U1 Object oriented modelling

Thursday, 13 August 2020 3:59 PM

Java programming

```
import java.util.Scanner;

class Factorial   // class name
{
  public static int getFact(int n) {
    int c, fact = 1;
    if ( n < 0 )
      System.out.println("Number should be non-negative.");
    else
        for ( c = 1 ; c <= n ; c++ )
        fact = fact*c;
    return fact ;
  }

  public static void main(String args[])
  {
    int n = 1;

    System.out.println("Enter an integer to calculate it's factorial");
    Scanner in = new Scanner(System.in);
    n = in.nextInt();
    System.out.println("Factorial of "+n+" is = " + getFact(n));


  }
}
//  must save as <class name>.java => Factorial.java
```

Four basic components of OOM
- Objects
- Messages
- Methods
- Classes

Four main concepts of OOM
- Abstraction
- Encapsulation
- Inheritance
- Polymorphism

OO Conceptual Terms

- Object
- Class
- Attribute
- Operation
- Interface
- Implementation
- Association
- Aggregation
- Composition
- Generalisation
- Super-Class
- Sub-Class
- Abstract Class
- Concrete Class
- Polymorphism

U2: Class and Object

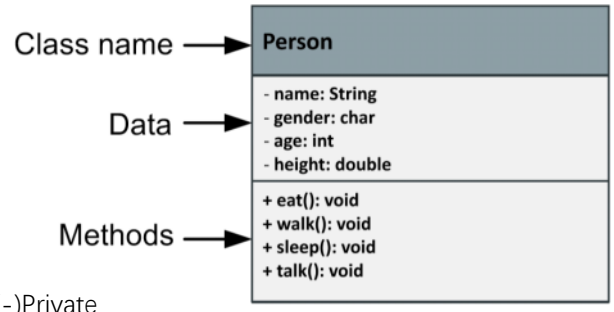
Friday, 14 August 2020 12:31 PM

Objects
- Characteristic
  - Identity (Object identifier OID)
  - State (Made up of attributes)
  - Behaviour (Things an object can do/ methods)

An object is a specific instance of a class
- Each object has unique characteristics

Class: defines the structure for creating objects
- Contains: Data properties and methods

UML (Unified Modelling Language) diagram:

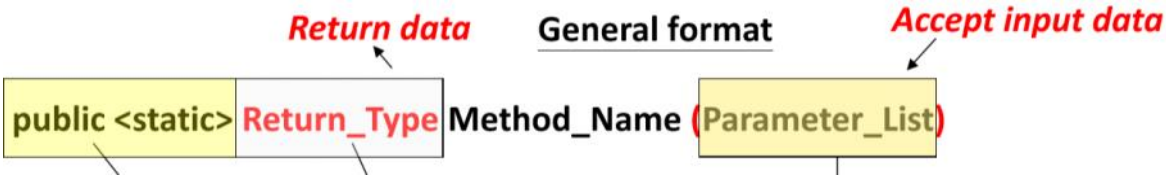


- (-)Private
- (+)Public

Class Diagram

Example:

```
public class Rectangle {
   // instance variables
   // instance methods
}
```



Example

public static **int** successor(*double num*) { ..... }

public **boolean** *savetoDB(String name, int age, char gender)*
{.....}

Creating objects:

Rectangle rect
Rect = new Rectangle ()
Int [ ] array = new int [10]

- Rect stores the memory address of Rectangle object

Constructors:
- Initialising instance variable
- Parameters of the "function"

Destructors:
- Finalizer
  - Finalize () method to release resources

Object_Variable_Name.Instance_Variable
Object_Variable_Name.Instance_Method(Argument_List);

Creating an application class to test it out:

Public class RectangleApp (){
    Public static void main (string[ ] args){

```
        Rectangle rect = new Rectangle ();
        System.out.println("Area of rectangle is:" + rect.findArea());
        System.out.println("Perimeter of rectangle is:" + rect.findPerimeter());
        }
}
```

- **copyObjects()** method:

```
public class Rectangle {
    // instance variables and methods
    ...
    public Rectangle copyObjects() {
        Rectangle rect = new Rectangle( width , height );
        return rect;
    }
}
```

The Keyword 'this'

```
public class Rectangle {
  private double width ;  // data properties
  private double height ;
  public Rectangle () {   // constructors
     width = 1.0 ;
     height = 1.0 ;
  }
  public Rectangle ( double w , double h ) {
     width = w ;
     height = h ;
  }
  public double findArea()    // methods
       { return width * height; }
  public double findPerimeter() {
      return( width + height ) * 2 ;
  } . . .
  public void print() {
    System.out.println("The area of rectangle is "
                       + this.findArea() );
    System.out.println("The perimeter of rectangle is "
                       + this.findPerimeter() );
  }
```

- Similar to "self" in python
- Can be used to call either methods or instance variables
- Eg. this(10.0,20.0) activates the rectangle constructor

Accessors and mutators (Getters and setters)

- It is not a good OO design if you can change the state of an object by another object or function directly

Static keyword

- Used in declaration of an instance variable or method
- Applied to the whole class instead of the individual objects
- Class methods can reference class variables and methods, but not instance variables and methods in the class

```java
public class Person{
  double height  ;
  double weight ;

  public Person(double h, double w)
    height = h ;
    weight = w ;
  }
  public double getHeight() {
    return height ;
  }
  public double getWeight() {
    return weight ;
  }
  public double calculateBMI() {
     calculateBMI( height,  weight) ;
  }
  public static double calculateBMI(double height,
                                    double weight) {
    return (weight/(height * height));
  }
```

Category
Very severely underweight
Severely underweight
Underweight
Normal (healthy weight)
Overweight
Obese Class I
Obese Class II
Obese Class III

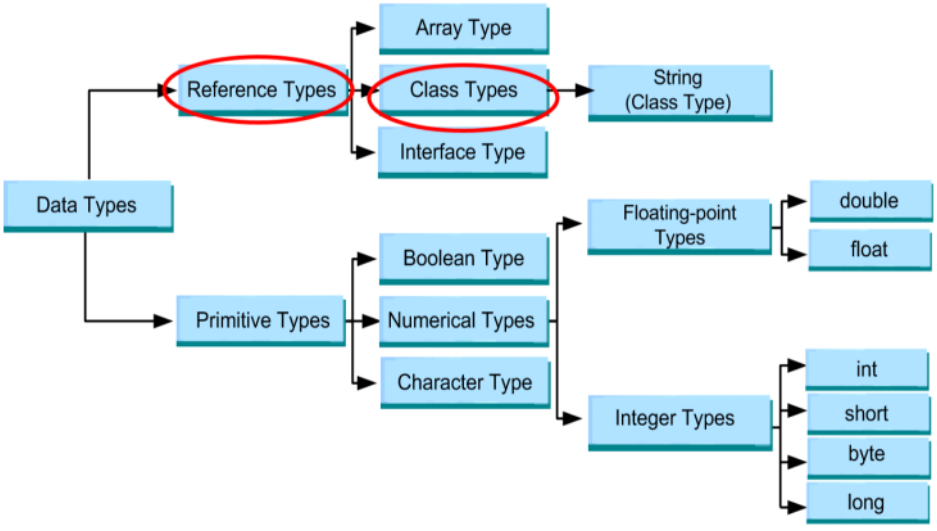
```java
public class PersonApp{
  public static void main(String[] args) {
    Person bill = new Person(1.80 , 70);
    double bmi = bill .calculateBMI(); //instance method

    System.out.println("BMI is " + bmi);

    bmi =  bill .calculateBMI(bill.getHeight(), bill.getWeight());

    System.out.println("BMI is " + bmi);

    bmi = Person.calculateBMI (1.90, 90);

    System.out.println("BMI is " + bmi);
  }
}
```

Encapsulation and information hiding
- Default , private, public and protected

Object composition
- "has a" relationship
- A class contains object reference from other classes as its instance variables

```java
public class Student {
   private int    studentId;
   private char   gender;
   private double height;
   private int    age;
   private Course course;
```

```java
Course course1 = new Course(103,2004,2,80);
StudentInfo myStudent1 =
    new Student(11456,'M', 1.75,25,course1);
```



String class
- java.lang.string is a class representing strings
- Double quotes

  String Variable_Name = new String(String_Value) ;

- or

  String Variable_Name = String_Value ;

```
String aString = new String( "Java Programming" ); OR
String aString = "Java Programming" ;
```

aString = "Java" ;

String can be used as arguments for System.out.println()

```
System.out.println( "Java Programming" ); OR
System.out.println( aString );
```

aString.charAt(0)

1. **Lowercase** is where all the letters in a word are written without any capitalisation (e.g., packagename). **For package name.**

2. **Uppercase** is where all the letters in a word are written in capitals. When there are more than two words in the name use underscores to separate them (e.g., MAX_HOURS, FIRST_DAY_OF_WEEK). **For constants, enums.**

3. **CamelCase** (also known as *Upper* CamelCase) is where each new word begins with a capital letter (e.g., CamelCase, CustomerAccount, PlayingCard). **For Classes and Interfaces.**

4. **Mixed case** (also known as *Lower* CamelCase) is the same as CamelCase except the first letter of the name is in lowercase (e.g., **h**as**C**hildren, **c**ustomerFirstName, For **methods, variables**.

U3: Inheritance

Tuesday, 18 August 2020 5:50 PM

```
public class Rectangle extends Shape {
  private double width  ;
  private double height ;
  public Rectangle()
    { super() ; width = 0 ; height = 0 ; }
  public Rectangle( double xCoor , double yCoor ,
                    double w     , double h     )
    { super( xCoor , yCoor ); width = w ; height = h ; }
  public double getWidth()       { return width  ; }
  public double getHeight()      { return height ; }
  public double findPerimeter() { return 2*(width+height) ; }
  public double findArea()       { return width * height   ; }
  public void   print() {
      System.out.println( "Rectangle print() method: " );
      System.out.print(   "Center " );
    super.print();
      System.out.println( "Width = "      + width
                        + ";Height = "    + height );
      System.out.println( "Perimeter = " + findPerimeter() );
      System.out.println( "Area = "       + findArea()       );
  }
```

Inherits superclass's attributes and methods

Constructors

From superclass

- Subclass is defined using the extends keyword.
- super:
  - Can be used in subclass constructor to call the superclass's constructor. The call to **super()** must be the **first statement** in the constructor for the class.
  - Can also be used in method definition to call the superclass's method, i.e., **super.print();**.

Method overloading (similar to constructor overloading)

Same Method Name but:

1) **different number of parameters.**
2) **different parameter *types*.**

OR

Method overriding

- When subclass alters a method inherited from super class
- Through refinement or replacement
- @Overriding

1) Refinement: Reuse the implementation of superclass method with some refinement – using the super keyword.
2) Replacement: Replace the method completely.

```
class A
{
    int i;
    public void show()
    {
        System.out.println("in A");
    }
}
class B extends A
{
    int i;
    public void show()
    {
        super.i=8;
        super.show();
        System.out.println("in B");
    }
}

public class OverridingDemo
{

    public static void main(String[] args)
    {
        B obj1 = new B();
        obj1.show();
```

```
class A
{
    public void show()
    {
        System.out.println("in A");
    }
}
class B extends A
{
    public void show()
    {
        System.out.println("in B");
    }
}

public class OverridingDemo
{

    public static void main(String[] args)
    {
        A obj1 = new B();
        obj1.show();

    }
```

Dynamic method dispatch

Reference A but object B

- (Refer to left) however this object will not be able to call config() since it is not a method under class A

```
class B extends A
{
    public void show()
    {
        System.out.println("in B");
    }
    public void config()
    {
        System.out.println("config");
    }
}
```

- Visibility (accessibility/ access control):
  - public
    - Visible (accessible) to anywhere in an application.
  - private
    - Visible (accessible) only within that class's implementation.
  - protected *Stay in the family*
    - Visible (accessible) to methods of the class, the methods of subclasses, or any classes in the ***same package***.

- A package contains a set of classes that are **grouped** together in the same directory.
- Non-private data can be accessed by any object in the same package.

```
package packageOne;

import java.util.Scanner;

public class Class1 {
// the usual.....


}
```

If Visibility is (Access Level)	Visible			
	Within Class?	Within Package?	to a Subclass?	to the World? (outside Classes)
public	Y	Y	Y	Y
protected	Y	Y	Y	N
Not defined	Y	Y	N	N
private	Y	N	N	N

Final method: cannot be overridden in subclasses
Final class: cannot be a super class/ will not have a subclass

But if a superclass is too general, then no meaningful object can be created (instantiated) from it, such classes are called abstract classes (Abstract Base Class (ABC)).

