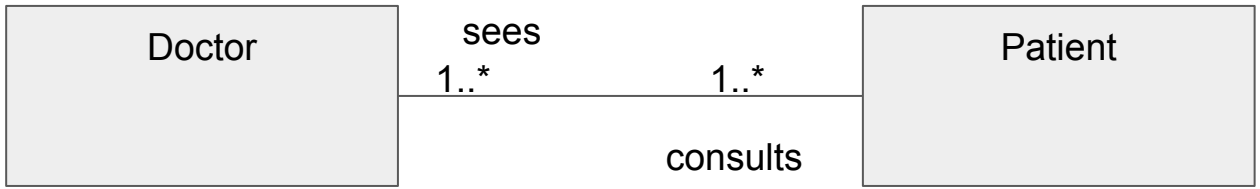
ESS 201: Programming in Java

Week 5
Term 1: 2018-19
T K Srikanth

Relationships between objects

Association: a weak relationship between objects. E.g. Doctor and Patient

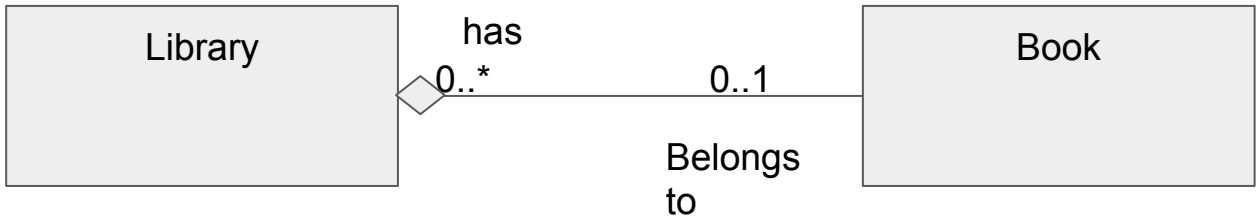
- A path of communication or link between objects, where one object “uses” possibly multiple instances of the other. No ownership or other semantics are implied
- The lifecycles of the objects are independently defined



Relationships: Aggregation

An object consists of (or is made up of) instances of objects of another class, but the lifecycle of the two objects are independent

E.g. Library and Books



Relationships: Composition

An object contains (and *owns*) instances of another object, and the second object does not exist if the owning object is deleted.

E.g. Bank and Account

Existence dependency: Component exists only when its container is around

Composition - Re-use of Implementation

Composition enables the design of complex classes by re-using existing classes: we re-use implementation of these classes

The new class is able to leverage all the functionality of one or more existing classes.

However, it typically hides the existence of these classes - hopefully keeping the references private, and exposes new interfaces.

Inheritance: Re-use of behaviour and state

Derive a new class by ***extend***-ing an existing class

Case 1: Reuse all (non-private) methods of a class and **add** new behaviour/state. Existing class used “as is”

Case 2: **Modify** one or more methods of a class to change the functionality of the derived class. **Override** behaviour

Case 3: **Implement** one or more methods of the base class, where the base class defines only the method interface but not the implementation. Concrete implementation of an **abstract** method

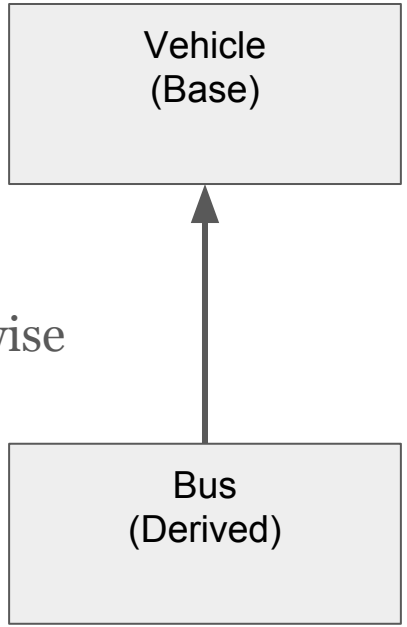
A given “**extends”** may involve one or more of these possibilities.

Java supports only “single inheritance”- can “*extend*” only one class

Constructors of Derived classes

```
class Vehicle {
      Vehicle(int id)  { … }
}

class Bus extends Vehicle {
       Bus(int id) {
             super(id); // compiler inserts super(); by default otherwise
       }
      // additional functionality of Bus
      …
}
```



Can invoke the constructor of the base class from that of the derived class.

Derived class constructor

- Can explicitly call a constructor of the parent class by using

  super(<params>); // depending on which constructor is provided/needed

- Must be the first statement of the derived class constructor
- If super(...) not explicitly called, then the compiler implicitly calls the no-parameter constructor

  super();

  as the first statement of the derived class constructor
- If the parent class does not have the appropriate constructor, then results in a compiler error.
- Thus, we get a chain of constructor calls, going up the hierarchy, with the body of the topmost getting executed first.

