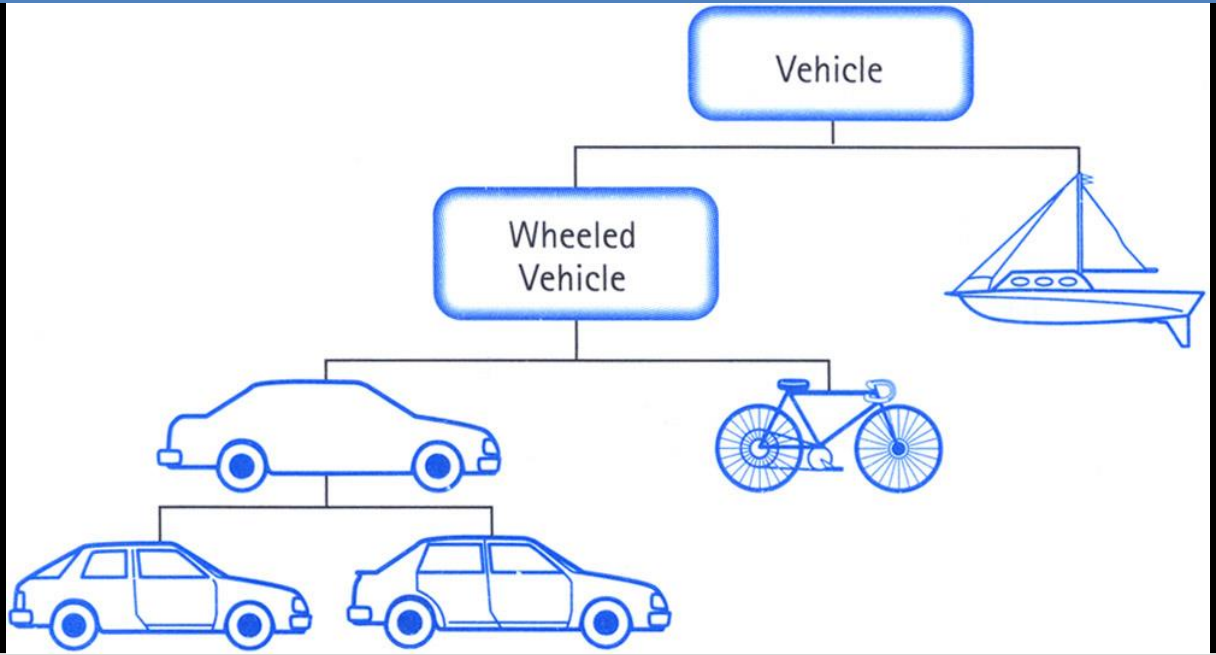



# INHERITANCE & POLYMORPHISM

BME 121 2016

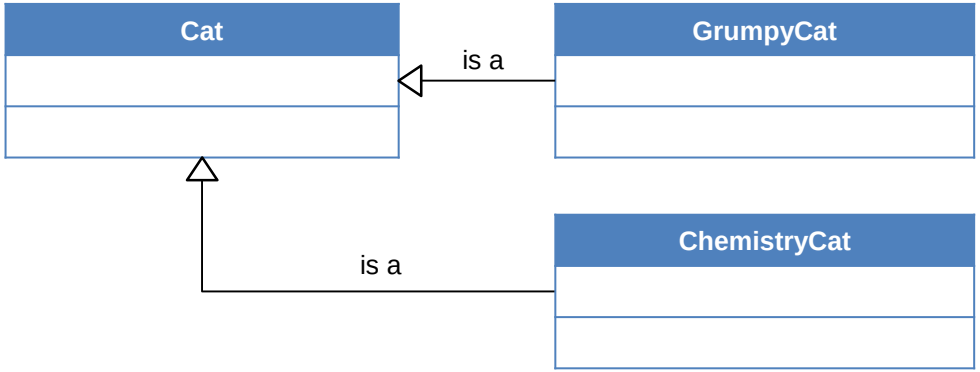
Rasoul Nasiri

# Topics

- Inheritance review
- Parent and child relation
- Virtual vs new
- Polymorphism
- Insertion sort in linked list
- WAx help