- Abstract method does not have the implementation in the abstract class.

```
public abstract class Figure
{
   private String color;
   public  Figure()           { color      = "black"; }
   public  Figure( String c ) { this.color = c;       }
```

```
public abstract class Figure
{
   private String color;
   public  Figure()            { color      = "black"; }
   public  Figure( String c ) { this.color = c;       }

   public String getColor() { return color; }

   // abstract methods - no method body
   public abstract double findArea()      ;
   public abstract double findPerimeter() ;
   public abstract void   print()         ;
}
```

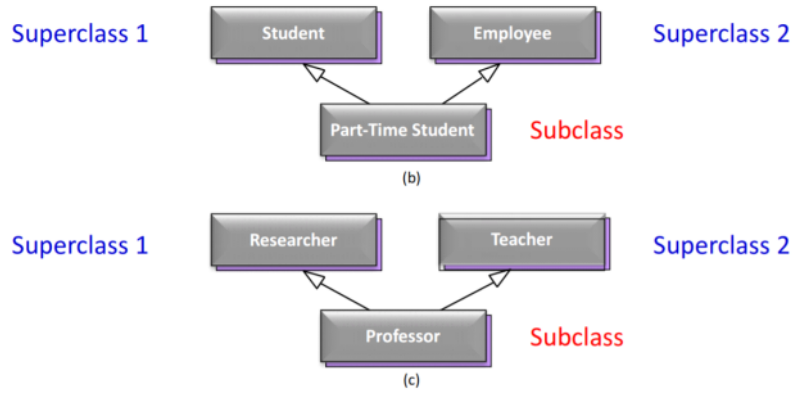
no method implementation

```
public class Rectangle extends Figure {
   private double width  ;
   private double height ;

   public Rectangle()
      { super()     ; this.width = 0 ; this.height = 0 ; }
   public Rectangle( String c , double w , double h )
       { super( c ) ; this.width = w ; this.height = h ; }
   public double getWidth()  { return width  ; }
   public double getHeight() { return height ; }

   // implementation of the abstract methods
   public double findArea()      { return width * height    ; }
   public double findPerimeter() { return 2*(width+height) ; }
   public void print() {
      System.out.println( "Rectangle print() method: " );
      System.out.println( "Width = "     + width
                        + "; Height = " + height           );
      System.out.println( "Perimeter = " + findPerimeter() );
      System.out.println( "Area = "      + findArea()      );
   }
```

Multiple inheritance and interfaces

- Java does not support multiple inheritance
- Java supports multiple interface

- An interface is like an abstract class, except it contains **only abstract methods** and **constants** (**with static final**).



```
public interface IOrdered
{
    public boolean precedes(Object other);

    public boolean follows(Object other);
}
```

```
public interface IShowablyOrdered extends IOrdered
{
    public void showOneWhoPrecedes();
}
```

```
public class Order implements IShowablyOrdered
{
    public boolean precedes(Object other) {….}
    public boolean follows(Object other) {….}
    public void showOneWhoPrecedes() {….}
```

- An abstract class is a real parent (base class); an interface is not a real parent (~~base class~~).
  type

Note: A Concrete class has implementation for all methods, i.e. NO abstract methods.

Abstract	Interface
May have some methods declared as `abstract`.	Can only have abstract methods.
May have `protected` properties and `static` methods.	Can only have `public` methods with no implementation.
May have `final` and non-final data attributes.	Limited to only constants (`static final`).

Both Abstract class and Interface CANNOT be instantiated with *new*, i.e.,
…… = new <AbstractClass>( ) ;
…… = new <Interface>( ) ;

```
public class SubClass1 extends SuperClass implements Interface1,
  Interface2, Interface3
```

U4: Error handling

Sunday, 23 August 2020 3:00 PM

- Try
  - Every method must first state the types of exceptions it can handle
- Throwing
  - When method detects an error or exception on a statement contained in it, it creates an exception object that contains information on the type of the exception, and the state of the program when the error occurred.
- Catching
  - When an exception is thrown, JVM looks for an exception handler that can catch and handle the exception
  - Exception handler must match the type of the exception thrown

```
public static void main( String[] args ) {
   int     i, numOfStudents;
  double  totalMarks = 0, avgMarks = 0 ;
   Scanner sc          = new Scanner( System.in );
   try {
      System.out.print( "Enter number of students: " );
      numOfStudents = sc.nextInt();
      if ( numOfStudents <= 0 )
        throw new Exception(
          "Error: no of students must not equal to 0!" );
      System.out.print( "Enter student marks: " );
      for ( i = 0 ;  i < numOfStudents ; i++ )
        totalMarks += sc.nextDouble() ;
      avgMarks = totalMarks / (double) numOfStudents ;
      System.out.println( "Average marks = " + avgMarks );
   }
   catch ( Exception e ) {
      System.out.println( e.getMessage() );
   }
   System.out.println( "End of program execution!" );
} }
```

Some Useful Exceptions (System Generate)

Exception (Predefined)	Description
ArithmeticException	This indicates division by zero or some kinds of arithmetic exceptions.
IndexOutOfBoundsException	This indicates that an array or string index is out of bound.
ArrayIndexOutOfBoundsException	This indicates that an array index is less than zero or greater than or equal to the array's length.
StringIndexOutOfBoundsException	This indicates that a string index is less than zero or greater than or equal to the string's length.
FileNotFoundException	This indicates that the reference to a file cannot be found.
IllegalArgumentException	This indicates that an improper argument is used when calling a method.
NullPointerException	This indicates that an object reference has not been initialised yet.
NumberFormatException	This indicates that illegal num format is used.

- Exception is the root class of all exceptions

```
public Exception()
public Exception( String message )
```

  - Instance methods

String getMessage(): Returns the message of the exception object.
String toString(): Returns a short description of the exception object.
void printStackTrace(): Print on screen a **trace** of all the methods that were called, leading up to the method that threw the exception.

- "at CreateException.method.."

- There are two types of exceptions
  - Checked exceptions
    - Can be analysed by complier
    - Can be caught by the method that threw that try/catch block
    - Possible to delay the handling of an exception when it is not clear how to handle it
  - Unchecked exceptions
    - Exceptions that belong to:
      - Any of the subclasses of class RuntimeException
      - Class Error
    - Not easy to be checked explicitly and are always avoided by programming
    - If not handled, Java's default exception handlers will take over

- Exception propagation
  - If an exception is thrown and not caught by the handlers after the try block it occurred, control is then transferred to the method that invoked the method that threw the exception

```
public class ExPropagation {
   public static void main( String[] args ) {
      System.out.println( "Start program execution" );
      ExPropagation exp = new ExPropagation();
      exp.method1();
      System.out.println( "End of program execution" );
   }
```

1

- Methods can throw more than one exception
  - Can declare more than one catch block
    - **Recommendation**: First put the catch blocks for the more **specific**, then the derived exceptions, and then the more **general** ones near the end.

- Creating your own exception classes

```
public class IntNonNegativeException extends Exception
{
  // constructors
  public IntNonNegativeException() {
    super( "Integer input is a negative number!! " );
  }

  public IntNonNegativeException( String message ) {
    super( message );
  }
}
```

with parameter

  - Must be derived from existing Exception class
  - Define only the constructors and call the default constructor using super

```
import java.util.Scanner ;
public class IntNonNegativeExceptionApp
```

```
import java.util.Scanner ;
public class IntNonNegativeExceptionApp
{
  public static void main( String[] args ) {
    IntNonNegativeExceptionApp sumEx
                = new IntNonNegativeExceptionApp() ;
    int      inputNum ;
    int      sum = 0 ;
    Scanner sc   = new Scanner( System.in );

    System.out.print( "Enter total no. of integers: " );
    int total = sc.nextInt();

    for ( int i = 0 ; i < total ; i++ ) {
      inputNum = sumEx.getInteger();
      sum += inputNum ;
    }
    System.out.println( "The sum of integers: " + sum );
  }
```

①

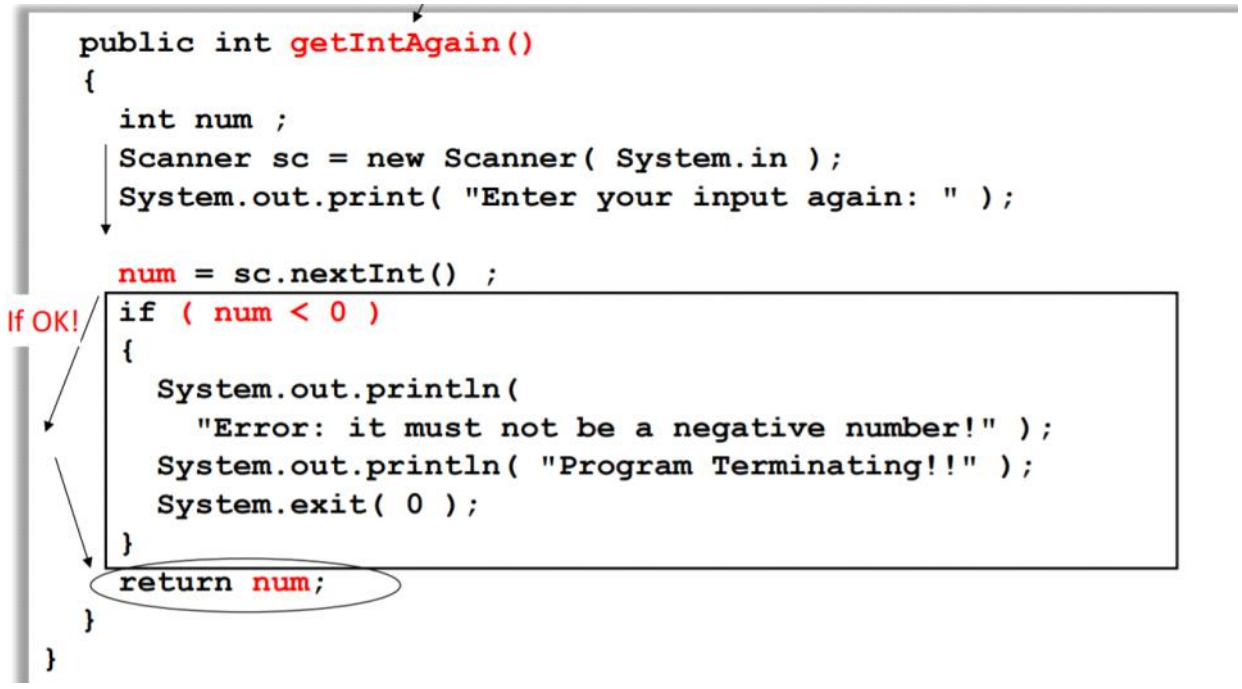
```
    public int getInteger() {
      int      num = 0 ;
      Scanner sc   = new Scanner( System.in ) ;
      try
      {
        System.out.print( "Enter the integer: " );
        num = sc.nextInt() ;
        if ( num < 0 )
          throw new IntNonNegativeException();
      }
      catch ( IntNonNegativeException e ) {
        System.out.println( e.getMessage() );
        num = getIntAgain() ;
      }
      return num ;
    }
```

Default constructor

Exception!! ②

If no more Exception

③


public int getIntAgain()
{
int num ;
Scanner sc = new Scanner(System.in);
System.out.print("Enter your input again: ");
num = sc.nextInt() ;
If OK!
if (num < 0)
{
System.out.println(
"Error: it must not be a negative number!");
System.out.println("Program Terminating!!");
System.exit(0);
}
return num;
}
}

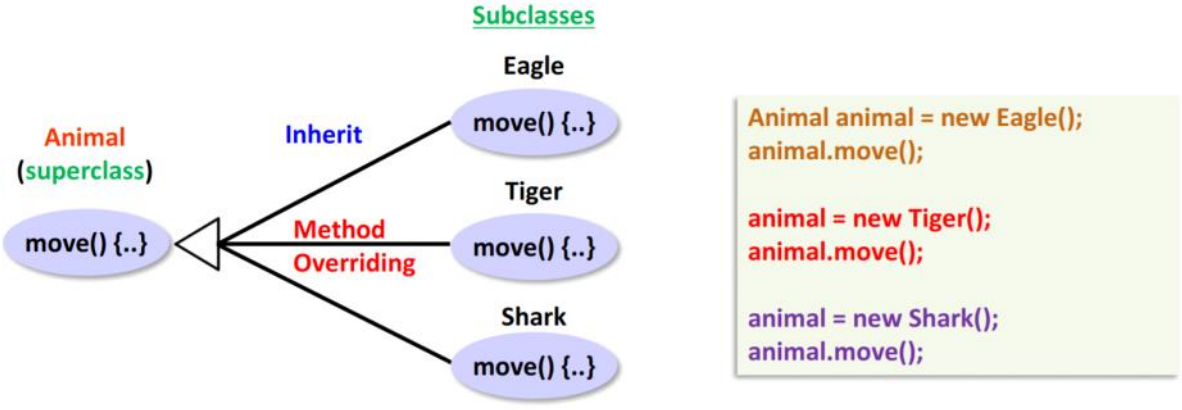
U5: Polymorphism

Friday, 4 September 2020 4:12 PM

- Overriding - a necessary tool for polymorphism.
- A subclass can override a method in the parent class by defining a method with exactly the same signature and return type.
- The subclass method can replace or refine the method in the parent class.

```
public class Person{
  int age ;
  String name;
// usual constructor, accessor mutator methods
…..
  public void printInfo() {
     System.out.println("Name is " + name);
     System.out.println("Age is " + age);
  }
  ……..
}
public class Employee extends Person{
  double salary ;
// usual constructor, accessor mutator methods
…….
  public void printInfo() {  // refine printInfo method
    System.out.println("Employee name is :" + getName()+
                       " and age is " + age );
    System.out.println("Salary is " + salary) ;
  }
}                  // replace printInfo method
```

method overriding




```
Animal animal = new Eagle();
animal.move();

animal = new Tiger();
animal.move();

animal = new Shark();
animal.move();
```

The Liskov Substitution Principle

- Barbara Liskov first wrote in 1988:

  "If for each object o_1 of type S there is an object o_2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o_1 is substituted for o_2 then S is a subtype of T."