Everything is an **Object**

Base class of all classes in Java: **Object**

Thus, any object can be *upcast* to Object

Useful if we just want to manage references, but don't care about the type

Provides a useful set of methods (that can be **overridden**). Important ones:

```
String toString();                  // Default generates a string with the package/class
name and hashCode. Invoked when an object needs to be "cast" to String

boolean equals(Object obj);         //Default implementation: == (or "shallow equals")

int hashCode();                     // should be a unique and consistent value for objects
                                    // that satisfy "equals". Needed if HashMap used, e.g.
```

Overriding methods

```
class Vehicle {
        Vehicle(int id)  { vid = id; }
        public String toString() { return (“Vehicle” + vid); }
        private int vid;
}

class Bus extends Vehicle {
        Bus(int id) {
                super(id); // compiler inserts super(); by default otherwise
        }
        public String toString() { return (“Bus” + super.toString());}
        // … Other methods of Bus
}
```

Overriding methods

```
class Vehicle {
        Vehicle(int id)  { vid = id; }
        @Override
        public String toString() { return (“Vehicle ” + vid); }
        private int vid;
}
class Bus extends Vehicle {
        Bus(int id) {
                super(id); // compiler inserts super(); by defa
        }
        @Override
        public String toString() { return (“Bus: ” + super.to
        // ... Other methods of Bus
}
```

An Annotation:
Informing the compiler that this is an override of a base class method (in this case, of Object). Allows the compiler to check if the method signature matches one in a base class. (if not, flags a compile time error)

Can now use:

```
Bus bus = new Bus(27);
…
System.out.println (bus);
```

Produces:
Bus: Vehicle 27

Dynamic or late-binding

When a method is overridden in a derived class, the method corresponding to the derived class that was instantiated (using new) is the version of the method that will be invoked

```
Bus bus = new Bus(17);
Vehicle v = bus;
System.out.println(v);
// prints out “Bus: Vehicle 17”

v = new Vehicle(27);
System.out.println(v);
// prints out “Vehicle 27”
```

Overriding static methods

Static methods are always resolved with static or early binding. Hence the type of the reference determines whether the base or derived class method is called.

```
public class Animal {
    public static void testClassMethod() {
        System.out.println("The static method
in Animal");
    }
 }

public class Cat extends Animal {
    public static void testClassMethod() {
        System.out.println("The static method
in Cat");
    }

    public static void main(String[] args) {
        Cat myCat = new Cat();
        Animal myAnimal = myCat;
        Animal.testClassMethod();
        myAnimal.testClassMethod();
    }
}
```

Both these call the static method in Animal

Polymorphism

The ability of an object to have “many forms”: object of a certain type can be manipulated as that of a parent/ancestor type.

Thus, we can add a Bus or Car object to a list of Vehicle objects and then manipulate that list as if it were a list of Vehicles, without worrying about the sub-type

Thanks to dynamic binding, if any method of Vehicle has been overridden in the derived classes, then that derived method would be invoked

We can explicitly move up and down the derivation hierarchy of an object using *casts*

Upcasting and downcasting

Casting a reference to a parent class - **upcasting** - is always safe. And is implicit when a reference is assigned to a parent class reference (or its ancestors). Or a reference is passed as a parameter of a method that expects an ancestor class reference

Downcasting: Need to explicitly cast when casting a reference to that of a derived type (or its descendants). Type is checked at run time to ensure the type of the reference is compatible with that mentioned in the cast. *All other casts are illegal*

```
Bus b1 = new Bus(11);
Vehicle v1 = b1;      // Upcast - implicit cast and works fine
Bus b2 = (Bus) v1;    // Downcast - will be checked at run time. This one is fine
Car c1 = (Car) v1;    // Downcast (assuming Car is derived from Vehicle) -
                      //  will fail at run-time
```

Access specifiers and visibility

Normal rules of access of data members and methods of a parent class applicable for derived classes too.

public/private/package-default specifiers work the same

private data members not accessible from the derived class

private methods cannot be overridden - since they are not visible to the derived class.

	package p1			package p2	
Specifier (in class A)	class A	class B	class D extends A	class C	class E extends A
public	Y	Y	Y	Y	Y
<package-private>	Y	Y	Y	N	N
private	Y	N	N	N	N

protected class members

To provide further granularity in access specifiers, we have an additional qualifier: **protected**

Members of class A marked **protected:** visible to derived classes of A (or their descendants) but not to other classes that are not descendants of A

	package p1			package p2	
Specifier (in class A)	class A	class B	class D extends A	class C	class E extends A
public	Y	Y	Y	Y	Y
protected	Y	Y (!!)	Y	N	Y
<package-private>	Y	Y	Y	N	N
private	Y	N	N	N	N

Preventing extension

A method marked **final** cannot be overridden

Sometimes needed to ensure that a derived class does not change some critical behaviour

A class marked **final** cannot be extended

Used when you have no intention to allow the class to be extended. E.g. for security reasons

When a method is marked final is that compiler optimizations are possible - since the method cannot be extended, it can do a static binding.