# Inheritance

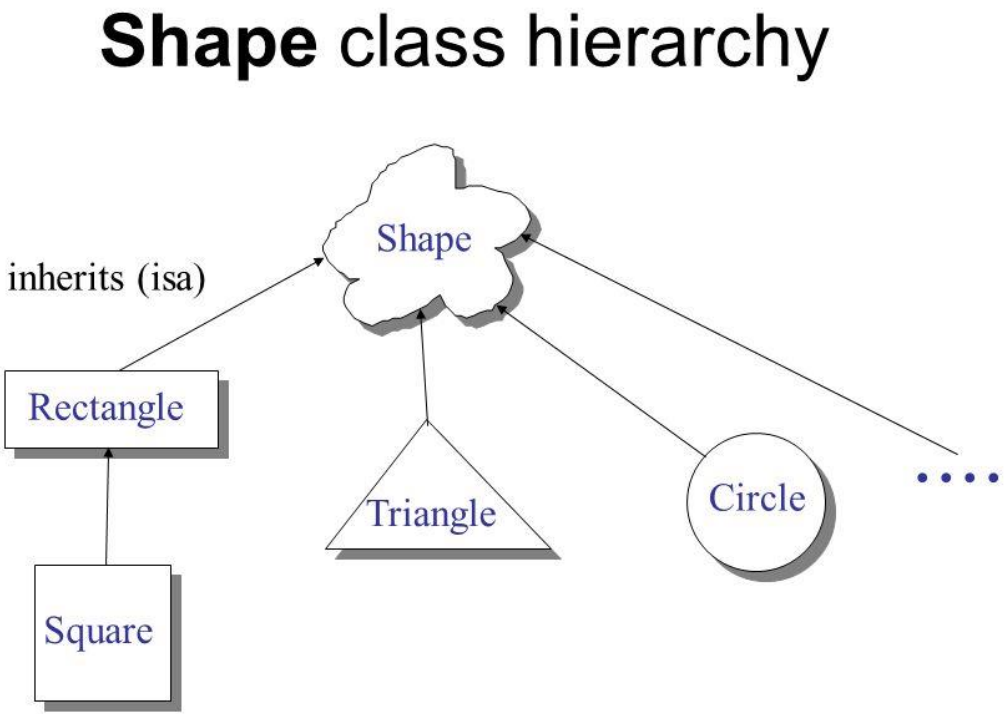
- White triangle points to the parent class



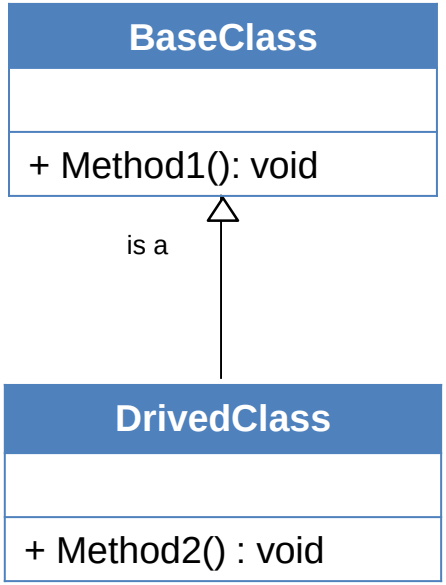


```
public class Cat
{
      public Cat() { }
}
public class GrumpyCat : Cat
{
      public GrumpyCat() { }
}
public class ChemistryCat : Cat
{
      public ChemistryCat() { }
}
```

# Inheritance in shape concept



```
class BaseClass
{
    public void Method1()
    {
        Console.WriteLine("Base : M 1");

    }
}
class DerivedClass : BaseClass
{

    public void Method2()
    {
        Console.WriteLine("Derived : M 2");
    }
}
```



# What is the result of method calls

```
BaseClass bc = new BaseClass();
DerivedClass dc = new DerivedClass();
BaseClass bcdc = new DerivedClass();
// impossible
// DerivedClass dcbc = new BaseClass();

bc.Method1();
//bc.Method2(); // impossible

dc.Method1();
dc.Method2();

bcdc.Method1();
//bcdc.Method2(); // impossible
```

```
class BaseClass
{
    public void Method1()
    {
        Console.WriteLine("Base : M 1");

    }
}
class DerivedClass : BaseClass
{

    public void Method2()
    {
        Console.WriteLine("Derived : M 2");
    }
}
```