- Paraphrased

  P → T o_2

  - Subtypes must be substitutable for their base types.

- Binding
  - Static binding (Early binding)
    - It occurs when the method call is "bound" at compile time
  - Dynamic binding (Late binding)
    - The selection of the method body to be executed is delayed until execution time (based on the actual object referred by the reference)
    - Execution time = run time

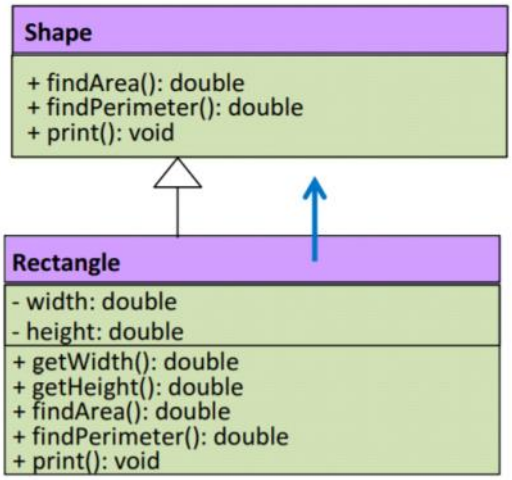
Example of Dynamic Binding

```
public class DynamicBindingTest {

    public static void main(String args[]) {
        Vehicle vehicle = new Car(); // here Type is vehicle but object will
                    // be Car
        vehicle.start();  //Car's start called because start() is overridden
             // method
    }
}

class Vehicle {
    public void start() {
        System.out.println("Inside start method of Vehicle");
    }
}

class Car extends Vehicle {
    // Override
    public void start() {
        System.out.println("Inside start method of Car");
    }
}

class Bus extends Vehicle {
    // Override
    public void start() {
        System.out.println("Inside start method of Bus");
    }
}
Output: Inside start method of Car
```

Java uses dynamic binding for all methods (except private, final, and static methods).

- Upcasting
  - When an object of a derived class is assigned to a variable of a base class (or any ancestor class)

```
Shape shape = new Rectangle(15,10));
```



- Downcasting
  - When a type cast is performed from a base class to a derived class (or from any ancestor class to any descendant class)

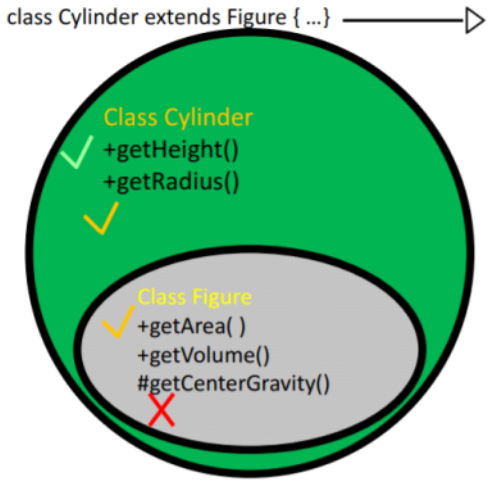
```
Shape shape = new Shape() ;

Rectangle rect = shape ;
// will have compilation error

Rectangle  rect = (Rectangle)shape ;
// explicit cast, accept by compiler
//will produce runtime error !!!!

int height = rect.getHeight();
// compiler OK
```

- **Downcasting** is very useful when you need to compare one object to another:

```
public class Person {
  private String name;
  private int age;
// usual constructor, accessor & mutator methods
…
  public boolean equals(Object anObject) //inherit fr. Object Class
  {
     if (anObject == null)
         return false;
    else if (!(anObject instanceof Person))
     /* else if (getClass( ) != anObject.getClass()) */
         return false;
     else
     {
        Person aPerson = (Person) anObject; // downcast
        return ( name.equals(aPerson.getName())
                 && (age == aPerson.getAge()));
     }
```

null String Object Person Student Human?

```
Object anObject = new String("Tom");
Person aPerson = (Person)anObject ;
```



Upcast

```
Figure fig = new Cylinder() ;
fig.getArea() ;
// compile & runtime both OK
```

Downcast

```
Figure fig = new Figure()
Cylinder cy = fig;  // compile NOK

Cylinder cy = (Cylinder) fig;
// explicit cast, compile OK,
// runtime NOK
cy.getHeight() ; // compile OK
```

<reference type> <reference name> = new <Object Type> ()

use at compile time to check — use at runtime to check

Allowed Assignments Between Superclass and Subclass Variables

- Superclass and subclass assignment rules
  - Assigning a superclass reference to a superclass variable is straightforward **Object ob = new Object()**
  - Assigning a subclass reference to a subclass variable is straightforward **String st = new String()**
  - Assigning a subclass reference to a superclass variable is safe because of the *is-a* relationship **Object st = new String()**
    - Referring to subclass-only members through superclass variables is a compilation error **st.charAt(0)**
  - Assigning a superclass reference to a subclass variable is a compilation error **String ob = new Object()**
    - Explicit downcasting can get around this error **String ob = (String)new Object()**

- Three ways of overriding
  - Method overriding
    - Methods of a subclass override the methods of a superclass
  - Method overriding (implementation) of the abstract methods
    - Methods of a subclass implement the abstract methods of an abstract class
  - Method overriding (implementation) through the Java interface
    - Methods of a concrete class implement the methods of the interface

```
interface IAnimal {
        void move();
        void speak();
}
class Cat implements IAnimal{
    public void move() {
        System.out.println("Cat
        moves....");
  }
    public void speak() {
        System.out.println("Meow !"); }
}
class Lion implements IAnimal {
    public void move() {
        System.out.println("Lion
        moves...");}
    public void speak() {
        System.out.println("ROAR !"); }
}
```

```
class CareTaker
{
  public void takeAWalk(IAnimal pet)
    {       pet.move();
            pet.speak();
     }

}

class  AnyClass {
…….
   CareTaker ct = new CareTaker();
       Cat cat = new Cat();
       Lion lion = new Lion();

       ct.takeAWalk(cat);
       ct.takeAWalk(lion);
……..
}
```

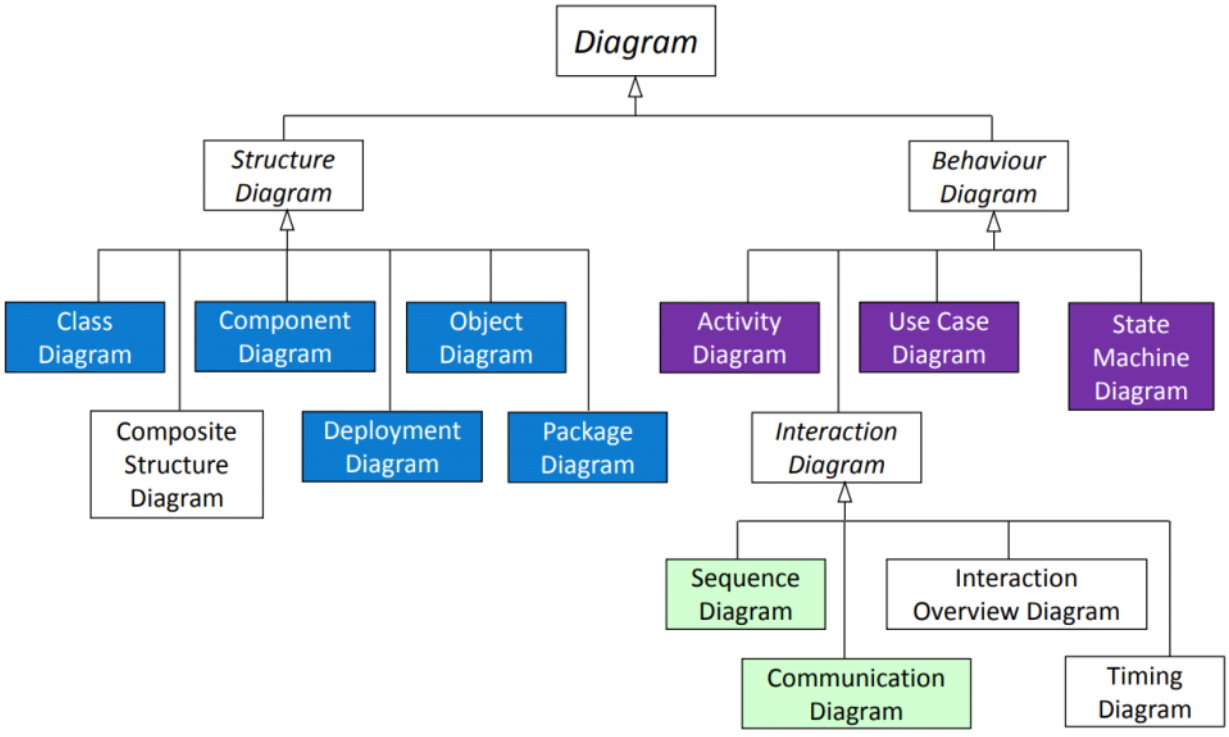
```
abstract class LivingThings {
      public  void grow( ) {
            System.out.println("LivingThings
            grow!"); }
     public abstract void speak( );
 }
 abstract class Mammal extends LivingThings {
    public void move( ) {
            System.out.println("Mammal move!"); }
     static
    public void grow() {
            System.out.println("Mammal  grow!"); }

    public void eat() {
            System.out.println("Mammal eat!"); }
    }
  class Dog extends Mammal { // concrete class
    public void speak( ) {
            System.out.println("Woof!"); }
    public  void move( ) {
    static System.out.println("Dog move!"); }
    public void grow( ) {
            System.out.println("Dog grow!"); }
  }
```

```
class House {
   public static void main(
          String[] args) {
      Dog mDog = new Dog( );
      mDog.speak( );
      mDog.move( );
      mDog.eat();
      mDog.grow();

      Mammal m = new Dog( );
      m.speak( );
       m.move( );
      m.eat();
      m.grow();


    }
  }

}
```

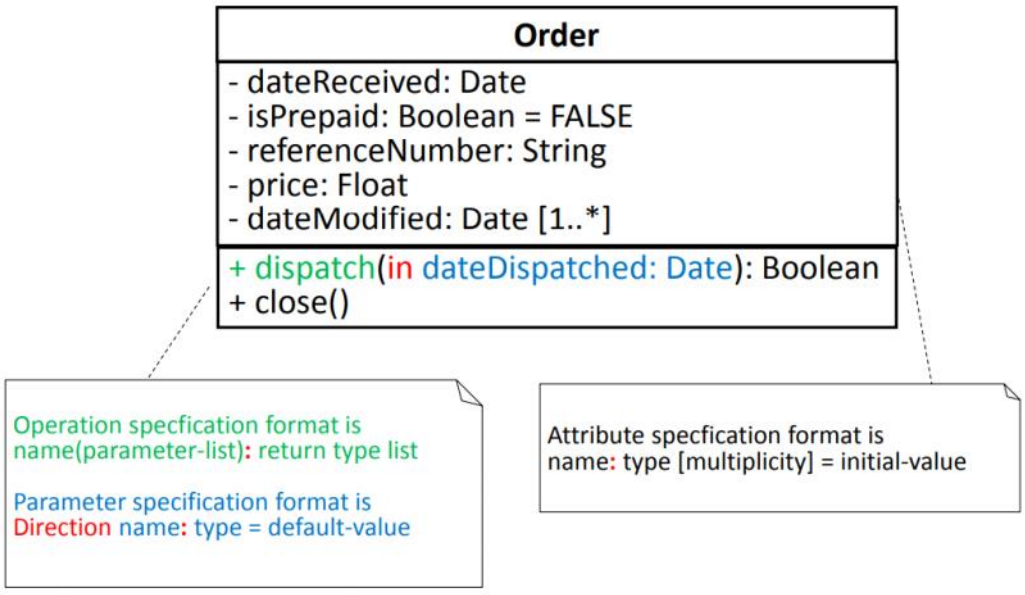
U6: UML diagram: Relationships - Class Diagram

