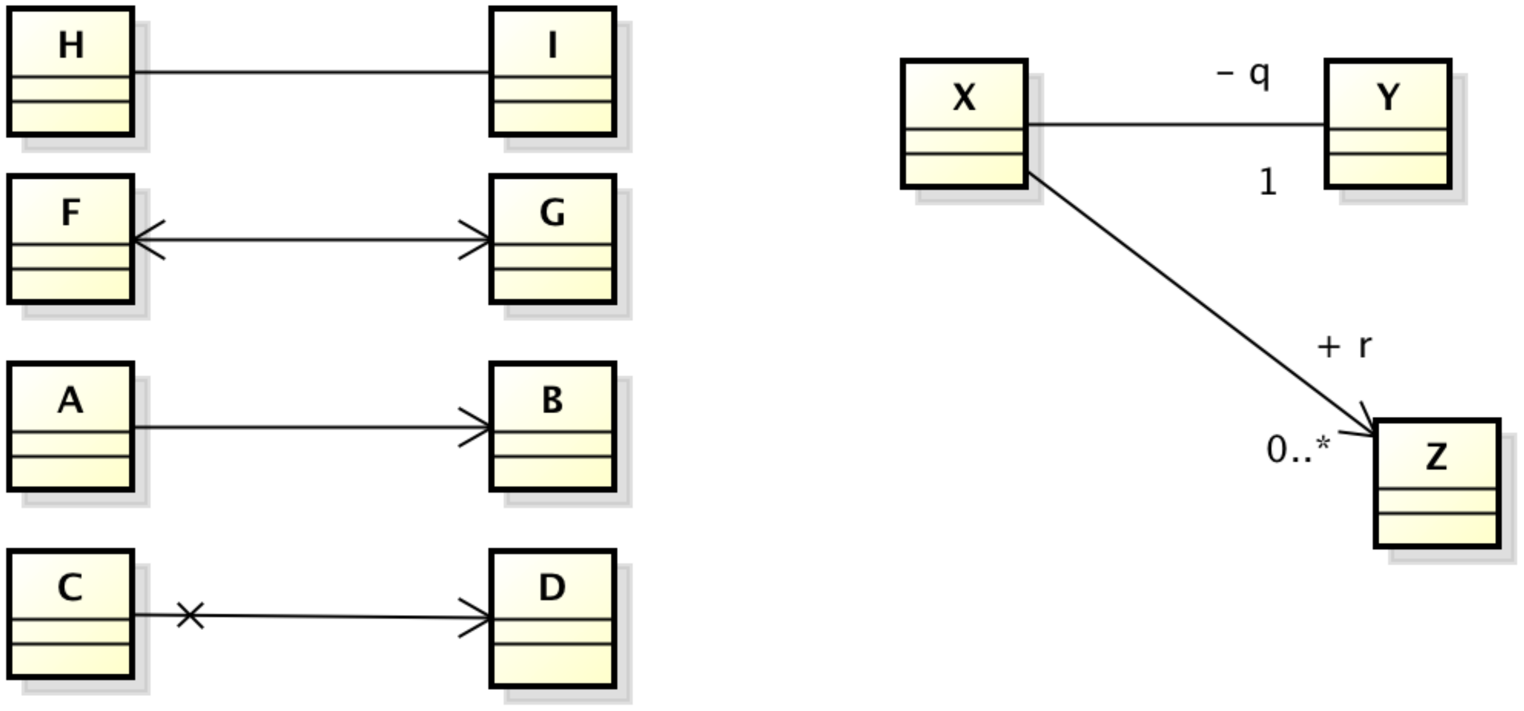
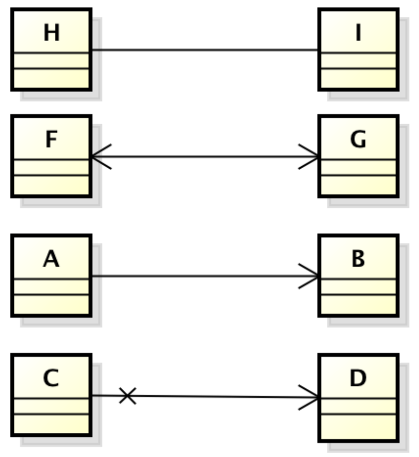
CMPE 202

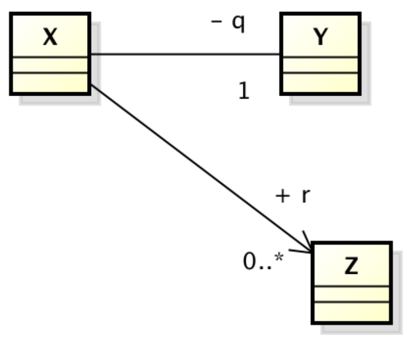
Translating UML into Java, C++ and C#

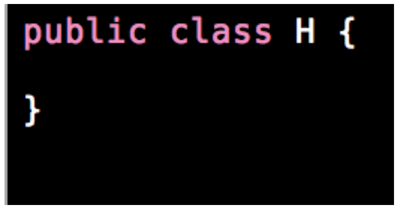
(How about JavaScript?)



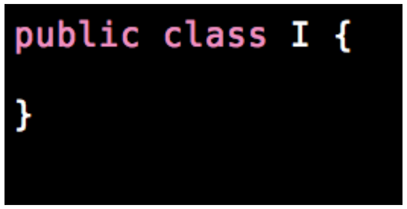
What are the Class Definitions for these Classes?



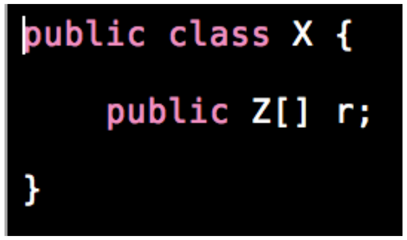




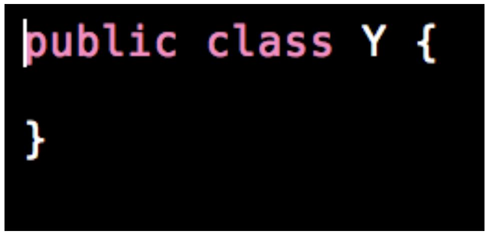
```
public class H {

}
```



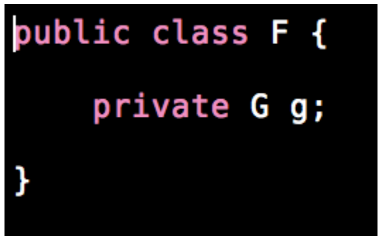
```
public class I {

}
```



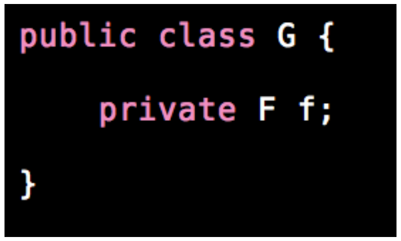
```
public class X {

    public Z[] r;

}
```



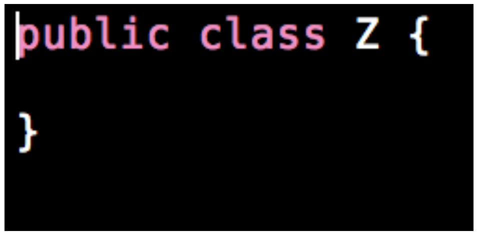
```
public class Y {

}
```



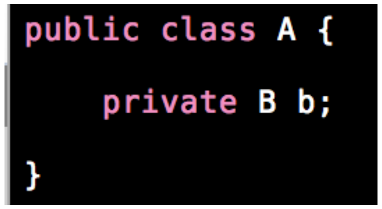
```
public class F {

    private G g;

}
```



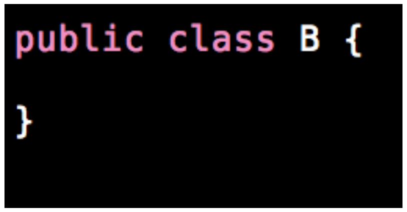
```
public class G {

    private F f;

}
```



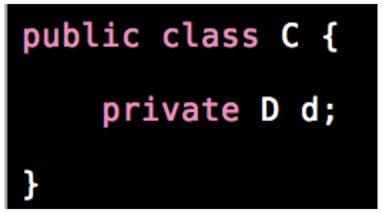
```
public class Z {

}
```



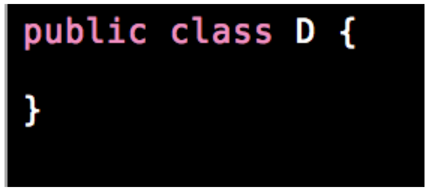
```
public class A {

    private B b;

}
```



```
public class B {

}
```

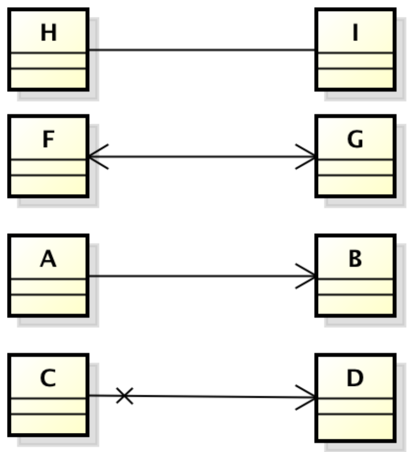


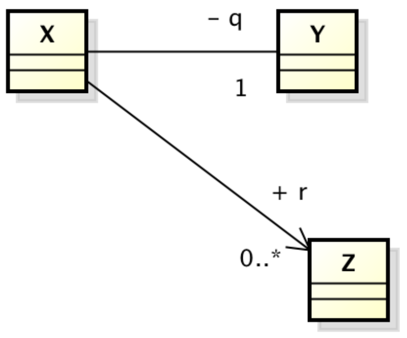
```
public class C {

    private D d;

}
```

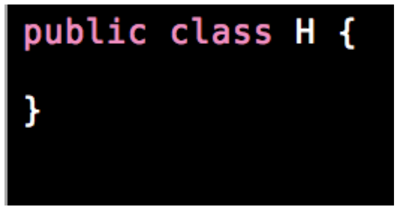


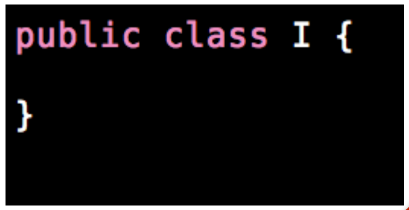
```
public class D {

}
```

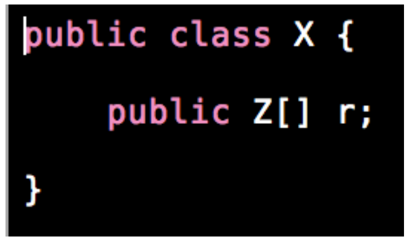
Which ones are incorrect?

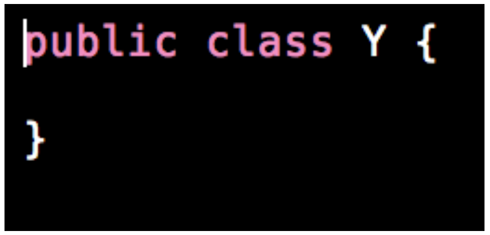


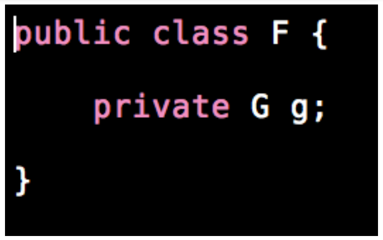


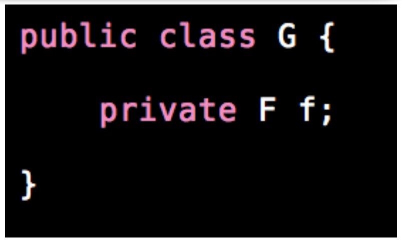

```
public class H {
}
```

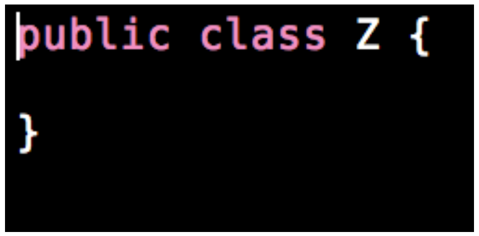

```
public class I {
}
```

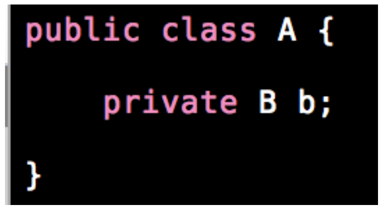

```
public class X {
    public Z[] r;
}
```

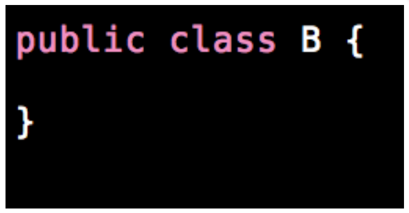

```
public class Y {
}
```

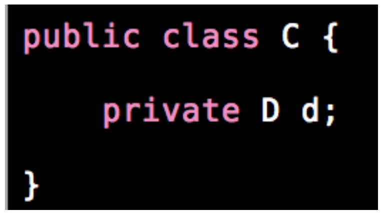

```
public class F {
    private G g;
}
```

