Programming Techniques

Polymorphism; Abstract Classes & Interfaces

RMIT UNIVERSITY

Pre-Lecture Videos

- Polymorphism

  https://goo.gl/vfbN3x (03:22)
  https://goo.gl/D1KJFR (02:14)

- Static

  https://goo.gl/C4pAqN (05:22)

- Abstract Classes

  https://goo.gl/VmNOhj (03:33)

Interfaces

https://goo.gl/ZJynK9 (04:30)

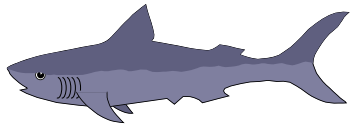
Contents

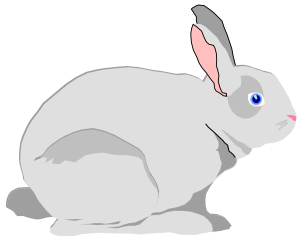
- Polymorphism
- Abstract Classes
- Interfaces
- Interfaces with default & static methods (Java 8)
- Declaring objects with an interface reference variable
- Casting objects to and from interface reference types to class references
- When to use an Interface vs an Abstract Class

RMIT UNIVERSITY

Polymorphism

- When someone tells you 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 by calling the method in a generic way.
- For example we can say deduct $5.00 monthly charges by calling the withdraw() method on all account objects. But depending on the type of account the correct version of withdraw() will be called.





PolyMorphism

Biology - The occurrence of different forms among the members of a population or colony, or in the life cycle of an individual organism:

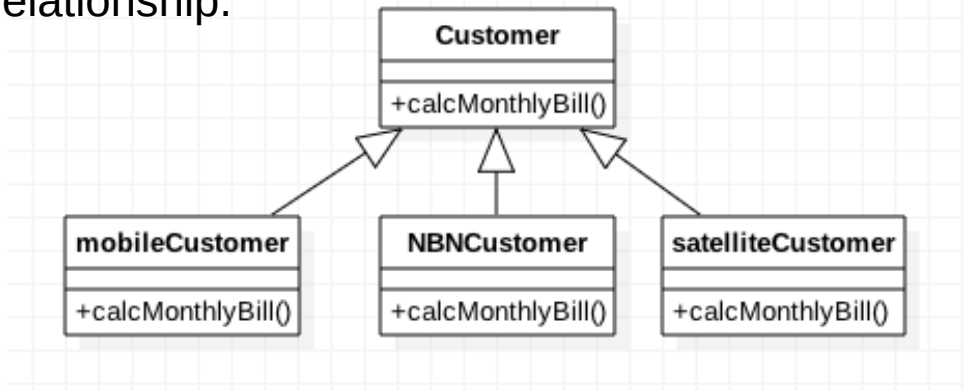
"..*the workers of this species exhibit polymorphism, specialised physical cases*"

http://www.oxforddictionaries.com/definition/english/polymorphism

In **object-oriented programming**, polymorphism refers to a programming language's ability to process objects differently depending on their data type or class.

PolyMorphism - applied

Imagine a Telco company that has to generate hundreds of thousands of monthly invoices for their customers. There are several different types of customers but all have to have their broadband and data comms bills calculated. The classes modelling the customers are in an **inheritance** relationship.



The way the bill is calculated is different for each type of customer. Accordingly, each class has its own calcMonthlyBill() method.

If you like, real time batch processing of the objects in an arbitrary number of sub-classes can occur.

The program doesn't need to know the subclass for the next object, it will dynamically identify its type and it will call the correct calcMonthlyBill() method

PolyMorphism - applied

Normally we would explicitly call a method for an object.

```
m24501.calcMonthlyBill( )     // calc bill for mobile customer 24501 object
nbn0054.calcMonthlyBill( )    // calc bill for nbn0054 customer object
sat9987.calcMonthlyBill( )    // calc bill for satellite customer 9987 object
```

But what if you have thousands of objects to process….?

