Topic 2

Inheritance and Polymorphism

Learning Outcomes

- **Explain** the difference between method overloading and method overriding and **apply** in Java code
- **Describe** the purpose of the `java.lang.Object` class and **apply** in Java code
- **Describe** the purpose of Inheritance and **apply** in Java code
- **Describe** the behaviour of attributes, methods and visibility in a Java inheritance hierarchy and **apply** in Java code
- **Describe** inheritance related Java keywords such as `extends`, `this` and `super` and **apply** in Java code
- **Describe** the purpose of Polymorphism and **apply** in Java code
- **Describe** the intrinsic relationship between Inheritance and Polymorphism and **apply** in Java code
- **Describe** the mechanics of constructors in an inheritance hierarchy and **apply** in Java code

Method Overloading

- Before looking at inheritance and the related idea of *overriding* a method it is useful to first consider (or revisit?) the simpler concept of method **overloading**
- Java allows two or more methods to have the same *name* provided the methods vary in the type or number of arguments (NOTE: a different return type is not sufficient)
- The actual method invoked is determined based on the passed argument type
- This process is known as method **overloading**
- It is useful when we perform similar operations on different types and also applies to Constructors where it is particularly useful
- Consider a Library Member class. We may want to `update()` or construct based on different types, especially when some details are optional
- A good API example is the overloaded `println(…)` method in class `java.io.PrintStream`

Inheritance 101: The 'supreme' superclass Object

Every class (even one that does not use extends) is a subclass of `java.lang.Object`

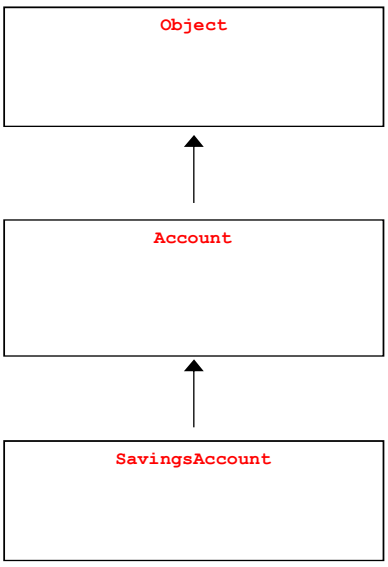
Therefore every time you instantiate a Java object inheritance is involved

Why?

The `Object` class has a small number of methods that are useful for all classes

- `equals()`
- `toString()`
- etc.

Check the rest in the API docs!



Inheritance on its own? (without Polymorphism?)

- In general Inheritance and Polymorphism are strongly linked as we will see as we cover the topic further
- However when learning it is useful to treat them separately and start with Inheritance independently
- Furthermore it is possible to actually use inheritance on its own without any polymorphism so it is also reasonable from a practical perspective to start with this simple case
- When using inheritance in this way the idea is to remove code duplication and promote code reusability by placing code that will be reused (and would otherwise have been duplicated across multiple classes) into a 'superclass'
- This superclass is then extended using the Java `extends` keyword
- i.e. `class X extends Y { … }`
- Using general OO terminology the Java *extends* process is called *inheritance* since the *child/decendent/sub-* class inherits form the *parent/ancestor/super-* class (yep, lot's of alternative terminology again!)

Object Oriented Relationships

Two commonly used clauses in class relationship

- **has-a** ← Association/Composition/Aggregation relationship discussed in Topic 1
- **is-a** ← Inheritance relationship