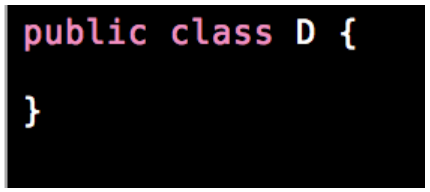

```
public class G {
    private F f;
}
```


```
public class Z {
}
```

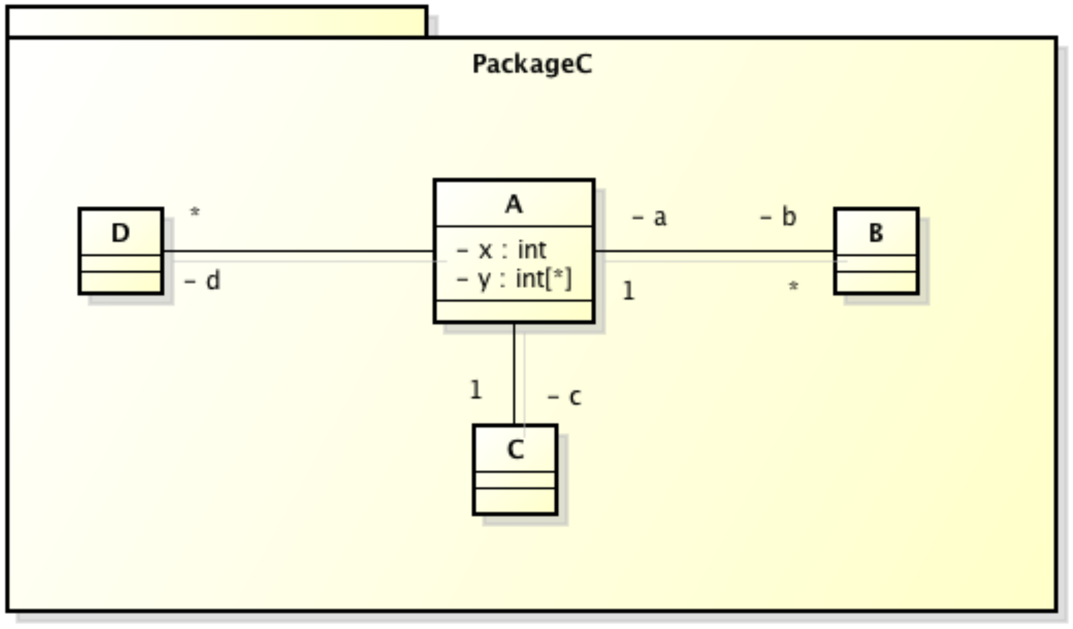

```
public class A {
    private B b;
}
```


```
public class B {
}
```

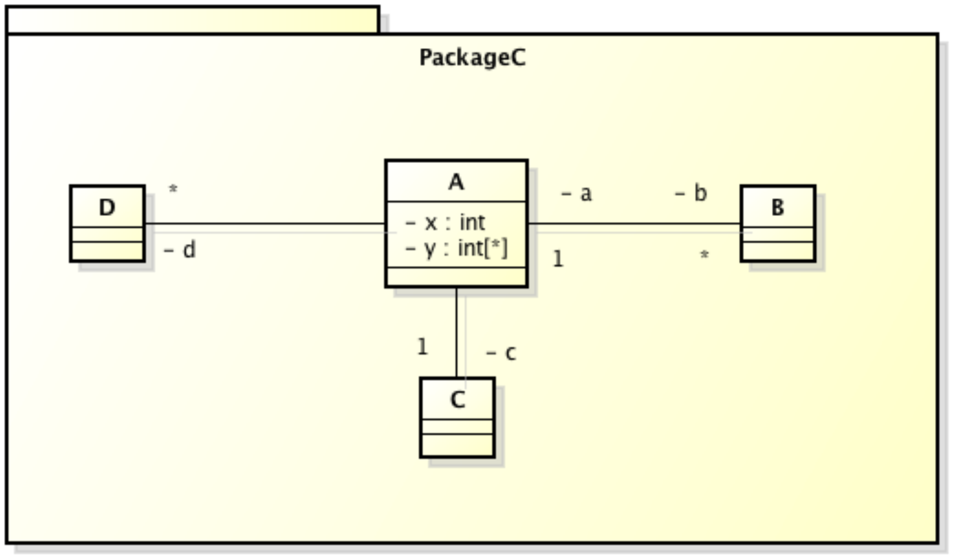

```
public class C {
    private D d;
}
```


```
public class D {
}
```

Why are these incorrect?



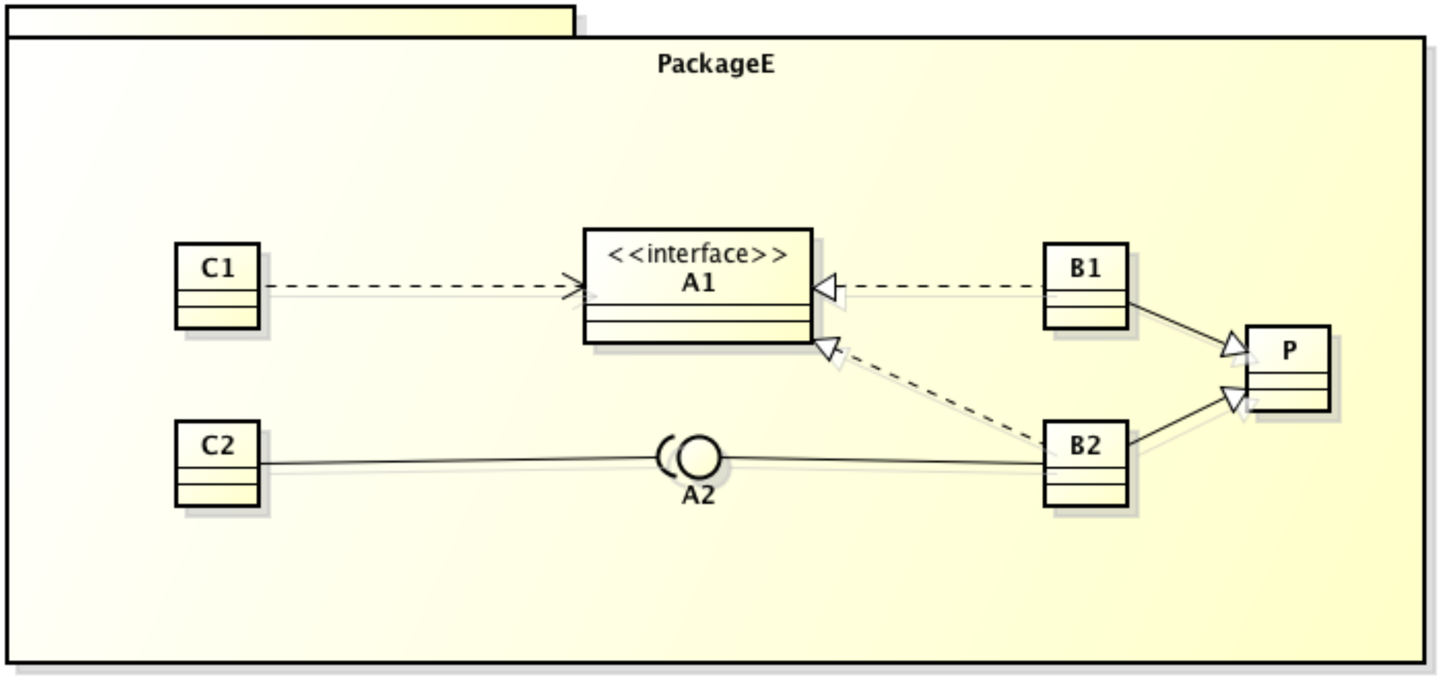
What is the Class Definition for Class "A" in C++, C# and Java?



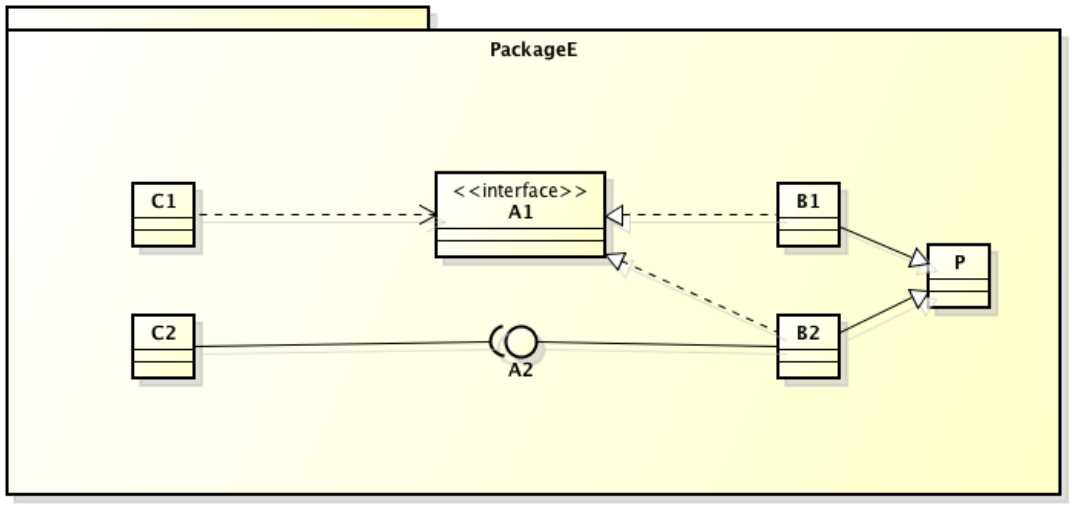
```
package PackageC;

import java.util.Collection;

// Java
public class A {

    private int x;

    private int[] y;

    private Collection<B> b;

    private C c;

    private Collection<D> d;

}
```

```
// C++
namespace PackageC
{
    class A
    {
    private:
        int x;
        int y[];
        B b[];
        C c;
        D d[];
    };
}  // namespace PackageC
```

```
namespace PackageC
{
    // C#
    public class A
    {
        private int x;

        private int[ ] y;

        private ICollection<B> b;

        private C c;

        private ICollection<D> d;
    }
}
```



What are the Class Definitions
for Classes "B2" and "C2"
in C++, C# and Java?



```
package PackageE;

// Java
public class B2 extends P implements A1, A2
{

}
```

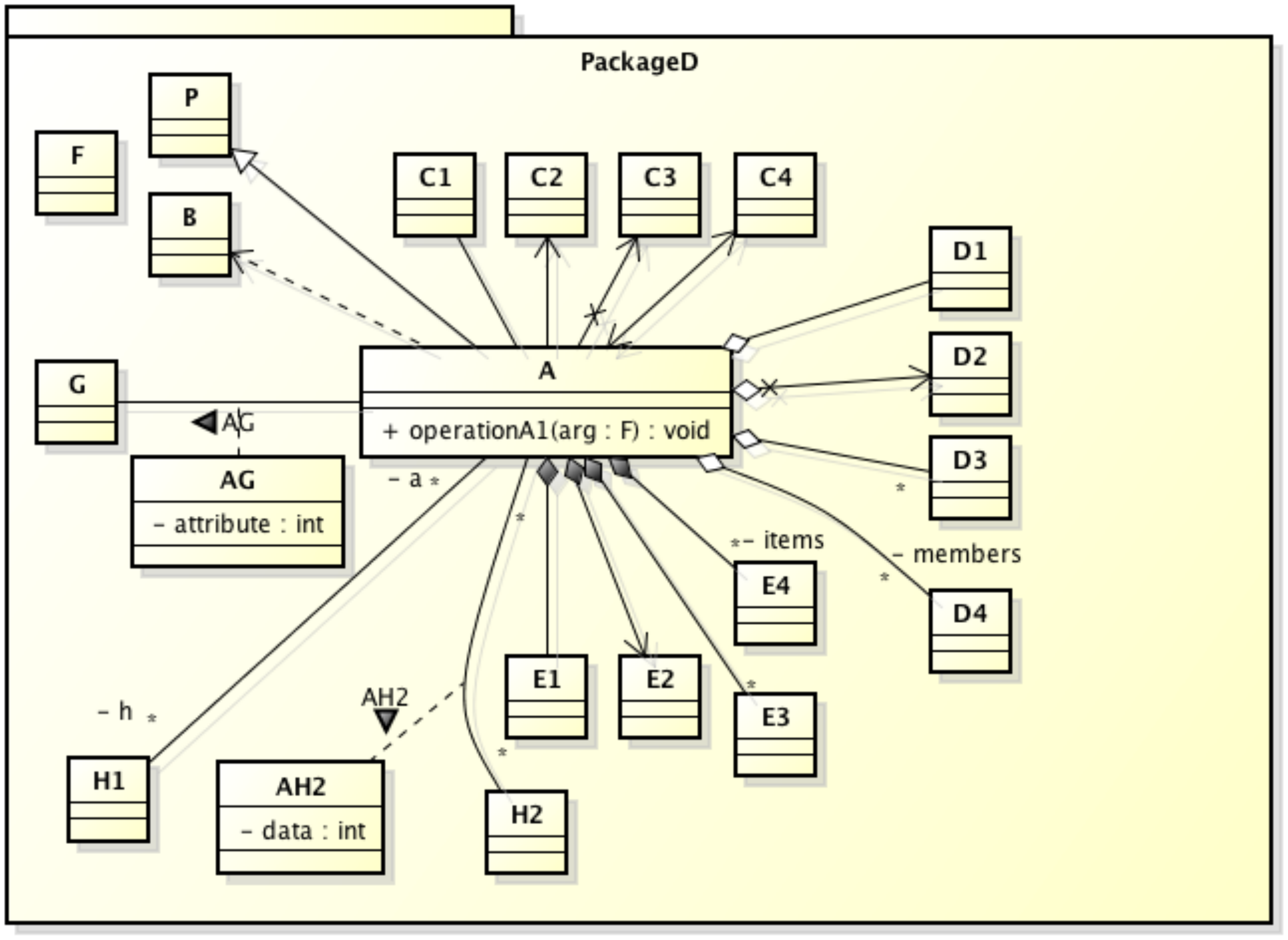
```
package PackageE;

// Java
public class C2
{
    // uses A2 - could be as
    // instance var, param, etc...
}
```

```
// C++
namespace PackageE
{
    class B2 : public P, public A1, public A2
    {
    };
}  // namespace PackageE
```