We could process a batch of certain object types. We could put say, all the **mobile** customer objects into an array and loop through the array with something like:

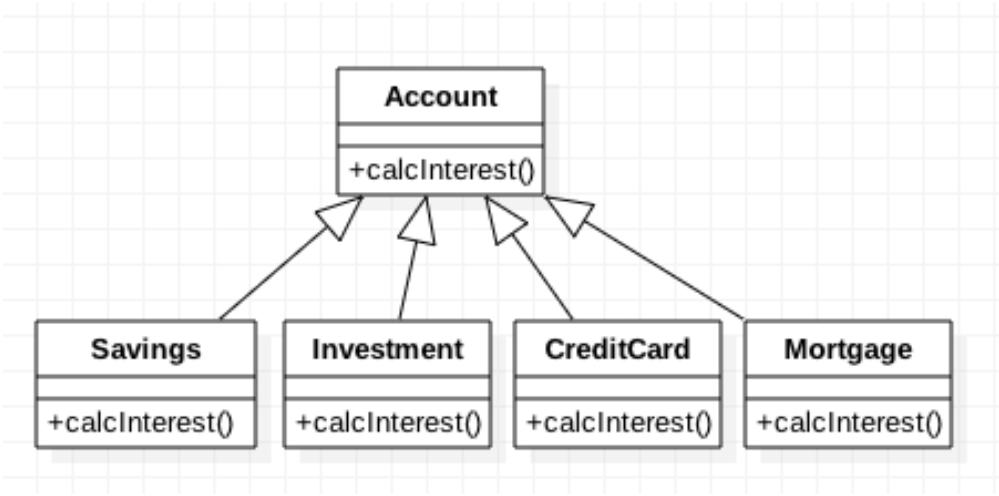
```
for (mobileCustomer cst : mobileArray)
      cst.calcMonthlyBill()  //call method for current mobile object
```

But how do we process **ANY** child object that needs to have their monthly bill calculated?

Dynamic polymorphism gives us the ability to do this.

Another applied example of PolyMorphism

Imagine you needed to create a system that could calculate the interest on different types of bank accounts. It would be ideal to be able to process the different types of account objects together in a form of batch processing.



For this example of polymorphism (with **dynamic binding**), we create an array of the parent class which can store **any** type of child object.

Account [ ] acctArray = new **Account**[4]; **//declare array of the parent class**

Setting up for PolyMorphism

Account [ ] acctArray = new **Account**[4]; *//*declare array of the parent class

With the array created, child objects can be instantiated but note how this is done differently.

The reference is to the parent class **NOT** the child

```
Account   s1 = new Savings( );          //create child objects with a super class reference
Account   inv1 = new Investment( );
Account    cc1 = new creditCard( );
Account    m1 = new Mortgage( );
```

*//*assign the different objects to the array

```
acctArray[0 ] = s1;
acctArray[1 ] = inv1;
acctArray[2 ] = cc1;
acctArray[3 ] = m1;
```

PolyMorphism & dynamic binding

Polymorphism is demonstrated when we process objects in the array to call the **calcInterest( )** method

```
Account[] acctArray = new Account[NUM_ACCTS];

for (Account acct : acctArray)
        acct.calcInterest();
```

One by one each object in the array calls the method and automatically the matching child class method is called. This is dynamic PolyMorphism in action

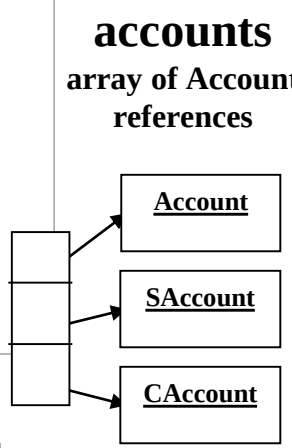
Dynamic Binding or Late Binding

As there is a parent reference variable for each child object, at compile time (**static binding**) the compiler doesn't know which method to call, the parent or sub class.

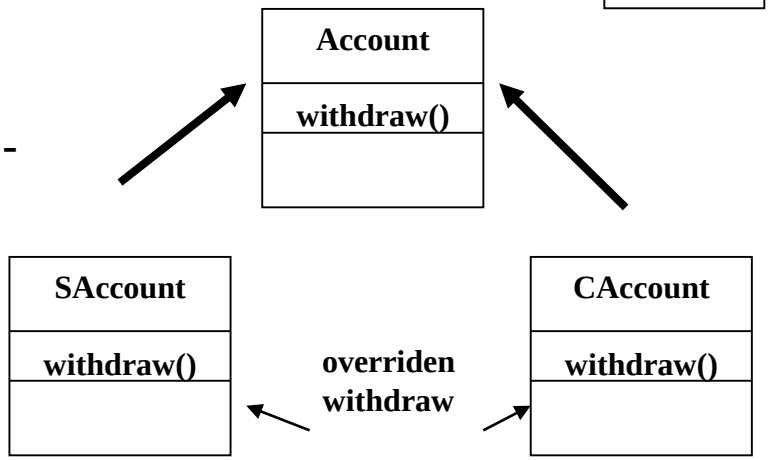
Dynamic binding means that the compiler decides at run-time which method to call. The parent class version of the method is overridden, as a result.

