Object Oriented Programming Lecture

Inheritance II

School of Computing and Information Technology
Faculty of Engineering and Computing
University of Technology, Jamaica

Email: dwwhite@utech.edu.jm, taedwards@utech.edu.jm

Object Oriented Programming

Expected Outcome

At the end of this lecture the student should be able to:

- Understand why objects of derived classes can be treated as objects of their base class
- State the order in which constructors and destructors are called in an inheritance hierarchy, and implement multiple inheritance
- Differentiate between concrete classes and methods and abstract classes and methods
- State the Yo-Yo problem

Object Oriented Programming

Topics to be covered in this lecture:

- Review of inheritance
- Objects of base versus derived classes
- Constructors and inheritance
- Destructors and inheritance
- Multiple inheritance
- Abstract methods and classes
- The Yo-Yo problem

Inheritance: Review

In the lecture on Inheritance – Part One, we learnt:

- Inheritance facilitates reuse of code
- Occurs where one class derives the state and behaviour from another class
- One or more child classes inherit the attributes and methods of the parent class

Inheritance: Review

Implementing single inheritance in code

In C++ we use:

```
class Child : public Parent
{



};
```

In Java we use:

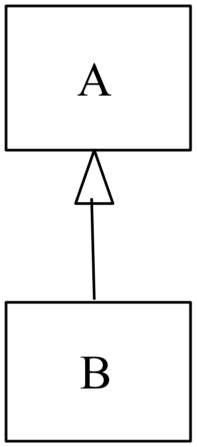
```
public class Child extends Parent
{



}
```

Inheritance: Base Class versus Derived Class

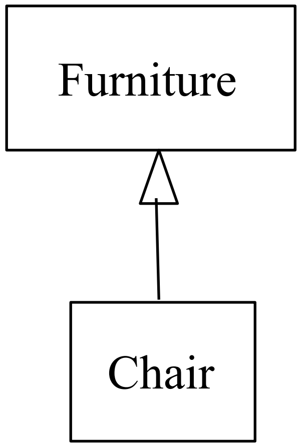
- Every derived class object is also automatically an object of the base class
- This propagates down an inheritance hierarchy, so children and grandchildren objects are also objects of the parent class
- However, objects of the base class are not objects of the derived class

Inheritance: Base Class versus Derived Class

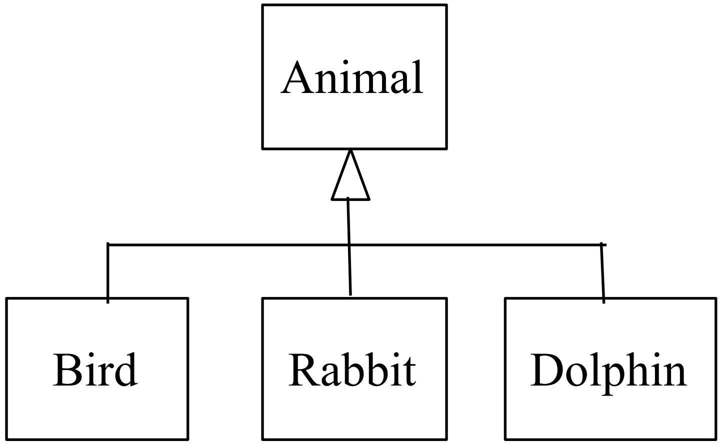
Every B is an A, but all
A's are not B's



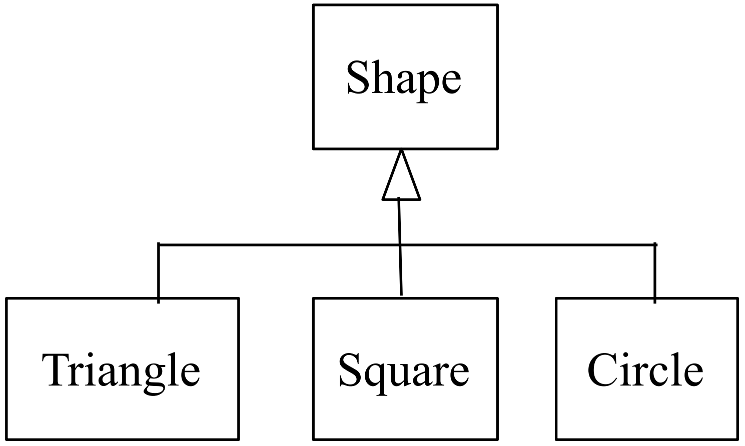
All chairs are furniture but not
all furniture are chairs