Tuesday, 25 August 2020 12:00 PM

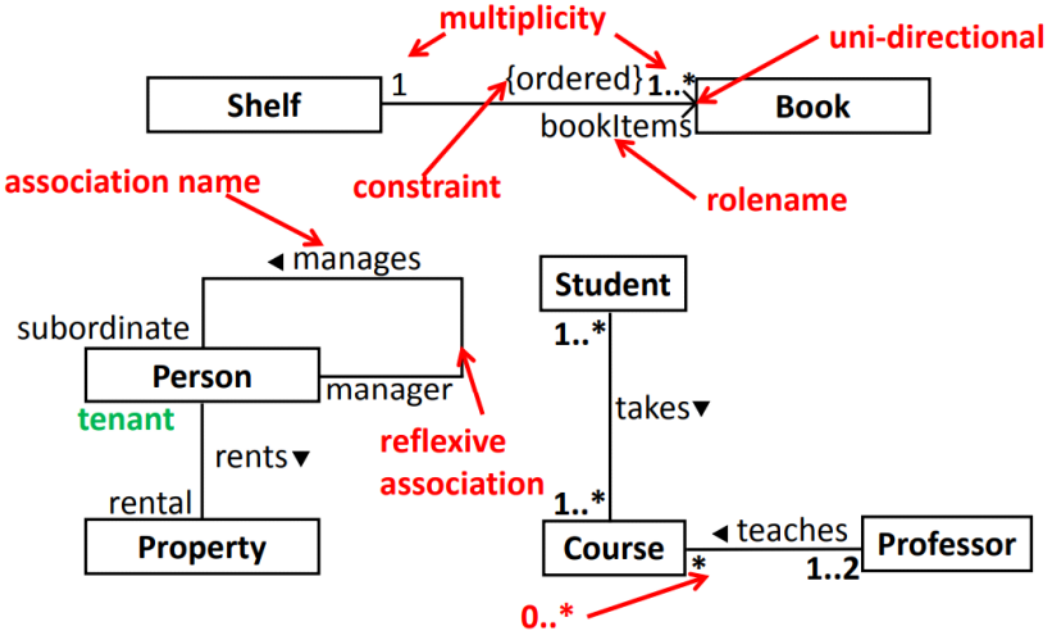
Reading Left to Right	Each A must be assigned to exactly one B	Each A must be assigned to one or many of B	Each A must be assigned to zero or one of B	Each A may be assigned any number of Bs
UML	A 1 B	A 1..* B	A 0..1 B	A * B
Martin/Odell (1st edition)	A || B	A |< B	A o| B	A o< B
Booch (2nd edition)	A 1 B	A 1..N B	A 0..1 B	A N B
Coad/Yourdon	A 1 B	A 1,m B	A 0.1 B	A 0,m B
Jacobson (unidirectional)	A [1] → B	A [1..M] → B	A [0..1] → B	A [0..M] → B
OMT	A — B	A 1+ ● B	A — ○ B	A — ● B



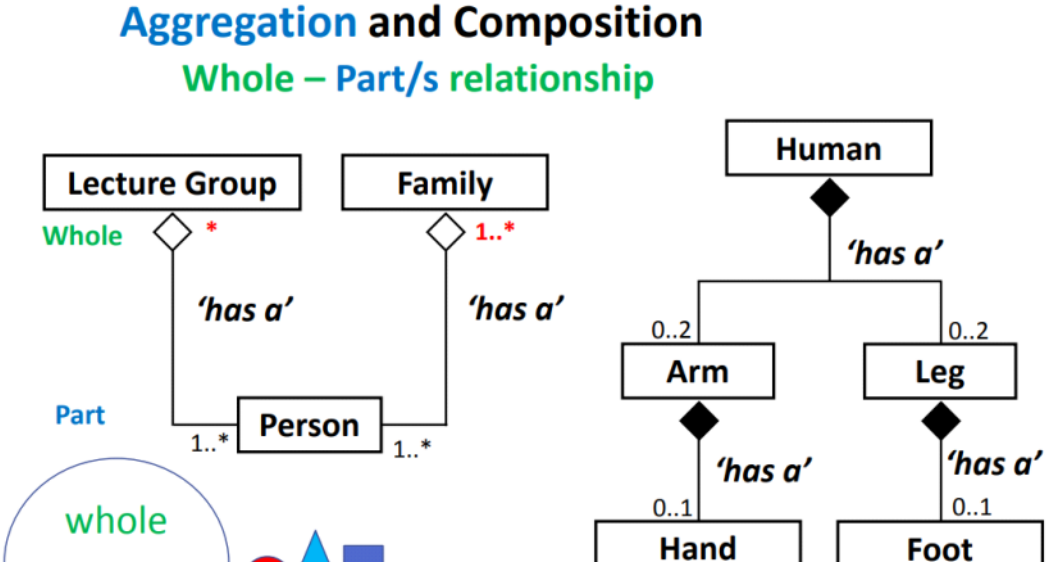
- Class diagram

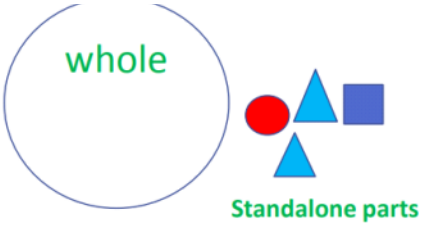


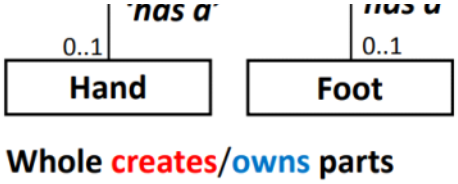
- Association



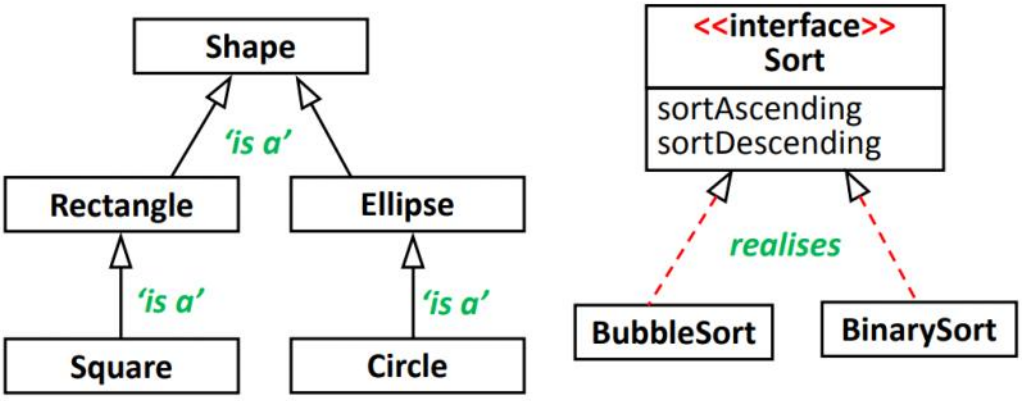
- Aggregation and composition



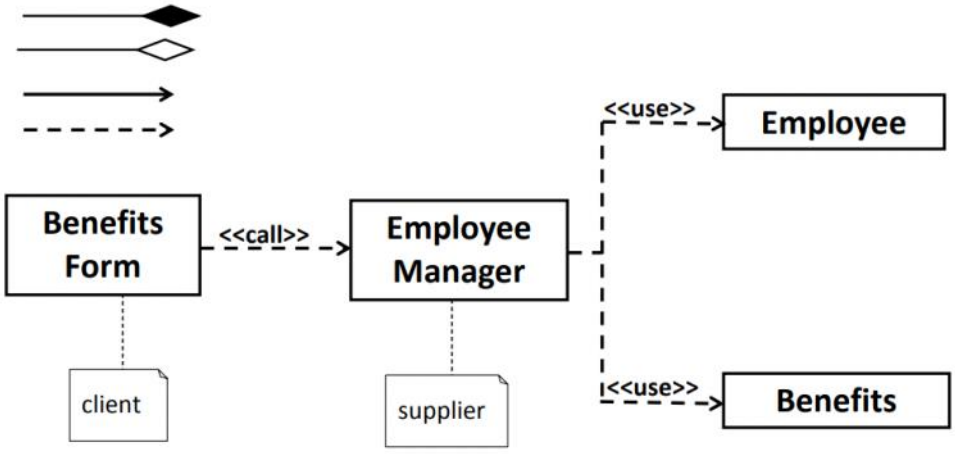




- Generalisation and interface realisation

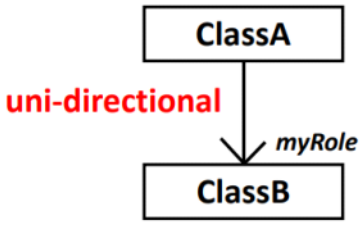


- Dependency

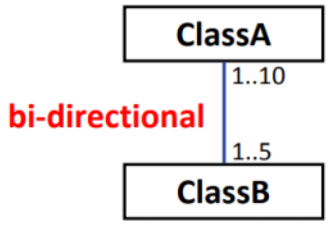


- UML class diagram to Java code

Association

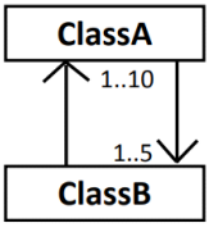



```
class ClassA {
    ClassB myRole ; // attribute
    ……..
    …..
}
```

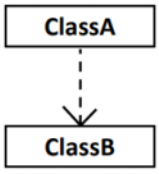



```
class ClassA {
    ClassB[ ] bObjs = new ClassB[ 5 ]; // attribute
    ……..
…….                bObjs[0] = new ClassB(this) ;
}
```

```
class ClassB {
   ClassA[ ] aObjs = new ClassA[ 10 ]; // attribute
    ……..
…….public ClassB(ClassA a) { aObjs[0] = a ; …}
}
```



Dependency

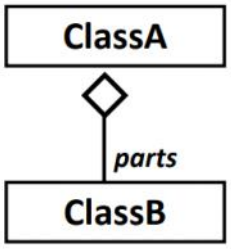



```
class ClassA {
    …….
    ClassB doSomeThing(…) { // as method return type
    ……..
    …..
    }
}
```

```
class ClassA {
    …….
    <ReturnType> doSomeThing(ClassB b, …) { // as method parameter type
    ……..
    …..
    }
}
```

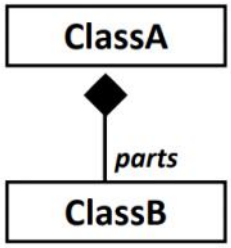
```
class ClassA {
    …….
    <ReturnType> doSomeThing(……) {
          ClassB b = new ClassB() ; // as variable type within the method (local)
          …..
    }
}
```

Aggregation




```
class ClassA {
    Vector parts = new Vector() ; // attribute
    ……. // parts stores a collection of ClassB objs
    public void addClassB(ClassB b ) {
        ……..// added through reference
          parts.add(b) ;
    }
    ……..
} // use of Vector class is just to show dynamic (*) Collection,
// can be ArrayList, etc OR array[ ] (if size is fixed)
```

Composition




```
class ClassA {
    Vector parts = new Vector(); // attribute
    ……..
    public ClassA( ) {
        ClassB b1 = new ClassB() ; // created in class
          ClassB b2 = new ClassB() ;
          parts.add(b1) ;
          parts.add(b2) ;
    }
    ……..
}
```

- visibility:
  - \+ public
  - # protected
  - \- private
  - ~ package (default)
- underline static methods

For Java

Entity

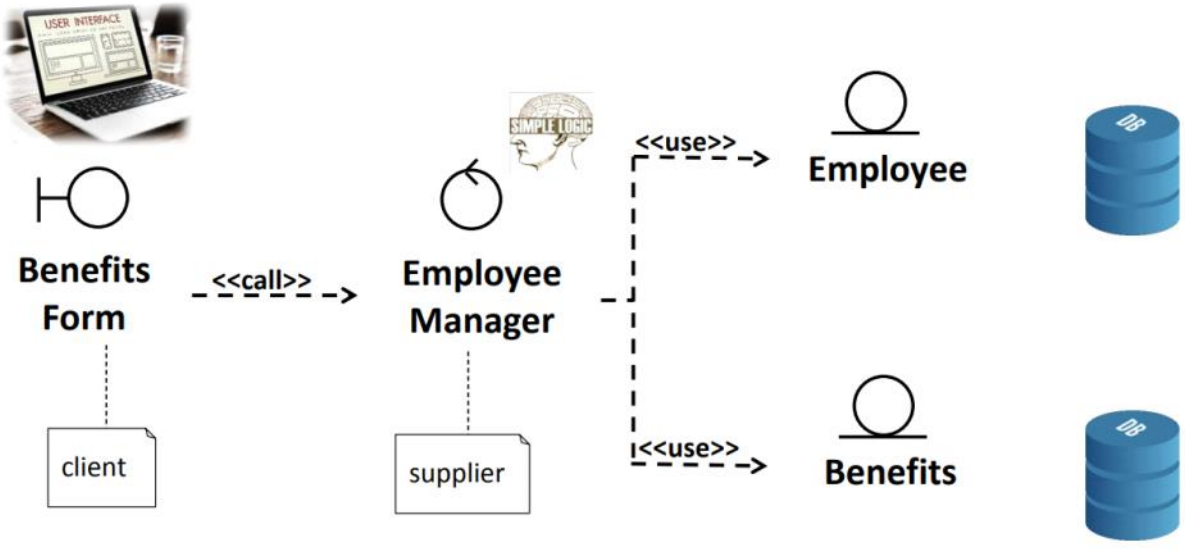
- Example: Student, Course, Group

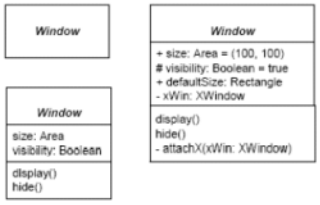
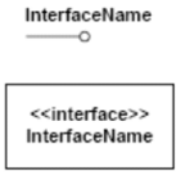
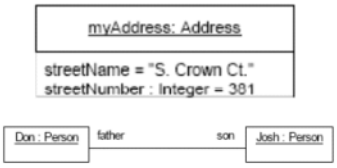
Boundary

