Java

Inheritance

Lecture objectives

- Another fundamental object-oriented technique is called inheritance, which enhances software design and promotes reuse

- To be able to understand
  - deriving new classes from existing ones
  - the `protected` modifier
  - Overriding

Inheritance

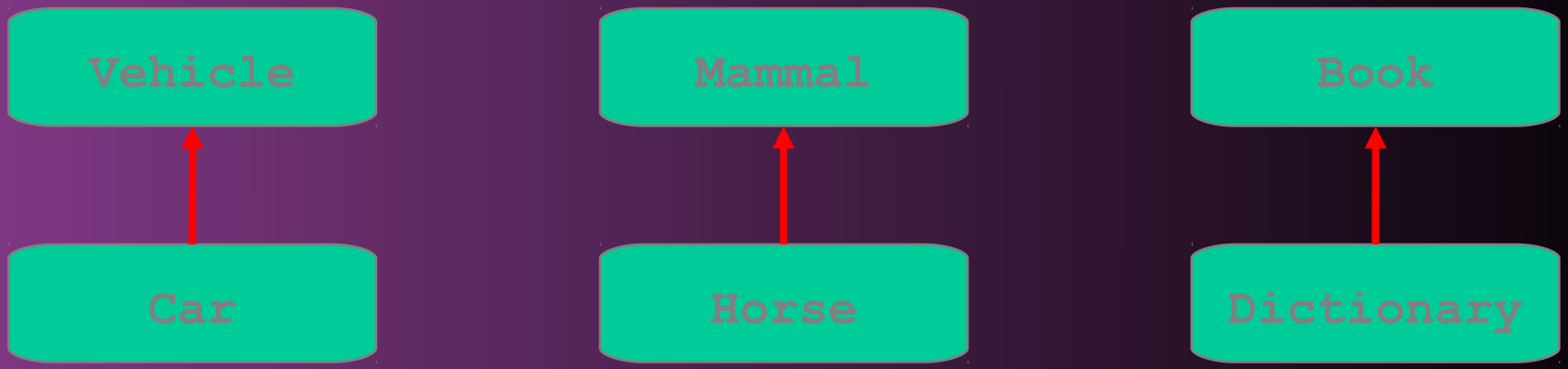
- Genetic inheritance - facts
  - Eg Your blood type, hair colour, height, body shape
  - Many facts about you (ie, your instance data) are inherited
- Genetic inheritance – behaviour
  - eg Sports and leisure interests, loudly or quietly spoken
  - Much of your behaviour (ie your methods) are inherited
- Many objects have similarities to other objects
  - Eg cars and trucks are similar and examples of vehicles
- A class (car) can have a ‘parent’ (vehicle) from which it inherits some of its data and behaviour

Inheritance

- *Inheritance* allows a software developer to derive a new class from an existing one
- The existing class is called the *parent class, base class* or *superclass*
- The derived class is called the *child class* or *subclass*
- As the name implies, the child inherits characteristics (the methods and data) of the parent

Inheritance

- Inheritance relationships are often shown graphically in a *class diagram*, with the arrow pointing to the parent class



Inheritance should create an *is-a relationship*, meaning the child *is a* more specific version of the parent

An Employee Class

```
public class Employee
{
    private int empNumber;
    private double empSalary;

    public Employee()
    {
        empNumber = 0;
        empSalary = 0;
    {

    public int getEmpNumber()
    {
        return empNumber;
    }
```

```
    public double getEmpSalary()
    {
        return empSalary;
    }

  public void setEmpNumber(int num)
    {
        empNumber = num;
    }

    public void setEmpSalary(double sal)
    {
        empSalary = sal;
    }
}
```