A home **is-a** house that **has-a** family and a pet

If House, Family and Pet are existing classes then in java we write

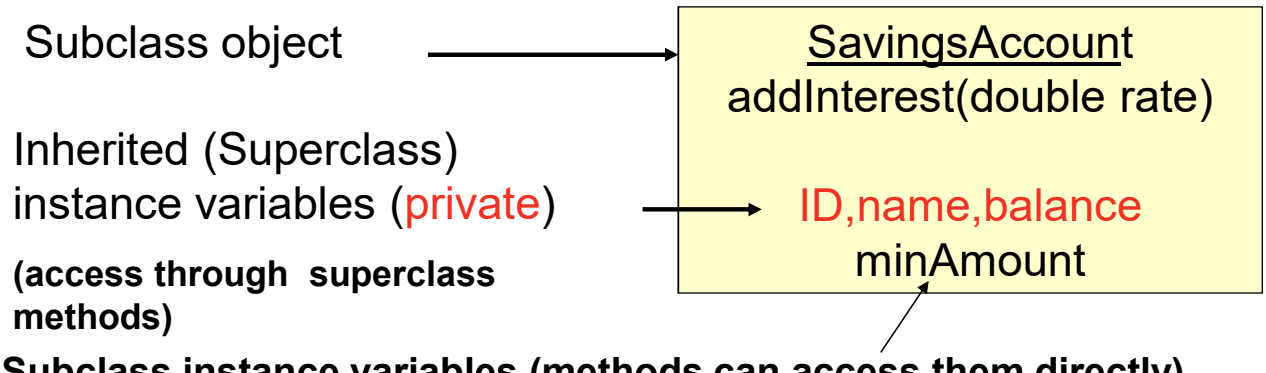
```
public class Home extends House
{
      private Family inhabitants;
      private Pet thePet;
}
```

Inheritance Overview

- So we have identified that with inheritance a `Home` is a `House` but what does this actually mean in practice?
- When we declare `class Home extends House` on the previous slide the `extends` (inheritance) keyword means that the `Home` class effectively contains all code from the `House` class that is not private
  - i.e. it *inherits* all of the non-private attributes and methods of `Home` even static methods
  - These attributes can be modified and methods called as if they were part of the `Home` class without having to rewrite/copy them
  - This facilitates code *reuse!*
- There are some extra rules here such as `final` methods but these will be covered later in the lecture/topic

Instance variable of superclasses

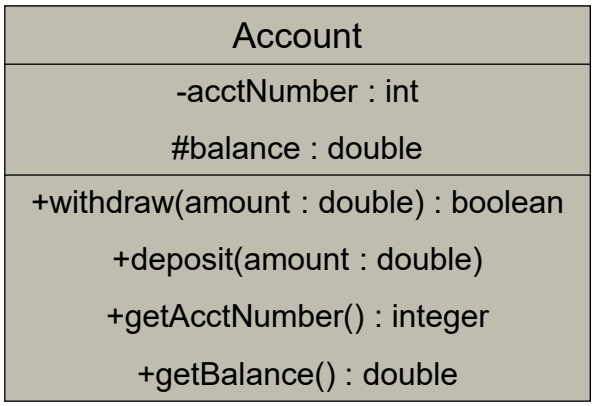
- As stated on the previous slide any **non-private** code artefact (attributes/methods) are automatically inherited by subclasses.
- But since attributes are usually **private** they cannot be accessed directly by subclasses (see next slide for detailed visibility options)
- Hence the only way to change them is through superclass mutators/methods



Modifier on members in a class	Accessed from the same class	Accessed from the same package	Accessed from a subclass	Accessed from a different package
public	✓	✓	✓	✓
protected	✓	✓	✓	–
default	✓	✓	–	–
private	✓	–	–	–

UML visibility: '-' private, '+' public, '#' protected, '~' package

Inheritance: Credit Account Example



Consider a simple Account class

The class has:

A *Name* (should be unique and descriptive)

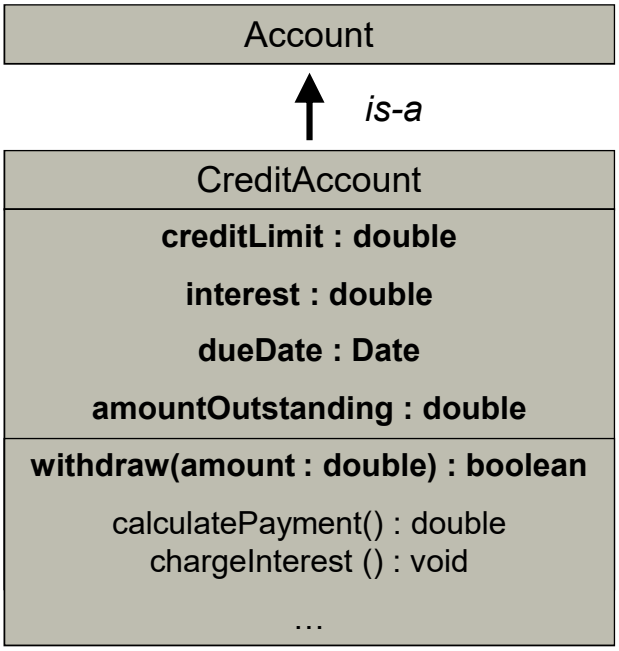
Attributes (also called fields/instance variables which are variables representing the **state** of the class)