- Example: xxxUI, xxxForm, xxxInterface
- Example: InvoiceForm

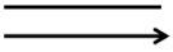
Control

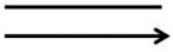
- Example: xxxMgr, xxxCtrl, xxxController
- Example: CourseMgr

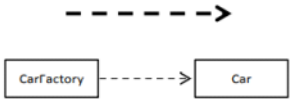


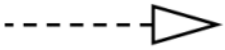
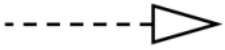
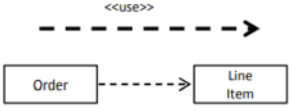
Node Type	Notation	Description
Class		Specifies a classification of objects, as well as, the features that characterise the structure and behaviour of those objects.
Interface		An interface is a kind of classifier that represents a declaration of a set of coherent public features and obligations. An interface specifies a contract; any instance of a classifier that realises that the interface must fulfill that contract.
Instance Specification		Instances of any classifier can be shown by prefixing the classifier name by the instance name followed by a colon and underlining the complete name string. An instance specification whose classifier is an association describes a link of that association.

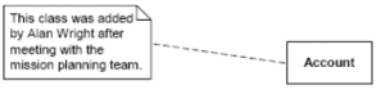
Path Type	Notation	Description
Aggregation		An aggregation represents a whole/part relationship.

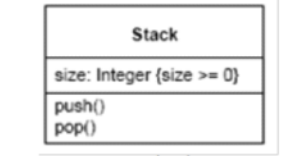
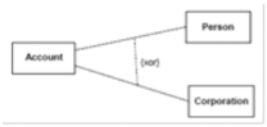
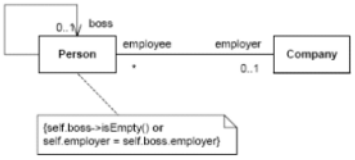
Association		An association specifies a semantic relationship that can occur between typed instances.
		
		An open arrowhead on the end of an association indicates the end is navigable. A small x on the end of an association indicates the end is not navigable.
		
		Notations that can be placed near the end of the line as follows:
		• name – name of association end
		• multiplicity
		• property string enclosed in curly braces
		  ○ {ordered} to show that the end represents an ordered set.
		  ○ {bag} to show that the end represents a collection that permits the same element to appear more than once.
		  ○ {sequence} or {seq} to show that the end represents a sequence

Link		An instance of an association

Path Type	Notation	Description
Composition		Composite aggregation is a strong form of aggregation that requires a part instance be included in at most one composite at a time. If a composite is deleted, all of its parts are normally deleted with it. Deleting an element will also result in the deletion of all elements of the subgraph below that element.
Dependency	 **instantiate dependency**	A dependency is a relationship that signifies that a single or a set of model elements requires other model elements for their specification or implementation. The modification of the supplier may impact the client model elements.
Generalisation		A generalisation is a taxonomic relationship between a more general classifier and a more specific classifier. Generalisation hierarchies must be directed and acyclical.

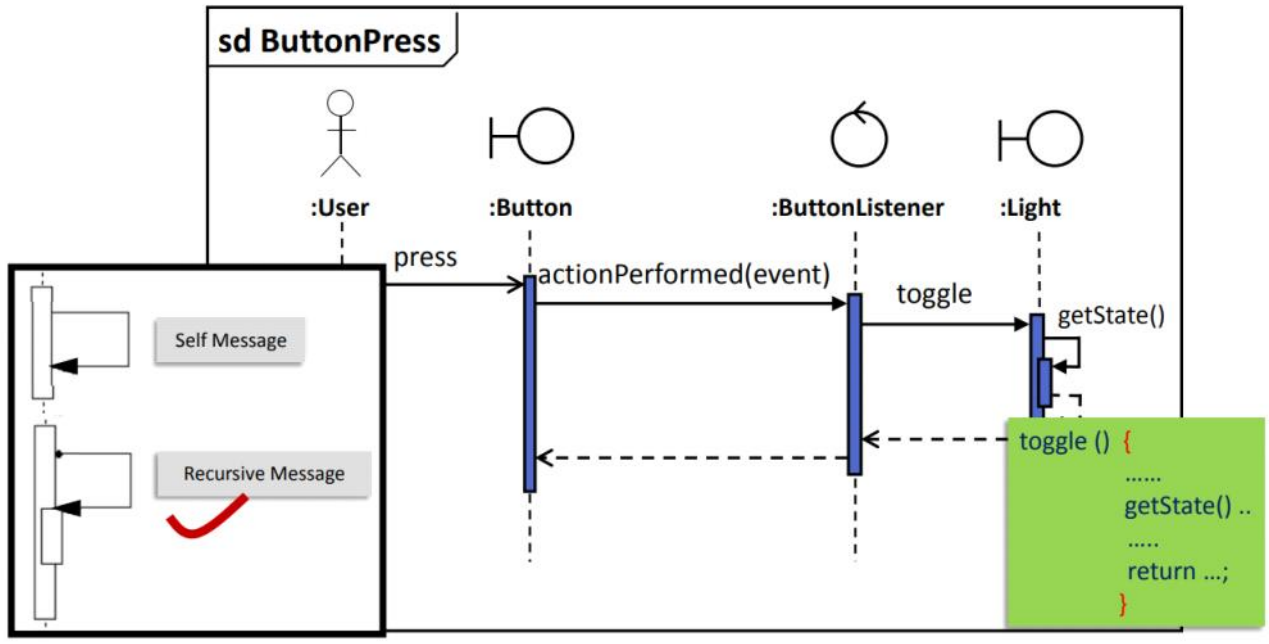
Path Type	Notation	Description
Interface Realisation		A specialised realisation relationship between a Classifier and an Interface. This relationship signifies that the realising classifier conforms to the contract specified by the Interface. A classifier may implement a number of interfaces. The set of interfaces implemented by the classifier are its provided interfaces and signify the set of services the classifier offers to its clients.
Realisation		Signifies that the client set of elements are an implementation of the supplier set, which serves as the specification.
Usage	 An Order class requires the Line Item class for its full implementation.	A usage is a relationship in which one element requires another element (or set of elements) for its full implementation or operation.

Type	Notation	Description
Association Class		Defines a set of features that belong to the relationship itself and not to any of the classifiers.
Comment		A comment is a textual annotation that can be attached to a set of elements. A comment gives the ability to attach various remarks to elements. A comment carries no semantic force, but may contain information that is useful to a modeler. The connection to each annotated element is shown by a separate dashed line.

Type	Notation	Description
Constraint	 **Constraint attached to an attribute**  **{xor} constraint**  **Constraint in a note symbol**	A constraint is a condition or restriction expressed in natural language text or in a machine readable language for the purpose of declaring some of the semantics of an element. One predefined language for writing constraints is OCL.

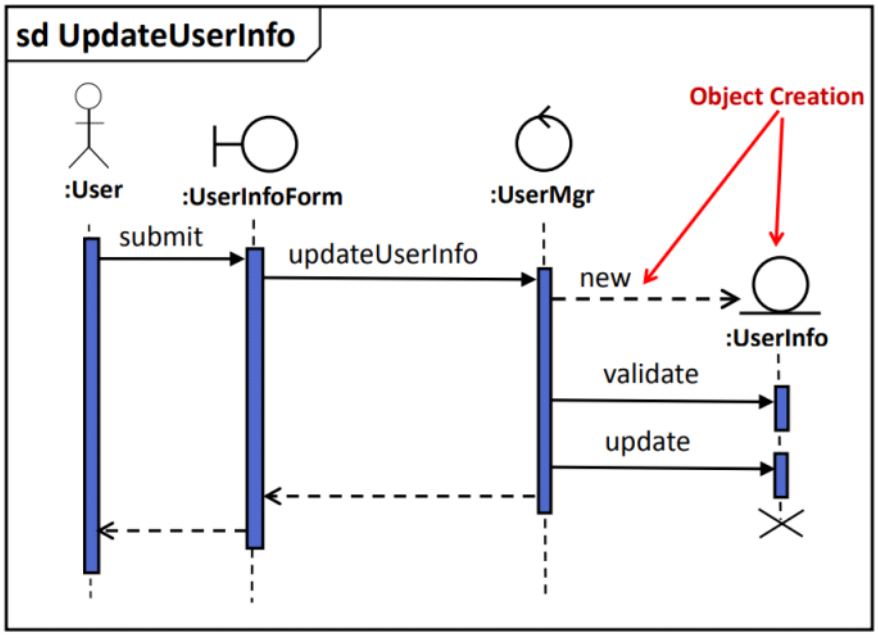
U7: UML Model: Object

Monday, 9 November 2020 10:15 AM



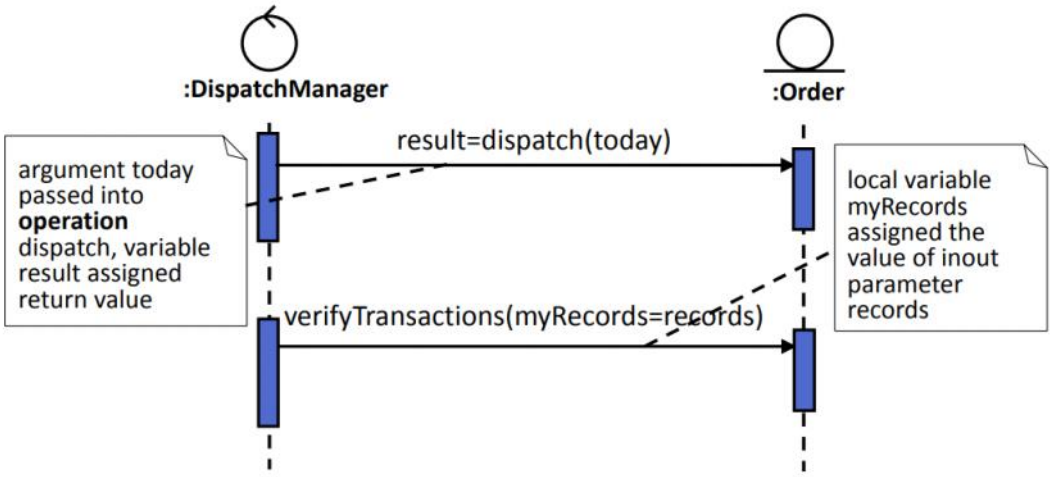
6

- Creating and deleting participants



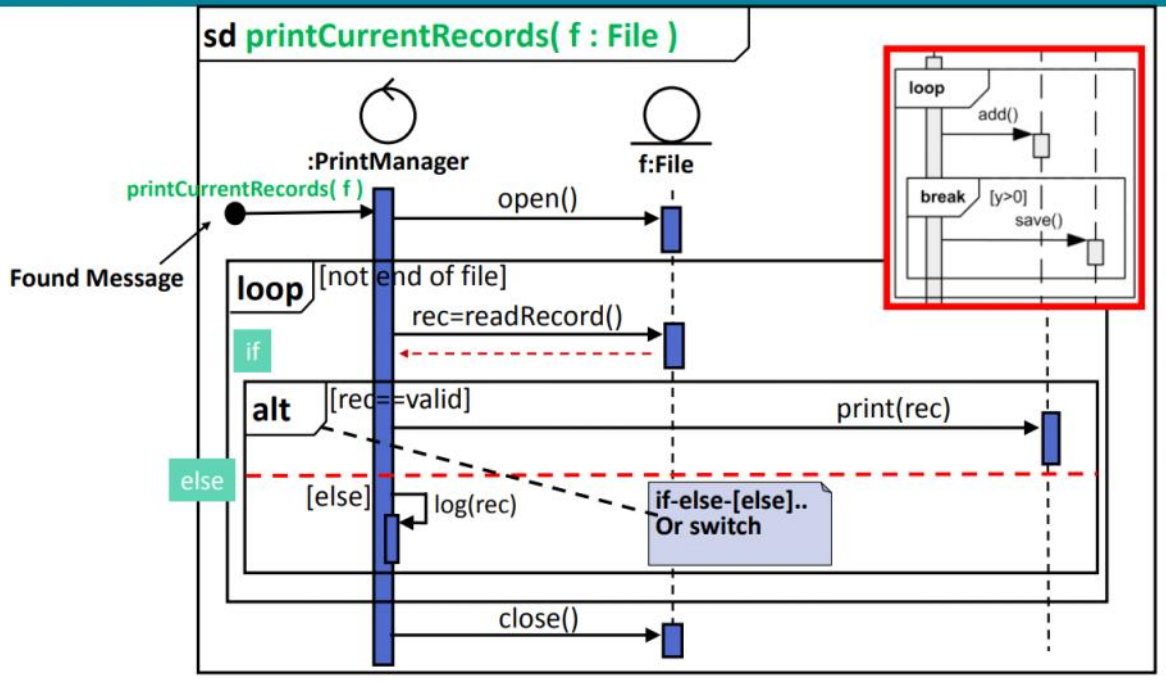
Message Specification

Refer to video at: 06:22

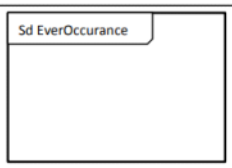
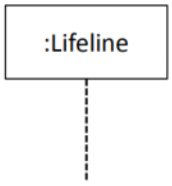


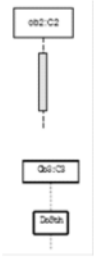
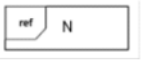
Order
dispatch(in dateDispatched: Date): Boolean verifyTransactions(inout records: List<Transaction>)

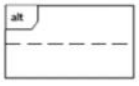
Interaction Fragments



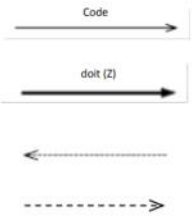
A **Sequence Diagram** describes an Interaction by focusing on the sequence of Messages that are exchanged, along with their corresponding Occurrence Specifications on the Lifelines.