Use of Polymorphism

```
Account[] accounts = new Account[3];

accounts[0] = new Account("a12345","Charles",1000);
accounts[1] = new SAccount("s12346","Craig",1200,1000);
accounts[2] = new CAccount("c12347","George",200,1000);

// Deduct fixed amount $500 from all accounts
for (int i=0; i<3; i++)
        accounts[i].withdraw(500);

(or use for…each)
```



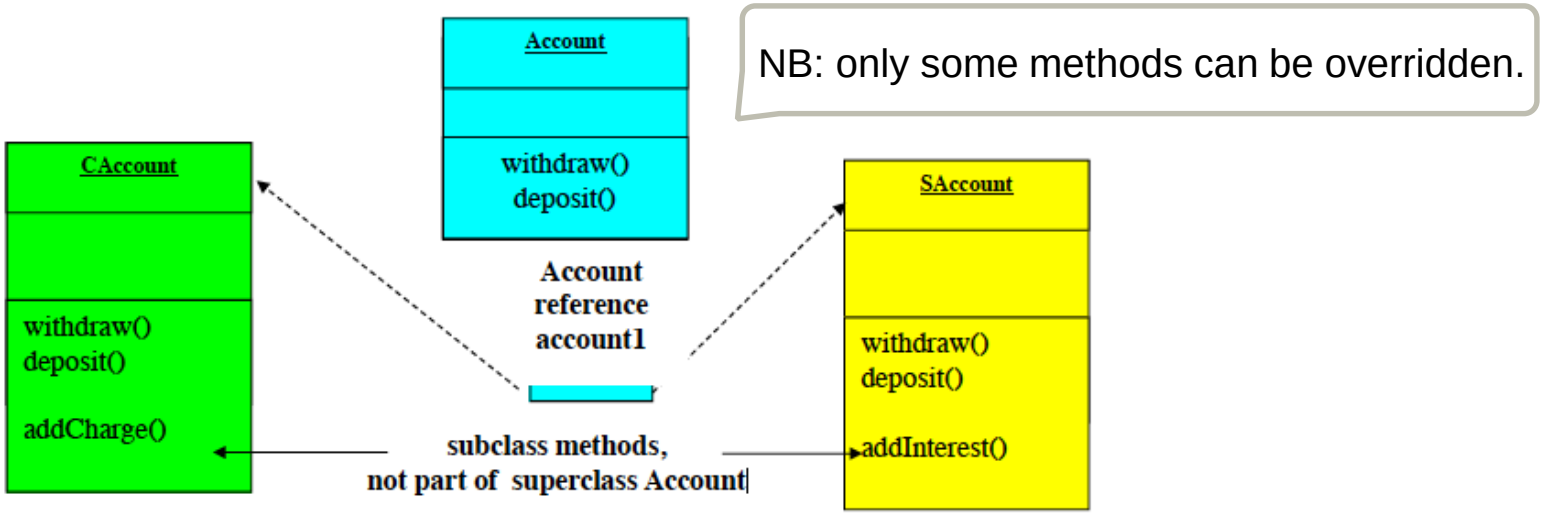
- Which withdraw() methods are called?
- Though Accounts[i] is a Account reference in Java, actual method called is determined at run-time based on the type of object being referred.
- This feature is Called Polymorphism.



Not all methods can be called

However, the compiler will report an error if we attempt to call the addInterest() of SAccount or addCharge() of CAccount through an Account reference.

```
Account account1 = new SAccount("s123","Tom",100, 0);

account1.addInterest(1.0);      //   Error !!!!!!
```

