

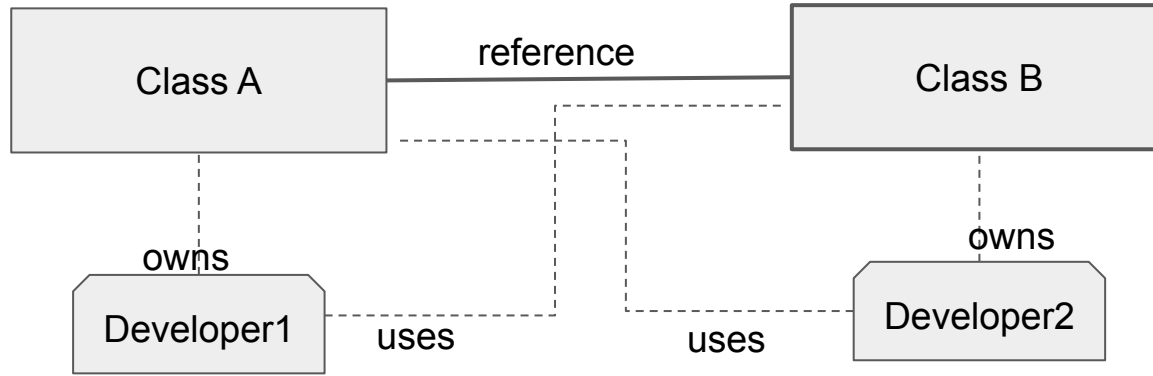
# ESS 201: Programming in Java

Encapsulation and Modularity

Term 1: 2020-21

T K Srikanth

# Useful design perspective



Note: Developer1 and Developer2 can be the same person.

**Owner** of a class implements the methods/data members of the class, and should ensure that *any object of that class is always in a correct and consistent state* - starting with the constructors.

The **User** of a class can access that class only through the methods (and data members) that are *accessible* to this user - as decided by the Owner of that class. That is, the methods of class B can manipulate objects of class A only through the methods (and data members) accessible to this class.

# 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. They provide flexibility to the author of the class, insulate the user from changes in the implementation, and can help improve robustness of the implementation.

# 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, *package-private*, *default-package* (i.e. no specifier) and **protected**) 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?

# Class definition

```
class IceCreamBar {  
    IceCreamBar(String iname, String iflavour, float itemp, float iprice)  
    {  
        name = iname; flavour = iflavour;  
        temp = itemp;  
        price = iprice;  
    }  
    String getName()      { return name; }  
    String getFlavour()   { return flavour; }  
    float getTemp()       { return temp; }  
    float getPrice()      { return price; }  
    void setTemp(float t) { temp = t; }  
    void setPrice(float p) { price = p; }  
  
    private String name, flavour;  
    private float temp, price;  
}
```



Cannot be accessed or used in  
a method of a different 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

# Invoking class methods - restricted access

What is the outcome of the following statement for the current example?

```
System.out.println("Flavour " + ic1.flavour +  
                    " from " + ic1.name +  
                    "costs " + ic1.price +  
                    " and is stored at " +  
                    ic1.temp + " degrees.");
```

Assume this is a statement in a method

- of a different class (say, IceCreamStore)
- of the same class IceCreamBar

# 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 directly 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 one 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];  
  
// ... Initialize 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;  
}
```

So how does garbage collection work?

# 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.

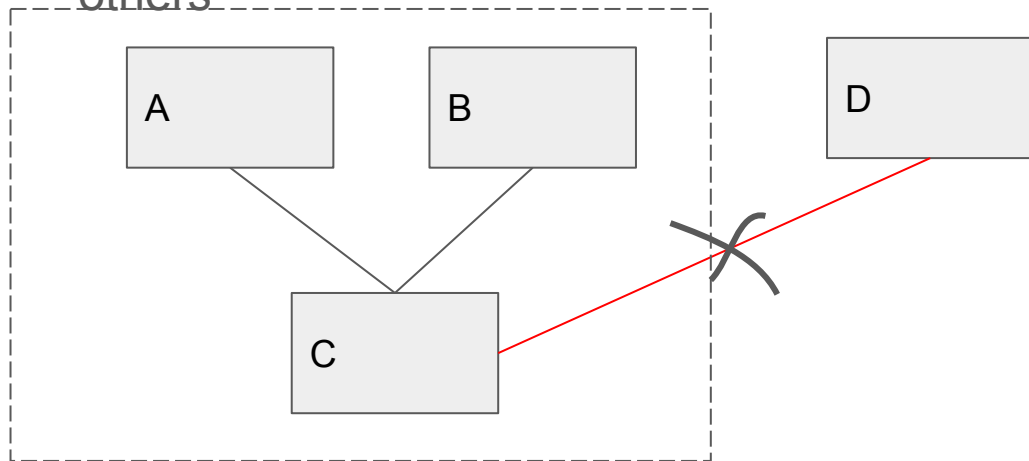
For example, in the IceCreamBar example, the name and flavour should be defined final, if the intent is that these cannot change once the object is initialized.

# 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.*
```



# 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
    int f1() {}             // visible to all classes of this package aOne
    private void f2() {}    // visible only from within this class
}
```

# 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.

# Constructors

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

- Package-private constructors?
- Private constructors?

# 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<sup>TM</sup> 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).

# Some rules (best practices)

1. When defining access specifiers, use *principle of least privilege*
  - a. All data members are private
  - b. classes and methods should have the most restrictive access possible
2. No casts\*
3. No compiler warnings

\* except in very specific situations - to be discussed later