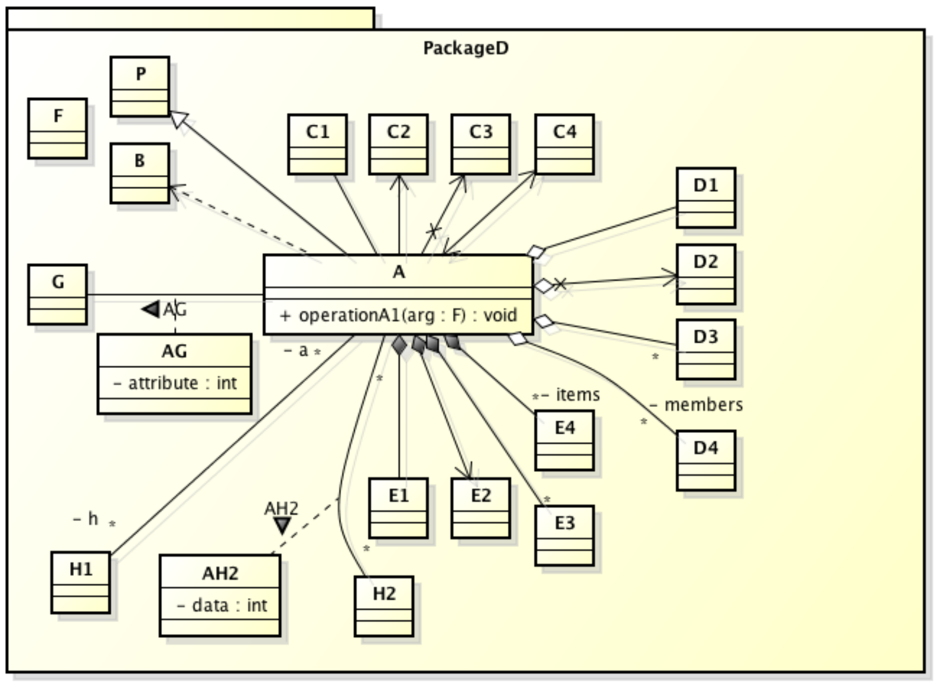
```
// C#
namespace PackageE
{
    public class B2 : P, A1, A2
    {
    }
}
```

```
// C++
namespace PackageE
{
    class C2
    {
        // uses A2 - could be as
        // instance var, param, etc...
    };
}  // namespace PackageE
```

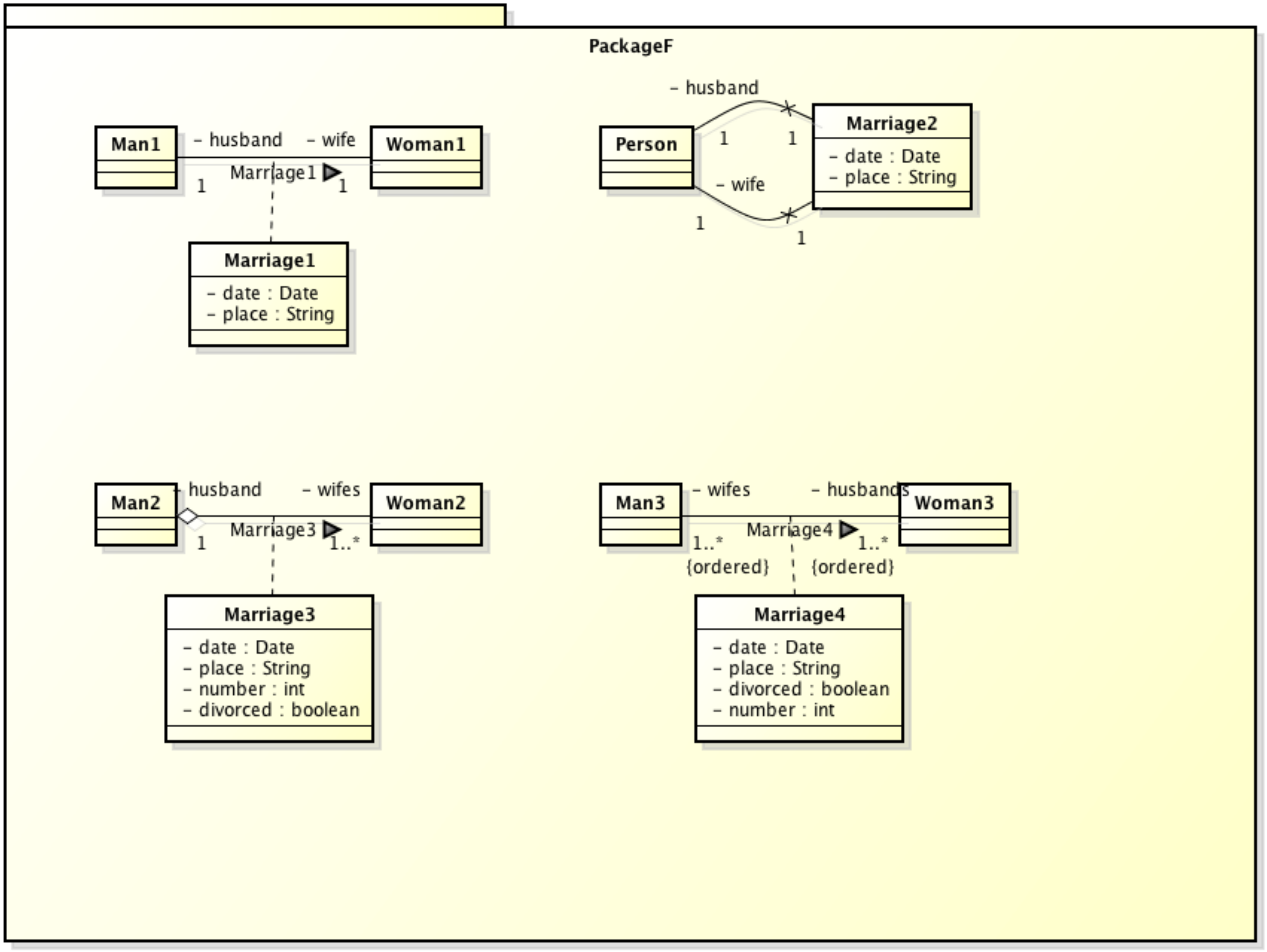
```
// C#
namespace PackageE
{
    public class C2
    {
        // uses A2 - could be as
        // instance var, param, etc...
    }

}
```



What is the Class Definition for Class "A" in Java?



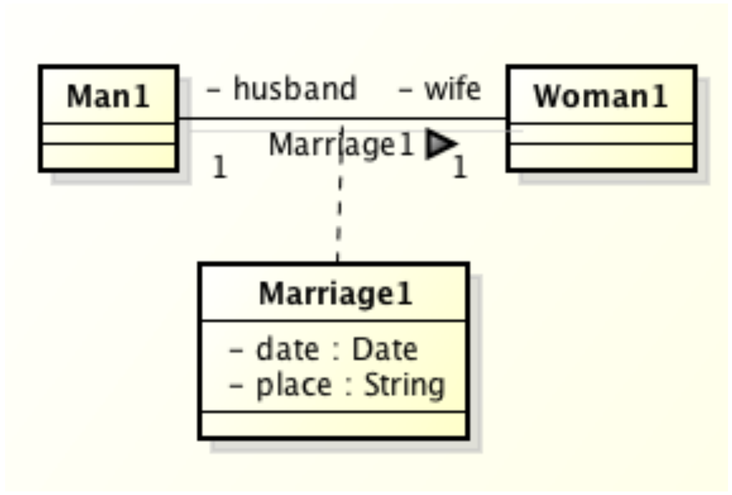
```
package PackageD;

import java.util.Collection;

public class A extends P {

    private C1 c1;

    private D1 d1;

    private E1 e1;

    private D2 d2;

    private E2 e2;

    private C2 c2;

    private C3 c3;

    private C4 c4;

    private Collection<E3> e3;

    private Collection<E4> items;

    private Collection<D3> d3;

    private Collection<D4> members;

    private AG aG;

    private Collection<H1> h;

    private Collection<H2> aH2;

    public void operationA1(F arg) {

    }

}
```



Write the Class Definitions (in Java) for all of the Classes above

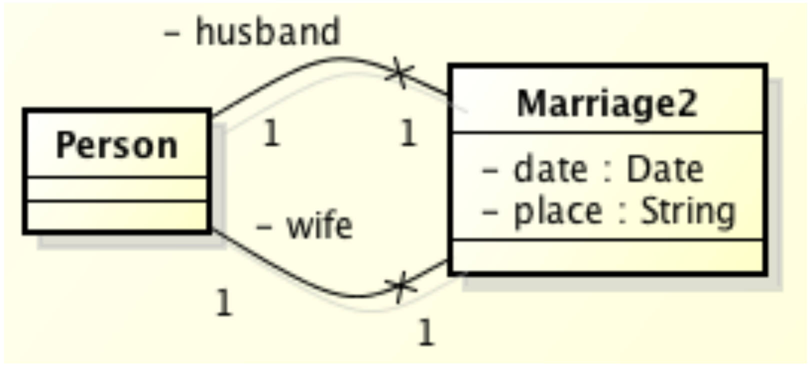
```
public class Man1 {

    private Marriage1 marriage1;

}
```

```
public class Woman1 {

    private Marriage1 marriage1;

}
```



```
public class Marriage1 {

    private Date date;

    private String place;

    private Woman1 wife;

    private Man1 husband;

    // accessor methods for attributes here...

}
```

note: one possible implementation.



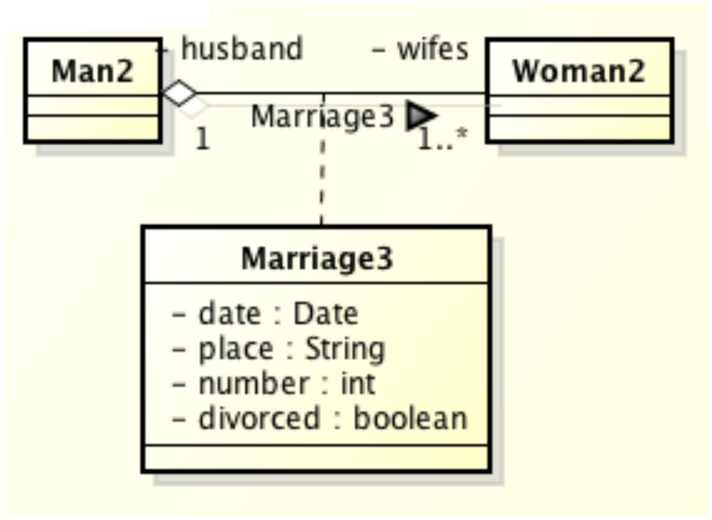
```
public class Person {

}
```

```
public class Marriage2 {

    private Date date;

    private String place;

    private Person wife;

    private Person husband;

    // accessor methods for attributes here...

}
```

<u>note</u>: one possible implementation.

```java
import java.util.Collection;

public class Man2 {

    private Collection<Marriage3> marriages;

}
```

```java
public class Woman2 {

    private Marriage3 certificate;

}
```

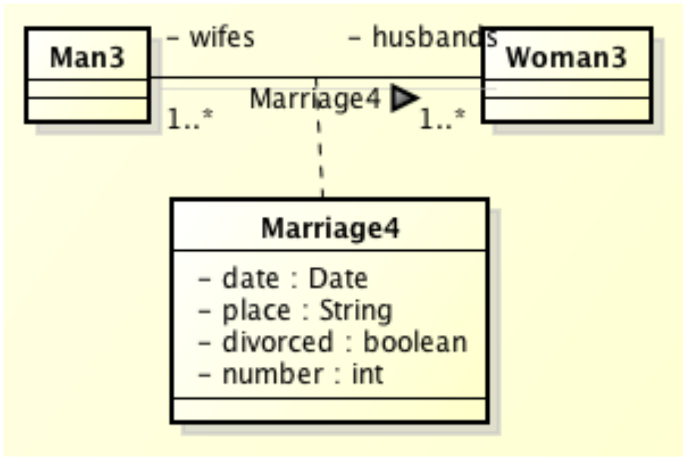


```java
import java.util.Date;

public class Marriage3 {

    private Date date;

    private String place;

    private int number;

    private boolean divorced;

    private Woman2 wife;

    private Man2 husband;

    // accessor methods for attributes here...

}
```

<u>note</u>: one possible implementation.