All rabbits are animals but not
all animals are rabbits

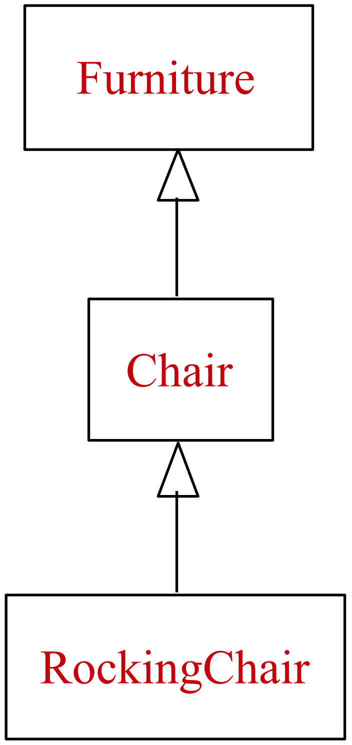


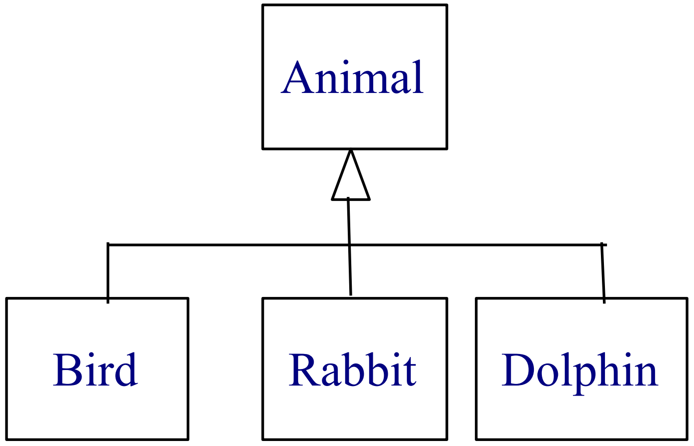
All circles are shapes but not
all shapes are circles



Inheritance: Base Class versus Derived Class

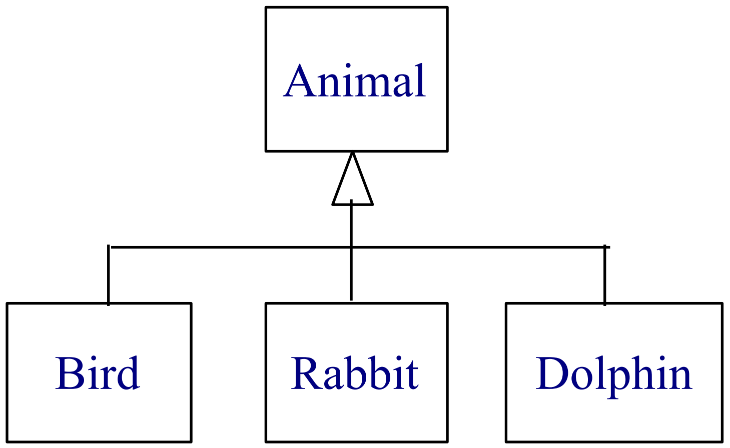
Since all chairs are furniture, anywhere a furniture is used, a chair can be used. So too can a rocking chair for the same reason – a rocking chair is an item of furniture





If all animals can move, then so too can all birds, rabbits and dolphins. You can replace an animal with a bird then invoke the same calls on the bird object as on the animal object

Inheritance: Base Class versus Derived Class



If all animals can move, you can replace an animal with a bird then invoke the same calls on the bird object as on the animal object without causing any errors. The program will still function correctly.

C++

```
Animal x;
x.move();
```

$\longrightarrow$

C++

```
Bird x;
x.move();
```

Java

```
Animal x = new Animal();
x.move();
```

$\longrightarrow$

Java

```
Bird x = new Bird();
x.move();
```

Inheritance: Constructors

- Constructors are not inherited by derived classes
- When an object is created, a constructor for that object gets called automatically
- Since a child class objects is also an object of its parent class, then:
- When a child class object is created, it implies that a parent class object is created
- Thus the constructors for both the child class and the parent class will be called automatically

Inheritance: Constructors

- When a child class object is created, its parent class' constructor is called first, followed by the child class' constructor
- The same applies in an inheritance hierarchy:
- when an object of the grandchild is created,
- the grandparent's constructor is called first,
- followed by the grandchild's immediate parent's constructor,
- followed by the grandchild's constructor