Methods (which are functions/procedures representing the **behaviour** of the class)

Relationships to other classes represented as attributes (which may be arrays/collections for 'many' relationships)

NOTE: relationships are not explicit i.e. you need to understand the code to identify a relationship which is why UML is useful!

Inheritance: Credit Account Example (contd.)



The bank introduces a new product, the credit account. This has a credit limit, an interest rate, a payment due date, and a payment amount outstanding.

A customer may withdraw up to the credit limit minus the account balance.

Every month, a credit charge based on the outstanding amount is be charged to the account.

Inheritance promotes **reusability** (code reuse) and **extensibility** (extending or modifying existing code).

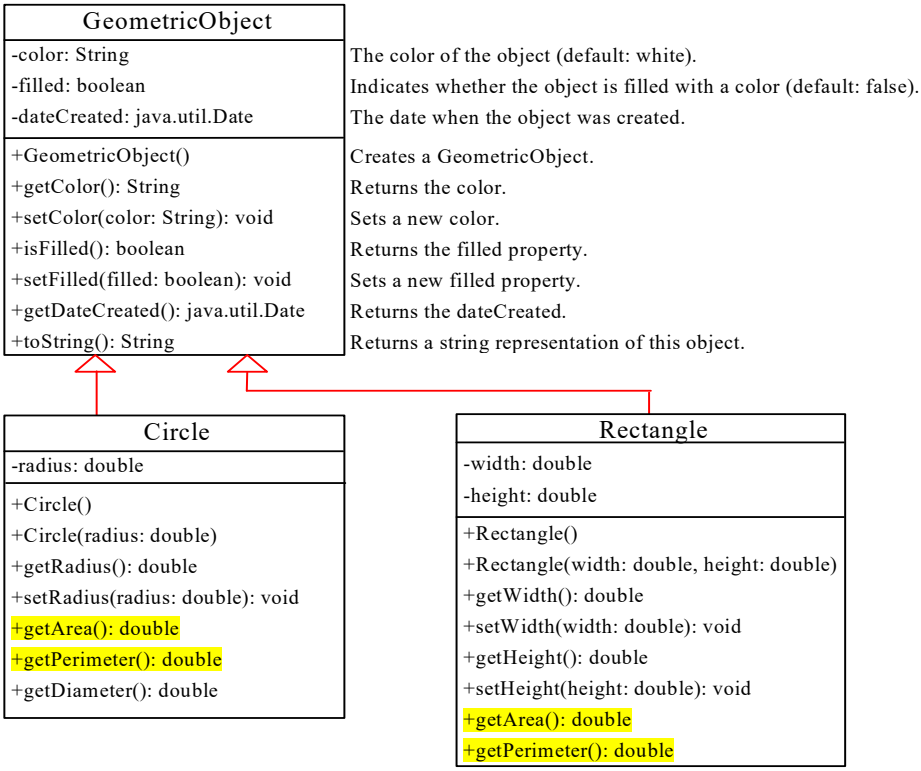
As stated previously every object in Java, either explicitly or implicitly, is a subclass of `java.lang.Object`.

Bank Account Example Summary

In summary, the `CreditAccount` class:

- inherits non-private properties (attributes/methods) from the `Account` class
- calls the superclass constructor to help create itself
- overrides some methods with a new implementation
- defines some new attributes and methods unique to `CreditAccount`

Example From Liang Textbook



See example code in package `liang.chapter9`

The super and this Keywords

A subclass may explicitly access a method or attribute in its superclass with the `super` keyword. When `super` is used to access the constructor of the superclass, it must be the first statement in the subclass's constructor (constructors are covered further later in the lecture).

