Increased maintainability

Class extension

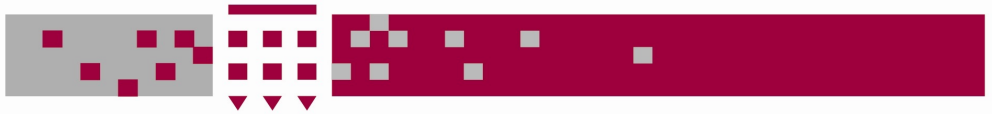
- Members of subclasses inherit variables and methods from their superclass(es)
  - they **do not** inherit private variables & methods, or constructors
- But they also can have their own special instance variables and methods that are not present in the superclass

```
public class Employee extends Person {
... }
```

Class extension in Java

```
public class Employee extends Person {
... }
```

- Employee is a subclass of Person
- Person is the superclass of Employee
- Employee inherits from Person

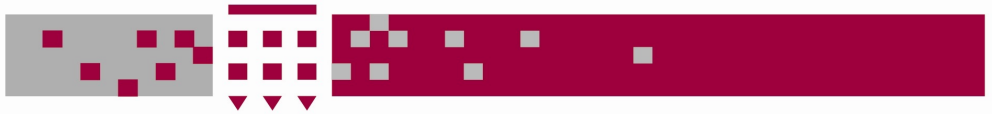


Example – Class Person

```
public class Person {
 private String name;
 private String dob;

public Person(String n, String d) {
      name = n;
      dob = d;
    }
Public Person (String n){
    name = n;
}
  public void setName(String newName) {
      name = newName;
    }
  public String getName() {
      return name;
    }
  public String getDoB() {
      return dob;
    }

}
```



Example – Class Employee

```
public class Employee extends Person {
  private double salary;

  public Employee(what goes here?) {
    // What to write here?
    }
  public double getSalary() {
      return salary;
    }
  public void setSalary(double newSalary) {
      salary = newSalary;
    }
}
```

Exercise 1

- Remember: Employees are also Persons
- What methods can we call on Employee objects?
  - setSalary
  - getSalary
  - setName (inherited)
  - getName (inherited)
  - getDoB (inherited)
- If `emp` is an `Employee` object:

```
String n = emp.getName();
emp.setName("John Smith");
emp.setSalary(25000.0);
```

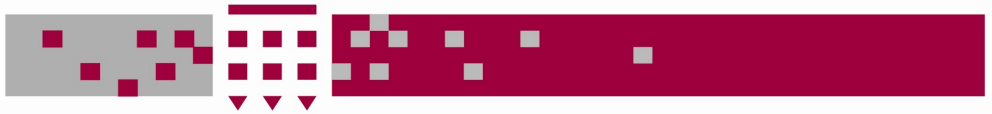
Exercise 2

- if `p` is a `Person` object, we can have:

```
p.setName("Dr Who");
String n = p.getName();
```

- Can we have `p.setSalary(25000);` ?

NO! Because the method is declared in the subclass!



Overview so far

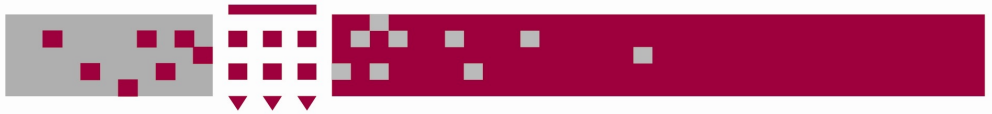
- `Employees` are also `Persons`, so they inherit methods from `Persons`
  - Java looks for a method first in the class to which the calling object belongs
  - If it does not find it there, Java looks in the class's superclass, ...
- `Persons` are not always `Employees`, so `Persons` will not have access to the methods that are defined only within `Employees`
- what about variables?

Inheritance & instance variables

- An instance of a subclass **stores all** the instance variables of the superclass (even private ones), plus all the instance variables defined in the subclass.
- **Be careful** though, **private instance variables** of the superclass are **NOT INHERITED**

`Employee` object

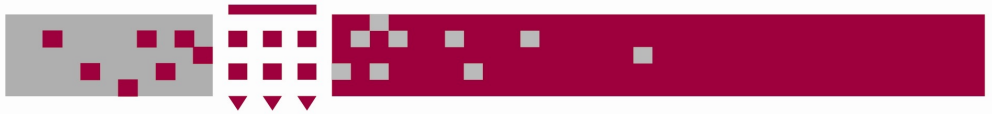
Name : Ted White Dob: 11/12/1990 Salary: 25000



Instance Variable

Problem: superclass variables are likely to be private

- Why private?
- Why is this a problem?



