Department of Computer Science
Trent University

COIS 2020H: Data Structures and Algorithms

Terminology

Object-Oriented Programming (OOP)

Object-oriented programming is a paradigm that rests on the three tenets of encapsulation, inheritance and polymorphism (dynamic binding).

Class

A class is a syntactic unit that encapsulates both data members and methods. It defines a reference type.

Object

An object is an instance of a class, created using the **new** command (see Reference Type).

Data Member

A data member is a field that defines (part of) the representation of a class.

```
class A { … int x; … }          // x is a data member of class A
```

Method

A method is a process that defines (part of) the behavior of a class.

```
class A { … void F ( ) { … }  … }      // F is a method of class A
```

Value Type

A value-type variable directly contains its data. For example, int, float, char and bool.

Reference Type

A reference-type variable contains a reference to its data (object). The object itself is created using the new command.

```
A a;              // "a" will store a reference to an object of class A
a = new A( );     // an object of class A is created and is referenced by "a"
```

Access Modifier

An access modifier defines the visibility of a data member or method. The five definitions that follow assume a **public** class. The definitions are the same for an **internal** class except access, whenever possible, is restricted to classes within the same compiled unit. Hence, a public method within an internal class can only be accessed by other classes within the same compiled unit.

1) **Private Modifier**

The private modifier restricts access to a data member or method to within the class itself.

```
class A { … private int x; … }          // data members are generally private
```

By default, data members and methods are private.

2) **Public Modifier**

The public modifier has no restrictions on access to a data member or method.

```
class A { … public void F ( ) { … } … }     // methods are generally public
```

3) **Protected Modifier**

The protected modifier restricts access to a data member or method to its own and descendent classes (see Inheritance).

4) **Internal Modifier**

The internal modifier restricts access to a data member or method to those classes within the same compiled unit. By default, the modifier of a class is internal.

5) **Internal Protected Modifier**

The internal protected modifier restricts access to a data member or methods to its own or descendent classes within the same compiled unit.

Static Class

A static class is a class that contains only static methods and data members. It is preceded by the keyword static.

```
public static class A { … }
```

Static Data Member or Method

A static method or data member is associated with the class itself, not an instance of the class. It is accessed or invoked using the class name. They are both preceded by the keyword static.

```
public class A {
…
private static int x;                  // data member x belongs to class A
public static void F ( ) { … }         // static method F can only be invoked by its class, i.e. A.F();
…
}
```

Instance Data Member or Method

An instance data member or method is associated with an instance of the class. Therefore, an instance of the class must be created to access the data member or invoke the method.

```
public class A {
…
private int x;
public void F( ) { … }
…
}
…
A a = new A( );
a.F( );                                // an instance of A must be created to invoke F
```

Generic Class

A generic class encapsulates data members and methods that are not specific to a particular type.

```
public class A<T> { … }
```

When an object of type A is created, T is then specified.

```
A<int> a = new A<int>( );
```

Constrained Generic Class

A constrained generic class restricts T to those types that satisfy a certain condition.

```
public class A<T> where T : IComparable { … }
```

Type T must have implemented the CompareTo method of IComparable (see Interface).

Abstract Method

An abstract method is a method with no body (no definition). Abstract methods are implicitly virtual.

```
public abstract int F( );
```

Abstract Class

An abstract class is a class with at least one abstract method. It cannot be instantiated.

```
public abstract class A { … }
```

Inheritance

Inheritance allows one class (the derived class) to reuse, extend and modify the behavior of its ancestor classes (base classes). In C#, a class can only inherit from one parent class.

```
public class B : A { … }              // class B inherits from class A
```

Object Class

The object class is found at the root of the type hierarchy from which all classes implicitly derive.

Virtual/Override Method

A virtual method is a method that can be overridden by a descendent class.

```
public virtual     float F( ) { … }     // in the parent class
…
public override float F( ) { … }     // in the descendent class, F is redefined
```

Interface

A syntactic structure, similar to a class, that includes the signatures (headers) of methods only *. Hence, each method is implicitly public, virtual and abstract. An interface contains no data members.
* An interface can also contain the signatures for properties, events and indexers.

```
interface A { float F( ); }
```

Each method of an interface must be implemented by the class that inherits it. Hence, it is typically said that a class “implements” an interface.

Polymorphism

Polymorphism (dynamic binding) allows one to assign an object of a derived type to a reference variable of its ancestor type. If the derived type has overridden a method of its ancestor, the declared reference of the ancestor type will invoke the method of its descendent (and not its own).

```
public class A      { … public virtual    void F ( ) { … } … }     // ancestor class A
public class B : A { … public override void F ( ) { ... } … }     // derived class B overrides method F

A a;
B b;
…
a = b;    // an instance of b can be assigned to any reference of its ancestor classes
a.F( );   // the F method of b is executed, not that of a
```

Time Complexity

Time complexity establishes a relationship between the running time of an algorithm T(n) and the size of the problem (data structure) given by n. Typically, the time complexity is described succinctly using the "Big-Oh" notation.

$T(n) = O(\log n)$	the running time increases logarithmically with the size of the problem
$T(n) = O(n)$	the running time increases linearly with the size of the problem
$T(n) = O(n^2)$	the running time increases quadratically with the size of the problem