Inheritance I

Inheritance

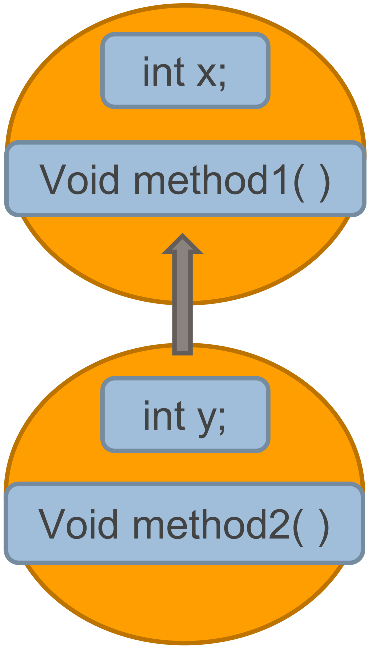
- A class inherits from another class to
- *Reuse*
  - use the actions or attributes of the original class
- *Extend*
  - adding action(s) or attributes to the original class
- *Modify*
  - change its action(s) the original class

Inheritance

```
public class Class1
{ public int x;
  public void method1()
  {
    Console.WriteLine("x={0}", x);
  }
}
```

```
public class class2:Class1
{ public int y;
  public void method2()
  {
    Console.WriteLine("y={0}", y);
     method1();
  }
}
```

Class1
(base)
(parent)
(super class)