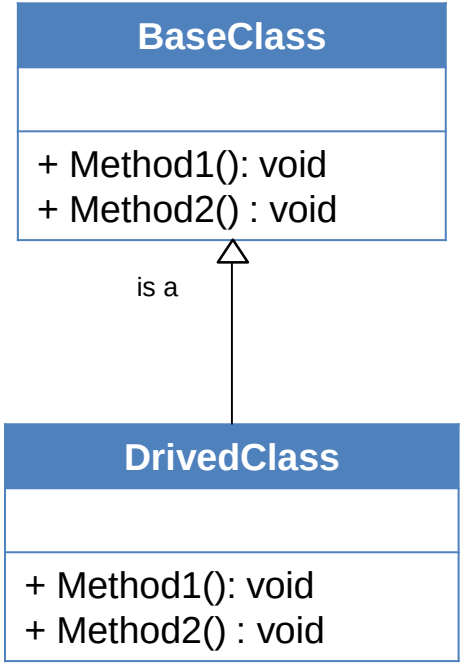
# What is the result of method calls

```
BaseClass bc = new BaseClass();
DerivedClass dc = new DerivedClass();
BaseClass bcdc = new DerivedClass();
// impossible
// DerivedClass dcbc = new BaseClass();

bc.Method1();
//bc.Method2(); // impossible

dc.Method1();
dc.Method2();

bcdc.Method1();
//bcdc.Method2(); // impossible
```

```
class BaseClass
{
    public void Method1()
    {
        Console.WriteLine("Base : M 1");

    }
}
class DerivedClass : BaseClass
{

    public void Method2()
    {
        Console.WriteLine("Derived : M 2");
    }
}
```

- Method1 and Method2 definitions in DerivedClass hide the definitions in BaseClass.
- Any reference of type BaseClass will go to the Base
- Any reference of type DerivedClass will go to derived class

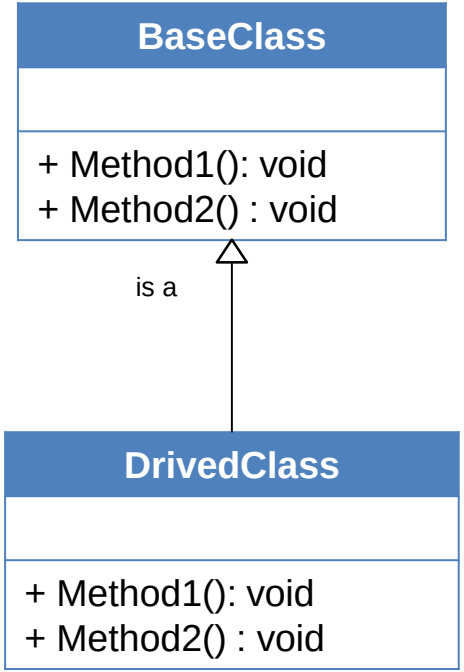
```
class BaseClass
{
    public void Method1()
    {
        Console.WriteLine("Base : M 1");

    }
    public void Method2()
    {
        Console.WriteLine("Base : M 2");
    }
}
class DerivedClass : BaseClass
{
    public void Method1()
    {
        Console.WriteLine("Base : M 1");
    }
    public void Method2()
    {
        Console.WriteLine("Base : M 2");
    }
}
```



# Practice: Write the program and see the result

```
BaseClass bc = new BaseClass();
DerivedClass dc = new DerivedClass();
BaseClass bcdc = new DerivedClass();
// impossible
// DerivedClass dcbc = new BaseClass();

bc.Method1();
bc.Method2();

dc.Method1();
dc.Method2();

bcdc.Method1();
bcdc.Method2();
```

```
class BaseClass
{
    public void Method1()
    {
        Console.WriteLine("Base : M 1");

    }
    public void Method2()
    {
        Console.WriteLine("Base : M 2");
    }
}
class DerivedClass : BaseClass
{
    public void Method1()
    {
        Console.WriteLine("Base : M 1");
    }
    public void Method2()
    {
        Console.WriteLine("Base : M 2");
    }
}
```



```
BaseClass bc = new BaseClass();
DerivedClass dc = new DerivedClass();
BaseClass bcdc = new DerivedClass();
// impossible
// DerivedClass dcbc = new BaseClass();

bc.Method1();
bc.Method2();

dc.Method1();
dc.Method2();

bcdc.Method1();
bcdc.Method2();
```

```
class BaseClass
{
    public void Method1()
    {
        Console.WriteLine("Base : M 1");

    }
    public void Method2()
    {
        Console.WriteLine("Base : M 2");
    }
}
class DerivedClass : BaseClass
{
    public void Method1()
    {
        Console.WriteLine("Base : M 1");
    }
    public void Method2()
    {
        Console.WriteLine("Base : M 2");
    }
}
```

```
BaseClass bc = new BaseClass();
DerivedClass dc = new DerivedClass();
BaseClass bcdc = new DerivedClass();
// impossible
// DerivedClass dcbc = new BaseClass();

bc.Method1();
bc.Method2();

dc.Method1();
dc.Method2();

bcdc.Method1();
bcdc.Method2();
```

```
class BaseClass
{
    public void Method1()
    {
        Console.WriteLine("Base : M 1");

    }
    public void Method2()
    {
        Console.WriteLine("Base : M 2");
    }
}
class DerivedClass : BaseClass
{
    public void Method1()
    {
        Console.WriteLine("Base : M 1");
    }
    public void Method2()
    {
        Console.WriteLine("Base : M 2");
    }
}
```