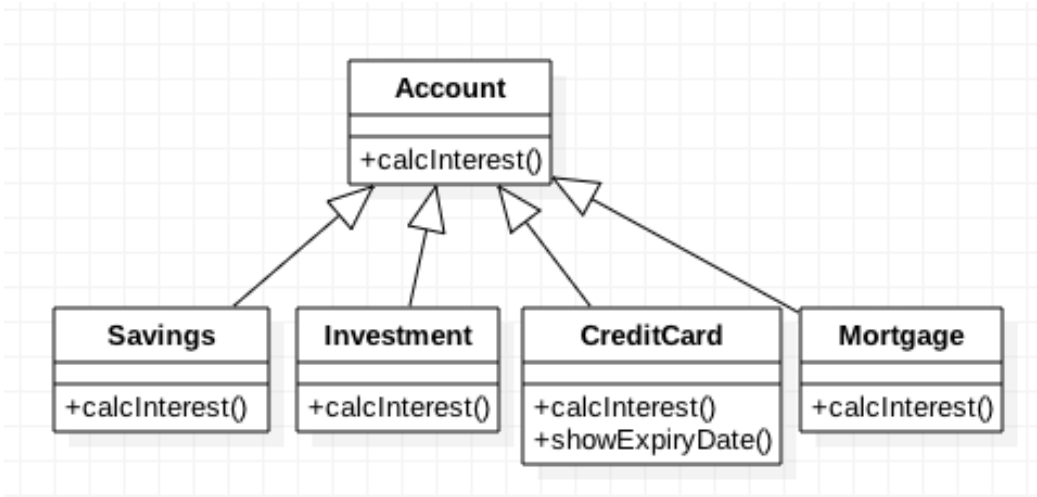


Problem !! Warning

When we use a parent class reference variable to refer to a subclass object it is great for enabling polymorphism but there is a problem.

Account cc5 = new CreditCard() //instantiate a credit object with super class reference

The type of the reference variable will determine the methods that it can invoke on the object. With the Account (parent) reference variable we **cannot** call a method from the CreditCard class without casting (converting) the object.



How to we call any of the other methods such as showExpiryDate( ) which are not part of the dynamic binding methods?

Casting

The statement below will not compile:

cc5.**showExpiryDate();** //call a method from the credit card class

The Java compiler throws the following message:

Exception in thread "main" java.lang.Error: Unresolved compilation problem:
The method showExpiryDate() is undefined for the type Parent

The compiler does not recognise the **showExpiryDate**( ) method as belonging to the **Account** (parent) class. Because this method belongs only to the child class, it cannot be 'seen' by this object which has a super class reference.

The solution is to cast (convert) the object from a super class reference to the relevant derived class reference.

Casting - how to

Account cc5 = new CreditCard() //the original instantiation

There are several approaches:

Create a new reference of the child class so:

Credit ccTemp = (CreditCard) cc5; //explicitly cast the existing object

new reference

cast

ccTemp.showExpiryDate(); //now call the method

More direct technique

((CreditCard)cc5).showExpiryDate; //cast & call the method in one statement

cast

method call

Exercise

- Create a console-based application with a menu to execute the following operations:

- Create Saving Account
- Create Credit Card Account
- Create Mortgage Account
- Create Investment Account

- After each account is created, it gets added to an array of Accounts

- Also, add two more menu options:

- Display Interest (calculates and display interests for ALL existing accounts)
- Show Expiry Date (displays expiry dates for all credit card accounts, you may need a hint)

Summary

- A superclass reference can refer to a *subclass* object
- However, only methods of that superclass interface can be called through the reference. The actual method called (whether it is of the specials or one of its subclasses) depends on the type of the object being referred.
- To call other methods not found in the superclass we need to cast *reference* to the appropriate type
- A superclass reference can be cast to a subclass reference if we are sure it is referring to a subclass object
- The program will through throw an exception and terminate at run-time if it is cast wrongly

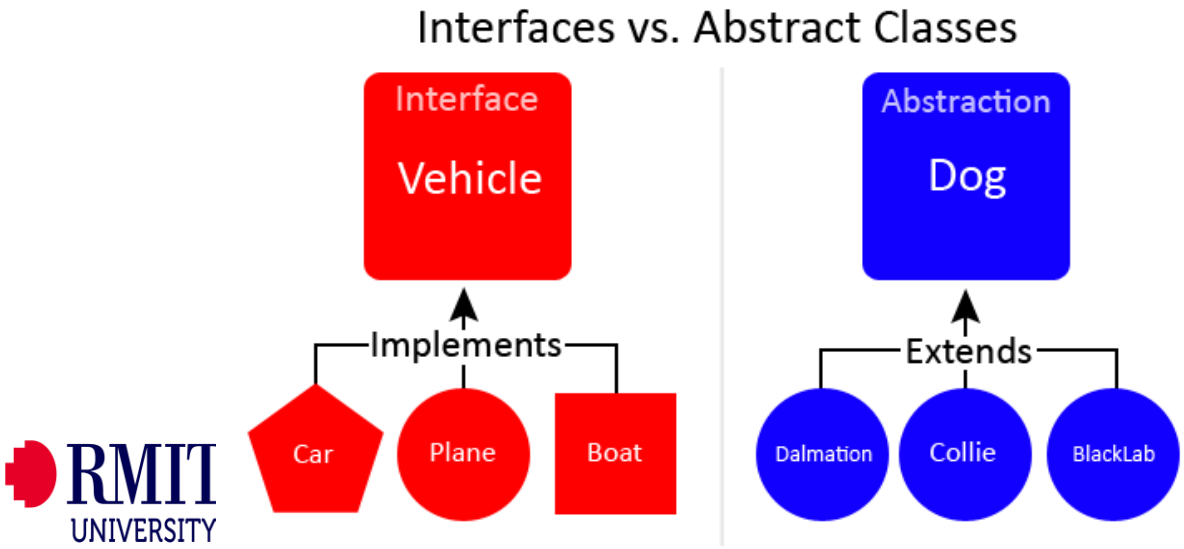
Abstract Classes & Interfaces

Abstract classes and Interfaces are key building blocks of the design aspects of OOP

