The Notion of a Class and Some Other Key Ideas (contd.)

Questions:
1. Right or wrong: What ‘#include’ does in C++ is achieved by ‘import’ in Java?
2. Code modularization achieved by packages in Java is achieved in C++ by what mechanism?
3. Do you see any problems with the following code fragment?

```cpp
void foo() {}

namespace Module1 {
    void foo() {}
}

int main()
{
    foo();
    using namespace Module1;
    foo();
}
```
4. Will this program compile?

```cpp
namespace Module1 {
    void foo() {};
}

namespace Module2 {
    void foo() {};
}

namespace Module3 {
    using namespace Module1;
    using namespace Module2;
}

int main(){ using namespace Module3; }
```
Access Control for Class Members (Section 3.11)

Every member of a class has associated with it an access control property.

In C++, a member can be *private*, *protected*, or *public*.

In addition to these three, Java also allows for the access control property of a member to be *package*.

When the access control modifier is left unspecified for a class member in Java, it is of type *package*.
Access Control in C++:

All members of a C++ class are *private* unless declared explicitly to be otherwise.

Public members of a class are accessible to all other classes and functions.

Members that are *private* to a class can be accessed by only those functions that are defined specifically for that class.

The access control modifier *protected* is used for a member if we wish for that member to be accessible to only the subclasses of that class. (A protected member of a class acts like a public member for the subclasses derived from that class, but like a private member for the rest of the program.)
The private members of a class are also accessible to the friends of that class.
//Friend.cc

#include <iostream>

class Y; // (A)

class X {
    int m;
    int n;
public:
    X( int mm, int nn ) { m = mm; n = nn; }
    friend Y; // (B)
    friend void print( X* ); // (C)
};

class Y {
    X* x;
    int t;
public:
    Y( X* xobj ) { x = xobj; t = x->m + x->n; } // (D)
    int get_t() { return t; }
};

void print( X* ptr ) {
    cout << ptr->m << " " << ptr->n << endl; // (E)
}
int main()
{
    X* ptr = new X(100, 200);
    Y y(ptr);
    cout << y.get_t() << endl; // 300

    print(ptr); // 100 200

    return 0;
}
A *friend* declaration may be placed in any section of a class, private, protected, or public. It carries the same meaning regardless of where it appears.
Access Control in Java:

In addition to the modifiers *public*, *private*, and *protected*, Java has one more: *package*.

The modifiers *public* and *private* carry the same meaning in Java as they do in C++.

The modifier *protected* also carries the same meaning, except that such members are like public members in the same package.

In other words, a protected class member will be accessible inside all classes in the same package, but to only the subclasses in other packages.
When no access modifier is specified in Java, that means the member is of type *package*. Such members are no different from public members within the same package. But they act like private members with respect to other packages.
Abstract Classes and Interfaces (Section 3.12)

Abstract classes are very important to object-oriented programming.

While the notion of an abstract class is common to both C++ and Java, the latter also supports a variant thereof – interfaces.

A Java interface is an abstract class that is not allowed to contain any implementation code at all for any of the member functions.

An interface is also not allowed any data members that can be given values on a per object basis.
Why abstract classes are useful:

- An abstract class can lend organization to the other classes in a class hierarchy.

- An abstract class can represent a specialized behavior, which when mixed with the other classes gives us classes with that behavior.

- Abstract classes can help build up an implementation incrementally.
An abstract class can help with knowledge organization in an OO program by serving as a root class of a hierarchy of other concrete classes and thus pull together what might otherwise be disparate bits of knowledge encapsulated in the subclasses.
C++:

```cpp
class Shape {
public:
    virtual double area() = 0;
    virtual double circumference() = 0;
    //....
};
```

JAVA:

```java
abstract class Shape {
    abstract public double area();
    abstract public double circumference();
    //....
}
```
interface Collection {
    public boolean add( Object o );
    public boolean remove( Object o );
    // other methods
}

The methods declared in an interface are always public, implicitly so if their access privilege is not so stated.
Comparing Objects (Section 3.13)

The basic issues in object comparison relate to

- what can be compared; and
- how to compare.

```
class Apple {
    Taste taste;
    Size size;
    Weight wt;
    // ...
};

class Orange {
    Taste taste;
    Size size;
    Weight wt;
    // ...
};
```
There are two kinds of comparisons that one can make for class type objects:

- We may wish to know whether or not two objects are identical on the basis of equal values for one or more of the data members. *The result of such a comparison is either true or false.*

- Or, we may wish to know whether one object is smaller than, equal to, or greater than another object, again on the basis of the values for one or more data members of the objects involved.
In C++, the first kind of comparison is yielded typically by the `==` operator. The programmer has to overload this operator for a given class.