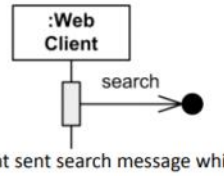
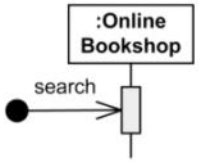
Node Type	Notation	Description
Frame		The notation shows a rectangular frame around the diagram with a name in a compartment in the upper left corner. The keyword sd followed by the Interaction name and parameters is in a pentagon in the upper left corner of the rectangle.
Lifeline		A lifeline represents an individual participant in the Interaction. A Lifeline is shown using a symbol that consists of a rectangle forming its "head" followed by a vertical line (which may be dashed) that represents the lifetime of the participant. The Lifeline head has a shape that is based on the classifier for the part that this lifeline represents. Often the head is a white rectangle containing the name. Information identifying the lifeline is displayed inside the rectangle in the following format: [name] : <class_name> (:class_name is mandatory but name is optional).

Node Type	Notation	Description
Execution Specification (Activation)		An ExecutionSpecification is a specification of the execution of a unit of behaviour or action within the Lifeline. The duration of an ExecutionSpecification is represented by the start ExecutionOccurrenceSpecification and the finish ExecutionOccurrenceSpecification.
InteractionUse		It is common to want to share portions of an interaction between several other interactions. An InteractionUse allows multiple interactions to reference an interaction that represents a common portion of their specification. The InteractionUse is shown as a CombinedFragment symbol where the operator is called ref.

Node Type	Notation	Description
Combined Fragment		A combined fragment is defined by an interaction operator and corresponding interaction operands. Through the use of CombinedFragments the user will be able to describe a number of traces in a compact and concise manner. **alt** designates that the CombinedFragment represents a choice of behaviour. At most one of the operands will be chosen. The chosen operand must have an explicit or implicit guard expression that evaluates to true at this point in the interaction. **opt** defines condition to a single call - the call will execute only if the supplied condition is true . Equivalent to an alt with only one trace. **par** defines that the calls within the fragment run in parallel. **critical** designates that the CombinedFragment represents a critical region, where the traces of the region cannot be interleaved by other OccurrenceSpecifications (on those Lifelines covered by the region). **loop** operand will iterate minimum the 'minint' number of times (given by the iteration expression in the guard) and at most the 'maxint' number of times. After the minimum number of iterations have executed and the Boolean expression is false the loop will terminate. **break** represents a breaking or exceptional scenario that is performed instead of the remainder of the enclosing interaction fragment.

Path Type	Notation	Description
Destruction Event		Destruction of the instance described by the lifeline containing the OccurrenceSpecification that references the destruction event.

Message		A Message defines a particular communication between Lifelines of an Interaction.
		
		A message is shown as a line from the sender message end to the receiver message end.
		
		Messages come in different variants depending on what kind of Message they convey. Here we show an asynchronous message, a call and a reply. These are all complete messages.
		 
		• Asynchronous Messages have an open arrow head
		• Synchronous Messages typically represent operation calls and are shown with a filled arrow head.
		• The reply message from a method has a dashed line.
		• Object creation Message has a dashed line with an open arrow.
		
		Examples of syntax:
		mymessage(14, - , 3.14, “hello”) // second argument is undefined
		v=mymsg(16, variab):96 // this is a reply message carrying the return value 96 assigning it to v
		mymsg(myint=16) // the input parameter ‘myint’ is given the argument value 16

Type	Notation	Description
Lost Message	 Web Client sent search message which was lost.	**Lost Message** is a message where the sending event is known, but there is no receiving event. It is interpreted as if the message never reached its destination. Lost messages are denoted with as a small black circle at the arrow end of the message.
Found Message	 Online Bookshop gets search message of unknown origin.	**Found Message** is a message where the receiving event is known, but there is no (known) sending event. It is interpreted as if the origin of the message is outside the scope of the description. This may for example be noise or other activity that we do not want to describe in detail. Found messages are denoted with a small black circle.

U8: Modelling OO Applications

Monday, 9 November 2020 10:15 AM

U9: Design Principles

Monday, 9 November 2020 10:15 AM

U10: OO Concepts in C++

Friday, 30 October 2020 3:51 PM

- **Header files (.h)** is used for function declarations: declare the interfaces.
- **Source code (.cpp)** is used for defining class/function implementation.

```
// in myclass.h
class MyClass {
   private :
      int value ;
   public:
      void foo() ;
      int evaluate()  ;
};
```

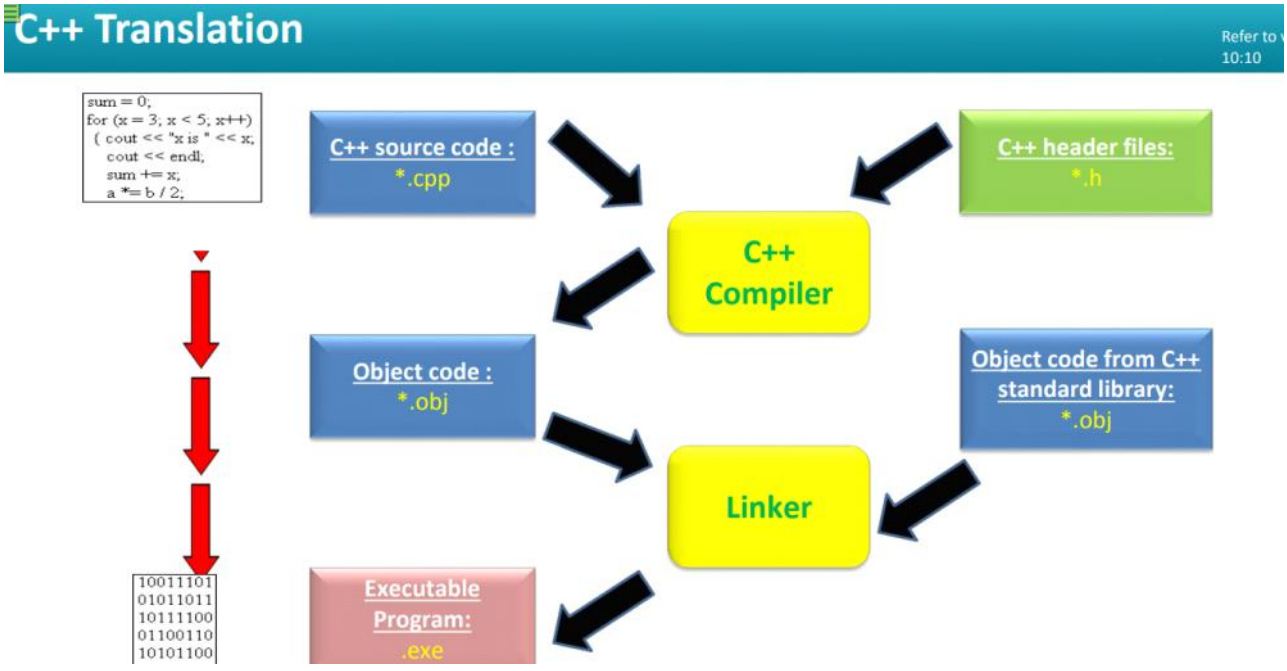
Interface Declaration (header)

```
// in myclass.cpp
#include "myclass.h"

void MyClass::foo() {
  ....// implementation
}
int MyClass:: evaluate() {
   .....
}
```

Implementation (implementation source code)

```
// in main.cpp
#include "myclass.h"

int main() {
    MyClass a;
    ......// usage
   return 0;
}
```

Using the Class



- Scope operator (::)

1) To access a global variable when there is a local variable with same name:

```cpp
// C++ program to show that we can access a global variable
// using scope resolution operator :: when there is a local
// variable with same name
#include<iostream>
using namespace std;

int x;  // Global x

int main()
{
  int x = 10; // Local x
  cout << "Value of global x is " << ::x;
  cout << "\nValue of local x is " << x;
  return 0;
}
```

Output:

```
Value of global x is 0
Value of local x is 10
```

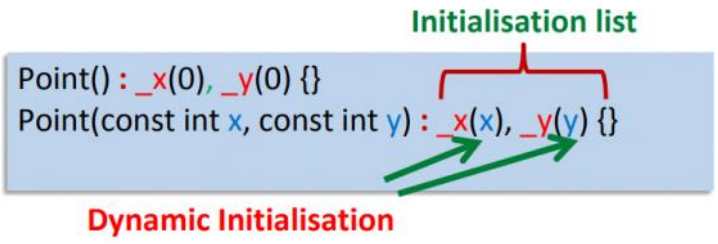
2) To define a function outside a class.

```cpp
// C++ program to show that scope resolution operator :: is used
// to define a function outside a class
#include<iostream>
using namespace std;

class A
{
public:

   // Only declaration
   void fun();
};

// Definition outside class using ::
void A::fun()
{
   cout << "fun() called";
}

int main()
{
   A a;
   a.fun();
   return 0;
}
```

```
fun() called
```

- Initialisation

};



- Destructor

```
~Point() { /* do clean up */ } // Destructor
//prefixed by a tilde (~ ) of the defining class
   .....
```

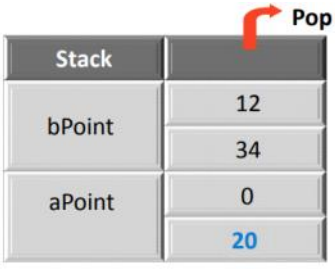
```
    ~Point() { /* do clean up */ } // Destructor
    //prefixed by a tilde (~ ) of the defining class
      .......
};
```

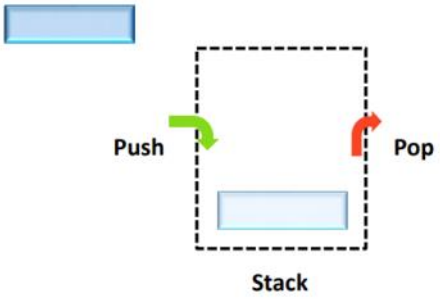
Ensure that the allocated memory is released!

- Object creation

Method 1 (stack)

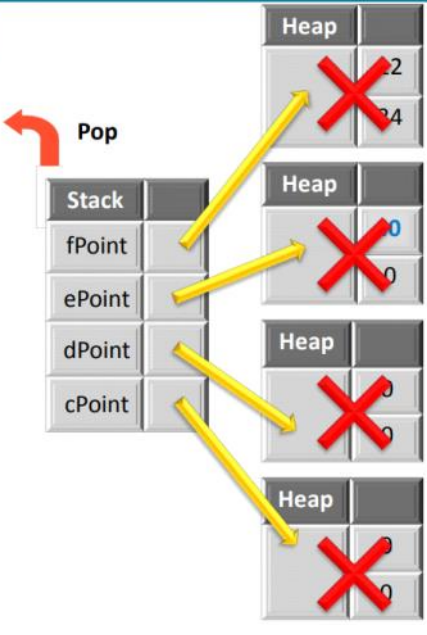
```
Point  aPoint;  // Point::Point()   Note : without ( )
Point  bPoint(12, 34); // Point::Point(const int, const int)
   aPoint.setY(20);
   int  y = aPoint.getY() ;
```





Method 2 (heap)

```
Point  *cPoint = new Point();  // Point::Point()
Point  *dPoint = new Point();  // Point::Point()
Point  *ePoint ;
    ePoint =  new Point();       // Point::Point()
Point  *fPoint = new Point(12, 34); // Point::Point(const int,
                                                  // const int)
ePoint->setX(40) ;
int x  = ePoint->getX() ;
delete  cPoint, dPoint, ePoint, fPoint ;
```



- Class inheritance

General Form :

class *derived-class-name* : [visibility-mode] *base-class-name*

{

.....