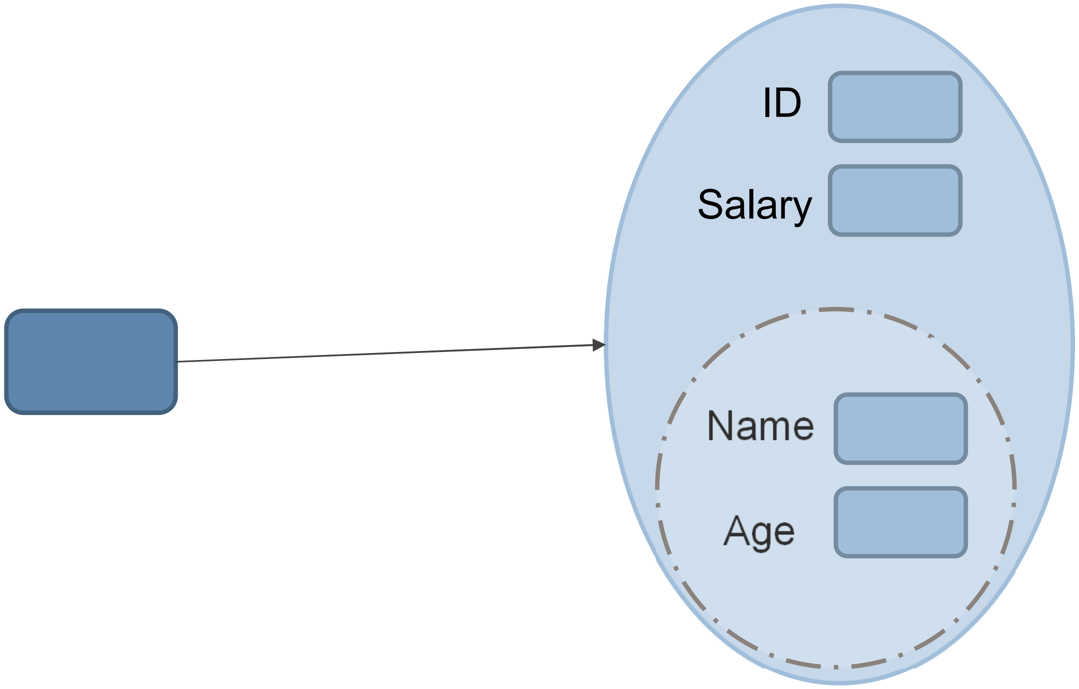


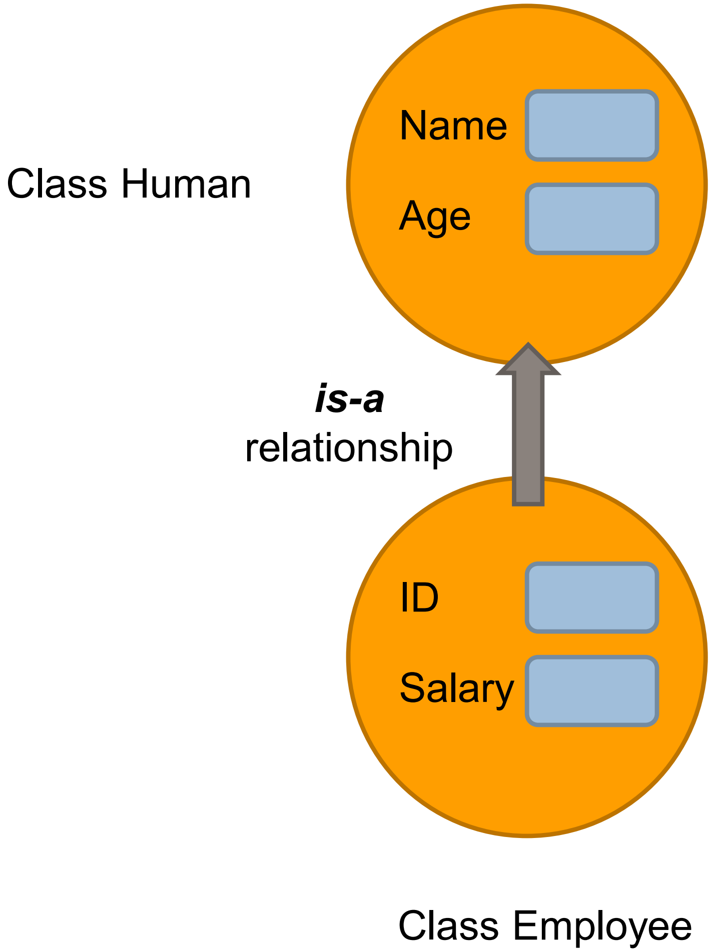
Class2
(derived)
(child)
(sub class)

Structure does not support inheritance

Inheritance

- EX: Employee Inherits Human
- Employee *is a* human





Inheritance and Access Modifier

- *public* Access Modifier
  - Has no effect (all members are inherited and accessible from within child class and anywhere else)
- *private* Access Modifier
  - All members are inherited but not accessible through child members
- *protected* Access Modifier
  - All members are inherited and accessible through child members but not accessible outside the child class

Inheritance and constructors

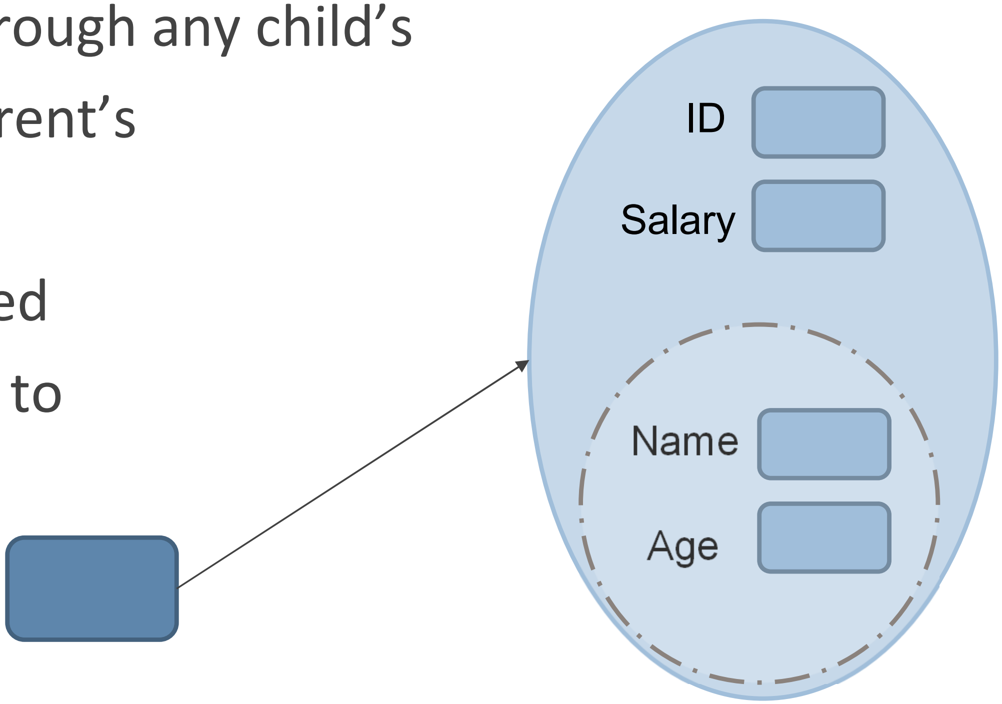
- Creating an Object of child class type cause creating an object of parent class type within it
- Creating an Object of child through any child's

