



Murdoch
UNIVERSITY

Topic 4: Advanced Java – objects, composition, inheritance, collections

ICT373: Software Architectures

Recap

- Java overview
- Objects
- Java revision
- O-O design and The Unified Modelling Language (UML)

Overview

- Reuse and Packages
- Composition (or aggregation)
- Inheritance, Polymorphism, Dynamic Binding
- Abstract Classes and Inner Classes
- Collections
- Exception Handling

Reading:

Textbook (11th ed.) Chapters 9, 10, 11, 14, 16 and 20

Objectives (1/3)

- Describe and explain the use of: abstraction, encapsulation, information-hiding, localization and name management.
- Use packages in Java.
- Explain the use of access modifiers in Java.
- Explain the term composition and be able to represent it in UML.
- Implement composition in Java including initialization of components.
- Explain inheritance and be able to represent it in UML.
- Implement inheritance in Java including initialization.
- Use, explain and implement overriding in Java.
- Make basic use of 'this' and 'super'.

Objectives (2/3)

- Explain the need for and use of the class **Object**.
- Correctly choose between composition and inheritance in a design.
- Explain the concepts of polymorphism and dynamic binding.
- Explain upcasting and downcasting and how to implement them in Java.
- Explain the uses for and use of abstract classes, interfaces and the 'final' keyword.
- Describe multiple inheritance and how it can be implemented in Java.
- Explain what is an inner class? Explain the uses of inner classes.
- Explain what is a call-back function? Explain the use of call-back functions.

Objectives (3/3)

- Explain the Java Collections Framework.
- Describe the use of data structures like collections in Java and the use of iterators.
- Explain the use of generics and enhanced for loop in Java.
- Explain how exceptions are handled in Java.



Murdoch
UNIVERSITY

Objects, composition, inheritance

Abstraction, Encapsulation and Re-use

- Important elements of OOSE.
- **Abstraction:** dealing with the essential features of something while ignoring the detail.
 - eg, defining a class instead of dealing with a bunch of separate objects.
- **Encapsulation:** providing things with a protected inside
 - eg, having a public interface so that client users do not need to know about the inner workings.
- **Information-hiding:** hiding or encapsulating the details
- **Localization:** allowing changes inside a class/object/thing to be made while the public interface is unaffected.
- **Name-management:** preventing confusion and clashes with identifier names

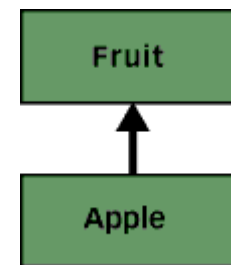
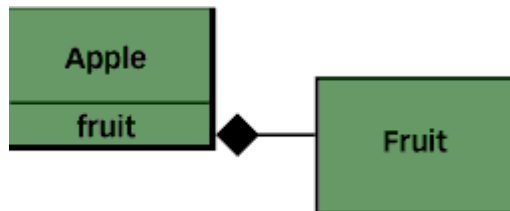
Abstraction, Encapsulation and Re-use

- **Reuse:** using the existing classes without changing their code:
 - Inheritance
 - composition
 - packages
 - Collection
 - Methods
 - Design patterns

Abstraction, Encapsulation and Re-use

Composition: simply create objects of your existing class inside the new class, meaning that the new class is composed of objects of existing classes. You are reusing the functionality of the code, not its form.

Inheritance: involves creating a new class as a *type* of an existing class. i.e., you take the form of the existing class and add code to it without modifying the existing class.



Abstraction, Encapsulation and Re-use

- Inheritance is implemented in Java with the ***extends*** keyword.
- You make a new class (derived class) and you say it extends an existing class (base class)
- Note that inheritance is an integral part of all OOP languages.
 - In java, we always use inheritance when a class is created, because unless we explicitly inherit from some other class, we implicitly inherit from Java's standard root class **Object**.
- Overall, OO Languages, i.e. languages which support these ideas, support reuse and pluggable components. This helps with speed and correctness of design and implementation.
 - Java supports these ideas at a class level.

Libraries and Packages

- Java also supports encapsulation above the class level with the **package** feature.
- A package is a collection of related classes that have been grouped together in a directory (and given a package name):
 - which may contain classes private to the package (supporting encapsulation) – to be used only by package classes; and
 - public classes - which can readily be imported together for use by other classes (supporting reuse).
- To make a package available we need to use **import** keyword, e.g. **import java.util.*;**

Libraries and Packages

- Many useful packages already exist and are ready for use by implementers as libraries. These include the standard class libraries (which we have met), e.g., Java API packages `java.lang`, `java.util`, `javax.swing` and Java Collection classes.
- Reuse also requires good documentation. A special feature of Java libraries is the ease of use of **javadoc** to directly generate online and fully cross-linked HTML descriptions.
 - Documentation is an important part of the public interface of an encapsulated class.

Package usage (1/3)

- Before a class can be imported into multiple applications, it must be placed in a package.
- In order to use a class in a package
 - the class must be declared to be part of the package (the class's first statement)
 - the package must be in the right directory
 - the class must be imported by the client class
 - To declare a class **A** as part of a package **X.Y** put
 - **package X.Y;**
 - as the first statement in the source file for class **A**.
 - (If there is no such declaration then the class belongs to a package called the **default package** and all such classes belong to that same package.)
 - Also, the class **A** must be declared a public class (otherwise it can only be used by other classes in this package).

Package usage (2/3)

- The classes belonging to a package must be compiled. Then make sure that all the **.class** files in the package are placed in an appropriate directory, eg

\myclassdirectory\X\Y

and that the operating system's environment variable

CLASSPATH includes **\myclassdirectory**

- For further information on the classpath, see:

<http://docs.oracle.com/javase/7/docs/technotes/tools/windows/classpath.html>

Package usage (3/3)

- Once the class is compiled and stored in its package, put

import X.Y.A; or

import X.Y.*;

in the client source file and refer to class **A**.

