Classes and Objects - Part I

Class and Object

- Object
  - A physical, conceptual or software entity
    - Has identity
    - Has state
    - Has behaviour
- Class
  - A specification of attributes and operations
    - Attributes
      - A named property that describes a range of values that instances of the property can hold
    - Operations
      - An implementation of a service that can be requested from any object of the class of which it is part

Members

- The variables and functions defined inside a class are called members of the class.
- Types of members
  - Data members
  - Member functions
    - Methods
    - Constructors
    - Properties
    - Finalizers
    - Operators
    - Indexer

Member variable default values

- Member variables are automatically assigned a default value.
- **`bool→ false`**
- **`Integer types→0`**
- **`Floating point types→0.0`**
- **`char →'\0'`**
- **`string and references→ null`**

Note that the compiler requires the local variables (variables declared within a method) EXPLICITLY initialized before use!!

Example of a customer class

Display.cs

```
class Customer{
public string name;
public uint custId;
public string address;

public void Display(){
System.Console.WriteLine(“Name:”+name);
System.Console.WriteLine(“ID:”+custId);
System.Console.WriteLine(“Address:”+address);
}
}
```

Data members

Method

Accessing members from the same class

Responsibility of this class is the integrity of details of customer that it encompasses.

Creating objects and accessing members

Accessing members from another class:

```
class Test{
…
Customer C1= new Customer();
C1.custId=10;
C1.name=“Alex”;
C1.address=“B-123, Swati Apts, Ramnagar, CBE”;
C1.Display();
}}
```

Creating Customer object

Accessing member using . operator

Question?

- Can custId be 0?
- Which class is responsible for the integrity of the custId?
- What should be done so that the class makes sure that the values of the variables are meaningful?

Object oriented principle number 1

“ Encapsulation is the process of compartmentalizing the elements of an abstraction that constitute its structure and behaviour; encapsulation serves to separate the contractual interface of an abstraction and its implementation.”

-- Grady Booch

Member Visibility

- public
  - Accessible from anywhere
- private
  - Accessible from only within a class
- protected
- internal
- protected internal

Unmarked members are private by default

Class visibility

- Top-level classes (, interfaces, structures, enumerations and delegates) can only be declared as
  - `public`
  - `internal`

- Implementing Encapsulation→ Traditional way

```
  public class Employee{
  private uint empID;
  private float pay;
  private string empName;
  public int GetEmpID(){return empID;}
  public void SetEmpID(uint empID){
      if(empID!=0)this.empID = empID;
      else{
      System.Console.WriteLine(“Invalid Value);
      System.exit(0);
}
//Accessors and Mutators for other Fields(if
  required)
}
Use:Employee e=new Employee();
e.SetEmpName(“Raj”);
```

Accessor

Mutator

Implementing Encapsulation → Using Properties

```
  public class Employee{
  private int empID;
  private float pay;
  private string empName;
//Property for empName
 public string Name
  {   get{return empName;}
      set{
          if(value!=“”)
              empName=value;
         }
  }
//properties for other attributes
}
Use:
Employee e=new Employee();
e.Name=“Raj”;
```

Employee.cs

“value” is not a keyword but a word that represents the implicit parameter used during property assignment in set property

Changing visibility level

```
public string Name
 {  get{return empName;}
   private set{
       if(value!=“”)
           empName=value;
      }
 }
```

Making read-only or write-only

- Read-only

```
public string Name{
 get{return empName;}
 }
```

- Write-only

```
public string Name{
 set{
           if(value!=“”)
                 empName=value;
 }}
```

Constructor

- A special method used to initialize members when object is constructed.
- Called automatically on creation of object.
- Classes for which explicit constructor is *not written*, *default constructor* is automatically provided.
- Name of the constructor is same as the name of the class.
- A constructor does not have return type.