The `this` keyword refers to the object through which the method or attribute is accessed (the current object.) In the absence of an object reference (or class name, for static items), this is the implied reference.

```
public CreditAccount(int num, double creditLimit,
                     double interest)
{
  super(num, 0); // set the balance to 0 for new account
  this.creditLimit = creditLimit; // this resolves the
  this.interest = interest;       // ambiguity
   balance=0; // this is not necessary since no ambiguity
}
```

Using the this reference

this refers to methods and instance variables of current object.

```
class SAccount extends Account
{
    public SAccount(String accountID, String
        accountName, double amount, double minAmount){
        super(accountID,accountName,amount);
        this.minAmount = minAmount;
    }
    . . .
    private double minAmount;
}
```

minAmount of current object (explicit)

Same name ?

Protected/Package Access

- Protected and package (default) are alternatives to private and public
- If an attribute/method is declared *protected* it can be accessed by methods of that *class,* its *subclasses (in ANY package)* and all *other classes* within the same *package*
- *Default/package* is similar but subclasses must be in the same package
- It can be argued that protected/package visibility strikes a balance between absolute protection and no protection at all
- However, it breaks the *encapsulation* rule as the designer of the superclass has no control over the authors of the subclass
  - Therefore the subclass can potentially break intended superclass behaviour
- Therefore safer to use protected getters and setters instead to maintain encapsulation
  - By ensuring that values stay within expected ranges, are modified in certain ways etc.
- ***TIP: Where possible keep it simple and stick to public and private!***

Inherited Methods and Visibility

A subclass cannot override a method to become less visible. For example, the compiler will not allow this.

```
class Account {
        public void withdraw(double amount) { . . . }
                . . .
}
class SavingsAccount extends Account {
        protected void withdraw(double amount) { . . . }
        . . .
}
```


Final methods and classes

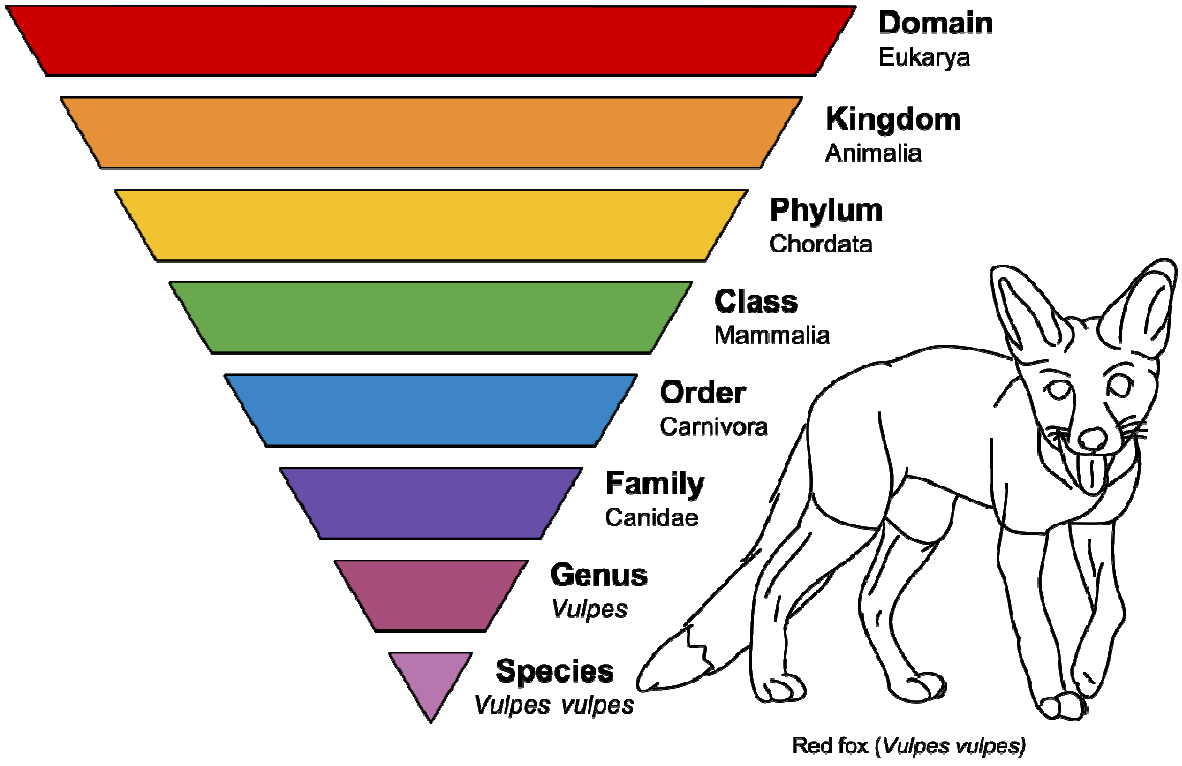
- For methods, final prevents the method from being overridden by any inheriting class.
- Ensures that the behaviour of a method is retained during inheritance and cannot be overridden.
- For classes final means it cannot be extended.
- When a class is declared final that class cannot be subclassed, and of course its methods cannot be overridden because the class itself cannot be extended.
- The class `java.lang.String` is defined:

  ```
  public final class String
  ```