- The new definition of method hide its old definition in parent
- But if the reference is parent type reference, the method call would go to parent

```
class BaseClass
{
    public void Method1()
    {
        Console.WriteLine("Base : M 1");

    }
    public void Method2()
    {
        Console.WriteLine("Base : M 2");
    }
}
class DerivedClass : BaseClass
{
    public void Method1()
    {
        Console.WriteLine("Base : M 1");
    }
    public void Method2()
    {
        Console.WriteLine("Base : M 2");
    }
}
```

# Override

- We can access to the extend version of Methods in derived class when object is created from derived class independent of reference type.
  - In this case we have to use override

```
class BaseClass
{
    public virtual void Method1()
    {
        Console.WriteLine("Base : M 1");

    }
    public virtual void Method2()
    {
        Console.WriteLine("Base : M 2");
    }
}
class DerivedClass : BaseClass
{
    public override void Method1()
    {
        Console.WriteLine("Derived : M 1");
    }
    public override void Method2()
    {
        Console.WriteLine("Derived : M 2");
    }
}
```

# Override

```
BaseClass bc = new BaseClass();
DerivedClass dc = new DerivedClass();
BaseClass bcdc = new DerivedClass();
// impossible
// DerivedClass dcbc = new BaseClass();

bc.Method1();
bc.Method2();

dc.Method1();
dc.Method2();

bcdc.Method1();
bcdc.Method2();
```

```
class BaseClass
{
    public virtual void Method1()
    {
        Console.WriteLine("Base : M 1");
    }
    public virtual void Method2()
    {
        Console.WriteLine("Base : M 2");
    }
}
class DerivedClass : BaseClass
{
    public override void Method1()
    {
        Console.WriteLine("Derived : M 1");
    }
    public override void Method2()
    {
        Console.WriteLine("Derived : M 2");
    }
}
```

- Try it and see the results

# Polymorphic Method Invocation

- What you just witnessed, the ability to use the right version of a method depending on what class it belongs to, is called Polymorphic Method Invocation

| **Reference Configuration** | | | **Which version of a common method will it use?** | |
|---|---|---|---|---|
| Reference Type | Object Type | Example | Declared normally in Parent and replaced in Child using new keyword | Declared virtual in Parent and replaced in Child using override |
| Parent Class | Parent Class | Cat c = new Cat(); | Parent Class | Parent Class |
| Child Class | Child Class | Grumpy g = new Grumpy(); | Child Class | Child Class |
| Parent Class | Child Class | Cat g = new Grumpy(); | **Parent Class** | **Child Class** |
| Child Class | Parent Class | Impossible | | |

Standard way in Java and C++

# Polymorphism

- More generally, there's a concept called Polymorphism:
  - From Greek: polys - many, much & morphe - form, shape.
  - In the programming sense: there are many versions of *something*
- There's actually 3 kinds of Polymorphism in programming:
  - **Method overloading** – multiple versions of the same method within the same class (formally called Ad hoc polymorphism)
  - **Generics** – a class which specializes to particular type(s) in each copy of its objects (formally called Parametric polymorphism); this class directly has may "versions"
  - **Inheritance** – each child class is a specialized version of the parent class, often adding something on top of the parent class (formally called Inclusion polymorphism, Subtype polymorphism, or simply Subtyping)
- Note that in industry, the term Polymorphism specifically refers to Inheritance and Polymorphic Method Invocation, the other 2 are directly referred as Overloading and Generics, respectively