- Or, in the client, refer to the class as **X.Y.A**
- The designers of Java recommend that a package name should start with your internet domain name in reverse order to provide unique names for packages. Eg, the package statement

package au.murdoch.it.ict373.topic4;

indicates that the class be placed in directory
au\murdoch\it\ict373\topic4

Access to Top Level Classes

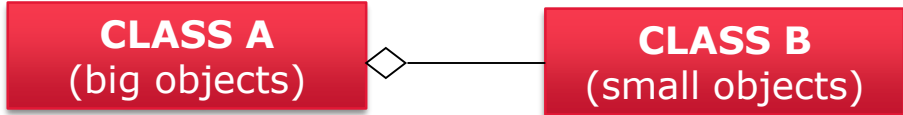
- Access modifiers are available to support encapsulation.
- A top level class is a class which appears in a source file not nested inside another class.
- These top level classes can have their access/visibility declared as
 - **public** – meaning that this class can be used by any other class. (Note that there can only be at most one top level class declared as public in any source file.) **OR**
 - **there may be no declaration** in which case this top level class can only be accessed within the package.
 - This is sometimes called “friendly” access.

General Access Modifiers

- save clients from having to know all about your implementation;
- stop clients from using things that you might want to change;
- stop clients from using methods (or changing variables) which should only be used very carefully in certain circumstances at the risk of making your program run incorrectly;
- hide local things to help prevent name clashes.

Access Levels				
Modifier	Class	Package	Subclass	World
public	Y	Y	Y	Y
protected	Y	Y	Y	N
no modifier	Y	Y	N	N
private	Y	N	N	N

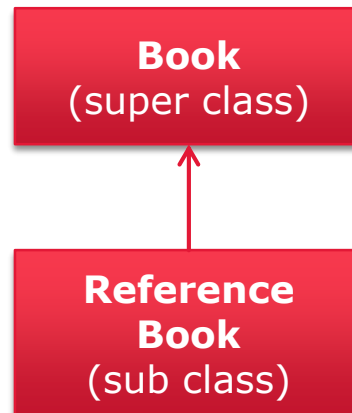
Composition

- One way of (re-)using another class B is to have a member variable of type B as part of the definition of your class A.
 - This is known as composition. It is the “**has-a**” relationship.
 - Each object of class A has an object of class B as part of itself.
 - For example, a staff member object might have an address object and each address object might have a postcode object.
 - Note that composition between classes is an example of an **association** which should be recorded as part of the design.
- Composition in UML,

```
classDiagram
    class A["CLASS A (big objects)"]
    class B["CLASS B (small objects)"]
    A "1" *-- "1" B
```
- Class A has a member variable of type B (and maybe lots of other member variables).
 - Note that this means that constructing an A object requires construction of a B object.

Inheritance

- We might want to say that each object of class B is also an object of class A, i.e. it has all the variables and methods that A objects have but it may have more.
- This is the “**is-a**” relationship.
- Class B is a bit more specific in its details. Class A is known as the **superclass** (**base class** in C++), class B as the **subclass** (**derived class** in C++).
- For example, in a library system, a reference book is a book. In UML,



Inheritance in Java

- We use the following syntax:

class ReferenceBook **extends** Book

{ method and variable declarations }

- Objects of the subclass type (e.g. ReferenceBook) will then have all the methods and variables of the Book class plus any extra ones which have been declared as belonging to ReferenceBooks.
- You do not need to re-declare the superclass methods and variables (unless you want to change the body of a method).

Inheritance in Java

- If Book has a **getTitle()** method and **bk1** contains a reference to a ReferenceBook, then you can call

bk1.getTitle();

and the method body defined in the Book class will be executed for the ReferenceBook. This will work because the ReferenceBook has all the other variables and methods that books are supposed to have.

- The methods defined specifically for ReferenceBooks can also call the Book methods, e.g. the method

printRedLabel();

(defined only for Reference books) can call

str = getTitle();

Initialize a member variable

- an initialization expression in the declaration **int i = 25;**
- a statement immediately after the declaration
int i ;
i = 25 ;
- assignment inside the *constructor* for class A
- default, primitives get a 0 value, other types get a **null** reference.

Overriding (vs Overloading)

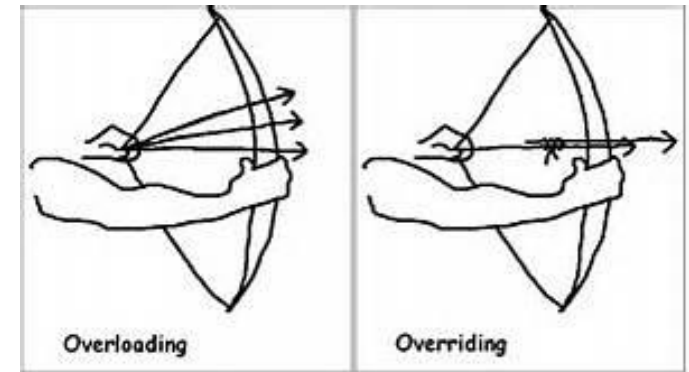
- We can also re-define methods inside ReferenceBook. Eg, Books might have a borrow() method but we can also give ReferenceBooks a new definition for a borrow() method which might just give an error message on a screen saying that the book can not be borrowed.
- This is known as **overriding** because calling the method on a subclass object will result in the subclass definition being used. The more general method is overridden.
- Note: do not get **overriding** confused with **overloading**, which is using the same name for several different methods but distinguishing them by the number of, and/or types of their arguments.

Overriding (vs Overloading)

Static vs dynamic binding

```
class Books {  
    Overriding  
    public void borrow(){  
        System.out.println("You can borrow this book");  
    }  
}  
  
class ReferenceBooks extends Books{  
    public void borrow(){  
        System.out.println("You can borrow this reference book");  
    }  
}
```

Same method name; same parameter



```
class Books {  
    Overloading  
    public void borrow(){  
        System.out.println("You can borrow this book");  
    }  
  
    public void borrow(int n){  
        System.out.println("You can borrow these books");  
    }  
}
```

Same method name; different parameter

Access: private vs protected

- Subclass methods have no access to the private methods or private members of the superclass.
- Eg, the implementer of the ReferenceBook class may not be able to change the location of a ReferenceBook in a ReferenceBook method except by calling the appropriate methods supplied with the Book class.
- If the implementer of the superclass wants to allow subclass methods to access some variables or methods then they should be declared as **protected**.