Note that private methods can also be statically bound

Unimplemented methods

Sometimes, we would like to specify the signature of a method, but not its implementation. In effect, define the behaviour (methods) of a class without necessarily implementing one or more methods

Option a: have an empty implementation

```
class Shape {
    float getArea() { return 0.0; }   // what is there is no good default value to return?
}
```

Option b: don’t provide implementation - mark as **abstract**

```
abstract class Shape {
    abstract float getArea();      // implementation left to a derived class
}
```

abstract methods and classes

Method with no meaningful implementation can be marked **abstract** and its implementation can be omitted

Such classes must be marked as **abstract**

abstract classes cannot be instantiated (i.e. cannot invoke a “new”)

abstract method is expected to be overridden and implemented in some descendant class - a concrete class

A derived class need not implement all abstract methods of its parent class - in which case it will also be abstract, and cannot be instantiated

```
abstract class GraphicObject {
    private int x, y;
    ...
    void moveTo(int newX, int newY)
    {
        ...
    }
    abstract void draw();
    abstract void resize();
}

GraphicsObject gObj =
        new GraphicsObject();   // Error
```

```
class Circle extends GraphicObject {
    void draw() {
        …
    }
    void resize() {
        ...
    }
}


Circle c1 = new Circle();  // OK
```

Abstract classes

Abstract methods provide a way of defining behaviour or interface, and defer implementation to a derived class. This allows us to focus on the “API” of classes at design time, and detail out the implementation laterNote: an abstract class can have abstract and well as “concrete” methods, as well as state variables/data members.

Can we have an abstract private method?

What if an abstract class has all its methods marked abstract? This would Is this correct? Is this useful?

Interface

An abstract class without any concrete methods or data members and with all its abstract methods public, can be defined as an *interface.* An **interface** is also a type - effectively a base class

Defines a set of interfaces - implementation left to derived classes

A class extends an interface by *implement*ing its methods:

```
interface iface {
...// one or more methods - by default they are public abstract
}
class A implements iface {
  … implement one or more methods of iface
}
```

Interfaces

Interfaces help enable a limited form of multiple inheritance

A class can **extend** only one class, but can **implement** any number of interfaces

class B extends A implements C, D, E {

}
Where A is a class and C/D/E are interfaces

If a class implementing an interface **does not implement all** the methods of an interface, then it **must** be marked abstract

You can create a new interface by **extend**ing one or more existing interfaces

Interfaces

An interface can be public or package-private

All abstract methods of an interface are **abstract public** by default. Hence the specifiers can be omitted

An interface can also define constants. These are always **public static final** and hence need not be specified.

[Note: Java 9 introduces *default* implementations and *private* methods in interfaces. We will ignore those for now… though you are welcome to read up on them and use them as appropriate. But **not** backward compatible.]

Abstract classes vs interfaces

Define an **abstract class** if:

- You have some methods implemented - want that re-used, typically by closely related classes
- Need to maintain data/state - non-static, non-final data members
- Need to have different levels of access - protected, private etc

Define as **interface** if:

- Want to have behaviour shared by potentially unrelated classes
- Focus on the behaviour, and not worry about the implementation
- Allow multiple inheritance

You could define an abstraction as an interface first, and then change it to an abstract class if needed.

Interface examples

Could our earlier examples such as Vehicle, GraphicsObject etc be better defined as interfaces?

We can define a class hierarchy of Shape and its derived classes, as pure geometric objects - without worrying about display.

We can define an interface Drawable with methods such as Draw, which add “drawability” to classes if they implement this interface. E.g.

class Circle extends Shape { ...}

class DrawableCircle extends Shape implements Drawable { … }

Class DrawableCircle extends Circle implements Drawable { … }

interface examples

We can have different kinds of File objects such as TextFiles, ImageFiles, AudioFiles, BinaryFiles, VideoFiles etc., all of which extend the class File and have methods such as open, close, copy etc.

Some of these types can be printed, so they can abstract that aspect with an interface Printable that support methods related to printing - preview, print, convertToGrayscale, etc. This makes sense for text and image files, but not for audio/video files, for example. These can be “played” instead. So, we may implement these as

class TextFile extends File implements Printable { … }

class AudioFile extends File implements Playable { … }