  Therefore it cannot be extended.

Polymorphism

- If we ask someone at home to feed the pets they know it means different things for different type of pets.
- Hence We can consider the action **feed** to be polymorphic.
- In OOP, polymorphism promotes code reuse and extensibility by calling the method in a generic way.
- For example we can deduct charges by calling an `applyFees()` method on all `Account` objects. But depending on the type of account the specific/different versions of operations such as `applyFees()` or `calculateInterest()` will be called for each account.



This makes for a deep inheritance hierarchy!
From: https://en.wikipedia.org/wiki/Animal

Polymorphism and Dynamic Binding (1)

When the bank manager asks his/her staff to deduct $5.00 from every bank account, the staff know that different deduction methods will apply to different types of account. Hence the deduction action is polymorphic.

When the method `withdraw()` is called on an `Account` object, depending on the actual type of that `Account` at the time, the correct version of `withdraw()` will be dynamically bound to that object and executed.

First, let's add several accounts of different types for a customer:

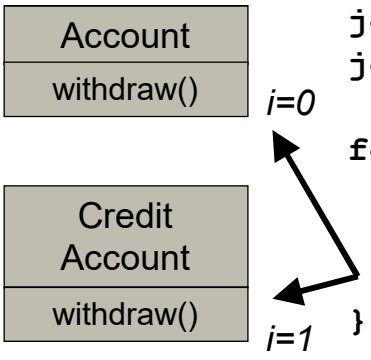
```
Customer john = new Customer("John", 123);
john.addAccount(new Account(1, 500));
john.addAccount(new CreditAccount(2, 0, 200));
```

Recall that the method `addAccount()` of `Customer` accepts an `Account` type as a parameter. Why can it also take in a `CreditAccount` type?

This is because `CreditAccount` "is-a" `Account`, due to inheritance.

Polymorphism and Dynamic Binding (2)

Now, to withdraw $5.00 from all of John's accounts. Which version of `withdraw()` is executed when?




```
Customer john = new Customer("John", 123);
john.addAccount(new Account(1, 500));
john.addAccount(new CreditAccount(2, 0, 200));

for (int i = 0; i < 2; i++) {

    Account a = john.getAccount(i);
    a.withdraw(5);
}
```

We can put different types of `Account` in a common `Account` array, and withdrawal can be performed polymorphically on them.

`Account` is the base class of all account subclasses. Polymorphism promotes code reuse by allowing methods to be called in a generic way.