Creating Employees

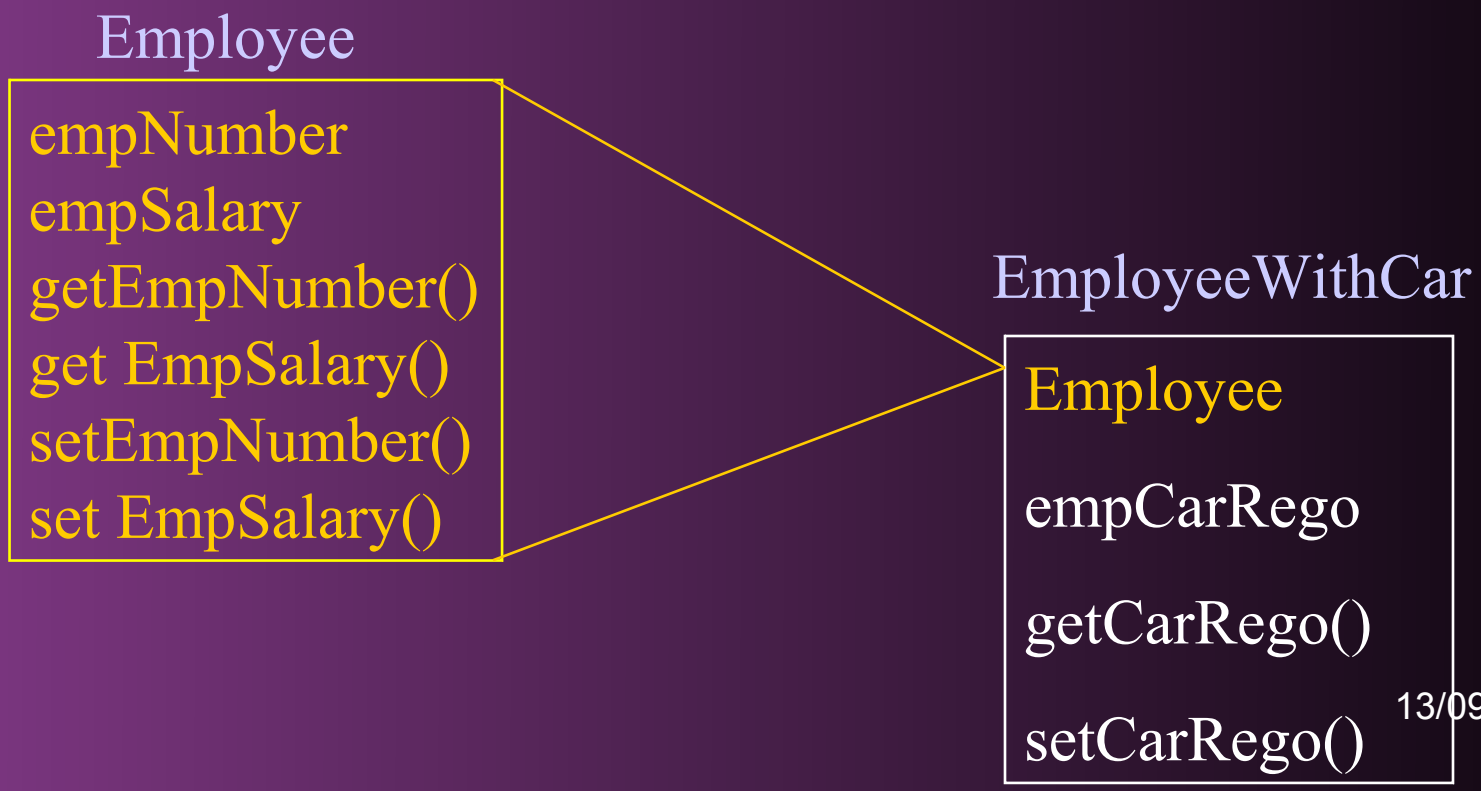
- Creating employee *objects* from the Employee class

```
Employee accountant = new Employee();
Employee programmer = new Employee();
```

- Other employees may require additional data to number and salary eg company car registration number
- We could change the class to have
  - 3 instance variables (empNumber, empSalary, empCarRego)
  - 6 methods (getters and setters for the 3 variables)
- It is better programming to keep it as it is and reuse it