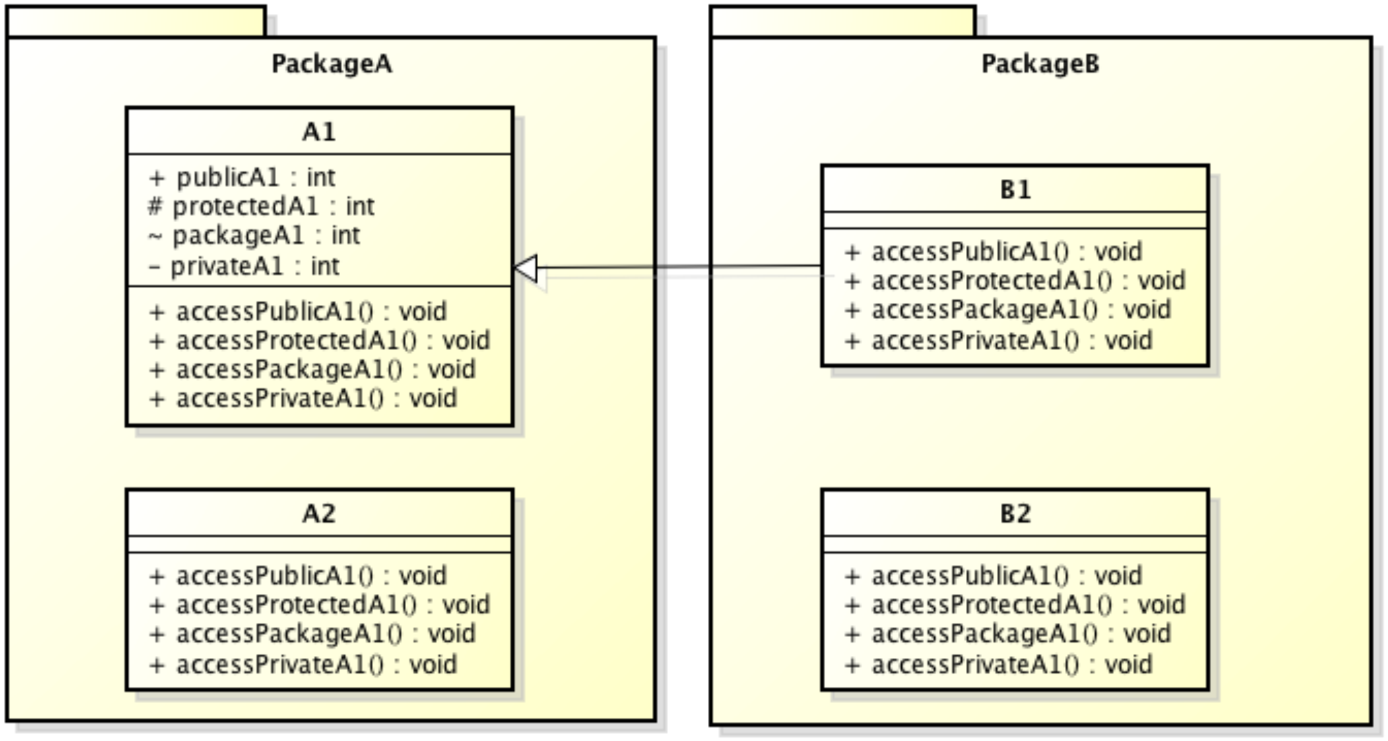
```
import java.util.List;

public class Man3 {

    private List<Woman3> wifes;
    private List<Marriage4> certs;

}
```

```
import java.util.List;

public class Woman3 {

    private List<Man3> husbands;
    private List<Marriage4> certs;

}
```

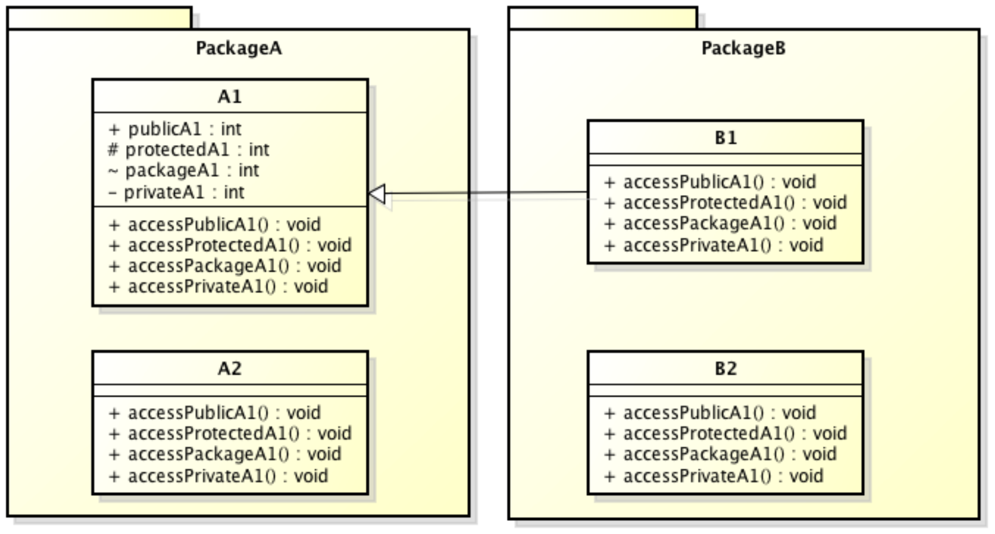


```
import java.util.Date;

public class Marriage4 {

    private Date date;

    private String place;

    private boolean divorced;

    private int number;

    // accessor methods for attributes here...

}
```

<u>note</u>: one possible implementation.

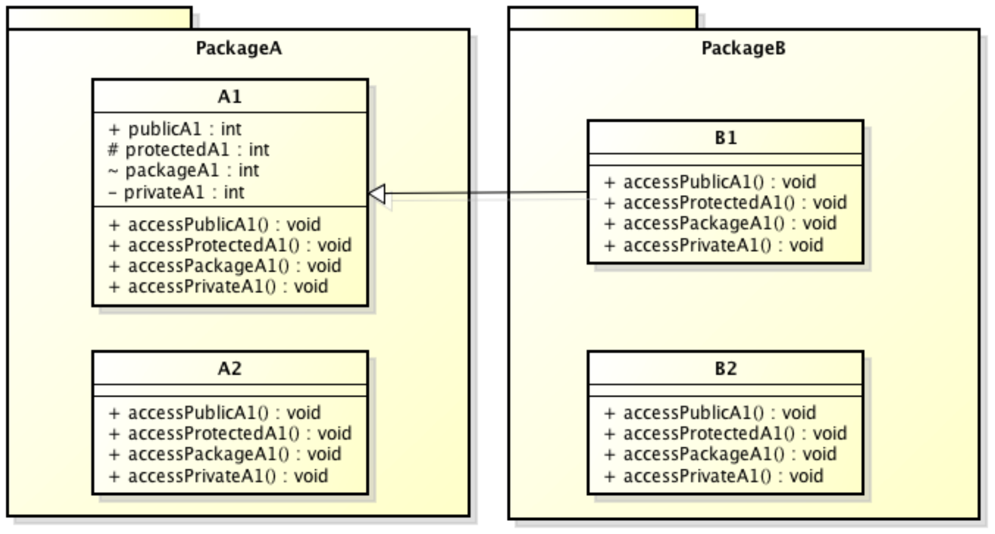

PackageA
A1
+ publicA1 : int
# protectedA1 : int
~ packageA1 : int
- privateA1 : int
+ accessPublicA1() : void
+ accessProtectedA1() : void
+ accessPackageA1() : void
+ accessPrivateA1() : void
A2
+ accessPublicA1() : void
+ accessProtectedA1() : void
+ accessPackageA1() : void
+ accessPrivateA1() : void
PackageB
B1
+ accessPublicA1() : void
+ accessProtectedA1() : void
+ accessPackageA1() : void
+ accessPrivateA1() : void
B2
+ accessPublicA1() : void
+ accessProtectedA1() : void
+ accessPackageA1() : void
+ accessPrivateA1() : void

Write a Java Test Program
to test the access modifiers
for Classes: “A1”, “A2”, “B1” and “B2”


PackageA
A1
+ publicA1 : int
# protectedA1 : int
~ packageA1 : int
- privateA1 : int
+ accessPublicA1() : void
+ accessProtectedA1() : void
+ accessPackageA1() : void
+ accessPrivateA1() : void
A2
+ accessPublicA1() : void
+ accessProtectedA1() : void
+ accessPackageA1() : void
+ accessPrivateA1() : void
PackageB
B1
+ accessPublicA1() : void
+ accessProtectedA1() : void
+ accessPackageA1() : void
+ accessPrivateA1() : void
B2
+ accessPublicA1() : void
+ accessProtectedA1() : void
+ accessPackageA1() : void
+ accessPrivateA1() : void

```java
package PackageA;

public class A1 {

    public int publicA1;
    protected int protectedA1 ;
    private int privateA1 ;
    int packageA1 ;

    public A1()
    {
        this.publicA1 = 444 ;
        this.protectedA1 = 333 ;
        this.privateA1 = 222 ;
        this.packageA1 = 111 ;
    }

    public void accessPublicA1() {
        System.out.println( publicA1 ) ;
    }

    public void accessProtectedA1() {
        System.out.println( protectedA1 ) ;
    }

    public void accessPackageA1() {
        System.out.println( packageA1 ) ;
    }

    public void accessPrivateA1() {
        System.out.println( privateA1 ) ;
    }

}
```

```java
package PackageA;

public class A2 {

    A1 a1 ;

    public A2()
    {
        a1 = new A1() ;
    }

    public void accessPublicA1() {
        System.out.println( a1.publicA1 ) ;
    }

    public void accessProtectedA1() {
        System.out.println( a1.protectedA1 ) ;
    }

    public void accessPackageA1() {
        System.out.println( a1.packageA1 ) ;
    }

    public void accessPrivateA1() {
        //System.out.println( a1.privateA1 ) ;
    }


}
```


PackageA
A1
+ publicA1 : int
# protectedA1 : int
~ packageA1 : int
- privateA1 : int
+ accessPublicA1() : void
+ accessProtectedA1() : void
+ accessPackageA1() : void
+ accessPrivateA1() : void
A2
+ accessPublicA1() : void
+ accessProtectedA1() : void
+ accessPackageA1() : void
+ accessPrivateA1() : void
PackageB
B1
+ accessPublicA1() : void
+ accessProtectedA1() : void
+ accessPackageA1() : void
+ accessPrivateA1() : void
B2
+ accessPublicA1() : void
+ accessProtectedA1() : void
+ accessPackageA1() : void
+ accessPrivateA1() : void