Any programmer has to decide whether it is beneficial for their Java classes to use and implement an abstract class or an interface or perhaps a combination of these.

The diagram below shows abstract classes are used with inheritance. The super class **Dog** can be made abstract.

The diagram also implies an interface is used with inheritance. However, classes implementing an Interface do not have to be in an inherited relationship. This is key difference.



RMIT UNIVERSITY

Abstract Classes & Interfaces

Both provide a level of abstraction that can improve a program's design and influence how classes are implemented.

Abstraction is a process of hiding the implementation details (how things work) from the user and other programmers.

In Java, abstraction is achieved using **Abstract classes, and Interfaces**.

RMIT UNIVERSITY

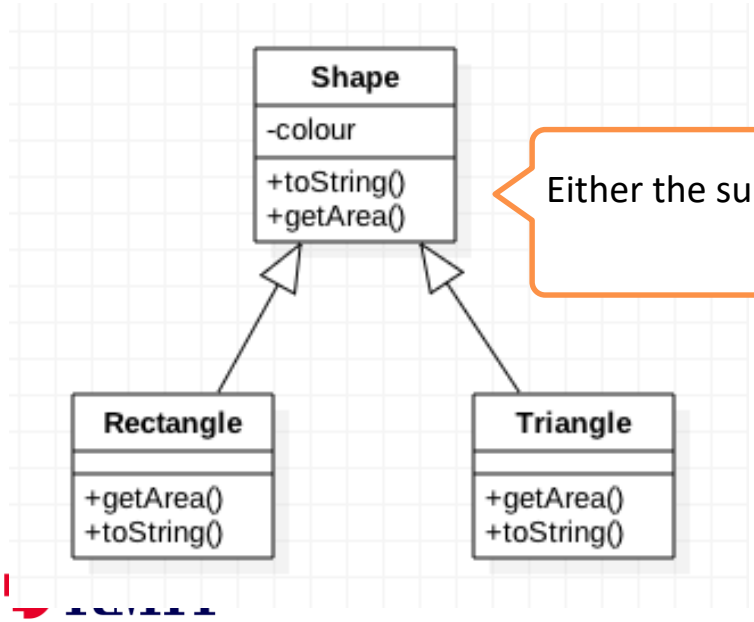
http://www.tutorialspoint.com/java/java_abstraction.htm

Inheritance with an abstract class

Imagine a simple inherited class structure depicted in the UML class diagram below.

We can observe that two methods are inherited by the sub - classes from the parent class **Shape.**