};

visibility-mode : [optional] private / public

private : privately inherited (default)

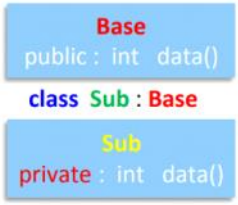
'public members' of base class become

'private members' of derived class.

public : publicly inherited

'public members' of base class remain

'public members' of derived class.




```
class Point3D : public Point {
   int _z;
public:
   Point3D() { setX(0); setY(0); _z = 0; }
   Point3D(const int x, const int y, const int z) {
        Point:: setX(x);   setY(y);   _z = z;
  }
   ~Point3D()        { /* Nothing to do */ }
   int getZ() { return _z; }
   void setZ(const int val) { _z = val; }
};
```

```
class Point3D : public Point {
   int _z;
public:
   Point3D( const int x, const int y, const int z) : Point(x, y)
        { _z = z; }
.....
};
```

Using Base class constructor

: Point(x, y) , _z(z) { }

Specify the desired base constructors after a **single colon** just before the body of constructor.

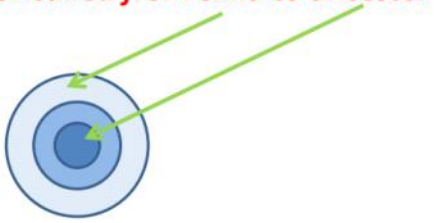
Class Multiple Inheritance

```
class DrawableString : public Point, public DrawableObject
{
.....
}
```

- Destruction

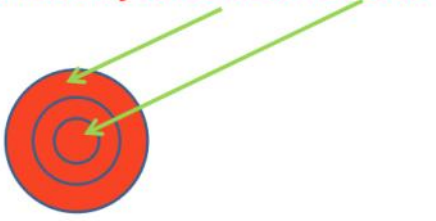
- ***Destruction***
- If an object is destroyed, for example by leaving its definition scope, the destructor of the corresponding class is invoked. If this class is derived from other classes their destructors are also called, leading to a recursive call chain.

Destructor called from child to ancestor



- ***Destruction***
- If an object is destroyed, for example by leaving its definition scope, the destructor of the corresponding class is invoked. If this class is derived from other classes their destructors are also called, leading to a recursive call chain.

Destructor called from child to ancestor



- Overloading

- **Just like in Java!**

```
int Add(int nX, int nY);   // integer version
double Add(double dX, double dY);   // floating point version
int Add(int nX, int nY, int nZ)

int GetRandomValue();
double GetRandomValue();
```

NOT Overloading

- A **default parameter** is a function parameter that has a default value provided to it. Example,

```
void PrintValues(int nValue1, int nValue2=10)
```

- **Rules**:
  1) All default parameters must be the **rightmost** parameters. The following is not allowed:
     void PrintValue(int nValue1=10, int nValue2); // not allowed
  2) The **leftmost** default parameter should be the one most likely to be changed by the user.

  ~~PrintValues(, ,3) ??~~
- **Constraints:**
  - Do NOT count towards the parameters that make the function unique. Example:

    PrintValues(3) ???

    void PrintValues(int nValue); **NOT ALLOWED !!**

    void PrintValues(int nValue1, ~~int nValue2=20~~);

- Ampersand

Let's start with the good old, better-known usages:

- `&` to declare a reference to a type
- `&` to get the address of a variable
- `&` as a bit-wise operator
- `&&` in a conditional expression

Use `&` to declare a reference to a type

If you use `&` in the left-hand side of a variable declaration, it means that you expect to have a reference to the declared type. It can be used in any type of declarations (local variables, class members, method parameters).

```
std::string mrSamberg("Andy");
std::string& theBoss = mrSamberg;
```

This doesn't just mean that both `mrSamberg` and `theBoss` will have the same value, but they will actually point to the same place in the memory. You can

When using it on the right-hand side of a variable, it's also known as the "address-of operator". Not surprisingly if you put it in front of a variable, it'll return its address in the memory instead of the variable's value itself. It is useful for pointer declarations.

```
std::string mrSamberg("Andy");
std::string* theBoss;

theBoss = &mrSamberg;
```

Use `&` as a bitwise operator

It is the bitwise AND. Its an infix operator taking two numbers as inputs and doing an `AND` on each of the bit pairs of the inputs. Here is an example. 14 is represented as *1110* as a binary number and 42 can be written as *101010*. So *1110* (14) will be zero filed from the left and then the operation goes like this.

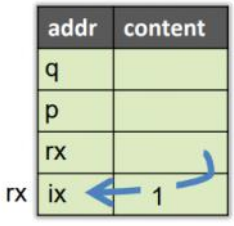
	32	16	8	4	2	1
14	0	0	1	1	1	0
42	1	0	1	0	1	0
14&42=10	0	0	1	0	1	0

Use `&&` for declaring rvalue references

- **Reference**
  - "alias" to "real" variable or object
  - Ampersand (**&**) is used to define a reference
  - Cannot be NULL eg, int &r ;
  - Example,

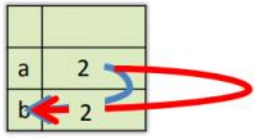
```
int ix;    /* ix is "real" variable */
int  &rx = ix;   /* rx is "alias" for ix */
ix = 1;   /* also rx == 1 */
rx = 2;  /* also ix == 2 */
int *p  = &ix ; // addressOf ix assigned to pointer p.
int &q = *p ;  // dereference pointer p and assign to q as alias.
```

```
int q = *p ;
```




```
#include <iostream>
using namespace std ;

void addIt(int a) { a += a ; }
void doubleIt(int &a) { a *= 2 ; }
void tripleIt(int *a) { *a *= 3 ; }

int main() {
    int b = 2 ;
    cout << "b is now " << b << endl;
        addIt(b) ;
    cout << "after addIt(int a), b is " << b << endl;
        doubleIt(b) ;
    cout << "after doubleIt(int &a), b is " << b << endl;
        tripleIt(&b) ;
    cout << "after tripleIt(int *a), b is " << b << endl;
    // cin >> b;
}
```

OUTPUT:
b is now 2
after addIt(int a), b is 2
after doubleIt(int &a), b is 4
after tripleIt(int *a), b is 12



Pointers: A pointer is a variable that holds memory address of another variable. A pointer needs to be dereferenced with * operator to access the memory location it points to.

References : A reference variable is an alias, that is, another name for an already existing variable. A reference, like a pointer, is also implemented by storing the address of an object. A reference can be thought of as a constant pointer (not to be confused with a pointer to a constant value!) with automatic indirection, i.e the compiler will apply the * operator for you.

- Pointers and references differ in:
  - Initialisation
  - Reassignment (pointers can be reassigned, references cannot)
  - Memory address (a pointer has its own memory address, references share the address)
  - NULL value (pointers can be assigned NULL directly, references cannot)
  - Indirection (Pointers can have extra pointers offering another level of indirection, references can only have one level)
  - Arithmetic operations (Can be performed on pointers, whereas for references: take the address of an object pointed by a reference and do pointer arithmetic)

- Polymorphism

- **Virtual**
  - To force method evaluation to be based on object type rather than reference type. [<ref type> <name> = new <obj type>(..)]
  - Without **virtual** => non polymorphic (no dynamic binding)
  - **Example : virtual** void area() { cout << "....." << endl ; }
  - **Virtual** function magic only operates on pointers(*) and references(&).
  - If a method is declared **virtual** in a class, it is **automatically virtual** in all **derived** classes.
- **Pure method => abstract** method (pure virtual)
  - By placing **"= 0"** in its declaration
  - **Example : virtual** void area() **= 0** ; // abstract method
  - **The class becomes an abstract** class

I saw that some function in C++ was declared as

```
virtual const int getNumber();
```

But what is the difference if the function is declared as the following?

```
const virtual int getNumber();
```

- No difference

- However,

```
virtual const int getNumber();
virtual       int getNumber() const;
```

In the first method, `const` refers to the returned value of type `int`.

In the second method, `const` refers to the object the method is called on; that is, `this` will have type `T const *` inside this method, - you will be able to call only `const` methods, modify only `mutable` fields and so on.

```
#include <iostream>

using namespace std ;

class Shape {
    public :
    Shape()  { }
    virtual void area() { cout << "undefined" << endl ;}
    void name() { cout << " a shape" << endl ; }
};

class Rectangle : public Shape {
   private :
     int _length ;
     int _height ;
   public :
     Rectangle(int x, int y) : _length(x) , _height(y) { }
     void area() { cout << "area is " << _length * _height << endl ; }
     void name() { cout << " a Rectangle " << endl ; }
};
```

```
int main() {
  Rectangle rect(10,20);

  Shape *shapePtr = &rect ;
  Shape &shapeRef = rect ;
  Shape shapeVal = rect ;

  shapePtr->area() ;
           shapeRef.area() ;
  shapeVal.area() ;
}
Output :
area is 200
area is 200
undefined
```

```
#include <iostream>

using namespace std ;

class Shape {
    public :
    Shape()  { }
    virtual void area() { cout << "undefined" << endl ;}
    void name() { cout << " a shape" << endl ; }
};

class Rectangle : public Shape {
   private :
     int _length ;
     int _height ;
   public :
     Rectangle(int x, int y) : _length(x) , _height(y) { }
     void area() { cout << "area is " << _length * _height << endl ; }
     void name() { cout << " a Rectangle " << endl ; }
};
```

```
int main() {
  Rectangle rect(10,20);

  Shape *shapePtr = &rect ;
  Shape &shapeRef = rect ;
  Shape shapeVal = rect ;
  Shape shapeDeref = *shapePtr;
  shapePtr->area() ;
           shapeRef.area() ;
  shapeVal.area() ;
}shapeDeref.area() ;
Output :
area is 200
area is 200
undefined
```

- Pure method: pointers or references to be used if abstract class is to be used as parameter type

```
#include <iostream>

using namespace std ;

class Shape {  // abstract class
    public :
    Shape()  { }
    virtual void area() = 0 ;  // pure method
    void name() { cout << " a shape" << endl ; }
};

class Rectangle : public Shape {
   private :
     int _length ;
     int _height ;
   public :
     Rectangle(int x, int y) : _length(x) , _height(y) { }
     void area() { cout << "area is " << _length * _height << endl ; }
     void name() { cout << " a Rectangle " << endl ; }
};
```

```
void showArea(Shape* s) { s->area();}
      showArea(&rect) ; // to call func
```

```
void showArea(Shape& s) { s.area() ;}
      showArea(rect) ;  // to call func
```

```
int main() {
  Rectangle rect(10,20);

  Shape *shapePtr = &rect ;
  Shape &shapeRef = rect ;
  Shape shapeVal = rect ;

  shapePtr->area() ;
           shapeRef.area() ;
  shapeVal.area() ;
}
```

- Ensure that the correct function is called for an object, regardless of reference used for the function call
- Used to achieve runtime polymorphism

Rules for Virtual Functions

1. Virtual functions cannot be static and also cannot be a friend function of another class.
2. Virtual functions should be accessed using pointer or reference of base class type to achieve run time polymorphism.
3. The prototype of virtual functions should be same in base as well as derived class.
4. They are always defined in base class and overridden in derived class. It is not mandatory for derived class to override (or re-define the virtual function), in that case base class version of function is used.
5. A class may have virtual destructor but it cannot have a virtual constructor.