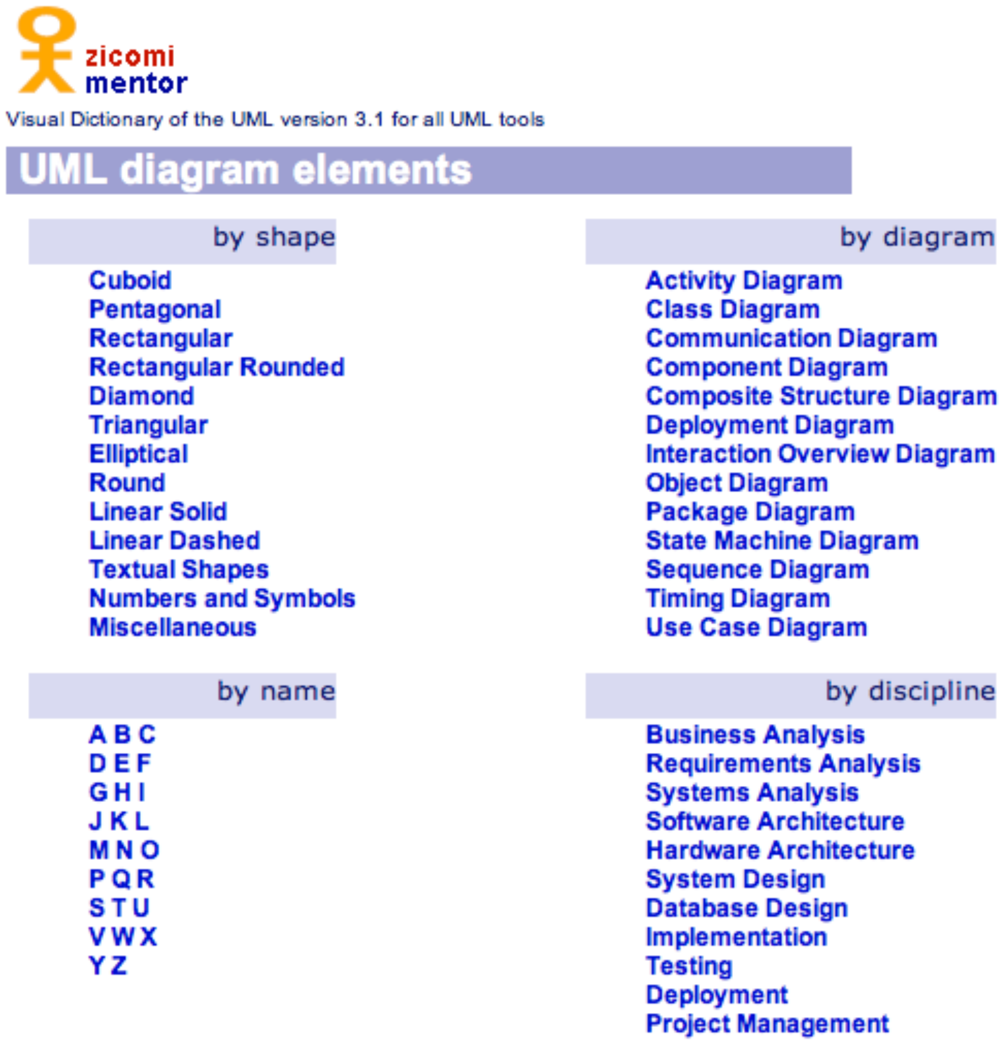
```java
package PackageB;

import PackageA.A1;

public class B1 extends A1 {

    A1 a1 ;

    public B1()
    {
        a1 = new A1() ;
    }

    public void accessPublicA1() {
        System.out.println( a1.publicA1 ) ;
    }

    public void accessProtectedA1() {
        //System.out.println( a1.protectedA1 ) ;
        System.out.println( this.protectedA1 ) ;
    }

    public void accessPackageA1() {
        //System.out.println( a1.packageA1 ) ;
    }

    public void accessPrivateA1() {
        //System.out.println( a1.privateA1 ) ;
    }

}
```

```java
package PackageB;

public class B2 {

    PackageA.A1 a1 ;

    public B2()
    {
        a1 = new PackageA.A1() ;
    }

    public void accessPublicA1() {
        System.out.println( a1.publicA1 ) ;
    }

    public void accessProtectedA1() {
        //System.out.println( a1.protectedA1 ) ;
    }

    public void accessPackageA1() {
        //System.out.println( a1.packageA1 ) ;
    }

    public void accessPrivateA1() {
        //System.out.println( a1.privateA1 ) ;
    }

}
```

Reference

UML Class Diagram Notation, UML Visibilities
& Access Modifiers in Programming Languages

http://www.zicomi.com/viewDictionaryHome.jsp



zicomi mentor

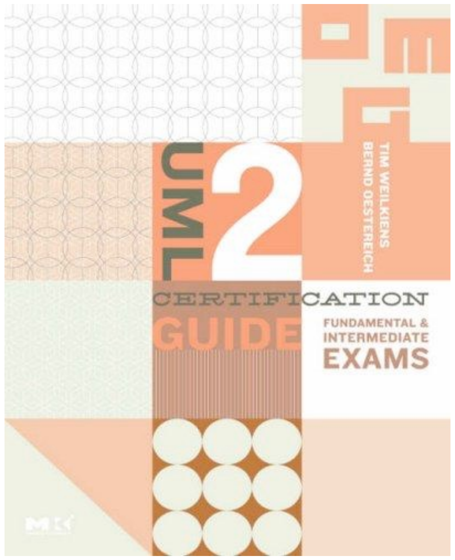
Visual Dictionary of the UML version 3.1 for all UML tools

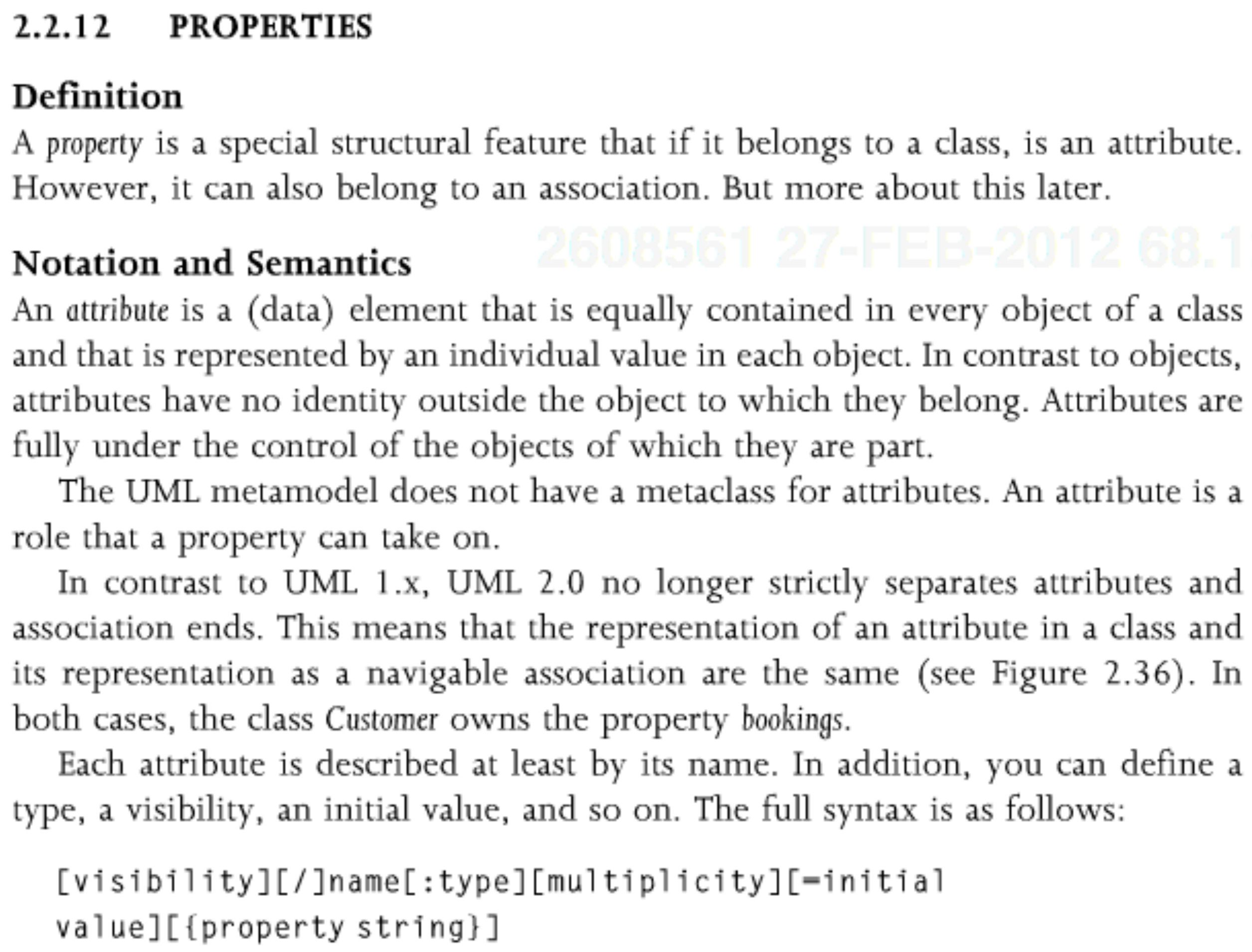
UML diagram elements

by shape	by diagram
Cuboid	Activity Diagram
Pentagonal	Class Diagram
Rectangular	Communication Diagram
Rectangular Rounded	Component Diagram
Diamond	Composite Structure Diagram
Triangular	Deployment Diagram
Elliptical	Interaction Overview Diagram
Round	Object Diagram
Linear Solid	Package Diagram
Linear Dashed	State Machine Diagram
Textual Shapes	Sequence Diagram
Numbers and Symbols	Timing Diagram
Miscellaneous	Use Case Diagram

by name	by discipline
A B C	Business Analysis
D E F	Requirements Analysis
G H I	Systems Analysis
J K L	Software Architecture
M N O	Hardware Architecture
P Q R	System Design
S T U	Database Design
V W X	Implementation
Y Z	Testing
	Deployment
	Project Management

UML Visibilities





2.2.12 PROPERTIES

Definition

A *property* is a special structural feature that if it belongs to a class, is an attribute. However, it can also belong to an association. But more about this later.

Notation and Semantics

An *attribute* is a (data) element that is equally contained in every object of a class and that is represented by an individual value in each object. In contrast to objects, attributes have no identity outside the object to which they belong. Attributes are fully under the control of the objects of which they are part.

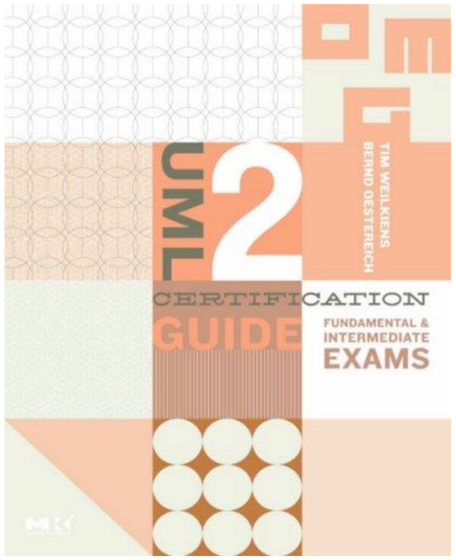
The UML metamodel does not have a metaclass for attributes. An attribute is a role that a property can take on.

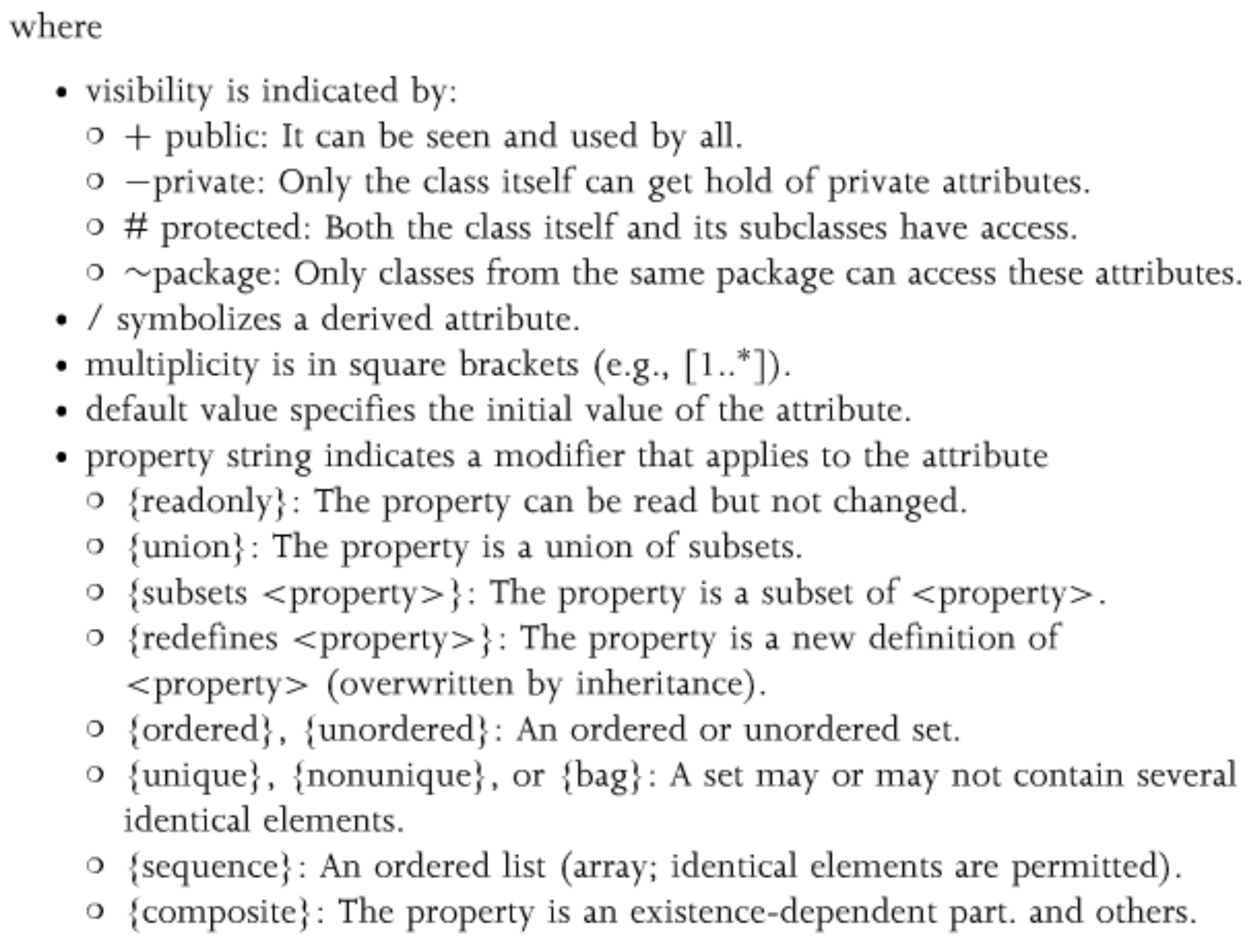
In contrast to UML 1.x, UML 2.0 no longer strictly separates attributes and association ends. This means that the representation of an attribute in a class and its representation as a navigable association are the same (see Figure 2.36). In both cases, the class *Customer* owns the property *bookings*.