Example-Constructor

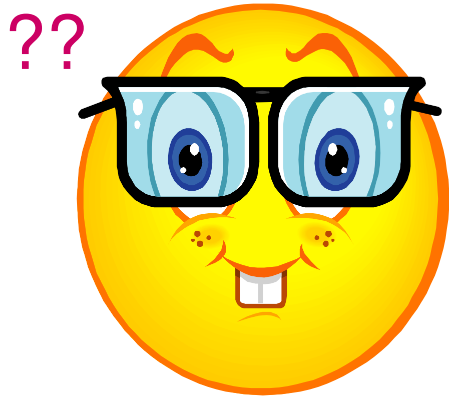
```
using System;

class Point{
private int x,y;

Point(int x,int y){
this.x=x;
this.y=y;
}
static void Main(){
Point p1= new Point(10,20);
Point p2= new Point(10,20);
}}
```

constructor

Creating object by calling the above constructor



Invoking new Point() generates error!

this

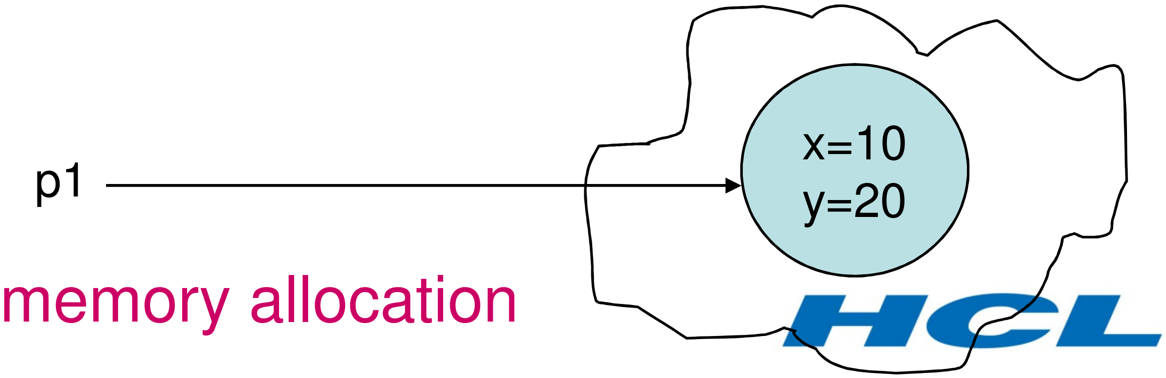
- The **this** keyword refers to the current instance of the class.
- It can be used to access members from within constructors, instance methods, and instance accessors.
- It can be used to the current object pass an object as a parameter to a method.
  - **call(this);**
- **this** is also used for constructor chaining and
- to declare indexers.

Constructor Chaining

```
public class Point{
public int x ;
public int y ;
public Point():this( 0,0 ){
}
public Point( int x, int y ){
this.x = x ;
this.y = y ;
}
// etc
}
```

References

- The types created for classes are called references.
- Unlike other basic types (like **`int, float`** etc. which are created on stack), references are created on the heap.
- They are implicit pointers to the object created.
- The space for the object allocation is not done until runtime.



Similar to C's dynamic memory allocation

Type classification

- Value types
  - Basic types like **`int, char, float`** etc.
  - **`struct`** types
  - **`enum`** types
- Reference types
  - Class types
  - Array types
  - Interface types
  - Delegate types

Arrays

- Arrays are references types.
- They are automatically of a predefined type

  `System.Array`.
- Creating an array:

```
 int[] n=new int[5];
```

If there is a mismatch between the declared size and the number of initializers, a compile time error is generated.

- Creating and initializing an array:

```
int[] n2=new int[4]{20,10,5,13};
```

Or simply

```
int[] n1={20,10,5,13};
```

Multidimensional array

- Two types of multidimensional array

  - Rectangular array
  - Jagged array

Rectangular Array

- A multidimensional array where length of each row is fixed.
- Creation:

```
int[,] matrix = new int[5,5];
```

- Accessing array elements:

```
matrix[0,1]=9;
```

Jagged Arrays

- A jagged arrays are array of arrays. The arrays may be of different sizes.
- Creation:

```
 int[][] myarr=new int[5][];
```

- Creating arrays in the jagged array,

```
for(int i=0;i<myarr.Length;i++){
 myarr[i]= new int[i+3];  }
```