Inheritance and constructors

- If class B extends class A then construction of a B object involves, in a certain sense, the construction of an A object. In fact, the basic A object should be built first and the extra B bits added on afterwards.
- Suppose a B object (derived from A) is **newly** created in a program.
 - If the B programmer does not explicitly request anything else, then the first step after memory allocation, is for the default constructor for A objects to be called.
 - When the A constructor is finished then the initializations in the B definition are executed.
 - Then body of the B constructor is executed.

this keyword (1/2)

- Recall that a constructor is used when client code uses the **new** keyword. Eg,

b = new B();

uses the constructor B() in the class B to build a new B object and puts a reference to it in the variable b.

- A constructor with no arguments is called a default constructor.
 - You can write your own constructor, or,
 - if you provide a class with no constructors then Java will automatically provide a default constructor for the class.
 - Note, if you write any constructors then none will be supplied automatically.

this keyword (2/2)

- You can also provide constructors which do have arguments. Eg, provide

B(int n) { constructor body using n}

and the client can use **b = new B(7);**

- If you have lots of constructors then they are distinguished by the number and types of arguments.
- If you are supplying several constructors then one can use another by means of the **this** keyword. Eg,

B (int n, char c)

{ this(n); initial=c; }

- Here a call to **new B(7, 'a')** results in the constructor call **B(7)** happening first to make the new B object, and then the **initial** field being given the value in the variable **c**.

super keyword

- If B is a subclass of A then we know that an A object will be built first before being turned into a B object.
- How do we control which A constructor is used?
- When **new B(args)** is called, then immediate appearance of **this** inside the **B(params)** constructor, might cause another B constructor to be called first.
 - Since recursion is not allowed here, eventually the last B constructor to be called is called.
- Now an A constructor (with empty parameter list) will be called unless the B constructor starts with a call to **super(args)**. This will result in the A constructor with matching parameters being called.
- Note that both **super** and **this** have other uses:
- Naively, **this** gives access to the current object (which owns the method that the code is in) and **super** gives access to the superclass version of the current object.

thisvssuper.Java example

/* test results:

-1 <- a non printable char is outputted here for the
 uninitialised variable 'initial'

5 ?

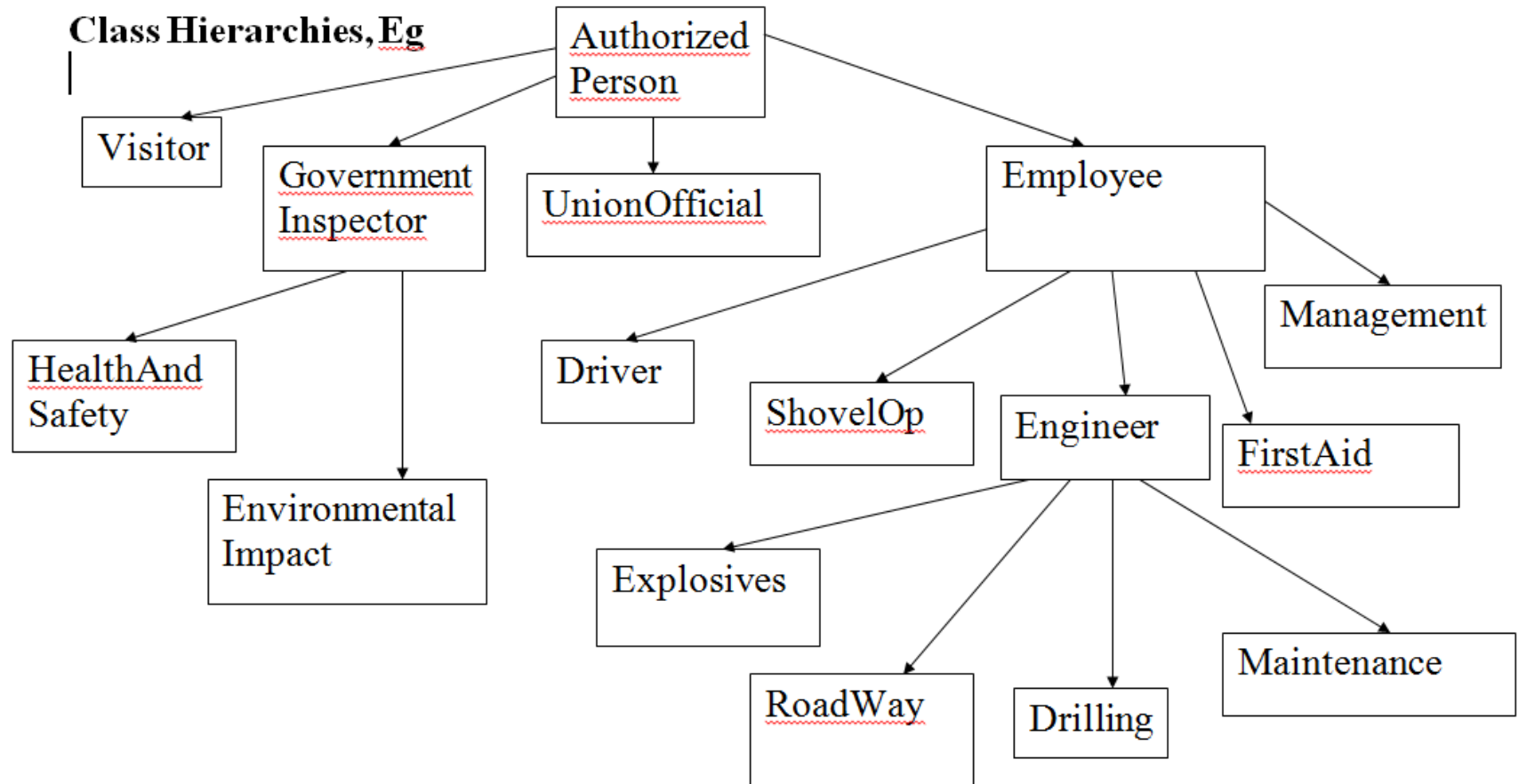
7 a

7

a

*/

Class Hierarchies



Objects

- For various reasons (e.g., garbage collection and collection classes), it is useful to have *a single-rooted inheritance hierarchy*, i.e. all objects belong to one class.
 - In Java all other classes inherit from the class called **Object** (found in library java.lang) but it is not necessary to declare that your classes extend **Object**.
 - all objects have the methods which belong to the Object class (e.g., clone() - for copying, equals for comparing two objects for equality, getClass() - for class determination during runtime, finalize() - for garbage collection, toString() - for string representation, and wait(), notify(), notifyAll() - to handle multithreading)
 - any object can be referred to by a variable of Object type (e.g., data structures can be built which can contain elements of any type).
 - Note that arrays are Objects too and so these two properties apply to them as well.