  Constructor would call the parent's

  default constructor
- This behavior could be changed

  Using *base* keyword to direct to

  Specific constructor
- Demo



Inheritance

- *base* Keyword
  - Used for identify base class *method* or *constructor*
- *sealed* Keyword
  - Prevent a class to be a parent for another class
  - Prevent members(method , property) from being overridden in child class

Inheritance and Type Conversion

Child to Parent

- The child class data type is – a parent class data type with extra (field or methods)
- The relation between derived class and base class is-a relation
- The child object could be referred as a parent
  - Ex: every Employee is-a Human

```
Employee emp = new Employee{Age=30};
Human h=emp;
Human h2= new Employee{Age=40};
```

- Conversion from child to parent achieved using ***implicit casting***

Inheritance and Type Conversion

Parent to Child

- Conversion from parent to child must be achieved through ***Explicit casting*** since not every human is an employee (he could be engineering or merchant , etc..)

```
Engineer eng = new Engineer{Age=30,Dept=“Elect”};
Human h= eng;
h= new Employee{Age=40};

Employee emp=(Employee)h;
```

Is operator , *as* operator

- *is* operator
  - Used for test if the object is a certain type or not

```
Human h = new Employee();
if (h is Employee)
   Console.WriteLine("True");
else
   Console.WriteLine("false");
```

- *as* operator
  - Used for explicit casting and evaluate to null if casting fails instead throwing exception

```
Employee emp = new Employee{Age=30};
Human h=emp;
```

```
//Employee emp = (Employee) h;
Employee emp = h as Employee;
```

Virtual Method
(run-time polymorphism)

- Derived class may need to provide customized implementation for inherited method (ex: Display method) this behavior called *Override*

```
public class Human
{…
  public void Dispaly()
  {
   Console.WriteLine($" { Name}/t{ Age}");
  }
}
```

```
public class Employee:Human
{
  public void Dispaly()
  {
   Console.WriteLine($"{ Name}   /t { Age} /t { ID} /t {Salary}");
  }
}
```

Virtual Method

- This scenario could be achieved by mark base class member as ***virtual*** and child class member as ***override***

```
public class Human
{…
  public virtual void Dispaly()
  {
   Console.WriteLine($" { Name}/t{ Age}");
  }
}
```

```
public class Employee:Human
{
  public override void Dispaly()
  {
   Console.WriteLine($"{ Name}   /t { Age} /t { ID} /t {Salary}");
  }
}
```

Virtual Method

- Run-time polymorphism achieved by using a reference of base class type with object to child class

```
Human h= new Employee{Age=40};
h.Display(); // call Employee method not Human method
```

- Both virtual and override methods must have the same signature (name + parameter)
  - Demo without virtual & override
- ***virtual*** modifier used with methods ad properties

Virtual Method

- Why virtual??
  - Demo Human , Employee Display method
  - Code in notes

Virtual Method

- ***new*** modifier
  - Derived class may need to hide inherited method this behavior called method hiding
  - This scenario could be achieved use new modifier
    - Ex: inherit class from external API and hide some members
  - Both old and new methods must have the same signature (name + parameter)
  - ***new*** modifier used with (const and static) fields, method and properties

Object class

- *Object* class is the parent Data type for all Data type in .NET directly or indirectly
- If a class has no parent it is implicitly inherited from *Object* class

Method	Description
public virtual bool Equals (object o)	if reference type check reference equality if value type check value(if different type return false even value is equal)
public Type GetType()	object type not reference type
public virtual string ToString()	Return a string (default return type as a string)
public virtual void Finalize()	implemented through destructor
public static bool ReferenceEquals (object a , object b)	check reference equality