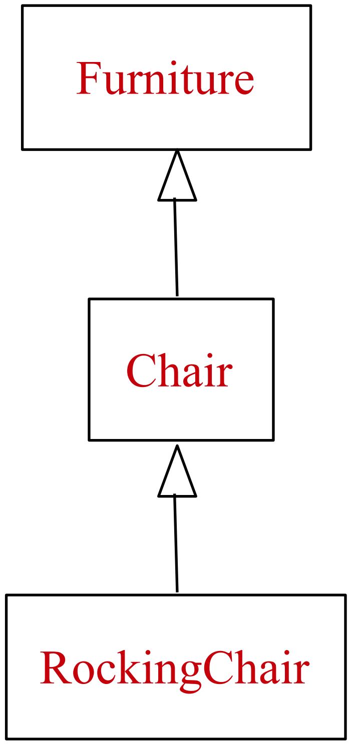
Inheritance: Constructors

When an object of RockingChair is created, the constructors are automatically called in the following order:

1st call : Furniture( )
2nd call : Chair( )
3rd call : RockingChair( )

When an object of Chair is created, the constructors are automatically called in the following order:

1st call : Furniture( )
2nd call : Chair( )



Inheritance: Constructors

- Every time a new child class object is created, usually the default constructor of its parent class' constructor is called before the constructor of the child class
- Instead of the default constructor, you can force another constructor in the child class to be called
- In C++ use : ParentClass(arguments) in the child class
- In Java use the keyword super in the child class

Inheritance: Constructors

C++

```
class Parent
{
        private:
        int x;

        public:
        Parent(int xv)
        {
                x  = xv;
        }
};

class Child : public Parent
{
        private:
        int y;

        public:
        Child(int xv, int yv)
        : Parent(xv)
        {
                y  = yv;
        }
};
```

Java

```
public class Parent
{
        private int x;

        public Parent(int xv)
        {
                x  = xv;
        }
}

public class Child extends Parent
{
        private int y;

        public Child(int xv, int yv)
        {
                super(xv);
                y  = yv;
        }
}
```

Inheritance: Destructor

- When ever an object is destroyed, its destructor is called
- Destructors are called in reverse order in an inheritance hierarchy
- If a child class object is being destroyed:
- its destructor is called first,
- followed by it parent's destructor
- C++ has destructors, but while Java does not have destructors, it has finalizers which are similarly called when an object is destroyed

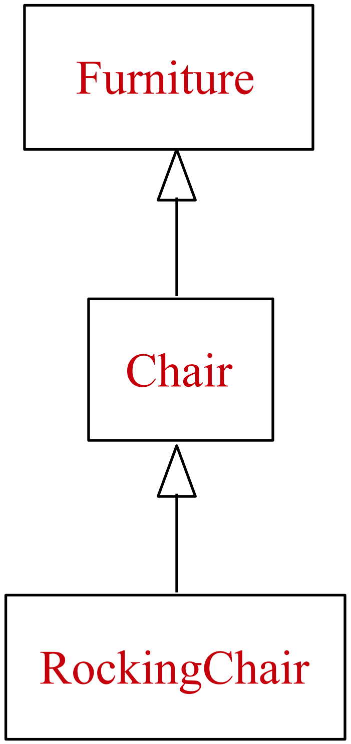
Inheritance: Destructors

When an object of RockingChair is destroyed, the destructors are automatically called in the following order:

1st call : RockingChair's destructor
2nd call : Chair's destructor
3rd call : Furniture's destructor

When an object of Chair is destroyed, the destructors are automatically called in the following order:

1st call : Chair's destructor
2nd call : Furniture's destructor



Inheritance: Multiple Inheritance

- The examples of inheritance we have looked at so far in code have all been of single inheritance
- Both C++ and Java support single inheritance
- Single inheritance involves one or more child classes inheriting from only one parent class
- Multiple inheritance involves one or more child classes inheriting from more than one parent at the same time

Inheritance: Multiple Inheritance

- Some OOP languages such as C++ readily support multiple inheritance
- C++ uses the same syntax for multiple inheritance as single inheritance, the parent classes being inherited are separated by commas
- While Java does not support multiple inheritance outright, it can be achieved through the use of interfaces
- Java uses keyword implements for simulating multiple inheritance using interfaces

Inheritance: Multiple Inheritance in C++

C++ Example

```
class Parent1
{

};

class Parent2
{

};

class Child : public
Parent1, public Parent2
{

};
```

Another C++ Example

```
class X
{

};

class Y
{

};

class Z
{

};

class A : public X, public Y,
public Z
{

};
```

Inheritance: Simulating Multiple Inheritance in Java using Interfaces

```
interface X
{
   void K();
}

interface Y
{
   float L(float r);
}

public class C implements X, Y
{
   public void K(){};
   public float L(float r){};
}
```

The methods in a Java interface are Implicitly public and can be either abstract, default or static, our focus is on abstract methods.