Each attribute is described at least by its name. In addition, you can define a type, a visibility, an initial value, and so on. The full syntax is as follows:

```
[visibility][/]name[:type][multiplicity][=initial
value][{property string}]
```

UML Visibilities

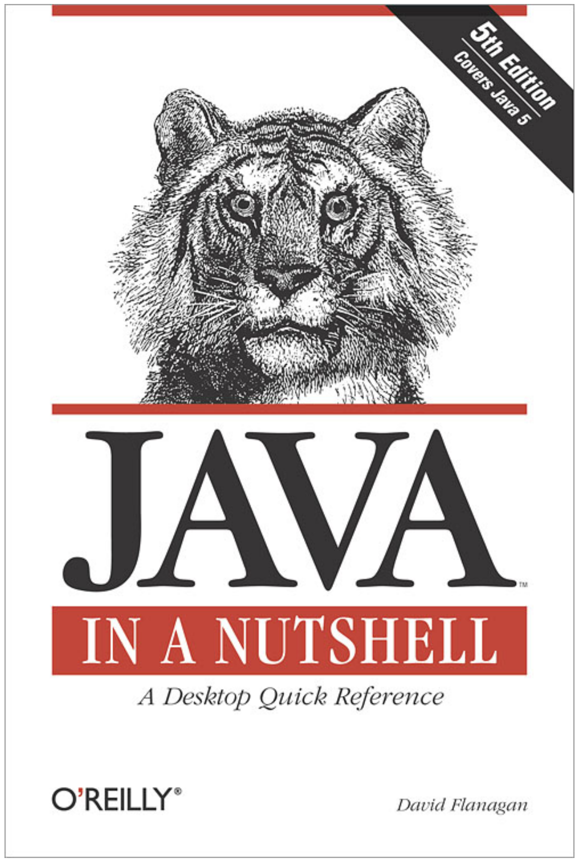


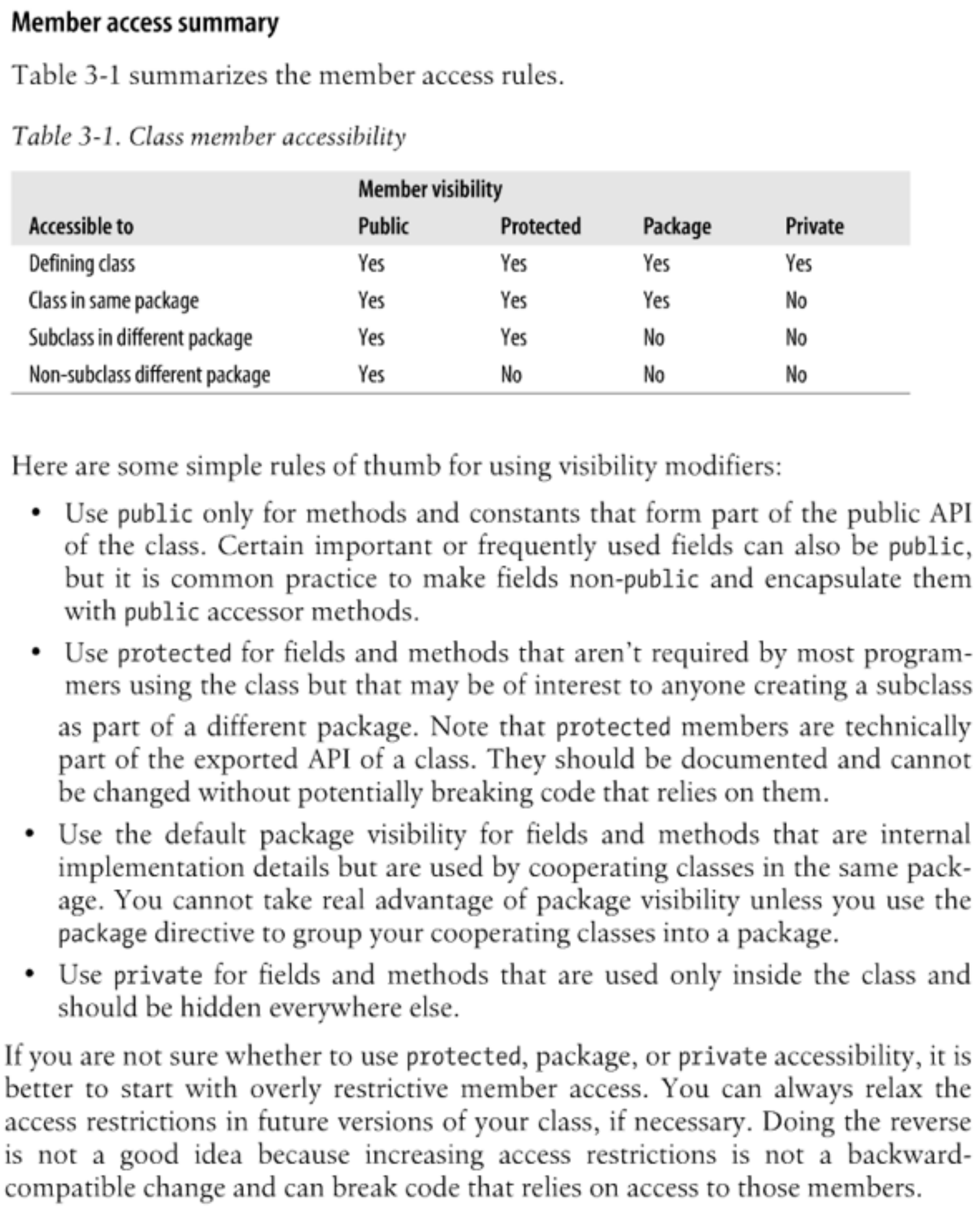


where

- visibility is indicated by:
  - + public: It can be seen and used by all.
  - −private: Only the class itself can get hold of private attributes.
  - # protected: Both the class itself and its subclasses have access.
  - ~package: Only classes from the same package can access these attributes.
- / symbolizes a derived attribute.
- multiplicity is in square brackets (e.g., [1..*]).
- default value specifies the initial value of the attribute.
- property string indicates a modifier that applies to the attribute
  - {readonly}: The property can be read but not changed.
  - {union}: The property is a union of subsets.
  - {subsets <property>}: The property is a subset of <property>.
  - {redefines <property>}: The property is a new definition of <property> (overwritten by inheritance).
  - {ordered}, {unordered}: An ordered or unordered set.
  - {unique}, {nonunique}, or {bag}: A set may or may not contain several identical elements.
  - {sequence}: An ordered list (array; identical elements are permitted).
  - {composite}: The property is an existence-dependent part. and others.

Java Access Rules





Member access summary

Table 3-1 summarizes the member access rules.

Table 3-1. Class member accessibility

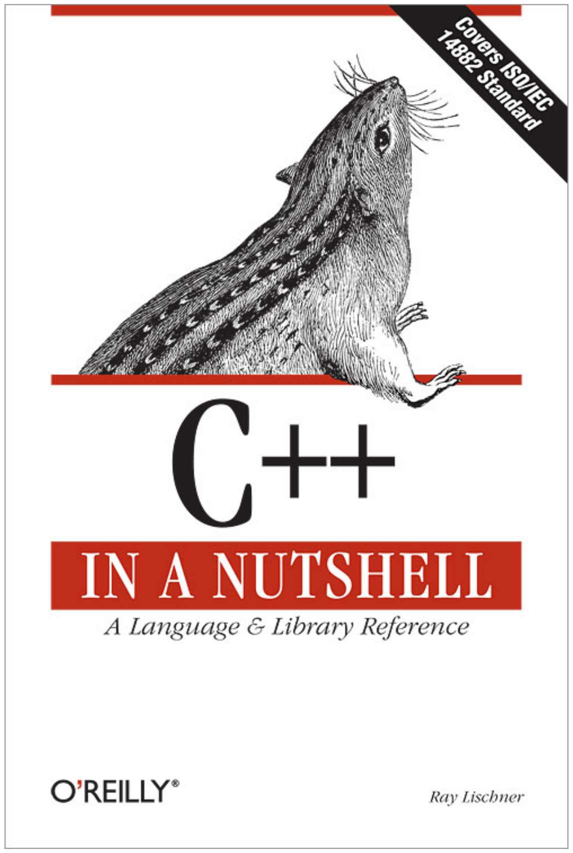
	Member visibility			
Accessible to	**Public**	**Protected**	**Package**	**Private**
Defining class	Yes	Yes	Yes	Yes
Class in same package	Yes	Yes	Yes	No
Subclass in different package	Yes	Yes	No	No
Non-subclass different package	Yes	No	No	No

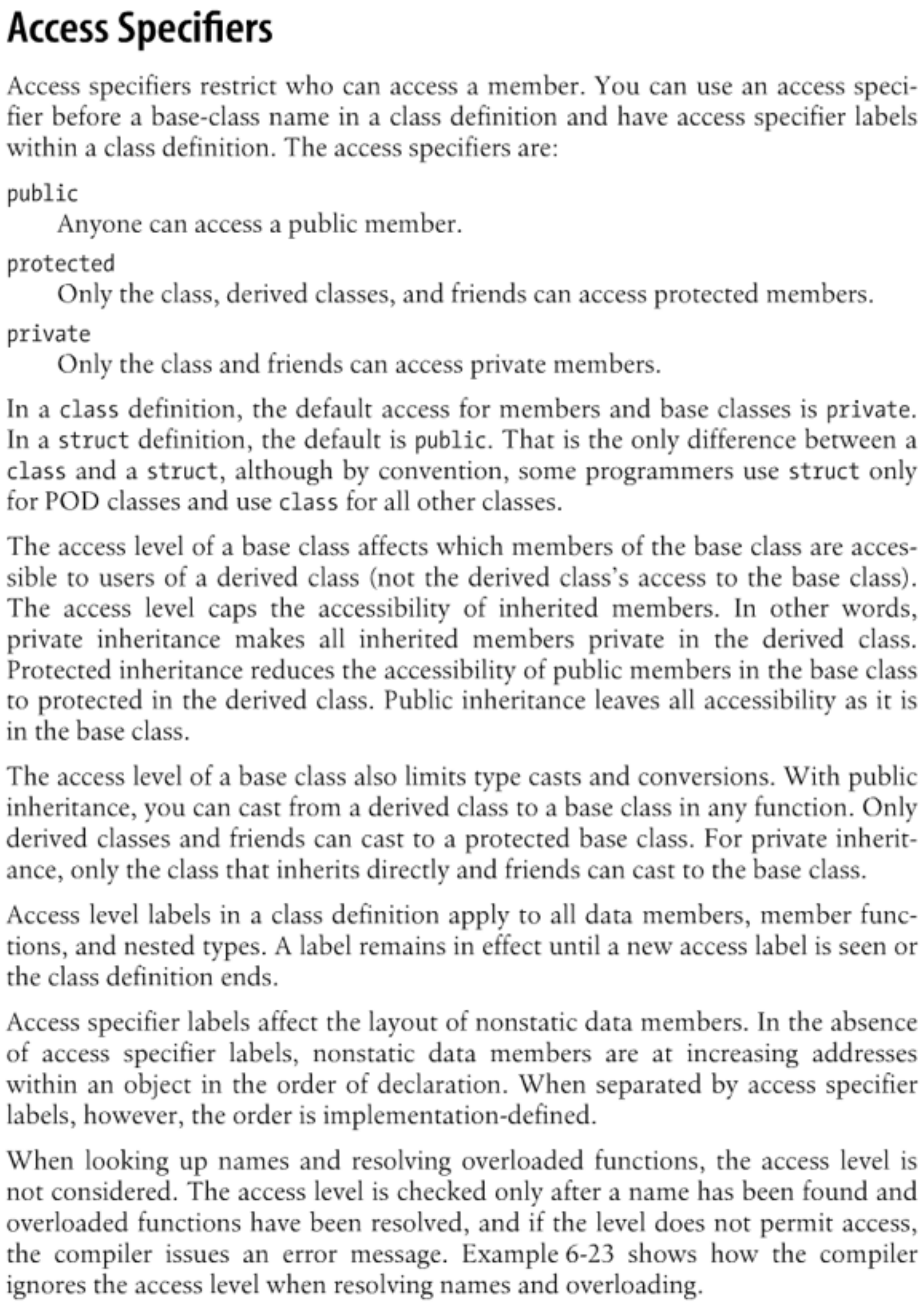
Here are some simple rules of thumb for using visibility modifiers:

- Use `public` only for methods and constants that form part of the public API of the class. Certain important or frequently used fields can also be `public`, but it is common practice to make fields non-`public` and encapsulate them with `public` accessor methods.
- Use `protected` for fields and methods that aren't required by most programmers using the class but that may be of interest to anyone creating a subclass as part of a different package. Note that `protected` members are technically part of the exported API of a class. They should be documented and cannot be changed without potentially breaking code that relies on them.
- Use the default package visibility for fields and methods that are internal implementation details but are used by cooperating classes in the same package. You cannot take real advantage of package visibility unless you use the `package` directive to group your cooperating classes into a package.
- Use `private` for fields and methods that are used only inside the class and should be hidden everywhere else.