- Accessing elements

```
    myarr[0][1]=8;
```

- Accessing the length of row 0,

```
   myarr[0].Length
```

Members of `System.Array`

- **`BinarySearch()`**→Searches array for a given item
- **`Clear()`**→Sets a range of elements in the array to empty values(0 for value types,null for reference types)
- **`CopyTo()`**→Copy elements from the source array into the destination array
- **`Length`**→ Determines the number of elements in an array(read-only property)
- **`Rank`**→ Returns the number of dimensions of the current array.
- **`Reverse()`**→Reverses the contents of a one-dimensional array.
- **`Sort()`**→Sorts one-dimensional array of intrinsic types.

`static` members

- Members which are accessible only at class level.
- Members cannot be accessed using instance.
- **`WriteLine()`** is a static method of **`System.Console`** class.
- **`Main`** is declared as static method.
- Static data is shared by all the instances of that class.
- Static member functions can access only static data members.

Static Property-Example

```
class Circle{
private static double PI;

public static double pi{
get{return PI;}
set{ PI=value;}
}

static void Main(){
Circle c= new Circle();
Circle.pi=3;
System.Console.WriteLine(Circle.pi);
}}
```

`c.pi` would give error!

static constructors

- Like constructors are used to initialize instance fields, static constructors are created to initialize static fields.
- But unlike regular constructors, static constructor
  - cannot have arguments.
  - cannot have any modifier
  - can be only single
- A static constructor executes before any other constructor.
- It gets called just before any of the class member (static or instance or constructor) is invoked.
- It gets called only once.

```
class BankAccount{
static double rate;
int acctid;
double bal;

public BankAccount(int acctid,
                 double bal){
this.acctid=acctid;
this.bal=bal;
}

static BankAccount(){
//assume the data is read from the
  database
rate=0.05;
}
public void calInterest(){
bal=bal+bal*rate;
}
public void display(){
System.Console.WriteLine("AcctID
="+acctid);
System.Console.WriteLine("Bal ="+bal);
}
static void Main(){
BankAccount bank= new
BankAccount(1,3000);
bank.calInterest();
bank.display();
}
}
```

const

- Constant members can be created using const keyword.
- Members of const type must be initialized during compile-time.
- Constants are implicitly static. Therefore they are accessed using class name only.

-
  ```
  class X{
   public const double PI=3.14;
  …
  }
  ```
- Accessing outside the class: **`X.PI.`**

Question?

- `public const Point center= new Point(10,20);`
- The above statement generates error.
- Why?

Because the object is created only at runtime and const requires the value to be known at the compile time.

Does that mean I cannot create constant objects?

Lets us see….

Read-only instances

- Read-only fields are assigned value only once either during compile time or runtime.
- They must be initialized either with the declaration or in the constructor.
- The keyword ‘readonly’ is used for this

```
readonly Point center= new
Point(10,20);
```

- Unlike constants, read-only fields are instance members and not static members.

readonly Example

```
class Point{
public int x,y;
public Point(int x,int y){
this.x=x;
this.y=y;}
}
class Circle{
public readonly Point center= new
  Point(10,20);
static void Main(){
Circle c= new Circle();
System.Console.WriteLine(c.center.x
  +", "+ c.center.y);}
}
```

Question?

- What is the difference between

  `const double PI=3.14` and

  `readonly double PI=3.14` ?

Try to answer

- What is **`static readonly`** field?

static classes

- A class defined as static cannot be created using new keyword.
- It can contain only static members.
- It is useful when we need to create a kind of a utility class which just has a set of utility functions.
- For instance, a class that contains all the sort and search methods.

static classes -Example

```
static class Common{
static int temp  ;
public static void bubble(int[]
  array){
   for(int
  pass=0;pass<array.Length;pass++)
   for(int j=0;j<((array.Length)-pass-
  1);j++)
        if(array[j]>array[j+1])  {
            temp = array[j];
            array[j] = array[j+1];
            array[j+1] = temp;
          }
   }
```