# Using Parent Class's methods within a Child Class

- A Child Class can utilize its Parent Class's methods using base:
- base is a reference to the Parent (Base) Class, just like this is a reference to the current class & object

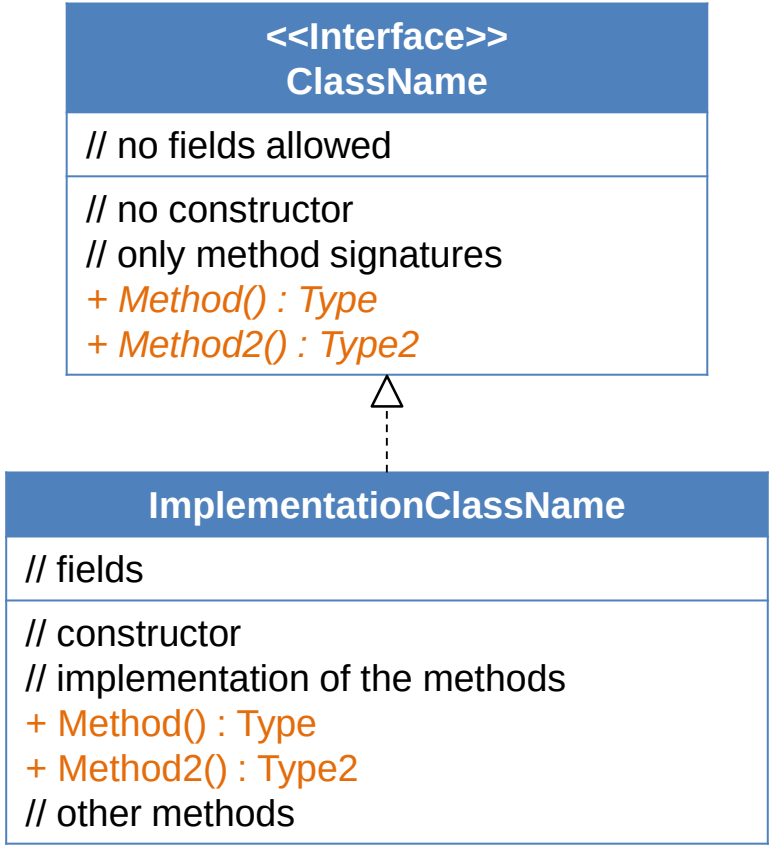
```
public class GrumpyCat : Cat
{
  public GrumpyCat(string quote) : base(quote)  // Call the Base Class constructor first,
      // passing in the variable quote
  {
    // Do nothing else
  }

  public override void Emote(string x)
  {
    base.Emote(x); // use the parent class's Emote method, then
    Console.WriteLine(x); // do something special in addition to the parent class's Emote
  }
}
```

- If a Parent Class has private fields, base.field won't work (can't access it directly because it's private), but base.Property and base.Method() will work.

# Interfaces

- An Interface is a special version of a Parent Class
- It has no fields or constructor
- It only defines methods signatures, e.g.:
  - public double GetArea();

- Used to establish a contract between two software subsystems or components:
- One component (eg Main) knows how to call these methods, what it needs to pass in and what it will get as a return
- The other component (the Implementation Class) implements the methods and provides the functionality
  - The implementation class only inherits the method signatures from the interface class



# Interfaces Example

```
public interface IAccount
{
   void PostInterest();
   void DeductFees(IFeeSchedule feeSchedule);
}

public class BusinessAccount : IAccount
{
   void IAccount.PostInterest()
   {
      // Code to post interest using the most favorable rate.
   }

   void IAccount.DeductFees(IFeeSchedule feeSchedule)
   {
      // Code to change a preferred rate for various services.
   }
}
```

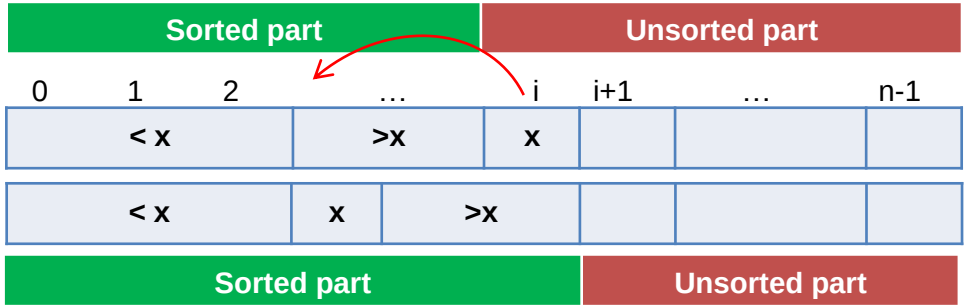
In C#, we have to define these methods with the Interface name prepended

# INSERTION SORT WITH LINKED LIST

# Insertion Sort

- Starting from the first index at first step, at each step we have array divided to two parts(sorted/unsorted)
- It means that at step i we have index 0 to i-1 sorted.