If you are not sure whether to use `protected`, package, or `private` accessibility, it is better to start with overly restrictive member access. You can always relax the access restrictions in future versions of your class, if necessary. Doing the reverse is not a good idea because increasing access restrictions is not a backward-compatible change and can break code that relies on access to those members.

C++ Access Rules





Access Specifiers

Access specifiers restrict who can access a member. You can use an access specifier before a base-class name in a class definition and have access specifier labels within a class definition. The access specifiers are:

`public`
: Anyone can access a public member.

`protected`
: Only the class, derived classes, and friends can access protected members.

`private`
: Only the class and friends can access private members.

In a `class` definition, the default access for members and base classes is `private`. In a `struct` definition, the default is `public`. That is the only difference between a `class` and a `struct`, although by convention, some programmers use `struct` only for POD classes and use `class` for all other classes.

The access level of a base class affects which members of the base class are accessible to users of a derived class (not the derived class's access to the base class). The access level caps the accessibility of inherited members. In other words, private inheritance makes all inherited members private in the derived class. Protected inheritance reduces the accessibility of public members in the base class to protected in the derived class. Public inheritance leaves all accessibility as it is in the base class.

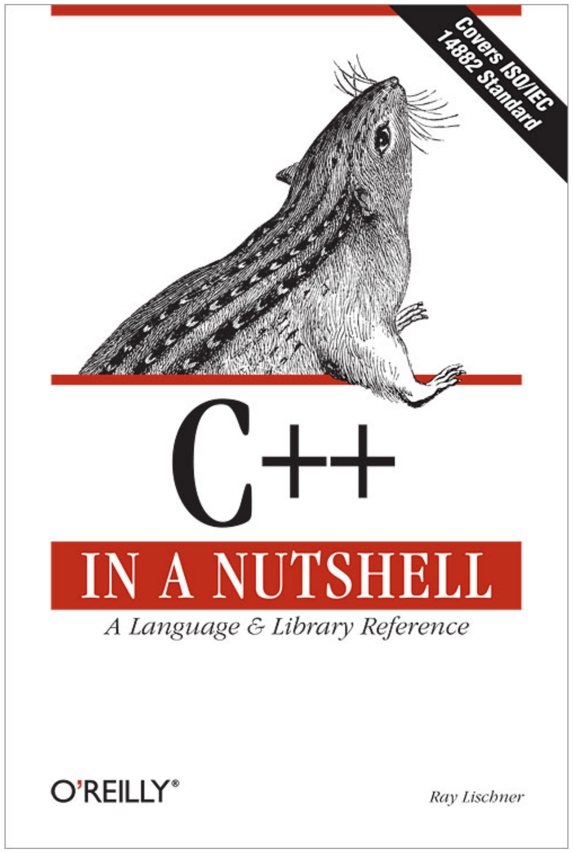
The access level of a base class also limits type casts and conversions. With public inheritance, you can cast from a derived class to a base class in any function. Only derived classes and friends can cast to a protected base class. For private inheritance, only the class that inherits directly and friends can cast to the base class.

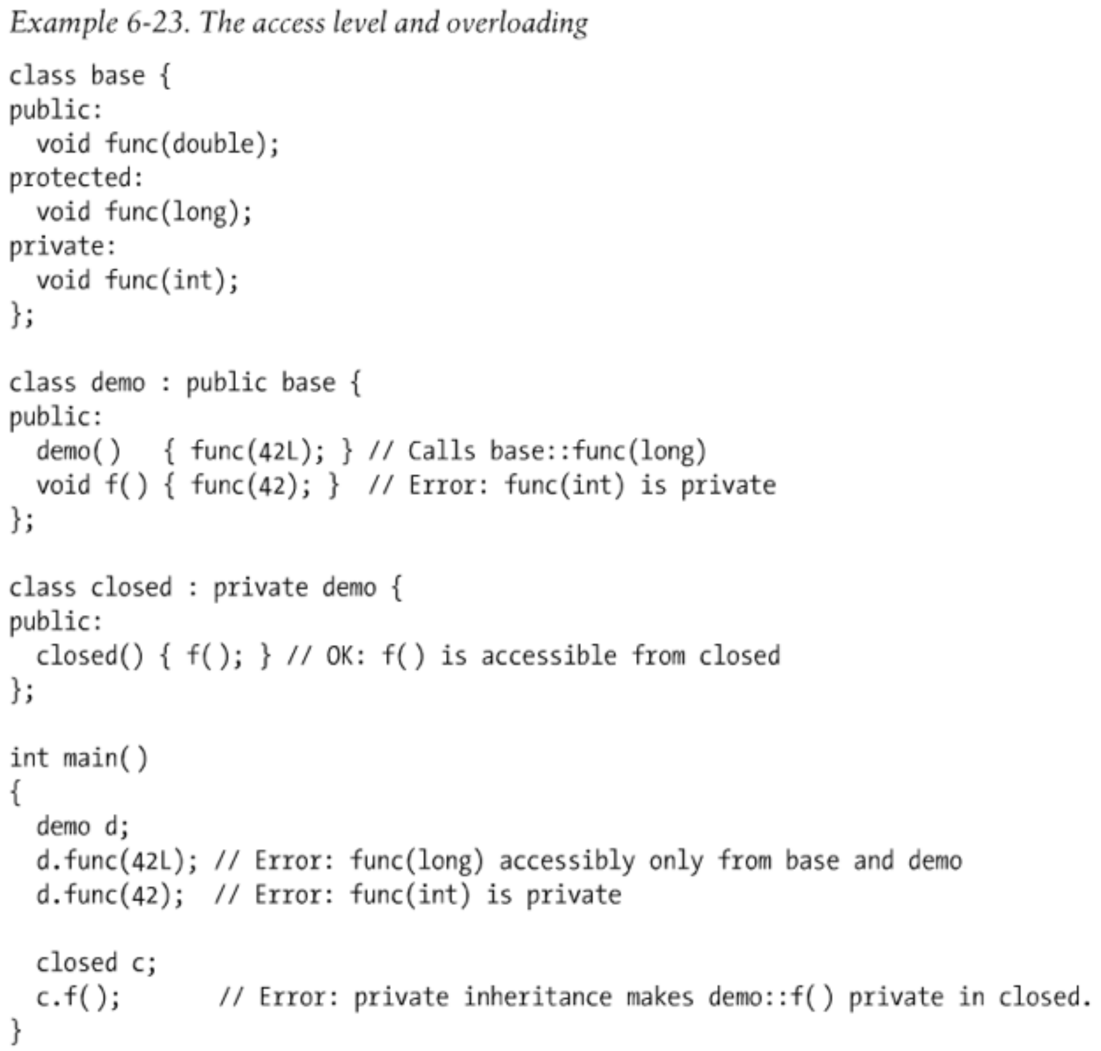
Access level labels in a class definition apply to all data members, member functions, and nested types. A label remains in effect until a new access label is seen or the class definition ends.

Access specifier labels affect the layout of nonstatic data members. In the absence of access specifier labels, nonstatic data members are at increasing addresses within an object in the order of declaration. When separated by access specifier labels, however, the order is implementation-defined.

When looking up names and resolving overloaded functions, the access level is not considered. The access level is checked only after a name has been found and overloaded functions have been resolved, and if the level does not permit access, the compiler issues an error message. Example 6-23 shows how the compiler ignores the access level when resolving names and overloading.

C++ Access Rules

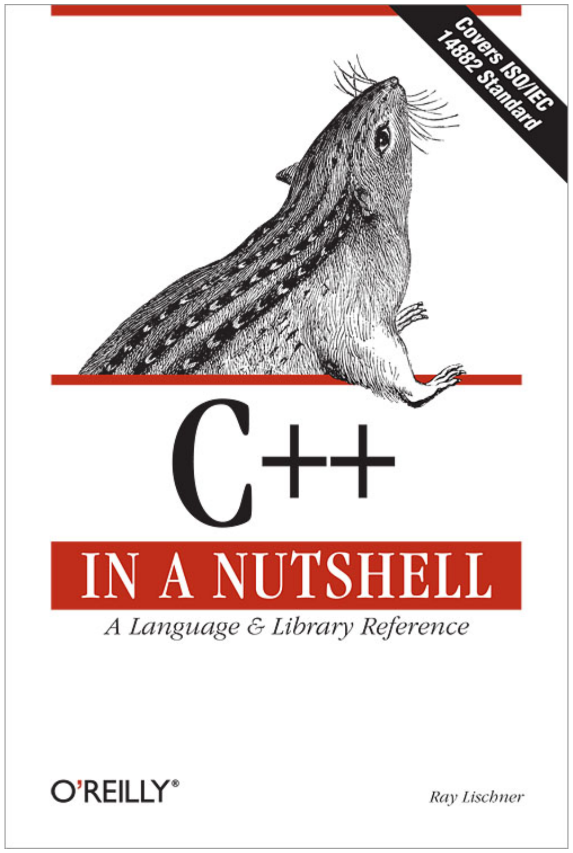




Example 6-23. The access level and overloading

```
class base {
public:
  void func(double);
protected:
  void func(long);
private:
  void func(int);
};

class demo : public base {
public:
  demo()   { func(42L); } // Calls base::func(long)
  void f() { func(42); }  // Error: func(int) is private
};

class closed : private demo {
public:
  closed() { f(); } // OK: f() is accessible from closed
};

int main()
{
  demo d;
  d.func(42L); // Error: func(long) accessibly only from base and demo
  d.func(42);  // Error: func(int) is private

  closed c;
  c.f();       // Error: private inheritance makes demo::f() private in closed.
}
```

C++ Friends



Friends

A *friend* is permitted full access to private and protected members. A friend can be a function, function template, or member function, or a class or class template, in which case the entire class and all of its members are friends.

Use the `friend` specifier to declare a friend in the class granting friendship. Note that friendship is given, not taken. In other words, if class A contains the declaration `friend class B;`, class B can access the private members of A, but A has no special privileges to access B (unless B declares A as a friend).

By convention, the `friend` specifier is usually first, although it can appear in any order with other function and type specifiers. The `friend` declaration can appear anywhere in the class; the access level is not relevant.

You cannot use a storage class specifier in a friend declaration. Instead, you should declare the function before the class definition (with the storage class, but without the `friend` specifier), then redeclare the function in the class definition (with the `friend` specifier and without the storage class). The function retains its original linkage. If the `friend` declaration is the first declaration of a function, the function gets external linkage. (See Chapter 2 for more information about storage classes and linkage.) For example:

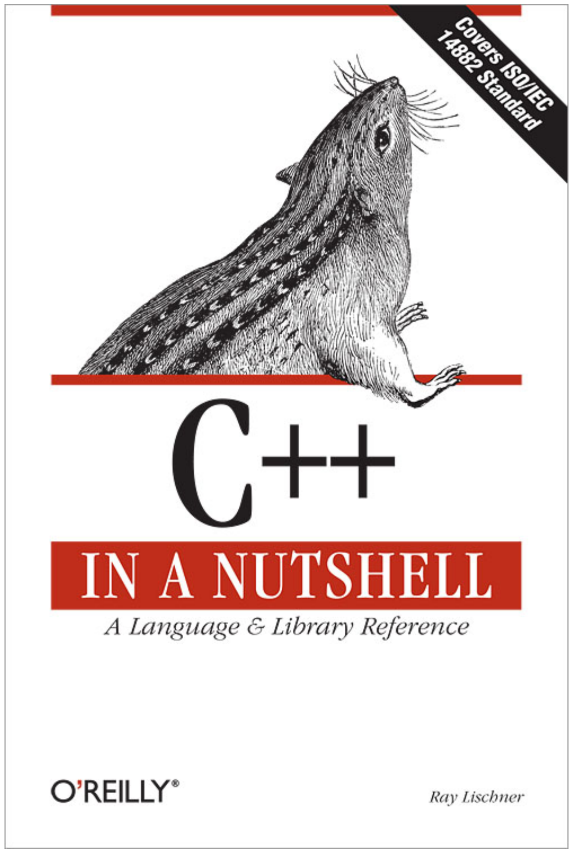
```
class demo;
static void func(demo& d);
class demo {
  friend void func(demo&);
  ...
```

Friendship is not transitive—that is, the friend of my friend is not my friend (unless I declare so in a separate `friend` declaration)—nor is a nested class a friend just because the outer class is a friend. (See the next section, "Nested Types," for more information.)