Must be declare explicitly as static

```
public static void exchange(int[]
array) {
    for(int i=0;i<array.Length-1;i++)
    for(int j = i+1;j<array.Length;j++)
        if(array[i]>array[j]){
          int temp=array[i];
          array[i]=array[j];
          array[j]=temp;
} } }
class Test{
static void Main(){
int[] array={1,6,4,3,7};
Common.exchange(array);
 for(int i=0;i<array.Length-1;i++)
System.Console.WriteLine(array[i]);}}
```

Parameter Passing

- Passing value types
- Pass by reference types

Passing basic types

```
class PassBasic{

static void f(int i){
i=5;
}
public static void Main(){
int i=10;
f(i);
System.Console.WriteLine(i);
}

}
```

Prints 10

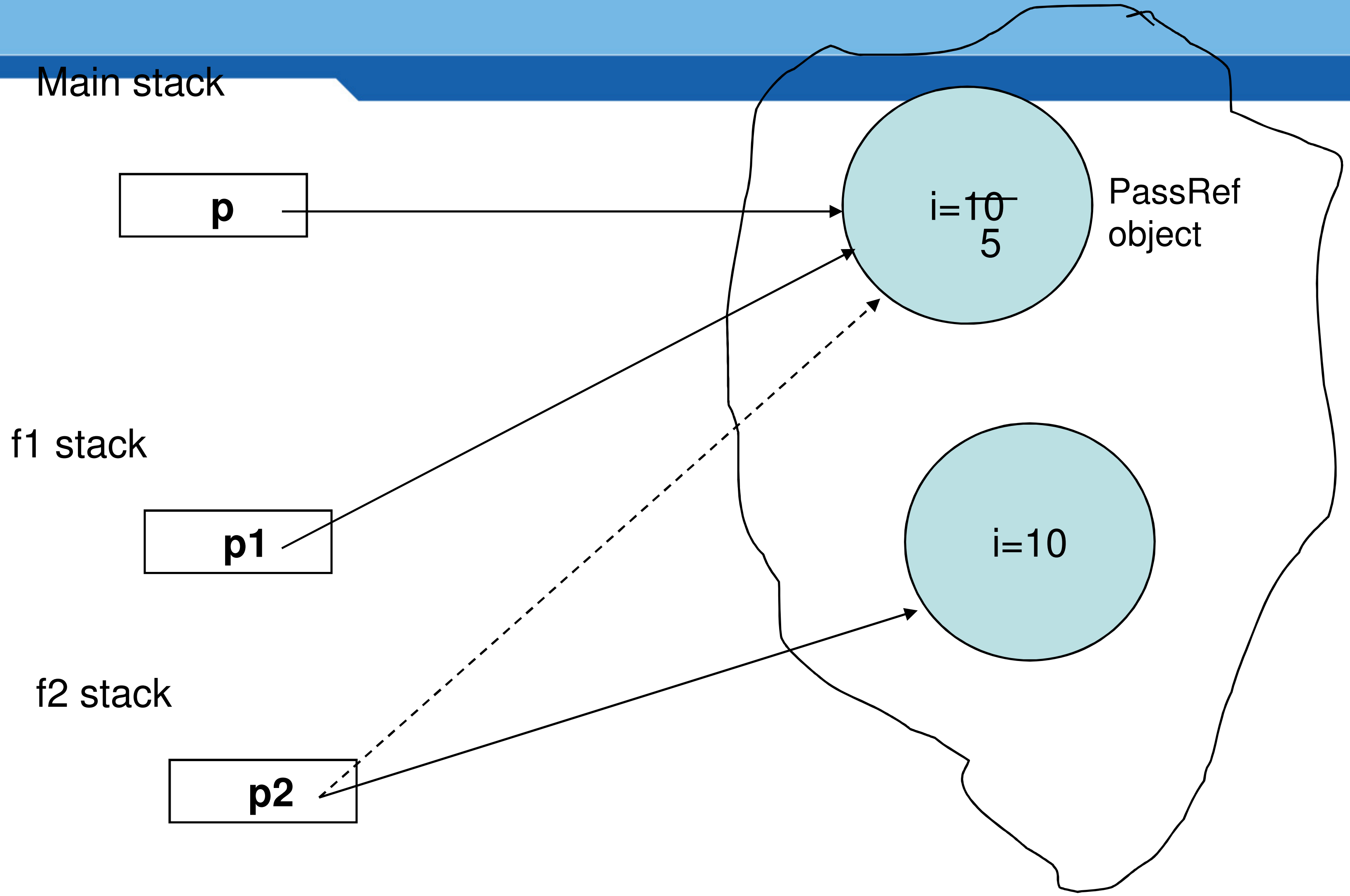
Conclusion →value types are passed by value

