Programming in Java

Preparatory Term, 2019
Lecture 2

T K Srikanth
IIIT-B

Hiding the implementation

"separating the things that change from the things that stay the same." - Bruce Eckel, Thinking in Java

The public methods that are used to create/access/modify objects should largely remain the same, although the implementation (including any other classes internal to this implementation) will likely change over time.

Access specifiers are one way to enable this separation.

Access specifiers

classes, methods and data fields can have specifiers added to them that control who all (which other classes) can access them

public, private (and later, **protected**, as well as a *default* (i.e. no specifier)) access

public: visible to all other classes

private: only visible from within (methods or class definition) of the same class

Does a *private class* make sense?

Public/private members

Visibility of members (methods or data) of a class A from within class B:

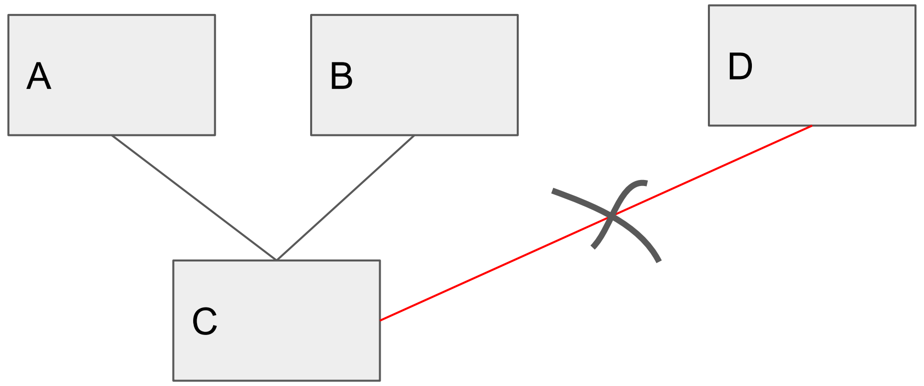
Specifier (in class A)	For class A	For class B
public	Y	Y
private	Y	N

Other controls of visibility

Public/private specifiers provide granular access to a class or its methods.

Need a mechanism to provide “partial” visibility

- A class (or a subset of its methods) visible to one set of classes but not to others



E.g. An algorithm that is needed by 2 or more developers of one team, but don’t want to expose it to others.

Packages

A mechanism for grouping classes together - typically a logical set of modules or library

Provides a *namespace*

- Can have the same class name in different packages
- Resolved by using <packageName>.<className>

Also provides another level of control over access

Package

Define a class (or rather the classes in a file) as belonging to a package p1 by adding the line

```
package p1;
```

as the first line of the file

By adding this in multiple files, all the classes in these files all now belong to the same package p1. (A file - i.e. the class in it - can be part of only one package)

Other packages import this using

```
import <package>.* ;
```

Not specifying a package name for a file implies it is part of the *default* package

Package

Packages can be nested into a hierarchy. So, we can have

```
package graphics.shapes;
```

Or

```
package a.b.c.d;
```

Source files are organized into directories that reflect this structure.

To avoid ambiguity, need a unique prefix for the package path, so that there is no chance of conflict. *What's a good scheme for this?*

You can import specific parts of packages with

```
import package.subpack.*
```

Access specifiers

classes, methods and data fields can have specifiers added to them that control who all (which other classes) can access them

public, private (and later, **protected**, as well as a *default* (i.e. no specifier)) access

public: visible to all other classes

private: only visible from within (methods or class definition) of the same class

Does a *private class* make sense?

Package access specifiers

Package access is the default for class/method/data fields etc.

That is, unless specifically specified as public or private (or protected), access is provided to all other classes of that package - and to no other. Also called *package-private*

```
package aOne;
public class A {
   public A() {}            // visible to all classes
   f1() {}                  // visible to all classes of this package aOne
   private f2() {}          // visible only from within this class
}
```

Public/private members

Visibility of members (methods or data) of a class A from within class B:

Specifier (in class A)	For class A	For class B
public	Y	Y
private	Y	N

Public/private and package(default) access

Visibility of members (methods or data) of a class A from within class B (both in package aOne) or class C (in package cOne):

	package aOne		package cOne
Specifier (in class A)	For class A	For class B	For class C
public	Y	Y	Y
<package-private>	Y	Y	N
private	Y	N	N

Same applies to classes in one package as seen by classes of the same or other package.

Some common packages in Java

java.io	Provides for system input and output through data streams, serialization and the file system.
java.lang	Provides classes that are fundamental to the design of the Java programming language.
java.math	Provides classes for performing arbitrary-precision integer arithmetic (BigInteger) and arbitrary-precision decimal arithmetic (BigDecimal).
java.net	Provides the classes for implementing networking applications.
java.sql	Provides the API for accessing and processing data stored in a data source (usually a relational database) using the Java^{TM}programming language.
java.text	Provides classes and interfaces for handling text, dates, numbers, and messages in a manner independent of natural languages.
java.util	Contains the collections framework, legacy collection classes, event model, date and time facilities, internationalization, and miscellaneous utility classes (a string tokenizer, a random-number generator, and a bit array).

https://docs.oracle.com/javase/7/docs/api/overview-summary.html

Constructors

Every class should have at least one constructor

Constructor that takes no arguments is called the *default constructor*

Compiler defines a default constructor if no constructors have been defined for the class. This default version sets all data members to their default initial value.

All instance variables are initialized to default values, if not explicitly initialized in a constructor

If any constructor is defined for the class, then the constructor does not define the default constructor

Note: no notion of "*destructors*". Facility for cleanup

Constructors

Can we have constructors that are not public? Why would we need them.