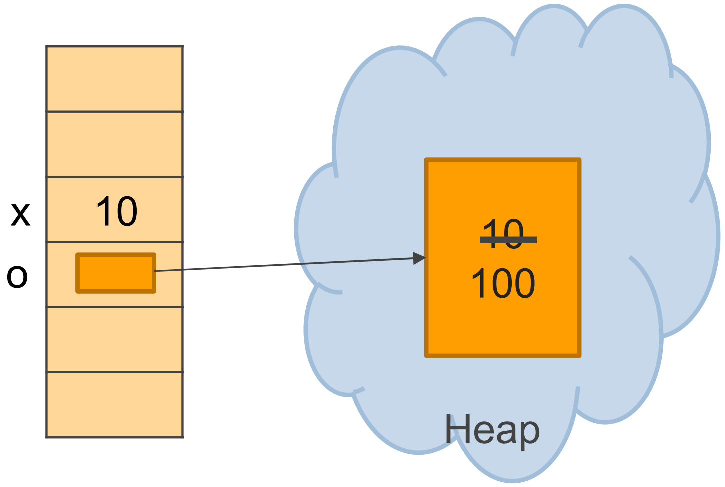
Object class

- Boxing
  - Boxing is the process of storing a value type inside an object

```
int x = 10;
object o = x; // boxing
```

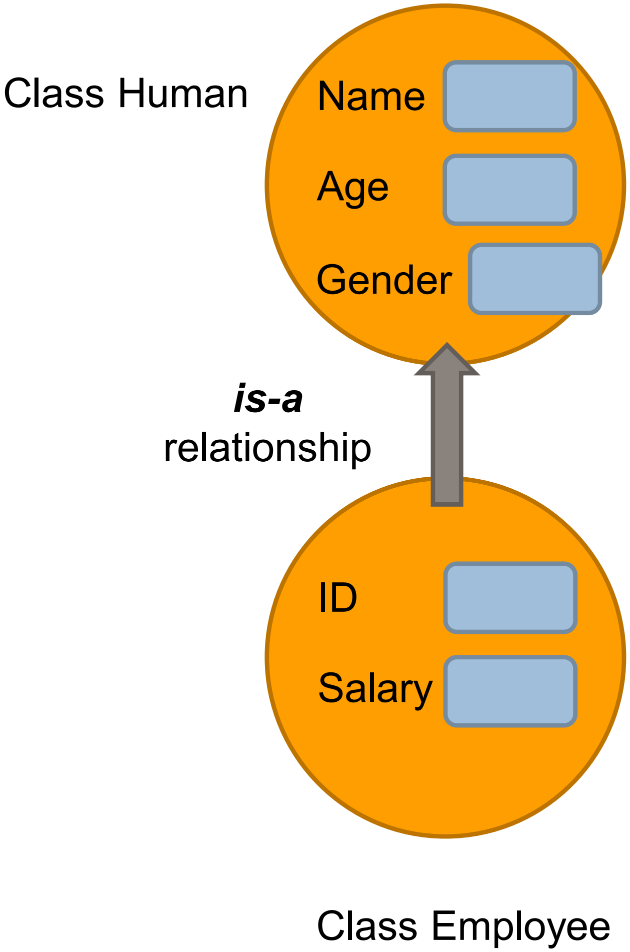
- Unboxing
  - Opposite of boxing

```
int x = 10;
object o = x; // boxing
o = 100;
int y =(int) o; // unboxing
```