A better way

- Create a new class *EmployeeWithCar* that inherits all the instance variables and methods of the *Employee* class
- Then just add the extra instance variable and methods



Deriving Subclasses

- In Java, we use the reserved word `extends` to establish an inheritance relationship

```
public class EmployeeWithCar extends Employee
{
        // class contents
}
```

EmployeeWithCar Class

```
public class EmployeeWithCar extends Employee
{
    private String empCarRego;

    // we will look at the constructor separately
    public String getCarRego()
    {
        return empCarRego;
    }

    public void setCarRego(String regoNum)
    {
        empCarRego = regoNum;
    }
}
```

Advantages of inheritance

- Saves time
  - reusing data and methods that already exist
  - makes programs easier to write
- Reduces errors
  - methods have already been used and tested

Another example

- A simple `Account` class may have this instance variable:

  private double balance;

- And these methods:

  public double deposit (double amount)

  public double withdraw (double amount, double fee)

  public double getBalance ()

- A `SavingsAccount` class may want to use these features and also have an interest rate
- A `SavingsAccount` *is an* `Account`, but with more features eg it has an interest rate

```
public class Account
{
    private double balance;

    public Account (double initial)
    {
        balance = initial;
    }

    public void deposit (double amount)
    {
      balance = balance + amount;
    }

    public void withdraw (double amount, double fee)
    {
        balance = balance - (amount + fee);
    }

     public double getBalance()
     {
        return balance;
     }

}
```

```
public class SavingsAccount extends Account
{
    private double interestRate;

    public SavingsAccount()
    {
          // constructor not defined yet
    }

    public double getInterestRate()
    {
         return interestRate;
    }

   public void setInterestRate(double  rate)
    {
         interestRate = rate;
    }
}
```

SavingsAccount class

- Instance variables of an object of the subclass:
  - `balance` (inherited)
  - `interestRate` (new)

- Methods that can be applied to `SavingsAccount` objects:
  - `getInterestRate` (new)
  - `setInterestRate` (new)
  - `deposit` (inherited)
  - `withdraw` (inherited)
  - `getBalance` (inherited)

A savingsAcct object

- If `account1` is an object

```
SavingsAccount account1 = new SavingsAccount()
```

- then these methods are legal:

```
System.out.print(“Rate: “ + account1.getInterestRate());
account1.setInterestRate(5.25);
account1.deposit(200.00);
account1.withdraw(100.00, 0.25);
System.out.print(“Balance: “ + account1.getBalance());
```

Writing Subclass Constructors

- A subclass doesn't inherit constructors from its superclass
- The constructor for the `SavingsAccount` class will need to initialize both the `balance` and `interestRate` variables
  - But the superclass instance variables are likely to be private so they cannot be referenced directly from the subclass

- The hard part of writing a constructor for a subclass is initializing the variables that belong to the superclass

Subclass constructors

- Writing constructors for a subclass is the same, except for initialisation of superclass variables (if private)

```
public SavingsAccount (double initialBalance,  double initialRate)
{
        balance = initialBalance;      //ERROR
        interestRate = initialRate;
}
```

- As `balance` was declared as `private` in the `Account` class this will not compile – the `SavingsAccount` class has no access to it

The super Reference

- A child's constructor is responsible for calling the parent's constructor
- The first line of a child's constructor should use the `super` reference to call the parent's constructor

The `super` reference

- To access a superclass private instance variable we need to use a superclass method…or in this case constructor
- We want to use the parent's constructor to set up the "parent's part" of the object
- The `super` reference can be used to refer to the parent class, and is often used to invoke the parent's constructor
- The **first line** of a child's constructor uses the `super` reference to call the parent's constructor

Invoking the superclass constructor

- The `SavingsAccount` constructor invokes the `Account` constructor using the word `super`

```
public SavingsAccount (double initialBalance,  double initialRate)
{
        super (initialBalance);    //invoke Account constructor
        interestRate = initialRate;
}
```