When to use composition instead

- We have said that inheritance is a way of getting B objects to be just like A objects but with a few extras.
- This allows us to re-use the A code and allows us to re-use the A public interface (ie the public methods).
- If it is really only code re-use that you are after, and not interface re-use, then it may be better (more correct) to use composition instead of inheritance.
- Thus we might put an A object inside as part of each B object. All the functionality (and code) of the A object is then available to the B implementer, but they can hide all of this inside a public interface which is just right for B objects.
 - a coloured square is a square with colour (inheritance);
 - a circle has a centre point (composition);
 - a professor is a staff member (inheritance);

Polymorphism

- How can inheritance be used?
- Suppose class B extends class A. This means that B objects can be used where an A object would be expected. The B object can be *upcast* to being an A object.

Eg, Variables of type A can refer to objects of type B.

A vara;

B varb = new B();

vara = varb; // upcasting – an A object was expected here

Eg, methods of class A can be called on B objects.

System.out.println(varb.name());

assuming that class A has a public **name()** method of String return type.

Polymorphism

- Being able to use a facility (variable, method, function etc) with different types of arguments is called ***polymorphism***.
- This is a basic feature of O-O languages and O-O design. It allows re-use of public interfaces and code in a very convenient form. Eg,

window.repaint(); (buttons, frames, dialogues, ...)

account.debit(n); (savings, cheque, credit, ...)

memb.printAddrLabel(); (staff, student, postgrad, ...)

Static vs dynamic binding(1/3)

```
package javaapplication28;

import java.util.Collection;
import java.util.HashSet;

/**
 *
 * @author 20150534
 */
public class StaticBindingTest {

    public static void main(String args[]) {
        Collection c = new HashSet();
        StaticBindingTest et = new StaticBindingTest();
        et.sort(c);
    }

    //overloaded method takes Collection argument
    public Collection sort(Collection c){
        System.out.println("Inside Collection sort method");
        return c;
    }

    //another overloaded method which takes HashSet argument which is sub class
    public Collection sort(HashSet hs){
        System.out.println("Inside HashSet sort method");
        return hs;
    }
}
```

Static vs dynamic binding(2/3)

```
public class DynamicBindingTest
{
    public static void main(String args[])
    {
        Vehicle vehicle = new Car(); //here Type is vehicle but object will be Car
        vehicle.start();              //Car's start called because start() is overridden method
    }
}

class Vehicle
{
    public void start()
    {
        System.out.println("Inside start method of Vehicle");
    }
}

class Car extends Vehicle
{
    @Override
    public void start()
    {
        System.out.println("Inside start method of Car");
    }
}
```

Static vs dynamic binding(3/3)

- Static binding in Java occurs during Compile time while Dynamic binding occurs during Runtime.
- private, final and static methods and variables uses static binding and bonded by compiler while virtual methods are bonded during runtime based upon runtime object.
- Static binding uses Type(Class in Java) information for binding while Dynamic binding uses Object to resolve binding.
- Overloaded methods are bonded using static binding while overridden methods are bonded using dynamic binding at runtime.

Downcasting

- Dynamic binding does not mean that the compiler gives up completely on type-checking of expressions. Methods and types of variables are checked just as thoroughly.
- Suppose that class B extends class A and that B has an **extra()** method which A does not have. The following code will not compile

```
A vara;    // class A object
```

```
vara = new B(); // class B object is assigned to A variable
```

```
vara.extra(); // class A does not have this method –  
                // vara has to be downcast first
```

because the compiler does not think that A objects have extra methods.

Downcasting

- There are situations in which we upcast (and so lose some type information) and then, later, want the type information back again.
- To do this we must ***downcast***, i.e. start referring to an object by a more detailed reference. Unlike, upcasting, downcasting involves explicit mention of the class:

`((B)vara).extra();`

- Upcasting from B to A can always be done. However, downcasting involves a risk of a run-time error: trying to cast an A type of object to a B reference although the object is not a B object. Programmer beware.!!

Abstract classes

- Abstract classes are intended to be used only as superclasses. They can not be used to instantiate objects. They are regarded as too generic (or incomplete) to create real objects.
- An abstract class normally contains one or more abstract methods.
- Suppose we have a bunch of similar class types B, C and D. Say that they share a lot of public interface and maybe some implementation. You sometimes want to treat them alike.
 - A good design is to identify the common interface, call it class A, and make B, C and D all extend A.
 - A client can then treat them all using A methods and A references when it is convenient.
 - Often, you do not even want there ever to be any A objects in existence.
- In that case class A can be made abstract by declaring it with keyword abstract: **abstract class A { fields and methods }**

Abstract classes

- The compiler will complain if someone tries to construct an A object. (But you can have many A variables.)
- You may want B, C and D to all have a **common(n)** method and to put that in class A but they may all implement it differently. In that case put
abstract void common(int n);
- in the class A with no method body. The compiler will check that you **override** the method in each subclass.
- Eg, we could have an abstract class Shape, and derive from it such classes as Square, Circle and Triangle. We could have an abstract method declared inside Shape
abstract void draw();
- Each of Square, Circle and Triangle classes will provide their own implementations of the draw method. Then **shape.draw()** can be called for drawing squares, circles and triangles.

Concrete Class

Concrete Classes – those used to instantiate objects. They provide implementations of every method they declare

final Methods and Classes

- The opposite of ***abstract*** is ***final***.
- A whole class can be declared **final** if you do not want to allow it to be inherited from (i.e., it cannot be a superclass).
 - This might be done to clarify a design decision, for security or safety reasons, or to allow for efficiency in implementation.
- The methods in **final** classes are implicitly **final**, e.g., class String.
- A method declared **final** in a class cannot be overridden.
 - This can be done for design or efficiency reasons.
 - The compiler can then know that it does not need to allow for dynamic binding when dealing with this method.
- The keyword **final** is also used for constant primitives and constant references in method arguments.