Subclass Constructor

- A subclass does not inherit constructors from the superclass
- The subclass constructor will need to initialise instance variables that belong to the subclass and to the superclass (see previous slide)

Exercise: which is the problem?

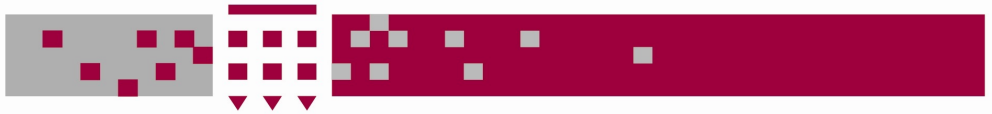
```
public class Employee extends Person {
...
public Employee(String initialName, double initialSalary) {
        name = initialName;
        salary = initialSalary; }
...
}
```

Solution: `Super` keyword

- We invoke superclass constructors by using the super keyword

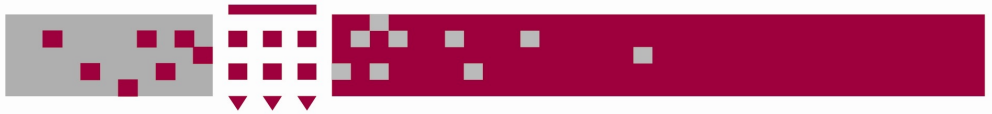
```
public Employee(String initialName, double initialSalary)
{
      super(initialName);
      salary = initialSalary;
}

Employee e = new Employee("Ted White", 25000.0);
```



Super: some rules

- super **must come first**, before the other statements in the body of the subclass constructor
- the order, type and number of the arguments we pass to super from the subclass must match those of the constructor of the superclass
- If a subclass has no constructors at all, Java will create a no-arguments constructor that contains only super();



A new access modifier: protected

- For making access to methods and variables easier between classes in an inheritance relationship, the protected access modifier is available
- private: can be accessed only in same class
- protected: can be accessed in the same class, or in a subclass, or in the same package
- public: can be accessed in any class

Access modifier: protected

- Protected variables and methods can be accessed by subclasses, subclasses of subclasses, etc.
- protected vs. private
  - declaring variables as protected exposes them to all subclasses
  - best to declare variables as private (even in inheritance relationships) and write getter and setter methods to provide access to variables