- The account constructor will initialise the `balance` variable to the value stored in `initialBalance`

Superclass constructor parameters

- The subclass constructor must provide all the parameters required by the superclass constructor in order eg.

```
public Account (String owner, int account, double initial)          // superclass constructor
{
    name = owner;
    acctNumber = account;
    balance = initial;
}

public SavingsAccount(String owner, int account, double initial, double rate)  // subclass
                                                                                //constructor
{
     super (owner, account, initial);
     interestRate = rate;
}
```

```
public class EmployeeWithCar extends Employee
{
    private String empCarRego;

    public EmployeeWithCar( String empCarRego)
    {
        super();
        this.empCarRego = empCarRego;
    }
    public String getCarRego()
    {
        return empCarRego;
    }

    public void setCarRego(String regoNum)
    {
        empCarRego = regoNum;
    }
}
```

Writing Subclass Constructors

- If a subclass constructor fails to include `super`, the compiler will automatically insert `super();` at the beginning of the constructor.
- If a subclass has no constructors at all, the compiler will create a no-arg constructor that contains `super();` but no other statements.

Controlling Inheritance

- Visibility modifiers determine which class members get inherited and which do not
- Variables and methods declared with `public` visibility are inherited
- Those with `private` visibility are not actually inherited in that we need to use the superclass constructor and methods to access them

- But `public` variables violate our goal of encapsulation!

The `protected` Modifier

- There is a third visibility modifier that helps in inheritance situations: `protected`
- `protected` behaves the same as `private` within a class but allows a member of a base class to be inherited into the child class

private	can only be accessed in the same class
protected	can only be accessed in the same class or in a subclass
public	can be accessed in any class

Using `protected`

```
public class Account
{
    protected double balance;
   . . .
}

public class SavingsAccount extends Account
{
     public SavingsAccount (double initialBalance,  double initialRate)
     {
         balance = initialBalance;    // legal
         interestRate = initialRate;
     }
}
```

`protected` versus `private`

- Declaring instance variables protected exposes them to all subclasses
- This is a potentially unlimited number and may weaken an object-oriented goal of encapsulation
- If a subclass needs access to these variables it should be able to call a getter or setter method of the superclass

- We will not use *protected* variables
- Always declare instance variables private

Methods available to a subclass

- Inherit methods from the superclass
  - superclass methods can be applied to subclass objects

```
account1.deposit(200.00);
```

- Define new methods

```
public void setInterestRate(double  rate);
```

- Override methods from the superclass
  - ie specify a method with the same signature (that is the same name, return type and parameter types) in the subclass
  - this method defined in the subclass will take precedence
  - Use `super` to call an overridden method

Using the right method

- Java looks first in the class of the calling object, then in the class's superclass, then its superclass etc

- Consider the following statement:

```
account1.deposit(500.00);
```

- Java first looks for the `deposit` method in the `SavingsAccount` class, then in the `Account` class.

```
public class AParentClass
{
    public void printClassName()
    {
        System.out.println("AParentClass");
    }
}


public class AChild extends AParentClass
{
    public void printClassName()
    {
        System.out.println("I am a child class");
        System.out.println("My parent is ");
        super.printClassName();
    }
}
```

```
public class DemonstrateMethodOverride
{
    public static void main(String[] args)
    {
        AChild child = new AChild();
        child.printClassName();
    }
}
```

```
I am a child class
My parent is
AParentClass
```