The second type of comparison in C++ is implemented by defining an appropriate comparison function that returns the three values needed.
Our discussion so far on object comparison has been centered primarily on a comparison of two objects on the basis of their content, meaning on the basis of the values of one or more the data members of the objects.

Java adds an additional twist to this – it allows object comparisons on the basis of equality of reference. Two objects are equal on the basis of equality of reference if they are the same object in the memory.
In Java, comparison of two objects on the basis of equality of reference is carried out by the == operator.

And a comparison on the basis of content can be carried out by a programmer-supplied definition for the `equals()` method that every class in Java inherits from the root class `Object`. 
class X {
    int p;
    X( int m ) { p = m; }
    // ...
}

X x1 = new X( 10 );
X x2 = x1;

x1 == x2; // true

X x1 = new X( 10 );
X x2 = new X( 10 );
x1 == x2; // false

If you do not supply your own override definition for equals that X inherits from Object, comparisons using equals behave in exactly the same manner as comparisons using ==:
X x1 = new X( 10 );
X x2 = x1;
x1.equals( x2 ); // true
X x3 = new X( 10 );
x1.equals( x3 ); // false
/EqualityTest.java

class X {
    int p;
    int q;
    X( int m, int n ) { p = m; q = n; }
    boolean equals( X other ) { return p == other.p; }
}

class Test {
    public static void main( String[] args ) {
        X x1 = new X( 10, 100 ); // x1 and
        X x2 = new X( 10, 10000 ); // x2 look very different
        if ( x1.equals( x2 ) ) // but they are equal
            System.out.println( "x1 and x2 are equal" );
    }
}
The Java platform also uses the notion of *natural ordering* for comparing class type objects.

The objects of a Java class exhibit natural ordering if the class has implemented the `java.lang.Comparable` interface.

Such a class must provide an implementation for the `compareTo` method – referred to as the class’s *natural comparison method* – that can then be used by the algorithms and the data structures for comparing data objects.

The `compareTo` method must return a negative integer, a zero, or a positive integer if the object on which it is invoked is less than, equal to, or greater than the argument object.
Many of the system supplied classes in Java possess natural ordering. These include `String`, `Integer`, `Float`, `Double`, `Date`, `File` and many others.

For the `String` class, the natural order is lexicographic; it is chronological for the `Date` class; lexicographic on the pathname for the `File` class, etc.
Static Members of a Class

A static member is global to all the objects of a class. For example, in the C++ class

```cpp
class SavingsAccount {
    string name;
    double balance;

public:
    static double interestRate;
    //....
};
```

In C++, a public static data member can be accessed directly through the class using the scope operator ::, as in

```cpp
SavingsAccount::interestRate = 6.5;
```
A private static data member may require appropriate access functions. The following example shows the use of static member functions for this purpose:

```cpp
class X {
  int m;
  static int n;
public:
  X( int p ) { m = p; }
  static int getn() { return n; }
  static void setn( int m ) { n = m; }
};

X::setn( 20 );
cout << X::getn();
```

This example is not meant to imply that the value of a static data member can only be modified by a static member function; it can also be modified by a non-static member function.
However, a static member function is not allowed to access non-static data members of a class.
The static members of a Java class behave in the same manner as the static members of a C++ class; however, the syntax for accessing such members is different since Java does not support the scope operator.

```java
class SavingsAccount {
    string name;
    double balance;
    public static double interestRate;
    //....
};

SavingsAccount.interestRate = 6.5;
```

An abstract class is not allowed to have static member functions.
Template Classes

A “templatized” C++ program can work with different types of data types.

For example, a templatized C++ linked-list can be used to hold elements of type int, double, char, etc.

```cpp
class X{
    T datum; // type T not known in advance
public:
    // constructor, etc.
};
```
template <class T> class X {
    T datum;
public:
    X( T dat ) : datum( dat ) {}
    T getDatum(){ return datum; }
};
```cpp
#include <string>
#include <iostream>

template <class T> class X {
   T datum;
public:
   X( T dat ) : datum( dat ) {}  
   T getDatum() { return datum; } 
};

int main()
{
   int x = 100;
   X<int> xobj_1( x );

   double d = 1.234;
   X<double> xobj_2( d );

   string str = "hello";
   X<string> xobj_3( str );

   string ret1 = xobj_3.getDatum();
   cout << ret1 << endl;  // output: ‘hello’

   // int ret2 = xobj_3.getDatum();  // Error
}