- Take index i (with value x) and place at right place in sorted part
- x's new index(j) is where values at lower indexes are less than and after that are more than x
- Then we have sorted part with length i+1

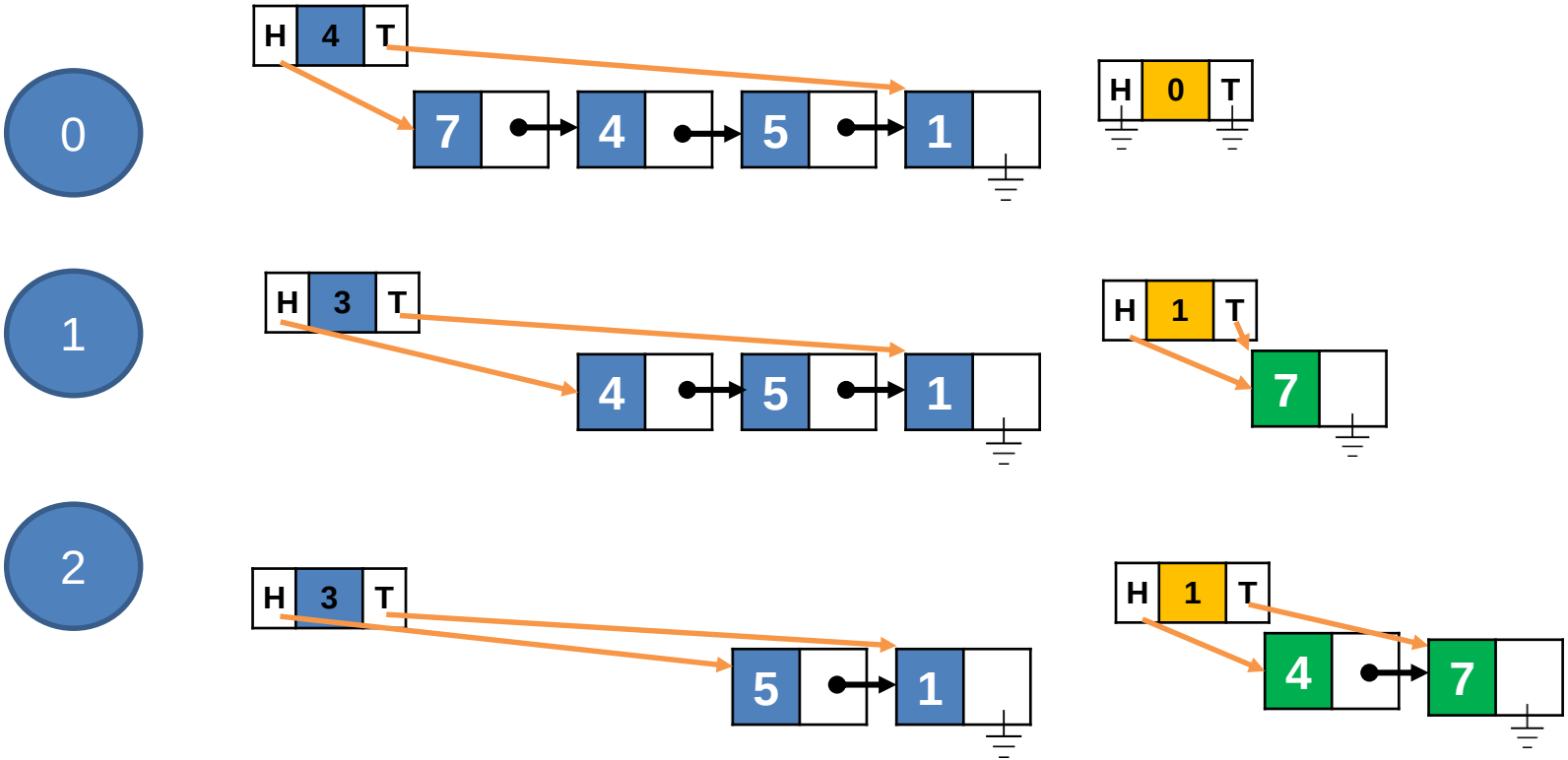


# Insertion sort: example

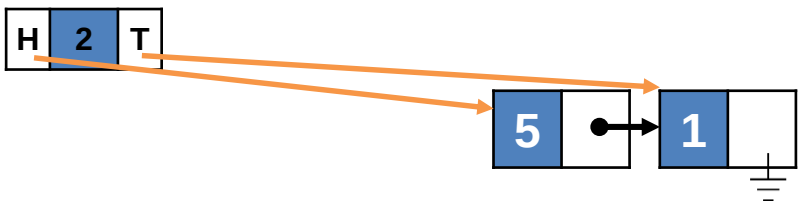
| | | | | | |
|---|---|---|---|---|---|
| 7 | -5 | 2 | 16 | 4 | unsorted |
| 7 | -5 | 2 | 16 | 4 | -5 to be inserted |
| ? | 7 | 2 | 16 | 4 | 7 > -5, shift |
| -5 | 7 | 2 | 16 | 4 | reached left boundary, insert -5 |
| -5 | 7 | 2 | 16 | 4 | 2 to be inserted |
| -5 | ? | 7 | 16 | 4 | 7 > 2, shift |
| -5 | 2 | 7 | 16 | 4 | -5 < 2, insert 2 |
| -5 | 2 | 7 | 16 | 4 | 16 to be inserted |
| -5 | 2 | 7 | 16 | 4 | 7 < 16, insert 16 |
| -5 | 2 | 7 | 16 | 4 | 4 to be inserted |
| -5 | 2 | 7 | ? | 16 | 16 > 4, shift |