Interfaces

- This is just a form of inheritance.
- An *interface* in Java is like a completely abstract class. All of its methods are abstract, i.e. they have headings only and no bodies.
 - It can contain fields but these are implicitly static and final (i.e. these are constants). An interface is used to provide a form for a class, but no implementation.

- To define an interface use, e.g.,

(public) interface externalTutor

{ method names and constant fields }

- No modifiers (public, abstract, static etc) should be used for the methods or fields: they are redundant.
- Interfaces can extend one or more other interfaces:

**interface externalTutor extends tutor, marker {
method names and constant fields }**

Interfaces

- Classes can implement interfaces:

class postgradExtTutor implements externalTutor

- and then either the class has to override all of the methods of the interface or the class has to be declared abstract.
- If the class is not abstract, then we can have an **externalTutor** variable with a reference to a **postgradExtTutor** object.
- A subclass that extends a superclass can also implement an interface to gain some additional behaviour (i.e., classes can both extend and implement):

class postgradExtTutor extends postgrad implements externalTutor {

// body of class postgradExtTutor }

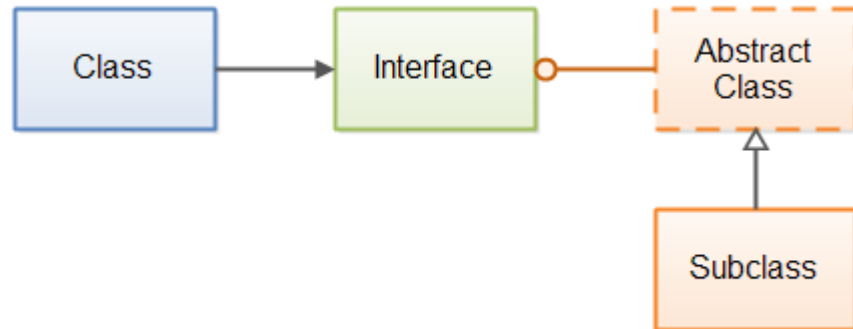
- See the interface Payable example in Section 10.9 (Creating and Using Interfaces) of textbook.

Interface vs Abstract class

- Methods of a Java interface are implicitly abstract and cannot have implementations. A Java abstract class can have instance methods that implements a default behaviour.
- Variables declared in a Java interface is by default final. An abstract class may contain non-final variables.
- Members of a Java interface are public by default. A Java abstract class can have the usual flavours of class members like private, protected, etc.
- Java interface should be implemented using keyword “implements”; A Java abstract class should be extended using keyword “extends”.
- An interface can extend another Java interface only, an abstract class can extend another Java class and implement multiple Java interfaces.
- A Java class can implement multiple interfaces but it can extend only one abstract class.

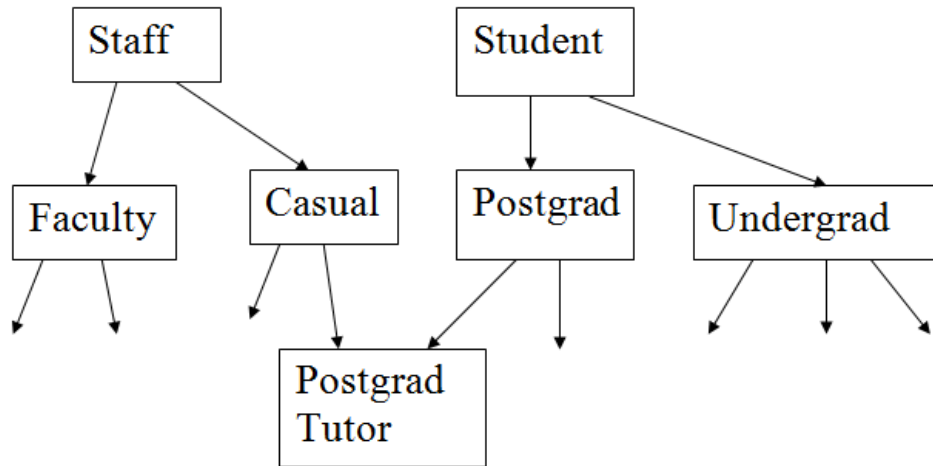
Interface vs Abstract class

- An example showing a class referencing an interface, an abstract class implementing that interface, and a subclass extending the abstract class



- Interface is used when you have scenario that all classes has same structure but totally have different functionality.
- Abstract class is used when you have scenario that all classes has same structure but some same and some different functionality.

Multiple inheritance

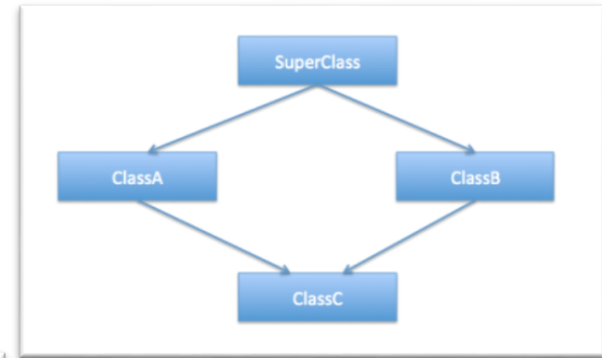


- By using various combinations of interface implementations and class extensions, each Java object can be allowed to act as completely different types of things at different times during the running of a program.
- Such a situation is called ***multiple inheritance***.