Your poll will show here

1

Install the app from pollev.com/app

2

Make sure you are in Slide Show mode

Still not working? Get help at pollev.com/app/help

or

Open poll in your web browser

Your poll will show here

1

Install the app from pollev.com/app

2

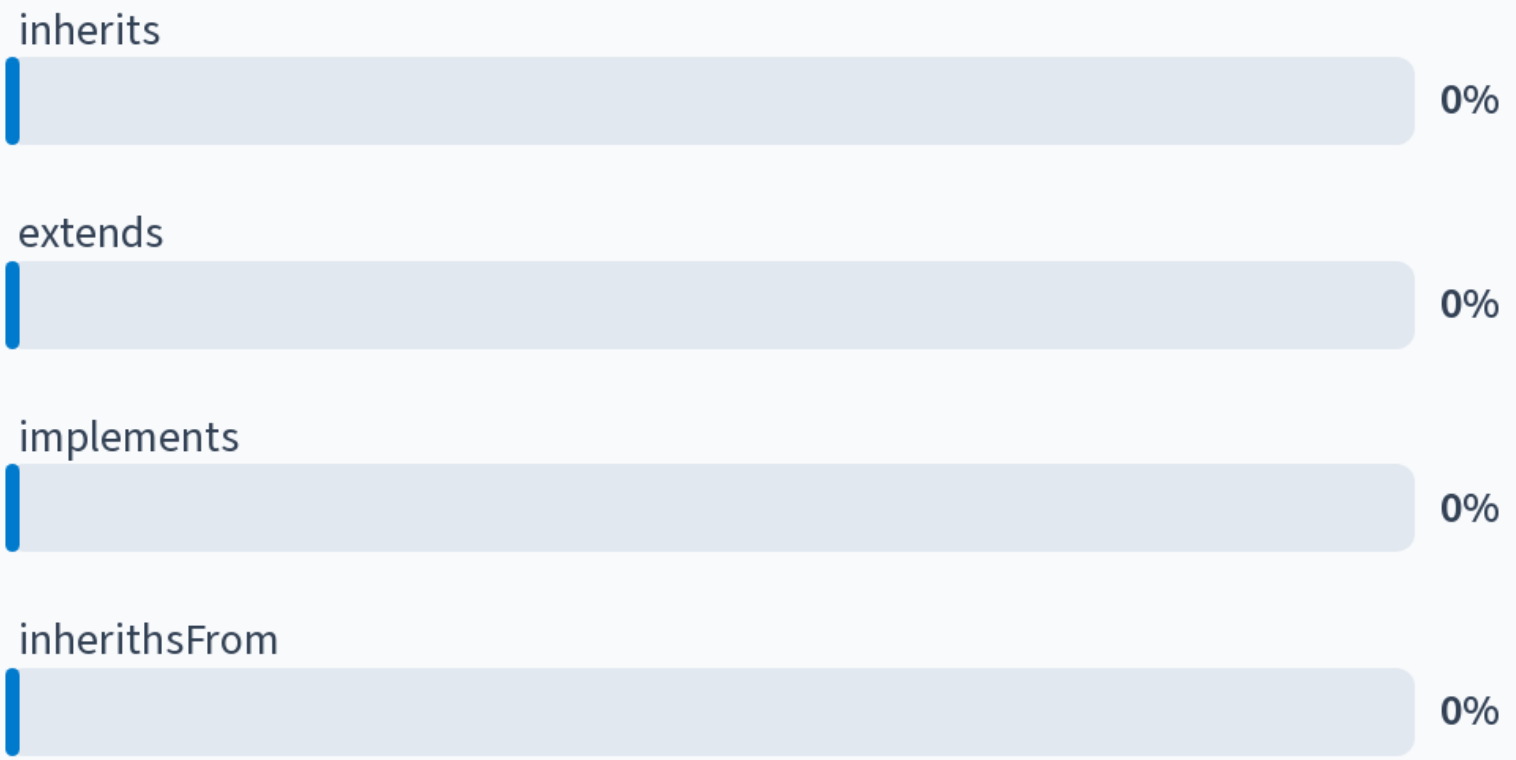
Make sure you are in Slide Show mode

Still not working? Get help at pollev.com/app/help

or

Open poll in your web browser

Which keyword is used to establish inheritance in Java?



Which of the following statements is true about the "super" keyword in Java?

It is used to access a subclass's methods and fields

0%

It is used to call a superclass's constructor

0%

It is used to create an instance of a superclass

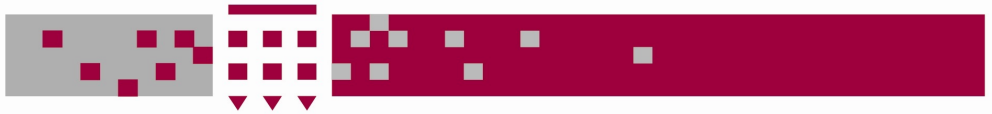
0%

It is used to override a superclass's method

0%

Overriding

- SavingAccount is-an Account (so SavingAccount is a subclass of Account)
  - Assume we can print a statement from a generic Account
  - We can print a statement from the SavingAccount with additional information
- A subclass can override an inherited instance method, by supplying a new method with:
  - the same name
  - the same number of parameters
  - the same type of parameters as the original inherited method



Example - Account

```
public class Account {

private double balance;
...

public void printStatement() {
     // print statement and account details
     }
}
```

Example – SavingsAccount

```
public class SavingsAccount extends Account {

private double interestRate;
. ..

public void printStatement() {
      print the normal statement like Account but
      after that print also details of interestRate,
and   other Savings specific information
      }
...
}
```

How can we call the overridden method printStatement (of Account) from within SavingsAccount?

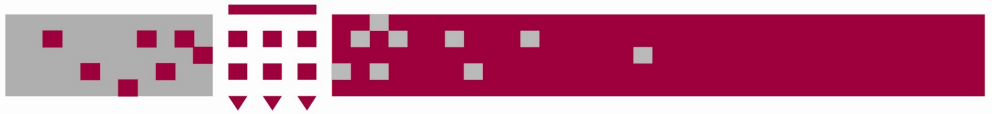
Example – SavingsAccount

```
public class SavingsAccount extends Account {

private double interestRate;
. ..

public void printStatement() {
      super.printStatement();
      //but after also print details of
      //interestRate, and other Savings specific
      //information
      }
  ...
}
```

Java will look first in the direct subclass for a method called printStatement. If it does not find it there, it will look in the superclass of that class, and so on ...