Overloading vs. Overriding

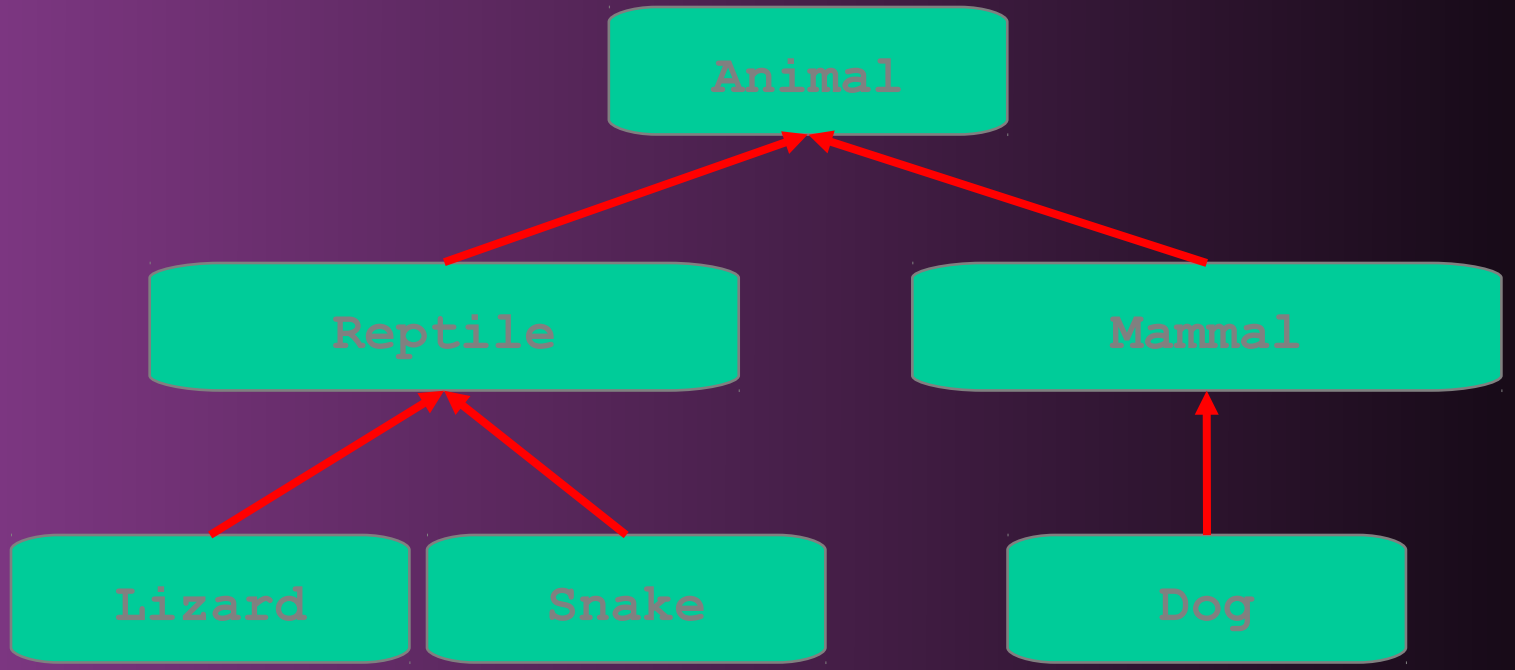
- Don't confuse the concepts of overloading and overriding
- Overloading deals with multiple methods in the same class with the same name but different signatures
- Overriding deals with two methods, one in a parent class and one in a child class, that have the same signature
- Overloading lets you define a similar operation in different ways for different data
- Overriding lets you define a similar operation in different ways for different object types

Practice

- Create a Circle class with
  - Instance variables *radius and area (doubles)*
  - A constructor that
    - receives a parameter value for *radius*
    - sets *area* to 0
  - A method called *calcArea* that calculates the circle area
  - A method called *getArea*
- Create a *Cylinder* class that extends *Circle* with
  - Instance variables for *length* and *volume*
  - A constructor that
    - receives parameters for *radius* and *length*
    - *Sets volume to 0*
  - A method *calcVolume* that calculates the cylinder volume
  - A method called *getVolume*
- Create a driver class to test the instantiable classes

Class Hierarchies

- A child class of one parent can be the parent of another child, forming *class hierarchies*



Lecture Outcomes

Today we have covered:

- deriving new classes from existing ones
- the `protected` modifier
- Overriding

- Questions?