Multiple inheritance: diamond problem

SuperClass.java

```
1 package com.journaldev.inheritance;
2
3 public abstract class SuperClass {
4
5     public abstract void doSomething();
6 }
```



ClassA.java

```
1 package com.journaldev.inheritance;
2
3 public class ClassA extends SuperClass{
4
5     @Override
6     public void doSomething(){
7         System.out.println("doSomething implemen
8     }
9
10    //ClassA own method
11    public void methodA(){
12
13    }
14 }
```

ClassB.java

```
1 package com.journaldev.inheritance;
2
3 public class ClassB extends SuperClass{
4
5     @Override
6     public void doSomething(){
7         System.out.println("doSomething implementation of B")
8     }
9
10    //ClassB specific method
11    public void methodB(){
12
13    }
14 }
```

ClassC.java

```
1 package com.journaldev.inheritance;
2
3 public class ClassC extends ClassA, ClassB{
4
5     public void test(){
6         //calling super class method
7         doSomething();
8     }
9
10 }
```

Multiple Inheritance in Interfaces

InterfaceA.java

```
1 package com.journaldev.inheritance;
2
3 public interface InterfaceA {
4
5     public void doSomething();
6 }
```

InterfaceB.java

```
1 package com.journaldev.inheritance;
2
3 public interface InterfaceB {
4
5     public void doSomething();
6 }
```

InterfaceC.java

```
1 package com.journaldev.inheritance;
2
3 public interface InterfaceC extends InterfaceA, InterfaceB {
4
5     //same method is declared in InterfaceA and InterfaceB both
6     public void doSomething();
7 }
```

InterfacesImpl.java

```
1 package com.journaldev.inheritance;
2
3 public class InterfacesImpl implements InterfaceA, InterfaceB, InterfaceC {
4
5     @Override
6     public void doSomething() {
7         System.out.println("doSomething implementation of concrete class");
8     }
9
10    public static void main(String[] args) {
11        InterfaceA objA = new InterfacesImpl();
12        InterfaceB objB = new InterfacesImpl();
13        InterfaceC objC = new InterfacesImpl();
14
15        //all the method calls below are going to same concrete implementation
16        objA.doSomething();
17        objB.doSomething();
18        objC.doSomething();
19    }
20
21 }
```

Multiple Inheritance in Java

- Java only allows a limited form of multiple inheritance (cf C++):
- a class can only inherit method body from one other class. It can only extend one other class. (But it can implement many interfaces.)
- eg, in the class hierarchy shown in the above diagram, the class PostgradTutor can be defined as follows:

```
class PostgradTutor extends Postgrad implements Casual  
{  
  
// body of class PostgradTutor  
  
}
```



Murdoch
UNIVERSITY

Nested Classes/Inner Class



Murdoch
UNIVERSITY

Inner Classes

- A class declared within the declaration of another class.
- An inner class can be:
 - **a nested static class**: subject to the scope of the outer class,
 - can be private within that class,
 - can be public and has to be referred to by **OuterClass.InnerClass**,
 - has access to *static* fields of the outer class,
 - can contain its own static inner classes.
 - a **member class**: each object belongs to an object of the outer class (who created it) and contains an implicit reference to that object, the inner object can access *all variables and methods* of the outer object, inner objects can continue to exist after their class has gone out of scope.
 - a **local class**: declared in a compound statement in a method, can access final local variables.
 - an **anonymous class**: declared in a statement after new and needs no class name, but only one object can be created.

Uses of Inner Classes (1/2)

- The main use of inner classes is to assist in the implementation of the outer class while encapsulating the details. (Information hiding and name management).
- By using inheritance or composition, the programmer can bring all sorts of useful classes inside the implementation of their own class.
- The inner object can access the variables of the outer object but behave like a really useful general type of object which can cooperate with a whole bunch of generally useful classes. (See next example – private class SSelector).
- Even more important but more subtle uses appear in nearly all practical programs, with call-back functions (eg in GUIs) as we will see.

Uses of Inner Classes (2/2)

- An inner object (implementing a widely used interface) can be sent out to be handled by library classes but, all the time, have its own access to your main class. (More later).
- In summary, there are three reasons why one would want to define inner classes:
 - organization: sometimes it seems most sensible to sort a class into the namespace of another class, especially when it won't be used in any other context.
 - access: nested classes have special access to the variables/fields of their containing classes (precisely which variables/fields depends on the kind of nested class, whether inner or static).
 - convenience: having to create a new file for every new type is bothersome, again, especially when the type will only be used in one context.

One detailed example of inner class

- This example is based on **Sequence.java** in chapter 8 of the Eckel reading.
- We suppose that an interface called **Selector** is generally available. A Selector operates on some collection of objects, being able to deliver one at a time in some order.
- Selector is a generally useful interface and someone has developed a useful method **SelPrint.selPrint(s)** which prints out all the objects produced by a Selector from the current object until the end.
- In class **Sequence** we make a collection of objects, deliver up a Selector (so we have to provide a particular implementation for the interface) and get selPrint to print out all our objects.
- The important thing to note is that our Selector has direct access to all the private fields of the Sequence object. (unlike in C++). These are completely hidden from selPrint.

InnerClass example

//SelPrint.java

```
public class SelPrint {
```

```
    public static void selPrint(Selector s){  
        while (!s.end()) {  
            System.out.println((String) s.current());  
            s.next();  
        }//end while  
    }//end selPrint
```

```
}//end of class SelPrint
```

//Selector.java

```
interface Selector {  
    boolean end();  
    Object current();  
    void next();  
}//end of interface
```

//Sequence.java

```
public class Sequence {  
  
    private Object[] o;  
    private int next=0;  
  
    public Sequence(int size){  
        o= new Object[size];  
    }  
  
    public void add(Object x){  
        if (next < o.length){  
            o[next]=x;  
            next++;  
        }//end if  
    }//end add  
}
```

```
//inner class
private class SSelector implements Selector {
    // provides the functionality of interface Selector
    int i=0;

    public boolean end() {
        return i == o.length; // an inner class has automatic access to
    }                          // members of the enclosing class.

    public Object current() {
        return o[i];
    }

    public void next() {
        if (i<o.length) i++;
    }
} //end of class SSelector
```

Callback functions

- This is **another use for inner classes**. With a callback, some other object is given a piece of information that allows it to call back into the originating object at some later point. We will see more details later in the unit. For now, a simplified example follows.
- Consider the situation of several different *important* objects managing the substantial tasks behind a GUI.
- Suppose that for each display button, some of these objects want to register their interest in being informed when the button is clicked.
- They obviously must all have some standard type of object to donate to the GUI but which can refer back to the important donator.