EXAMPLES: See `sadi.topic2.bank` and `liang.chapter9` Java examples

Type Conversion

So, a `CreditAccount` can be treated as an `Account`. However, the reverse is not true; an `Account` may not be a `CreditAccount`.

```
Account ac;
CreditAccount creditAc;
ac = creditAc;
creditAc = ac;
```

Can we convert an `Account` to a `CreditAccount`? Yes*, by **type casting**; this can be done only if the `Account` is actually a `CreditAccount`, otherwise a `ClassCastException` will be thrown.

```
creditAc = (CreditAccount) ac;
```

- A subclass reference is automatically a superclass reference, but the reverse is not true.
- It is possible to convert a superclass reference to a subclass reference if the former is actually referencing a subclass object, otherwise Java will throw a `ClassCastException`.

*** Avoid casting even with instance of checks and maintain correct references and use polymorphism to handle type differentiation (more detail next week!)**

Inheritance and constructors

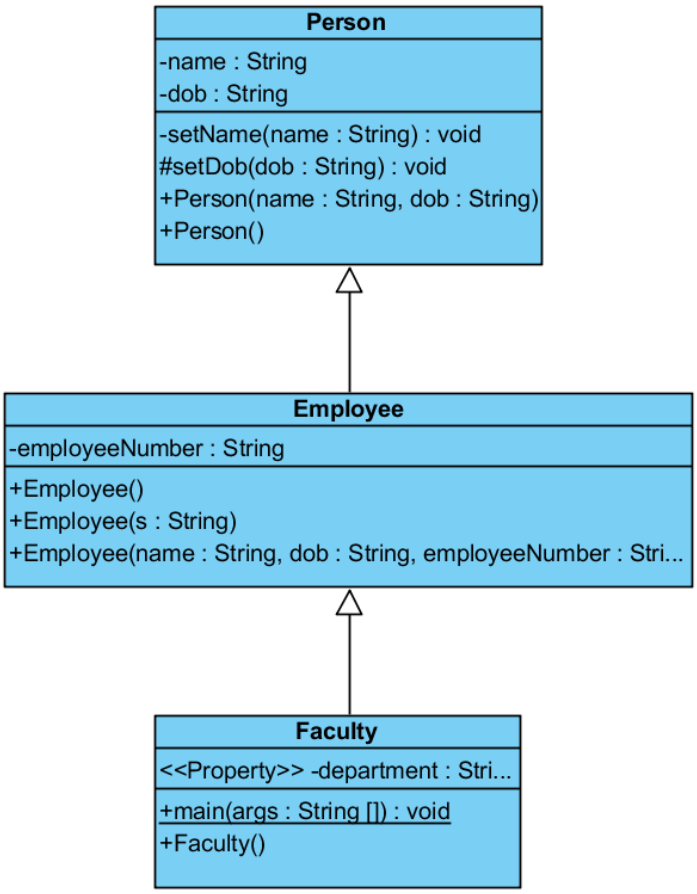
- The behaviour of constructors in Java can be tricky if you do not understand how they work!
- Classes do NOT inherit constructors
- However classes by default WILL call the superclass no argument constructor as their first line unless you explicitly call a different constructor with `super(...)` or `this(…)`
- `super()` must eventually be called (or will get a recursive call error)
- Therefore by default constructors actually execute in reverse order from normal polymorphism in a class hierarchy
  - i.e. they are called recursively from the bottom until the top constructor executes (`java.lang.Object`) and then execute and return in a top down fashion
  - This is different from method polymorphism where the VM starts at the most specialised subclass (at the bottom of the hierarchy) and works its way up until it finds a signature match

Inheritance and constructors

- The best (most cohesive) approach is to use partial construction where each class in the hierarchy constructs only the parts specific to that class

- See `liang.chapter9.Faculty.java` example and `package sadi.topic2.constuctors` example (I will debug through an example to demo the behaviour in this lecture)

Inheritance and constructors (example)



Summary

- Inheritance allows you to reuse existing code using Java `extends` keyword without needing to copy and paste it
- Common things you will implement in the subclass (the class that inherits from the superclass)
  - Add one more new attributes
  - Add one more new methods
  - Add one or more constructors (since not inherited)
- COVERED in more detail in future lectures
  - Override a superclass method with a completely new implementation
  - Override a superclass method but still call original method via `super.methodName()` but add some new code as well