Pass by reference types

```
class PassRef{
int i;
PassRef(int i){this.i=i;}

static void f1(PassRef p2){p2.i=5;}
static void f2(PassRef p){
p= new PassRef(15);        }
public static void Main(){
PassRef p= new PassRef(10);
f1(p);
System.Console.WriteLine(p.i);
f2(p);
System.Console.WriteLine(p.i);}
}
```

Prints 5



Conclusion →reference types are also passed by value

Method Parameter modifiers

- By default the parameters are passed by value.
- Parameter modifiers can be used to alter this behaviour.
- Parameter modifiers that alter the default behaviour :
  - `ref`
  - `out`
- There is yet another modifier that could be used with the parameters which is `params`.

ref

- **ref** keyword makes the parameter passing to be done by reference.

```
class PassRef{
int i;
PassRef(int i){this.i=i;}

static void f1(ref PassRef p){p.i=5;}
static void f2(ref PassRef p){
p= new PassRef(15);
}
static void f3(ref int i){
i=100;
}
public static void Main(){
int i=50;
f3(ref i);
System.Console.WriteLine(i);

PassRef p= new PassRef(10);

f1(ref p);
System.Console.WriteLine(p.i);

f2(ref p);
System.Console.WriteLine(p.i);
}
}
```

Must specify this while calling

Prints 100

Prints 5

Prints 15

refmod.cs

out

- Similar to **`ref`**, except that the initial value of an the argument provided by the calling function is not important.
- In other words, the called methods will ensure that the variable defined as out will have a valid value before function exits.

```
using System;
class OutParam{
public static void cal(int i,int j,
                         ref int k){
k=i+j;
}
public static void Main(){
int i=10,j=20,k;
cal(i,j,ref k);
Console.WriteLine(k);
}
}
```

Compile-time Error : Use of unassigned local variable 'k'

```
using System;
class OutParam{
public static void cal(int i,int j,
                                   ~~ref~~ int k){
k=i+j;
}
public static void Main(){
int i=10,j=20,k;
cal(i,j,~~ref~~ k);
Console.WriteLine(k);
}
}
```

out

Prints 30

OutParam.cs

`params`

- Sending any number of argument of a particular type.
- There can be only one **`params`** for any method.
- The **`params`** argument must be the last parameter specified.
- The **`params`** should be a single dimensional or a jagged array.

```
using System;
class Params{
static int sum(params int[] i){
int sum=0;
for(int k=0;k<i.Length;k++)
sum+=i[k];

return sum;
}
public static void Main(){
int s=sum(1,2,3,4);
Console.WriteLine(s);
s=sum(11,22);
Console.WriteLine(s);
}}
```

Or send an array of int

C# Nullable Types

- Default value for reference type is **`null`**.
- Reference values can also be explicitly assigned to null
  - **`string s=null;`**
- Value types cannot be set to **`null`**
  - **`int i=null;`**
  - **`bool b=null;`**

Not completely true because this works!!!

```
bool? b=null;
int? i=null;
```

and **`string? S=“Raj”;`** gives error!!!

Why would you want to assign value types as null? → we will discover in the boxing section.

The ?? Operator

- `??` operator allows us to assign a value to a nullable type if the retrieved value is **`null`**.

Some user defined method which retrieves value(say, student id) from the database

```
int? studid=GetIdFromDatabase()??1;
```

If GetIdFromDatabase() returns null then studid=1