Callback functions

- One solution is to have an interface, **clickable** say, which can cope with being **clicked()**. Get each important object to supply an inner class implementation of clickable for registration. When a button is clicked the button can go through its registration list of clickable objects and call all their **clicked()** methods.
- Being code for an inner class object means that the actual method body which gets executed can have access to the fields of the important object owning it.
- **clicked()** is a *call back* to the important object which supplied it.

```
public interface Callback {  
    //parameters can be of any types, depending on the event defined  
    void callbackMethod(String aParameter);  
}
```

```
public class CallbackImpl implements Callback {  
    void callbackMethod(String aParameter) {  
        //here you do your logic with the received parameters  
        //System.out.println("Parameter received: " + aParameter);  
    }  
}
```

```
//.... and then somewhere you have to tell the system to add the callback method  
//e.g. systemInstance.addCallback(new CallbackImpl());
```




Murdoch
UNIVERSITY

JAVA Collections



Murdoch
UNIVERSITY

Collections: Arrays

- A collection is a data structure that represents a group of objects, known as its elements.
 - Actually, it is an object—that can hold references to other objects.
- Usually, collections contain references to objects that are all of the same type (or subtype).
- Arrays are efficient ways to store and *randomly* access collections of objects.
 - They are efficiently implemented.
 - Bound checking is done during storage and recall.
 - They can be returned by a method: an array is an object.
 - They can hold primitives.
- **But** they have to be fixed in size.
 - Therefore, the Java SE and other Java libraries supply other sorts of collection classes.

Java Collections Framework (1/3)

- A framework is a set of classes that form the basis for building advanced functionality. It contains superclasses with useful functionality, policies and mechanisms.
- The user of a framework writes subclasses to extend the functionality without having to reinvent the basic mechanisms. Eg, Swing is a framework for user interfaces.
- The Java collections framework (JCF) is a hierarchy of classes, abstract classes, interfaces and algorithms that implement commonly usable collection data structures.
- There are two fundamental interfaces for Java collections (JCF):
 - **Collection** – defines a value for each item in the data structure;
 - **Map** – defines a pair of items, key-value, for each element/node in the data structure.

Java Collections Framework (2/3)

- **java.util.Collection** is a root interface in the collections hierarchy from which the **List**, **Queue** and **Set** interfaces are derived. It declares the `add()`, `remove()`, `toArray()` and `contains()` method for a collection.
- **java.util.List** interface is an ordered collection that can contain duplicate elements. In addition to the methods inherited from `Collection`, `List` provides (eg, `add`, `remove`, `get`, `set`) methods for manipulating elements via their indices, manipulating a specified range of elements (eg, `addAll`, `removeAll`, `subList`), searching for elements and obtaining a `ListIterator` to access the elements. Interface `List` is implemented by several classes, including **ArrayList**, **LinkedList** and **Vector**.
- **java.util.Queue** interface - typically, a FIFO collection that models a waiting line.

Java Collections Framework (3/3)

- **java.util.Set** interface - a collection that does not contain duplicates.
- **java.util.Map** interface that associates keys to values and cannot contain duplicate keys. Interface Map is implemented by classes **HashMap**, **TreeMap** and **LinkedHashMap**.
- All collections have an *iterator* that goes through all of the elements in the collection.

Collection classes (1/3)

- The following abstract classes supply many of the routine implementation of the interface methods: **AbstractCollection**, **AbstractList**, **AbstractSequentialList**, **AbstractSet**, **AbstractQueue**, and **AbstractMap**.
- The Java library provides the following concrete classes: **ArrayList**, **LinkedList**, **ArrayQueue**, **PriorityQueue**, **HashSet**, **TreeSet**, **HashMap**, and **TreeMap**.
- There is also a number of “legacy” container classes available before there was a collections framework: **Vector**, **Stack**, **Hashtable**, and **Properties**.

Collection classes (2/3)

- Thus there is a whole hierarchy of useful collection classes and interfaces available for use. Eg, one branch is

interface Collection

|

+--interface List

|

+--java.util.ArrayList

(Class ArrayList implements interface List which extends interface Collection)

Collection classes (3/3)

- The abstract classes and interfaces allow the user to make their own versions conform to the standard ways of use. Create a collection using a concrete class but then pass around a more abstract reference: this allows maximum potential for future changes.
- There are algorithms in these classes for binary search, reversing, sorting etc. There are constructors and methods for converting to and from Arrays.
- There is also a class Arrays (in java.util package) which contains static methods for doing useful things to arrays, eg `binarySearch()`, `equals()`, `fill()`, and `sort()`.

Objects Only

- Collections of Marsupials and collections of lists of addresses, for example, should have methods (eg, `elementAt`) with different argument and return types.
- In earlier versions of Java, it is not possible to write code that can implement a generic collection type, such as a `Stack`, which will work for all types of contained objects.
- In C++, this problem is solved using parameterized types or templates.
- Java does not have templates. Java solves the problem by having collections which only hold `Objects`.
- Since every class inherits from `Object`, this means that every reference type (i.e. not primitive) of object can be put in a collection.

Objects Only

- primitives must be wrapped (eg, **ints** converted to **Integers**) before being collected
- Other objects must be downcast if you want to regain type information after they come out of a collection.
- Recall that downcasting can cause problems: incorrect downcasting causes run-time errors. Take care to note the types of collected objects, or use RTTI= run-time type identification (see later).

Generic Types

- A new feature of J2SE 5.0 is *generics* which allow definitions that include parameters for types.
- This enables designing of methods/classes with common functionality that can be used with multiple data types.
- Generic methods enable programmers to specify a set of related methods with a single method declaration. Eg, we may write a generic method for displaying array elements, then invoke the method with Integer arrays, Double arrays, Character arrays, and so on.