Polymorphism

- ‘polymorphism' from the ancient greek poly (many) morph (shapes)

- In OOP, it describes the capability to use the “same code” to process objects of various types and classes, as long they have a common super class

Polymorphism

- Car is a Vehicle, Bicycle is a Vehicle
- Consider giving instructions to someone operating a Vehicle:
  - Start Vehicle
  - Release break
  - Accelerate
  - Apply break
  - Stop Vehicle
- These instructions will work for any kind of Vehicle, not only a Car
  - for a Bicycle, accelerate may just mean “pedal faster”

Substitution Principle

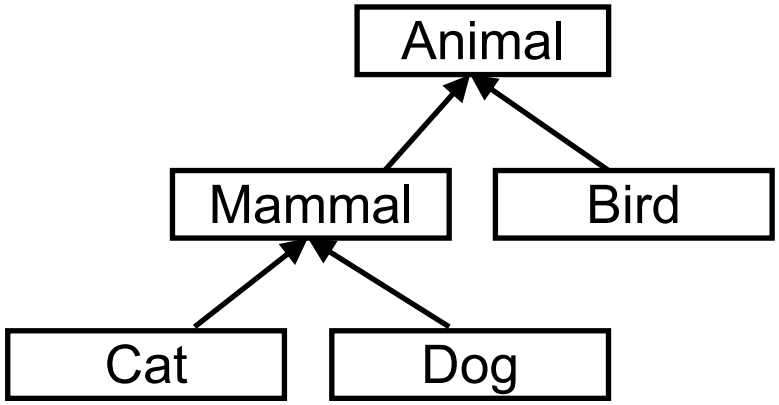
- In order to allow polymorphism, Java introduces the **Substitution Principle,** defined as:

An instance of a subclass can take the place of an instance of any of its superclasses

```
Vehicle v = new Car();
Vehicle v = new Bicycle();
```

- Variables holding object types are polymorphic variables - they can hold objects of different acceptable types
- acceptable types: the declared type, or any subtype of the declared type

Exercise: Which is correct?




```
1) Animal myBird = new Bird();
2) Mammal felix = new Cat();
3) myBird tweetie = new Bird();
4) Dog snoopy = new Mammal();
5) Bird littleBird = new Animal();
6) Mammal m = new felix;

7) Animal snoopy= new Dog();
```

NO

NO

NO

NO

Method polymorphism

- A Java version of the “algorithm” for operating a vehicle:

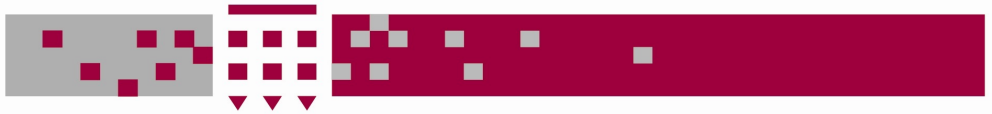
```
v.start();
v.releaseBreak();
v.accelerate();
v.applyBrake();
v.stop();
```

- It does not matter what exactly v is, as long as it is Vehicle or any of its subtypes

- **This algorithm is polymorphic**, it works for a variety of vehicle types, not only for a single type

Dynamic method binding

- How can `v.start()` work if the compiler at compile time does not know what type v refers to?
- We do not really know which version of start() is being called
- v will have to be tested during program execution **each time** it calls an instance method
- This process is known as **dynamic binding**
  - the exact method called will not be known until the program is actually run



Dynamic vs static binding

- static binding - what method to call is resolved at compile time (e.g. overloaded methods)

- dynamic binding - what method to call is resolved at run time (most overridden methods)

Dynamic binding

- If different objects are assigned to v during execution, different versions of start() may be called:

```
v = new Car();
v.start(); // Calls start method in Car class

v = new Bicycle();
v.start(); //Calls start method in Bicycle
           class
```

instanceof

- In case of polymorphic variables it is useful to be able to determine what the exact type is

```
if (myVehicle instanceof Car) ...
```

- object **instanceof** class
- this expression will return true if object is an instance of class, or if object is an instance of any subclass of class .

Casting object references

```
if (acct instanceof SavingsAccount) {
SavingsAccount savingsAcct = (SavingsAccount) acct;
savingsAcct.creditInterest(); }
```

- This is object casting: we cast object `acct` to type `SavingsAccount`
- but only after we have checked that it really is of `SavingsAccount` type