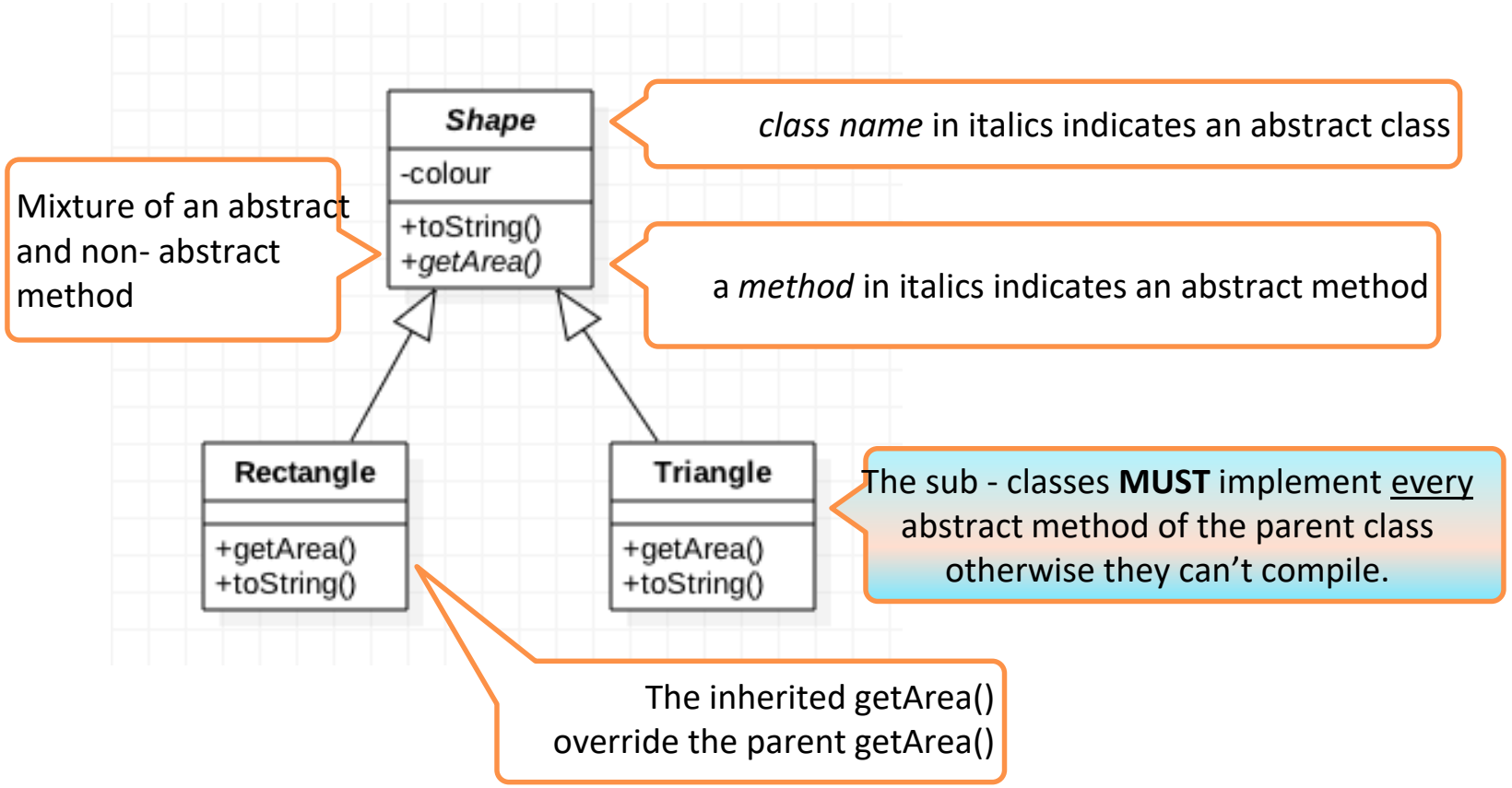
The next slide will show Shape as an abstract class



Either the super class methods can be called or the child class methods can override these with their more specialised methods.

Inheritance with an abstract class

Shape is an abstract class with **getArea**( ) being an abstract method.



RMIT UNIVERSITY

So what do these changes mean?

```
public abstract class Shape
{
    public abstract void getArea();
 }
```

abstract keyword used in declarations

The method contains no code & no { }

- Shape class can no longer be instantiated.
- The **getArea()** method in Shape class cannot be called. Without being abstract it still could.
- Inherited classes Rectangle & Triangle class MUST have a **getArea()** method. They can specify the code they want for this method.

By making **getArea( )** abstract you are forcing all child classes to implement that method. It effectively creates a sort of template, a requirement on the inheritance structure. If any further child classes are added, they will have no choice but to build this method in their class.

RMIT UNIVERSITY

Characteristics of an Abstract Class

A class which contains the **abstract** keyword in its declaration is known as abstract class.

- Abstract classes may or may not contain abstract methods i.e. methods with out code statements.
- But, if a class has at least one abstract method, then the class (super) must be declared abstract.
- If a class is declared abstract it **cannot be instantiated.**
- If you inherit an abstract class you have to provide implementations for all the abstract methods in it.

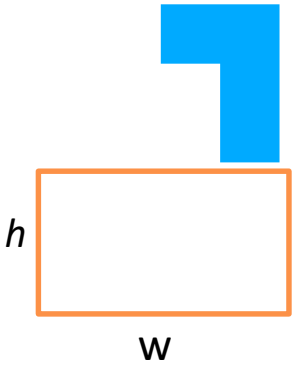
This forces design constraints on any inherited classes.
It is a means of specifying the minimum methods that the child class must have.

RMIT UNIVERSITY

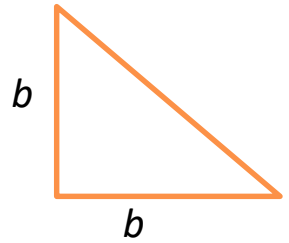
http://www.tutorialspoint.com/java/java_abstraction.htm

Exercise – Shapes

Create an abstract class called *Shape*, with a method called *computeArea()* that computes the area of of the shape. Should *computeArea()* be abstract?