Generics: an example

```
// generic method printArray
public static < T > void printArray( T[] inputArray )
{
    // display array elements
    for ( T element : inputArray )    // enhanced for loop
        System.out.printf( "%s ", element );
    System.out.println();
} // end method printArray
```

```
// create arrays of Integer, Double and Character
Integer[] integerArray = { 1, 2, 3, 4, 5, 6 };
Double[] doubleArray = { 1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7 };
Character[] characterArray = { 'H', 'E', 'L', 'L', 'O' };
```

```
printArray( integerArray );    // pass an Integer array
printArray( doubleArray );    // pass a Double array
printArray( characterArray );  // pass a Character array
```

Generic class

- Generic classes enable programmers to specify a set of related types with a single class definition. Eg, we might write a single generic stack class to manipulate a stack of objects, and then use it to process a stack of Integers, a stack of Doubles, a stack of Strings, and so on.
- The following example uses the JDK1.4 Collections API library:

```
ArrayList list = new ArrayList();  
list.add(0, new Integer(25));  
int total = ((Integer)list.get(0)).intValue();
```

- The above example with the JDK1.5 generified Collections API:

```
ArrayList<Integer> list = new ArrayList<Integer>();  
list.add(0, new Integer(25));  
int total = list.get(0).intValue();
```

Iterators (1)

- An object that allows a program to step through a collection of objects and do some action on each one is called an ***iterator***.
 - Eg, for arrays, an index variable can be used as an iterator, with the action of going to the next thing in the list being something like: `index++`;
- With all collection classes there is a need to be able to extract and examine the elements one at a time.
- To gain access to the internal workings of a collection class, an iterator class is usually defined as an inner class.
- In Java, collections can produce an **Iterator** via their **iterator()** method. Clients can then use **hasNext()**, **next()** and **remove()** methods.

Iterators (2)

- Suppose we have an ArrayList **a** (for example).
- To get an iterator for **a** ...

Iterator it = a.iterator();

- Then **it** is ready to return the first element. To get the first or subsequent elements ...

x = it.next();

- To test for the end (i.e., if there are any more objects) ...

if (it.hasNext()) ...

- To remove the last element returned by the iterator ...

it.remove(); // remove() is an optional operation.

- You can have several iterators going at once.

Iterator: example

```
import java.util.*;

public class Pet {
    private int petNumber;
    public Pet(int i) { petNumber = i; }
    public void displayPetId() {
        System.out.println("Pet No " + petNumber);
    }

    // client test method
    public static void main(String[] args) {
        List myPets = new ArrayList();
        // create 10 pet objects in the container myPets
        for (int i = 1; i <= 10; i++)
            myPets.add(new Pet(i));

        // create an iterator to select list objects
        Iterator it = myPets.iterator();

        // get each list object, and display it
        while(it.hasNext())
            ( (Pet)it.next() ).displayPetId();

    } // end main
} // end of class Pet
```

```
import java.util.*;

public class Pet1 {
    private int petNumber;
    public Pet1(int i) { petNumber = i; }
    public void displayPetId() {
        System.out.println("Pet No " + petNumber);
    }

    // client test method
    public static void main(String[] args) {

        List<Pet1> myPets = new ArrayList<Pet1>();
        // create 10 pet objects in the container myPet
        for (int i = 1; i <= 10; i++)
            myPets.add(new Pet1(i));

        // get each list object using enhanced for loop of jdk1.5
        for (Pet1 p : myPets)
            p.displayPetId();

    } // end main
} // end of class Pet1
```




Murdoch
UNIVERSITY

Exceptions



Murdoch
UNIVERSITY

Exceptions

- Another important Java class hierarchy deals with exceptions: **objects allowing unusual situations to be handled without disrupting the normal code.**
- The syntax for exception handling is roughly
 - We **try** to execute some code.
 - One of several possible exceptions might be **thrown** at some point.
 - Then execution is halted and resumed at a later place designed to **catch** and deal with that type of exception.
 - The **finally** block (if present) will execute whether or not an exception is thrown in the try block.

```
{ ...  
try {  
    ...  
    f();           //f might throw a badName exception  
    if x==0 throw new Exception("x=0");  
    ...  
}  
catch(badName e) {  
    System.out.println("a bad name was used");  
}  
catch(Exception e){  
    System.out.println(e.getMessage());  
}  
finally { // statement(s)  
  
}  
}
```

Throwing an exception

- When the values of variables are discovered to be unacceptable for current processing to continue and the local code is not the right place to deal with the problem then an exception should be thrown as follows:

throw new TypeOfException("optional string");

- The type of exception can be a built-in or user defined one.

Catching an Exception

- If the **throwing** code is inside a **try** block then the JVM starts at the inner-most such block and examines the following exception handling blocks (indicated by the **catch** keyword) in order.
- If the exception (which was thrown) matches the type of error being caught then that block is used. Matching works hierarchically, so general catchers can catch lots of types of exception. Eg,
catch (Exception id) will catch all exceptions.
- The handler block code can make use of the string message in the exception object via **id.getMessage()**
- If the JVM gets to the end of a method with an unmatched exception then it goes back to the method of which called that method to see whether the calling point was in a **try** block.
- This continues from nested **try** blocks outwards and all the way back through the stack of method calls.
- Thus the programmer can arrange for the exception to be handled at the right level.

Exception Specification

- The code for any methods you write should declare (for the benefit of any clients) any types of exceptions which might be thrown in your method but not handled there. In that case the client (user of your class) has to handle them.
- Do so as follows:

```
void f () throws badName           { ...  
    if name.equals("fred")  
        throw new badName("fred");  
    ...}
```

- The compiler will actually check that you make such a declaration and your code won't compile if you miss any potential exceptions.
- However, one class of common exceptions called **RuntimeExceptions** does not need to be declared. This class includes null references being called, array bounds being exceeded etc.

Creating your own exceptions

- Need to add your own new classes of exception to the hierarchy. To make use of all the benefits of the built-in exception handling procedures you need to inherit from an existing class. Eg,

```
class MyException extends Exception {  
    public MyException() {}  
    public MyException(String msg){  
        super(msg);  
    }  
}
```

Note the two constructors, one relying on the automatic calling of the base-class default constructor, the other handing over to the **Exception(String msg)** constructor.

- By carefully constructing your hierarchy of exceptions you can make sure that exceptions are handled at appropriate places in your code.

throw vs try-catch

```
private void calculateArea() throws Exception {  
    ....do something  
}
```

```
private void calculateArea() {  
    try {  
        ....do something  
    } catch (Exception e) {  
        showException(e);  
    }  
}
```



Murdoch
UNIVERSITY

Summary



Summary

- This topic looked at some of the ideas for creating programs and systems from components and how to reuse components to create others.
- UML notation was used for describing such systems and in particular for the relationships ***has-a***, ***is-a***, ***uses-a*** used to represent composition, inheritance, and association respectively.
- You should be able to answer questions like the ones given at the beginning of this document under the Objectives section.
- Java collections
- Java exception handling



Murdoch
UNIVERSITY