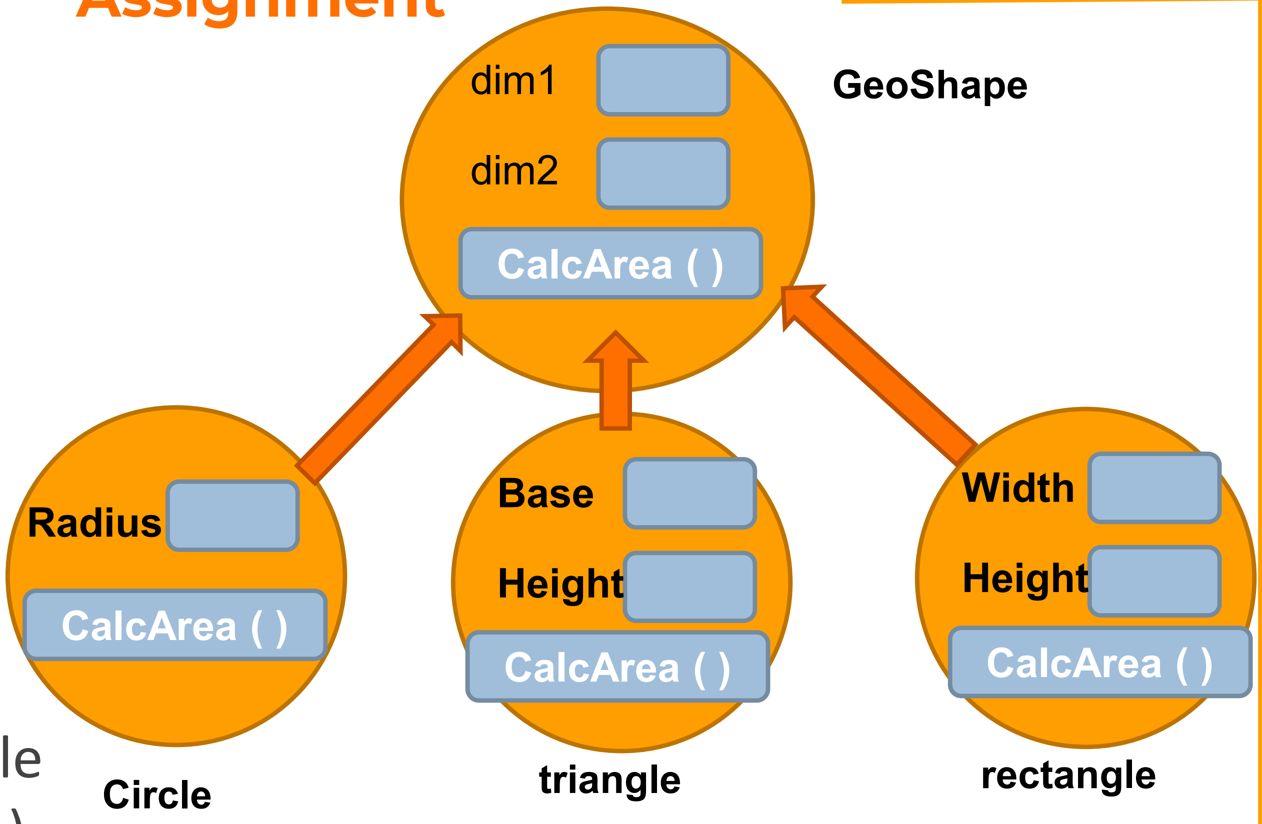
Assignment

- Modify menu Program by
- Design class Human (Age , Name, Gender) and modify employee class to inherit from it
- Override Tostring( ) method in both classes