Create a *Rectangle* class that extends *Shape* with width and height attributes.



Create a *IsoscelesRightTriangle* class that

```
public class ShapeDriver {

    public static void main(String[] args) {

        Shape shapes[] = new Shape[3];

        shapes[0] = new Rectangle(3,5); //width 3, height 5
        shapes[1] = new IsoscelesRightTriangle(5); //base 5
        shapes[2] = new Rectangle(5,5);

        for (int i = 0; i< shapes.length; ++i)
            System.out.println("Area is : " + shapes[i].computeArea());
    }
}
```

Abstract methods are optional

You do not need to implement an **abstract class**, however it helps defining a discipline and structure for your APIs to adhere to

- In particular, the **polymorphic** processing of child class objects becomes more precise and better structured
- Super class objects cannot be instantiated. Thus processing only has to deal with child objects
- A level of abstraction occurs because other programmers adding child classes in the future don't need to be concerned with the abstract methods. They only need to provide the implementations that are relevant to their codebase

RMIT UNIVERSITY

Abstract vs Concrete class

- Note an abstract method has no implementation.
- You cannot construct objects of classes with abstract methods - called an abstract class.
- In Java, you must declare all abstract classes explicitly with the keyword abstract.
- If a subclass of Shape such as Square overrides this method providing an implementation, and has no other abstract methods then Square will be considered a concrete class.
- Then we can create instances of Square class.

RMIT UNIVERSITY

Java Interface

The purpose of an interface is similar to an abstract class and abstract methods.

It acts as a template that specifies methods that must be implemented into every class that you choose to connect to it via the reserved **implements.**

An **Interface**, any class can 'connect' by using the reserved word **implements** in their class declaration.

Inheritance (class heirarchies) model an 'is a' relationship

Interfaces model a 'behaves like' relationship.

A class can implement many interfaces, but inherit from only one super class.

RMIT UNIVERSITY

Interfaces in Java 7

An `interface` is like a `Class`, with no bodies in the methods. It may define constants (`public static final`) but no runtime instance variables.

Usually, an **`Interface`** is **`public`**.

An interface provides a standard way to access a class which could be implemented in many ways.

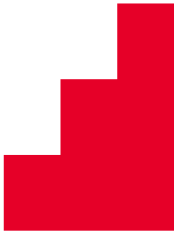
***The Java Tutorials*:**

"There are a number of situations in software engineering when it is important for disparate groups of programmers to agree to a 'contract' that spells out how their software interacts."

"Each group should be able to write their code without any knowledge of how the other group's code is written."

"Generally speaking, *interfaces* are such contracts."

RMIT UNIVERSITY



Interfaces in Java 8

In Java 8, an interface may contain

`default` implementations of instance methods, and

implementations of `static` methods.

In any OO language, an interface

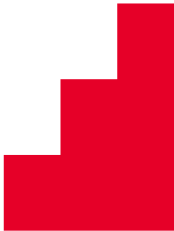
cannot be instantiated, and

defines a “contract” which any realization of the interface must fulfil.

Java is a strongly-typed language.

Java compilers *can* enforce contracts, by refusing to compile classes whose implementations might “partially realize” an interface.

RMIT UNIVERSITY



Java Interface

A Java Interface **is not** a class so it can never be instantiated. An interface is a collection of abstract methods.

A class implements an interface.

interface example

```
interface MyInterface
{
  /* All the methods are public abstract by default
   * These methods have no body…no code
   */
  public abstract void method1();
  void method2();
}
```

methods are always abstract even though the key word is not used. Are always public even if not declared public

```
class Accounts implements MyInterface
{
   // code statements
}
```

Just like with an abstract class the Accounts class is forced to implement **method1**() & **method2**() by the interface

RMIT UNIVERSITY

http://beginnersbook.com/2013/05/java-interface/

Java Interface - can have data (constants)

Variables can be declared in an interface.
If declared, they do **not** have to be utilised by a class implementing the interface.

```
interface MyInterface
{

  public static final total = 0;   //all variables are public + static + final
  int x = 100;                     // even if not explicitly declared as such


   public abstract void method1();
   void method2();
}
```

final …any variable is a constant & so value cannot be modified.
And what does static mean?