Friendship is not inherited. If a base class is a friend, derived classes do not get any special privileges.

You cannot define a class in a `friend` declaration, but you can define a function, provided the class granting friendship is not local to a block. The function body is in the class scope, which affects name lookup (see Chapter 2). The friend function is automatically inline. Usually, friend functions are declared, not defined, in the class.

C++ Friends

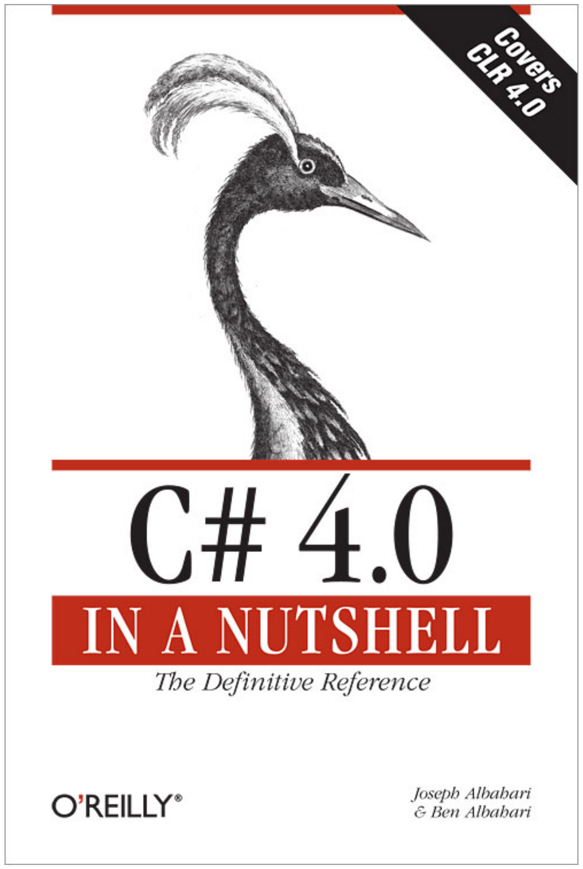


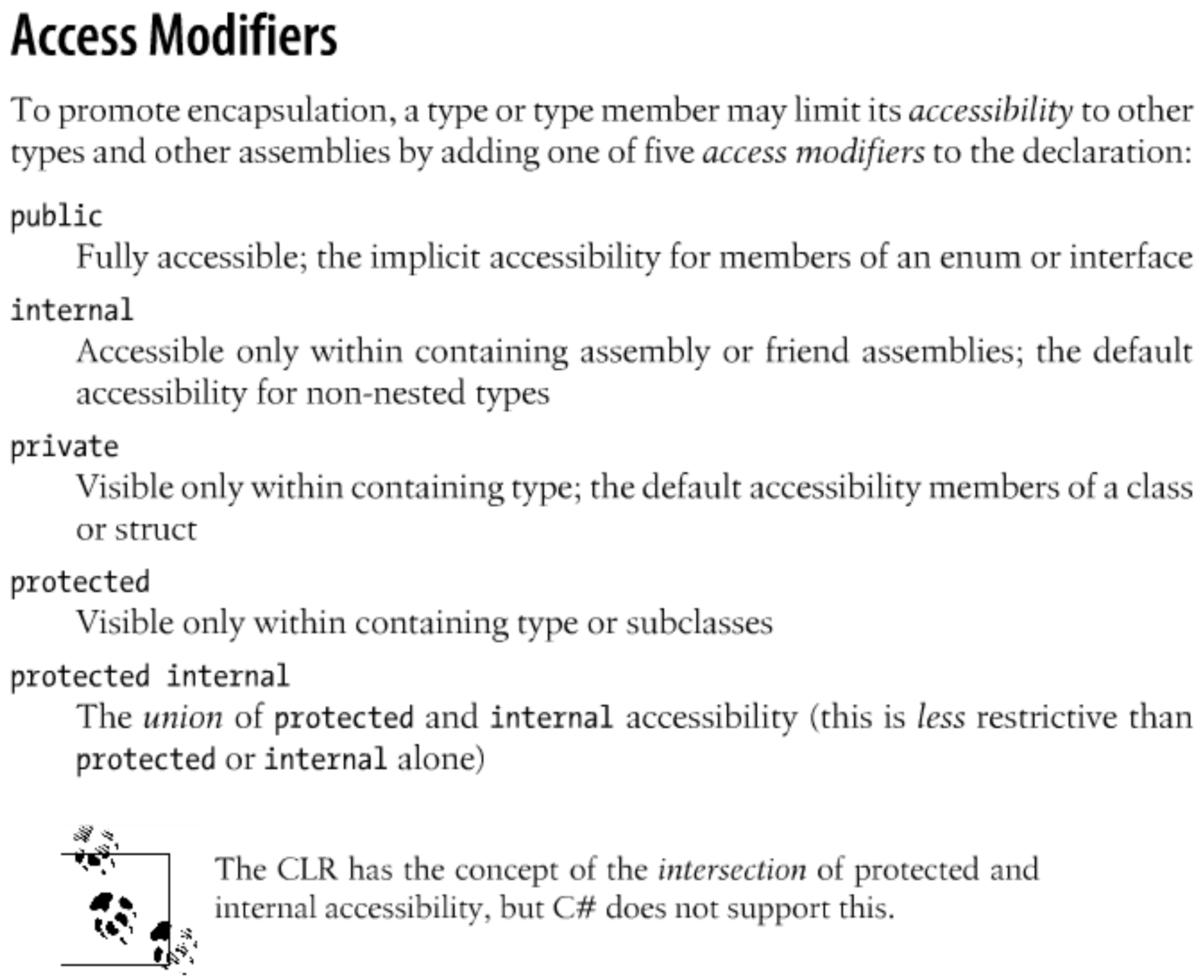
A declaration or definition of a friend function does not make that function a member of the class and does not introduce the name into the class scope. Example 6-26 shows several different kinds of friends.

Example 6-26. Friend functions and classes

```
#include <iterator>

// Simple container for singly-linked lists
template<typename T>
class slist {
  // Private type for a link (node) in the list
  template<typename U>
  struct link {
    link* next;
    U value;
  };
  typedef link<T> link_type;

  // Base class for iterator and const_iterator. Keeps track of current node, and
  // previous node to support erase().
  class iter_base :
  public std::iterator<std::forward_iterator_tag, T> {
  protected:
    friend class slist; // So slist can construct iterators
    iter_base(slist::link_type* prv, slist::link_type* node);
    slist::link_type* node_;
    slist::link_type* prev_;
  };

public:
  typedef T value_type;
  typedef std::size_t size_type;

  class iterator : public iter_base {
    // Members omitted for bevity...
  private:
    friend class slist; // So slist can call constructor
    iterator(slist::link_type* prev, slist::link_type* node)
    : iter_base(prev, node) {}
  };
```

C# Access Rules





Access Modifiers

To promote encapsulation, a type or type member may limit its *accessibility* to other types and other assemblies by adding one of five *access modifiers* to the declaration:

`public`
: Fully accessible; the implicit accessibility for members of an enum or interface

`internal`
: Accessible only within containing assembly or friend assemblies; the default accessibility for non-nested types

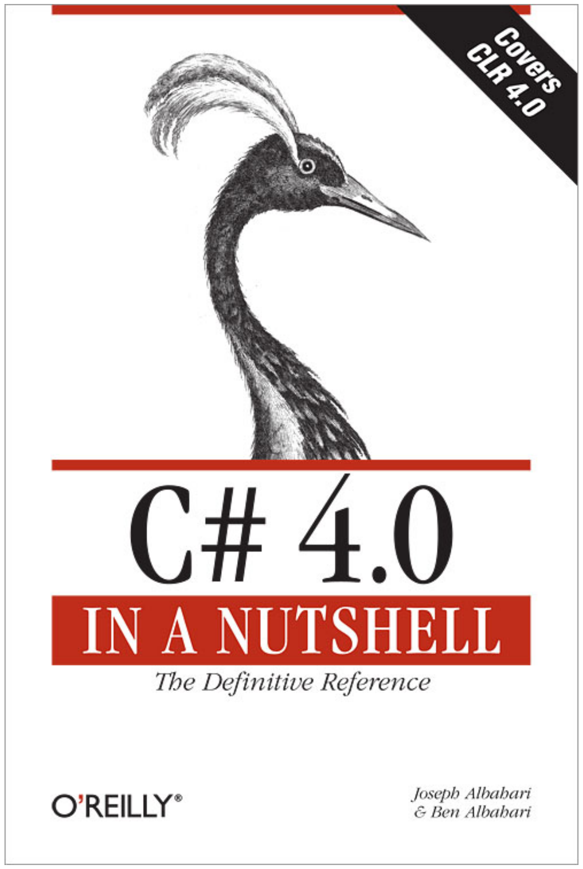
`private`
: Visible only within containing type; the default accessibility members of a class or struct

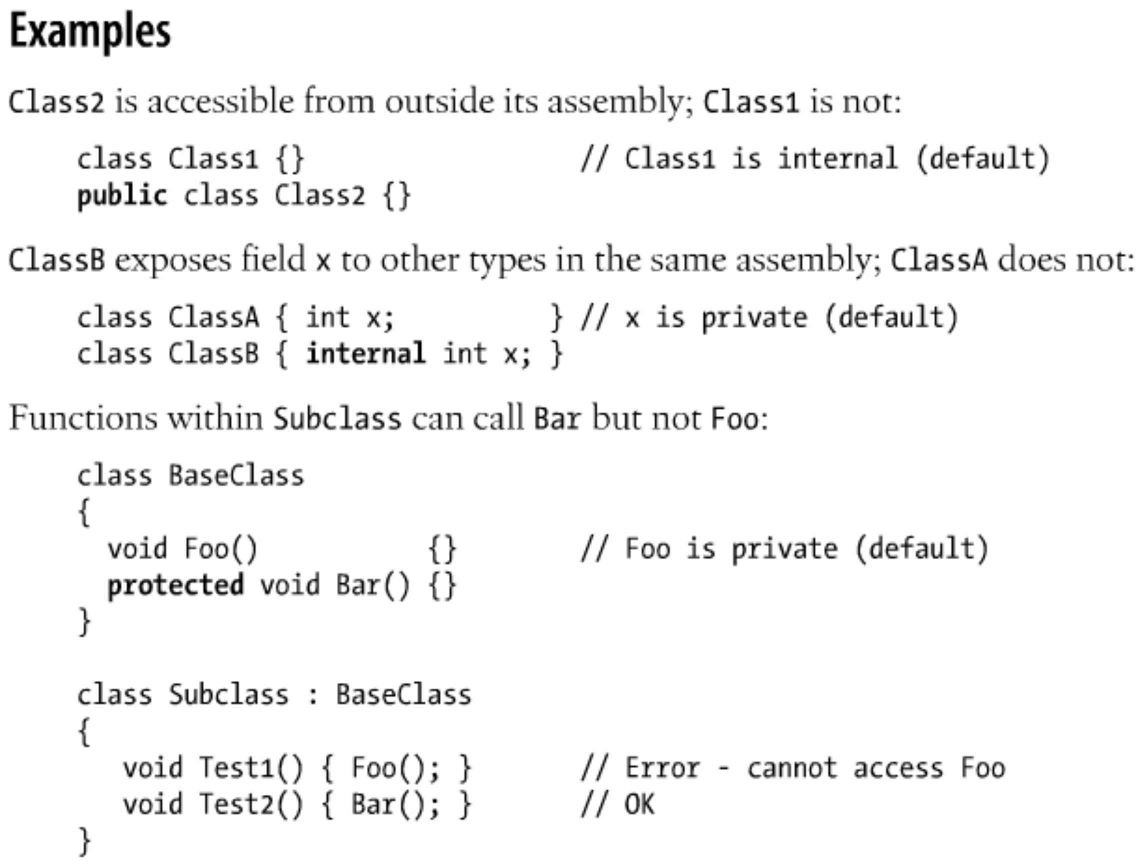
`protected`
: Visible only within containing type or subclasses

`protected internal`
: The *union* of `protected` and `internal` accessibility (this is *less* restrictive than `protected` or `internal` alone)

The CLR has the concept of the *intersection* of protected and internal accessibility, but C# does not support this.

C# Access Rules





Examples

`Class2` is accessible from outside its assembly; `Class1` is not:

```
class Class1 {}                    // Class1 is internal (default)
public class Class2 {}
```

`ClassB` exposes field `x` to other types in the same assembly; `ClassA` does not:

```
class ClassA { int x;          } // x is private (default)
class ClassB { internal int x; }
```

Functions within `Subclass` can call `Bar` but not `Foo`:

```
class BaseClass
{
  void Foo()           {}          // Foo is private (default)
  protected void Bar() {}
}

class Subclass : BaseClass
{
   void Test1() { Foo(); }         // Error - cannot access Foo
   void Test2() { Bar(); }         // OK
}
```