| -5 | 2 | ? | 7 | 16 | 7 > 4, shift |
| -5 | 2 | 4 | 7 | 16 | 2 < 4, insert 4 |
| -5 | 2 | 4 | 7 | 16 | sorted |

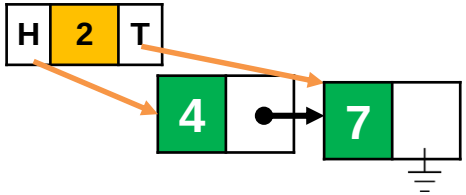
# Sorting items in linked list using insertion sort

- Steps:
  - Create an empty list that will keep the sorted list at the end.
  - Remove nodes one by one from the head of original list.
  - Insert the removed(current) node to proper location in the second list
    - The proper location is where before the first item larger than removed item
  - Insert removed node before the larger node

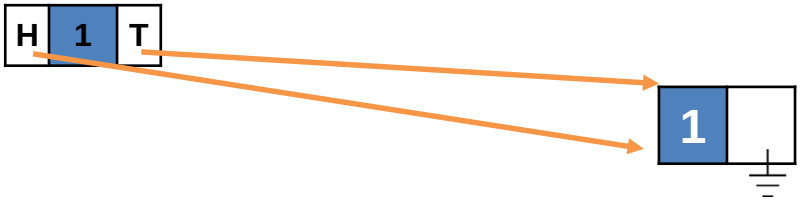

0
H
4
T
7
4
5
1
H
0
T
1
H
3
T
4
5
1
H
1
T
7
2
H
3
T
5
1
H
1
T
4
7

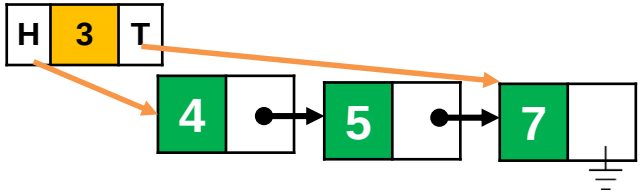
2



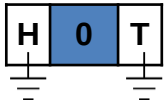



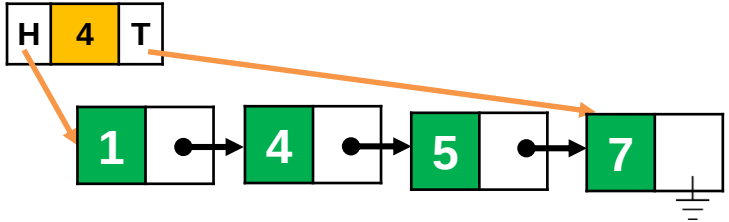

3




4



# THE END