Any variable needs to be initialised upon declaration otherwise it will not compile:

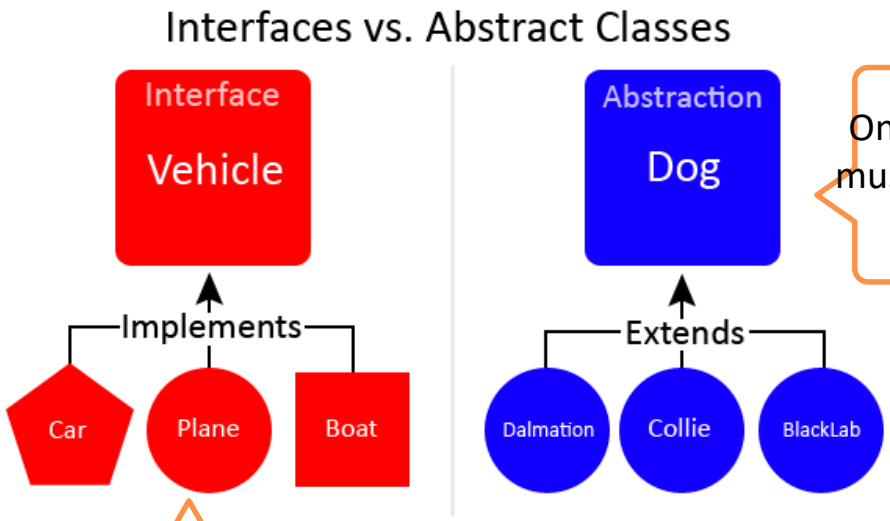
```
int x;     //error - invalid declaration
```

RMIT UNIVERSITY

http://beginnersbook.com/2013/05/java-interface/

difference - interface and an abstract class

There is a subtle but important difference in the consequences of implementing an interface and / or an abstract class



Only **abstract** methods in **Dog class** must be implemented in **every** child class.

All methods in the **Vehicle** interface **MUST** be implemented in **each** class that **implements** the interface.

RMIT UNIVERSITY

Why bother with an interface?

An interface can be used to provide a form of multiple inheritance but beyond that it allows you to specify methods or behaviours to classes that may not even be related in any way.

It is a way to specify compulsory methods to any class you wish in the same way an abstract class can impose method implementation on its child classes.

Imagine a need to be able to play a sound for any object type in your program:

objectType.**playsound()**;

It could be a car, plane, cat, bird, the wind, an explosion, a particular singer or band, breaking glass, a police siren, water fall, rain, a tv, static, etc. All possible objects are **unlikely** to be in an inherited relationship, especially as Java has only one level of inheritance. An interface solves this.

RMIT UNIVERSITY

Interfaces with default methods

Java's interface language feature lets you declare interfaces

with abstract methods that must be implemented in the classes that implement the interfaces.

You are required to implement all abstract methods

after publishing the interface you cannot add new abstract methods to it without breaking source and binary compatibility.

Java 8 addresses these problems by evolving the interface to support *default* and *static* methods.

A default method is an instance method defined in an interface whose method header begins with the default keyword;

it also provides a code body.

Every class that implements the interface inherits the interface's default methods and can override them.

RMIT UNIVERSITY

Default method example

```
public interface Addressable
{
   String getStreet();
   String getCity();

   default String getFullAddress()
   {
      return getStreet()+", "+getCity();
   }
}
```

```
public class Letter implements Addressable
{
   private String street;
   private String city;

   public Letter(String street, String city)
   {
      this.street = street;
      this.city = city;
   }

   @Override
   public String getCity()
   {
      return city;
   }

   @Override
   public String getStreet()
   {
      return street;
   }

   public static void main(String[] args)
   {
      // Test the Letter class.

      Letter l = new Letter("123 AnyStreet", "AnyCity");
      System.out.println(l.getFullAddress());
   }
}
```

RMIT UNIVERSITY

Abstract class & Interfaces

Which should you use? - http://docs.oracle.com/javase/tutorial/java/IandI/abstract.html

Consider using **abstract classes** if any of these statements apply to your situation:

- You want to share code among several closely related classes.
- You expect that classes that extend your abstract class have many common methods or fields, or require access modifiers other than public (such as protected and private).
- You want to declare non-static or non-final fields. This enables you to define methods that can access and modify the state of the object to which they belong.

Consider using **interfaces** if any of these statements apply to your situation:

- You expect that unrelated classes would implement your interface. For example, the interfaces (Java inbuilt) **Comparable** and **Cloneable** are implemented by many unrelated classes.
- You want to specify the behaviour of a particular data type, but not concerned about who implements its behaviour.
- You want to take advantage of multiple inheritance of type.

https://docs.oracle.com/javase/tutorial/java/IandI/abstract.html