A class using the interface must implement all abstract methods or that class must be declared as abstract

Inheritance: Simulating Multiple Inheritance in Java using Interfaces

```
interface X { void K(); }

interface Y { float L(float r); }

interface Z
{
   double PI = 3.142;
   int MAX = 100;
}

public class C implements X, Y, Z
{
   public void K(){
      for(int i=0; i < MAX; i++){
         System.out.print("i = " + (i*i));
      }
   }

   public float L(float r){
      return PI * (r * r);
   }
}
```

The data members / variables in a Java interface are implicitly public, static and final

A class using the interface can then access these data members

Inheritance: Simulating Multiple Inheritance in Java using Interfaces

- Note that Java does not support multiple inheritance (you can only inherit from or extend one class)
- However you can implement multiple interfaces
- Therefore interfaces in Java are an alternative to multiple inheritance
- You can also inherit from a class while implement one or more interfaces, for example:

```
public class C extends A implements B
{

}
```

Inheritance: Abstract Methods and Classes

- So far, we have looked at concrete methods, for which we have supplied their implementation, i.e. they have a body
- We now look at abstract methods, which are declared in a class but for which no implementation (body) is provided in that class
- Abstract methods in a parent class are inherited by a child class just as concrete methods are

Inheritance: Abstract Methods and Classes

- A concrete class is a class that has all its constituent methods fully implemented
- Objects can be instantiated (created as an instance) from concrete classes
- An abstract class is class which has one or more of its methods not implemented (i.e. it has at least one abstract method)
- Objects cannot be instantiated from an abstract class – i.e. an abstract class cannot be used to create objects

Inheritance: Abstract Methods and Classes

- If a parent class A is an abstract class, and is inherited by a child class B, then class B too becomes an abstract class unless it implements the abstract methods in the parent class A
- This follows through in an inheritance hierarchy, so if class A is abstract, and class B inherits from class A and does not implement its abstract methods, then if class C inherits from class B, class C also become abstract, and so on, if class C does not implement the abstract methods

Inheritance: Abstract Methods and Classes

C++ Concrete Class

```
//concrete class A
class A
{
   private:
      int x;

   public:
      A(int xv)//primary
constructor
      {
         x = xv;
      }

      void cm()//concrete method
      {
         x = x * 2;
      }
};
```

Java Concrete Class

```
//concrete class A
public class A
{
   private int x;

   public A(int xv)//primary
constructor
   {
      x = xv;
   }

   public void cm( )//concrete
method
   {
      x = x * 2;
   }
}
```

Inheritance: Abstract Methods and Classes

C++ Abstract Class

```
//abstract class A
class A
{
   private:
      int x;

   public:
      A(int xv){
         x = xv;
      }

   //abstract method ‘AM()’
      virtual void AM() = 0;
};
```

Java Abstract Class

```
//abstract class A
public abstract class A
{
   private int x;

   public A(int xv){
      x = xv;
   }

   //abstract method AM()
   public abstract void AM();
}
```

Inheritance: Abstract Methods and Classes

Inheriting C++ Abstract Class

```
//abstract class A
class A
{
   public:

   //abstract method AM()
   virtual void AM() = 0;
};

//concrete class B
class B : public A
{
   public:

   //concrete method AM()
   void AM(){}
};
```

Inheriting Java Abstract Class

```
//abstract class A
public abstract class A
{
   //abstract method AM()
   public abstract void AM();
}

//concrete class B
public class B extends A
{
   //concrete method AM()
   public void AM(){}
}
```

Inheritance: Abstract Methods and Classes

<u>Inheriting C++ Abstract Class</u>

```
//abstract class A
class A
{
   public:

   //abstract method AM()
   virtual void AM() = 0;
};

//abstract class B
class B : public A
{


};
```

<u>Inheriting Java Abstract Class</u>

```
//abstract class A
public abstract class A
{
   //abstract method AM()
   public abstract void
AM();
}

//abstract class B
public class B extends A
{


}
```

Inheritance: Abstract Methods and Classes

Inheriting C++ Abstract Class

```
//abstract class A
class A
{
   public:
   //abstract method AM()
   virtual void AM() = 0;
};

//abstract class B
class B : public A
{

};

//concrete class C
class C : public B
{
   public:
   //concrete method AM()
   void AM(){}
};
```

Inheriting Java Abstract Class

```
//abstract class A
public abstract class A
{
   //abstract method AM()
   public abstract void AM();
}

//abstract class B
public class B extends A
{


}

//concrete class C
public class C extends B
{
   //concrete method AM()
   public void AM(){}
}
```

The Yo-Yo Problem

- The Yo-Yo problem occurs when an inheritance hierarchy is too long and complicated, it is difficult for a person to understand
- A person trying to understand the complicated inheritance relationship has to flip back and forth among many pages, like a yo-yo toy, hence the name
- The Yo-Yo problem is due to excessive fragmentation and as such is counterproductive
- Refactoring the inheritance hierarchy can fix the Yo-Yo software engineering problem