- Dynamic cast
  - Safe down-cast
    - Use **dynamic_cast**
    - Type* t = **dynamic_cast**<Type*> (variable)
    - Returns NULL if the conversion was not possible
    - Only applicable to pointers
  - If used very often, there is a good chance of your design being flawed

```
int main() {
  Shape *s = new Rectangle(10,20);
  Rectangle* rect1 = dynamic_cast<Rectangle*>(s) ;

  if( rect1 != NULL ) {
          cout << "valid  cast" << endl ;
 }
}
Output :
valid cast
```

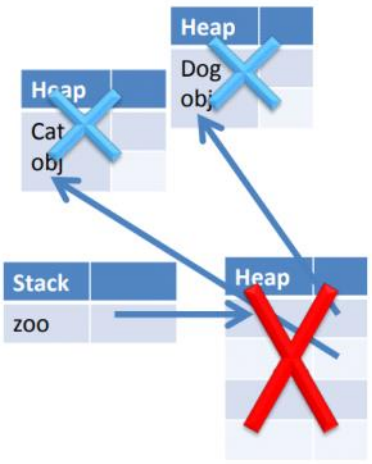
- For Java: use instanceOf to check whether an object is the same class or a descendant of the same class

- Array

```
float  figure[3] ;
figure[0] = 0.79 ;  // first element is always index 0
figure[1] = 0.88;
figure[2] = 0.32 ; // last element, since size is 3

int  a[4] = { 1,2,3,4} ;
int  b[] = { 1,2,3,4,5} ;
int  c[5] = {1,2 } ; // less is ok, more is not
```

```
Cat *cats = new Cat[5];  // Cat is concrete class
delete [] cats ;
// if Mammal is abstract class
Mammal **zoo = new Mammal*[4] ;
zoo[0] = new Dog() ;
Or
Mammal *zoo[4] ;
zoo[0] = new Dog() ;
```

```
for(j=0;j<4;j++)
delete zoo[j];

delete [] zoo;
```



- Operator overloading (different from function overloading)

```
// Overload + operator to add two Box objects.
Box operator+(const Box& b) {
   Box box;
   box.length = this->length + b.length;
   box.breadth = this->breadth + b.breadth;
   box.height = this->height + b.height;
   return box;
}
```

```
// box 1 specification
Box1.setLength(6.0);
Box1.setBreadth(7.0);
Box1.setHeight(5.0);

// box 2 specification
Box2.setLength(12.0);
Box2.setBreadth(13.0);
Box2.setHeight(10.0);

// volume of box 1
volume = Box1.getVolume();
cout << "Volume of Box1 : " << volume <<endl;

// volume of box 2
volume = Box2.getVolume();
cout << "Volume of Box2 : " << volume <<endl;

// Add two object as follows:
Box3 = Box1 + Box2;

// volume of box 3
volume = Box3.getVolume();
cout << "Volume of Box3 : " << volume <<endl;
```

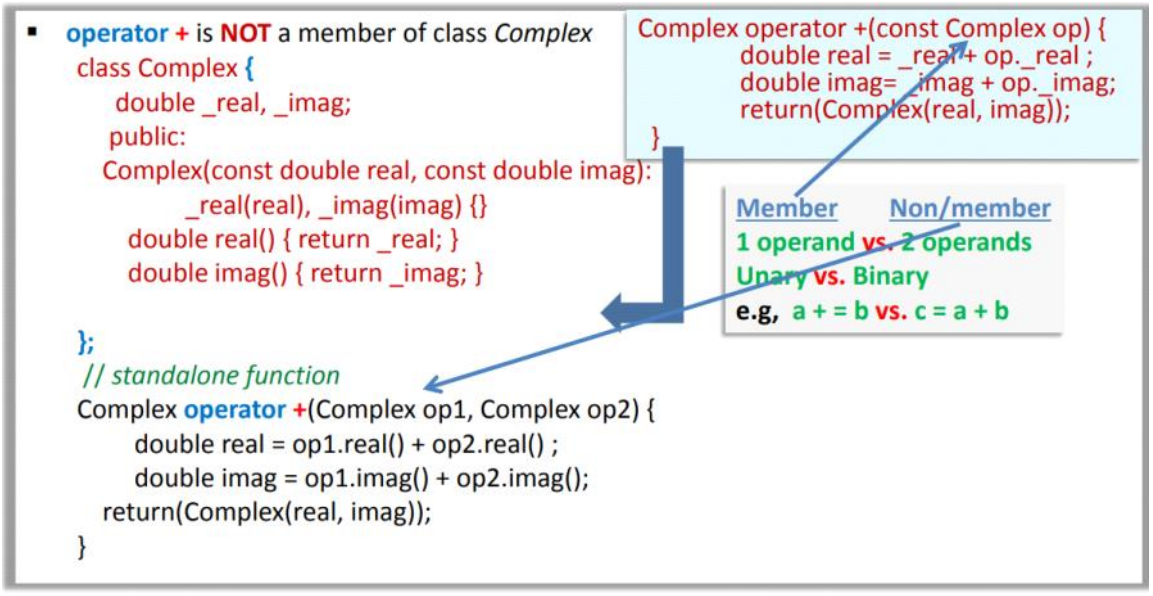
```
Volume of Box1 : 210
Volume of Box2 : 1560
Volume of Box3 : 5400
```

Following is the list of operators which can be overloaded −

+	-	*	/	%	^
&	|	~	!	,	=
<	>	<=	>=	++	--
<<	>>	==	!=	&&	||
+=	-=	/=	%=	^=	&=
|=	*=	<<=	>>=	[]	()
->	->*	new	new []	delete	delete []

Following is the list of operators, which can not be overloaded −

::	.*	.	?:



- Friend
  - Friend allows non-member function access to private data of a class

```
class Complex {
     double _real, _imag;  // private
    public:
            ....
    friend Complex operator +( const Complex , const Complex  );
};
 Complex operator +(const  Complex op1, const Complex op2) {
            double real = op1._real + op2._real ;
            double imag = op1._imag + op2._imag;
   return(Complex(real, imag));
}
```

- **Should not use friend unless necessary**
  - **Break the data hiding principle.**
  - **If used often it is a sign that it is time to restructure your inheritance.**

```
// friend class SomeClass;
```

- **const**
  - to declare particular aspects of a variable (or object) to be constant
  - **const** variable
- Examples,

```
int i;          // just an ordinary integer
int *ip;        // uninitialised pointer to integer
int * const cp = &i;      // constant pointer to integer
const int ci = 7;         // constant integer
const int *cip;           // pointer to constant integer
const int * const cicp = &ci; // constant pointer to constant integer
```

const int const int int

```
class Point {
      int _x, _y;    // point coordinates,  default private
   public:
     void setX(const int val); // definition
      int getX() const ; // definition
};
```

- **Function parameter/s**

  Example,

```
void Point::setX(const int val) {
   val = 5 ;  // error!!   val is constant and not modifiable
}
```

- **Member Function** (read-only function)

  Example,

```
int Point::getY()  const {
   _x = 0 ; _ y= 5 ; // error!!   Member variables are not modifiable
   return _x;
}
```

- Using namespaces, we can create two variables or member functions having the same name.

```
// in example.h
#include <iostream>
using namespace std ;

namespace root {
        namespace sub {
                class Test {
                        public :
                        void print() {
                            cout << "an example of namespace" << endl ;
                        }
                };
        }
}
```

```
#include example.h
using namespace root::sub ;
int main() {
        Test t ;
        t.print();
}
```

```
#include example.h
using namespace root ;
int main() {
        sub::Test t ;
        t.print();
}
```

```
#include example.h

int main() {
    root::sub::Test t ;
    t.print();
}
```

- Base class scoping :: used to access methods deeper in the hierarchy

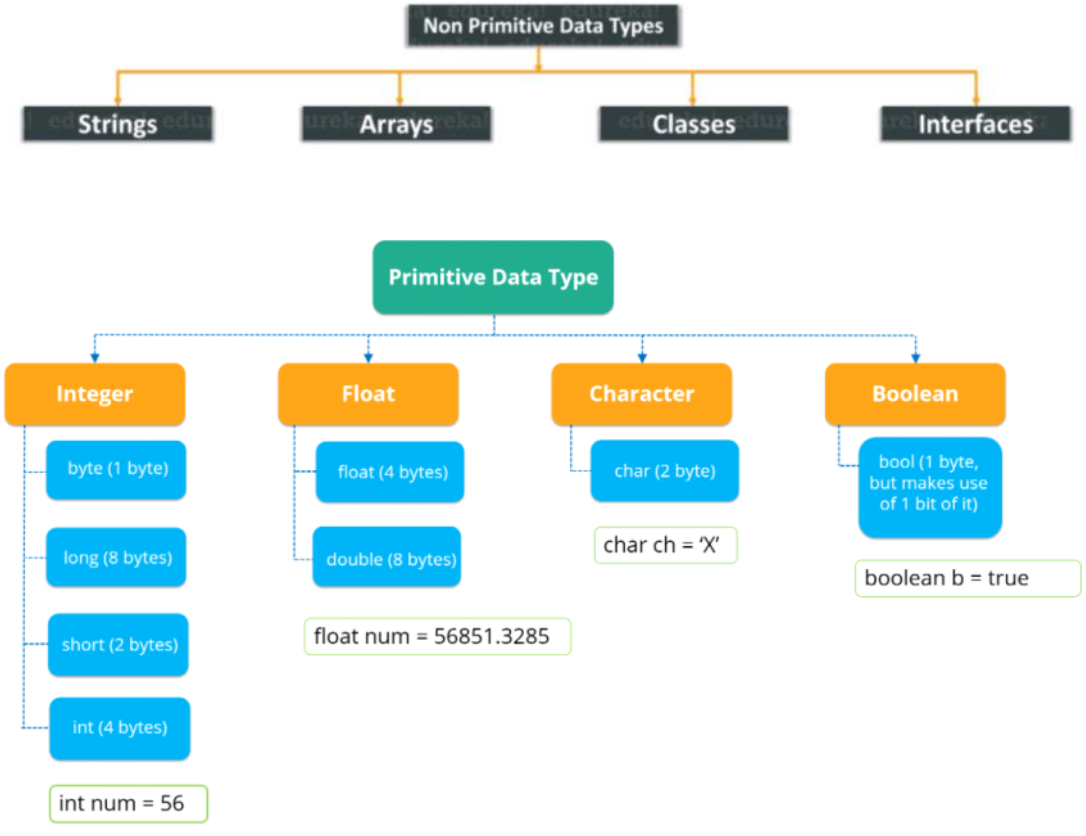
#include <string>

Function	Task
append()	Appends a part of string to another string.
at()	Obtains the character stored at a specified location.
compare()	Compares string against the invoking string.
empty()	Returns true if the string is empty; otherwise returns false.
erase()	Removes character as specified.
find()	Searches for the occurrence of a specified substring.
insert()	Inserts character at specified location.
length()	Gives number of elements in a string.
replace()	Replace specified characters with a given string.

- Java

Non-Primitive Datatypes

Non-Primitive data types refer to objects and hence they are called **reference types.** Examples of non-primitive types include Strings, Arrays, Classes, Interface, etc. Below image depicts various non-primitive data types.



- Java vs C++

- **Everything in Java must be in a class**. There are no global functions or global data.
- **There's no scope resolution operator :: in Java**. Java uses the dot for everything (including package). package (Java) vs. namespace (C++).
- **In Java, all objects of non-primitive types can be created only via new**. There's no equivalent to creating non-primitive objects "on the stack" as in C++.
- **In Java, Object handle (references) defined as class members are automatically initialised to null.** Initialisation of primitive class data members is guaranteed in Java.
- **There are no Java pointers in the sense of C and C++.** When you create an object with new, you get back a reference.
- **There are no destructors in Java.**
- **Java uses a singly-rooted hierarchy** inherited from the root class Object. In C++ you can start a new inheritance tree anywhere, so you end up with a forest of trees.

- **Inheritance in Java has the same effect as in C++, but the syntax is different.** (however, the super keyword in Java allows you to access methods only in the parent class, one level up in the hierarchy. Base-class scoping (::) in C++ allows you to access methods that are deeper in the hierarchy).
- **Java provides the interface keyword** which creates the equivalent of an abstract base class filled with abstract methods and with no data members.
- **There's no virtual keyword in Java** because all non-static methods always use dynamic binding. In Java, the programmer doesn't have to decide whether or not to use dynamic binding. The reason virtual exists in C++ is so you can leave it off for a slight increase in efficiency when you're tuning for performance (or, put another way, "if you don't use it you don't pay for it").
- **Java doesn't provide Multiple Inheritance (MI).**
- **There is method overloading, but no operator overloading in Java.**

- C vs C++

- C is primarily a subset of C++.
- **Header files** help in speeding up compile time, keep code more organised, and allow us to separate interface from implementation.
- C++ allows object to be created in the **stack** without the use of '**new**'.
- C++ requires object to be cleaned after use to **avoid memory leak**. As a rule, for every '**new**' for creating object, there should be a corresponding '**delete**' to clean up the object.
- C++ has a class **destructor** which is called when **delete** is used.
- C++ uses **dynamic initialisation** to initialise class attributes and base object.
- C++ allows **Multiple Inheritance**.
- C++ uses class scope resolution operator **::** to identify explicitly the class to use.
- C++ provides **reference (&)** as an alternative to using **pointer (*)**.
- C++ allows dynamic binding only for **virtual** function.
- C++ allows default parameters in the function parameters.
- C++ allows **operator overloading** using the **operator keyword**.

Lab 4

Monday, 12 October 2020 11:24 AM

1)

1. `int` is a POD, not an object. Boxing and unboxing from `int[]` to corresponding `Integer[]` is not automatically performed. You need to declare `intArray` as:

```
Integer[] intArray = {2,3,5,1};
```

2. `Integer` implements `Comparable<Integer>`, not `Comparable<Object>`. The two specializations are different and incompatible types. The right way to declare `selectionSort` is by using generics, as you suggest in the title:

```
public static <T extends Comparable<T>> void selectionSort(T[] input)
```

2)