- Package-private constructors?
- Private constructors?

static methods and data members

A method specified as **static** is considered as a method defined on the class, and not tied to any instance.

Can be invoked even without creating instances of that class

static methods cannot access data (or methods) that are non-static - even of the same class

Similarly, static data members are shared by all instances of that class - there is only piece of storage associated with that data member.

Static methods/members are accessed with

classname.member

static methods

```
class Account {
  …
  static Account max(Account[]
acs) {}

   private String name;
   private float balance;
}
```

Usage:

```
Account[] accounts = new
Account[10];

// …. Iniitalize array

Account largest =

  Account.max(accounts);
```

static data fields

```
class Account {
  …
   Account() {
    accountNumber = nextId++;
  }
  static Account max(Account[] acs) {}
  ...
   private static int nextId = 1;
   private String name;
   private float balance;
   private int accountNumber;
}
```

Scope and lifetime (of primitives, references, objects)

The scope of primitives and references defined by the block where they are declared. And so is their lifetime (memory no longer available once out of scope)

```
{
    int x = 7;
    Box b1 = new Box();
    ...
}
```

Objects live on independent of the block where they were constructed. Accessible as long as some reference to them is still alive.

Hence, objects with **static** references are not garbage-collected

final variables

A variable labelled as **final** implies value cannot be changed once it is initialized

final variables must be initialized

- when declared
- or, in every constructor (also called “blank final”)
- Compiler error if attempt to re-assign a value

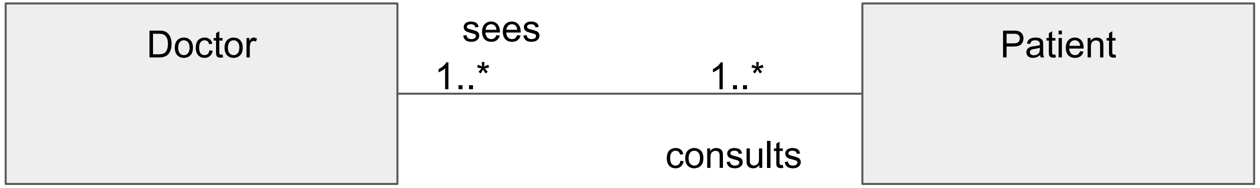
Note: if a reference to an object is final, the reference itself cannot change, however, the object it refers to can change.

For example, in the IceCreamBar example, the name and flavour should be defined final, if the intent is that these cannot change once the object is initialized.

Relationships between objects

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

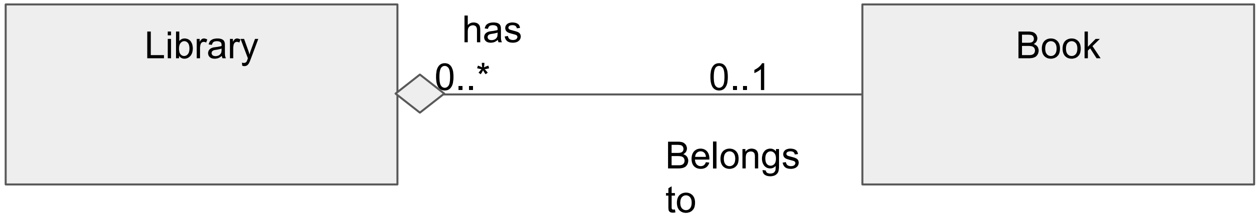
- A path of communication or link between objects, where one object “uses” possibly multiple instances of the other. No ownership or other semantics are implied
- The lifecycles of the objects are independently defined



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

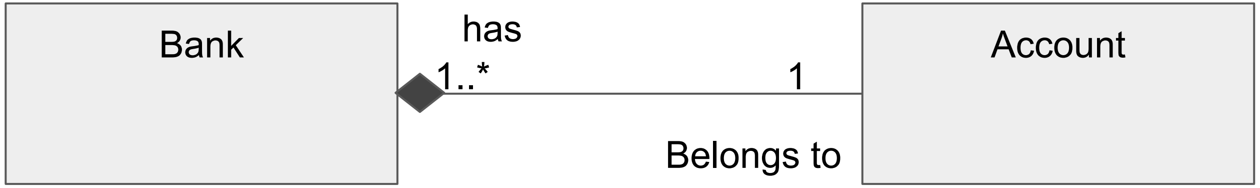


Relationships: Composition

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

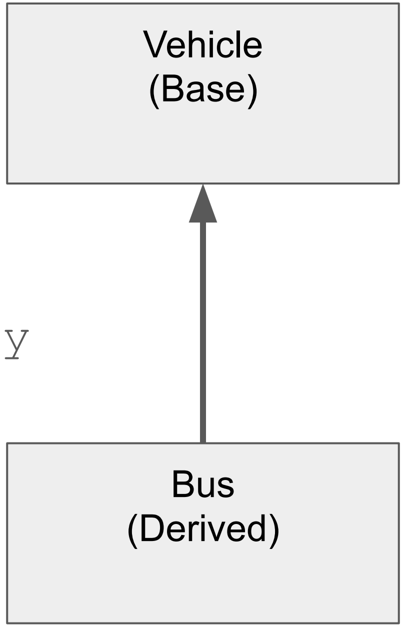
A given "**extends**" may involve one or more of these possibilities.

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);
    }
    @Override
    public String toString() { return ("Bus: " + super.toString());}
    // ... 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

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

Some rules (best practices)

1. Principle of least privilege:
   a. All data members are private
   b. Use <package-default> for classes unless they need to be public
   c. Use private/<package-default> for methods unless they are public interfaces
2. No explicit casts*
3. No compiler warnings
4. Use **final** where possible

* except is very specific situations - to be discussed later

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*

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”
```

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
```