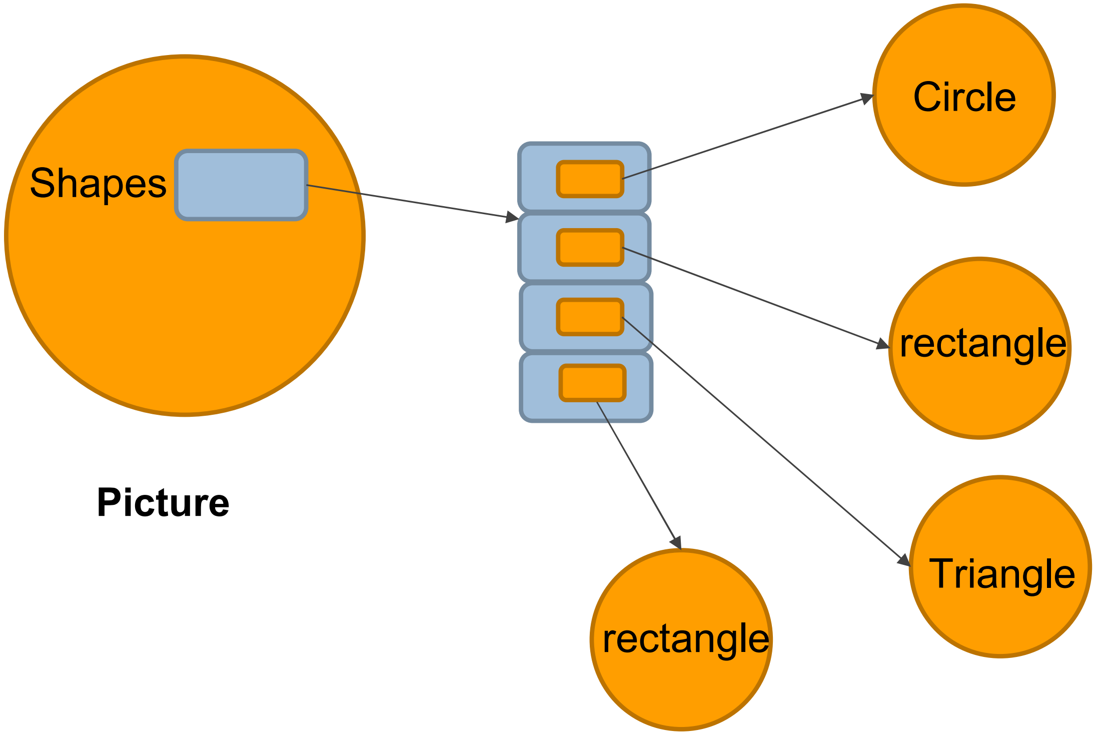
Assignment

- Design GeoShape , rectangle , triangle , circle classes
- And calculate areas individual then through Array of Geoshape type each element refer to different object (circle , Triangle , Rectangle)



Assignment

- Design a class **Picture** that encapsulate number of shapes (circle ,rectangle ,triangle)then calculate sum of their areas



Inheritance II: Abstract class and Interface

abstract **class**

- Is a class not intended to be instantiated , used for design Only , Used to define common member to its ***concrete*** subclasses
  - Ex: GeoShape class
- The major characteristic of abstract class that it contain at least one **abstract member** (method or property)

```
abstract class Geoshape
{
        protected int dim1, dim2;
    …
        public abstract float CalcArea();
}
```

abstract member

- Abstract member is a method or property that has no Implementation ,it can exist ***only*** in abstract class.
  - Ex: converting CalcArea () into abstract method since it does not has a logical meaning to return 0

```
public abstract float CalcArea();
```

- Inheriting from abstract class **enforce** subclasses to override (implement) abstract members
- Abstract members can not be private nor static
- Abstract method implicitly virtual method

Interface

- Interface like abstract class it contain only abstract members, it **can't** contain implementation **nor** member fields.
  - No *abstract* modifier is used
- Interface defines a contract any class implements(inherit) that contract must provides an Implementation of the members defined in the interface
- Interface members has not access modifier (since they must be public)

Abstract property
Not
Auto-implement property

```
interface Imyinter
{
   int prop { set; get; }
   void mymethod();
}
```

Interface

- A class can implements more than interface
- Interface support inheritance
  - Ex: : ***IQueryable*** :***IEnumerable***
- Interface support loose coupling (Example in notes)
- A type, regardless of whether it is a reference type or a value type, can implement any number of interfaces.

Implement interface

```
interface Imyinter
{
   int prop { set; get; }
   void mymethod();
}
```

- Implicitly
  - Through class reference
  - Through interface reference

```
class myclass : Imyinter
    {
        void mymethod()
        {…..}
    }
```

- Explicitly
  - No access modifier
  - Through interface reference only
  - Used in case of multiple implementation

```
class myclass : Imyinter
    {
        void Imyinter.mymethod()
        {…..}
    }
```

Why interface

- Capturing similarities among unrelated classes without artificially forcing a class relationship.
- Ex: PrintData for Employee ,Point

Assignment

- In Menu Program
- Add Sort button
  - Sort Array of Employee
    - Using Array.Sort(array) (hard coding)
      - Implmenting **IComparable** interface by Employee class
    - Using Array. Sort(array, ***IComparer***)
      - By implementing the way of sorting in classes that implements ***Icomparer*** Interface