




Lecture 12

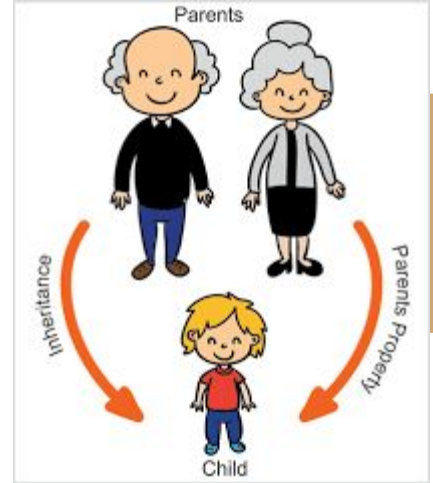
Inheritance
Adapter Design Pattern
Inner Class
Object Class



reminder

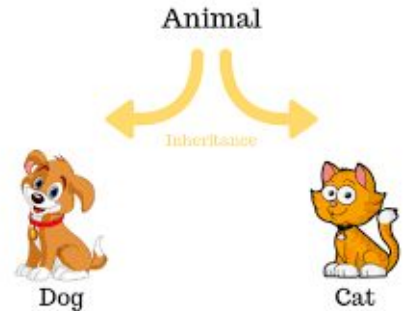
- Mic
- Exam on Friday

Inheritance



Inheritance in Java

- It is the ***mechanism*** in Java (Python) by which one class is allowed to inherit the features (fields and methods) of another class.
- The subclass can *add* its own fields and methods in addition to the superclass fields and methods.
- The subclass can *override* methods from the superclass



Inheritance

```
public class Foo {  
    int field1;  
    float field2;  
  
    void method() {  
        // body  
    }  
}
```

Inheritance

```
public class Foo {  
    int field1;  
    float field2;  
  
    void method() {  
        // body  
    }  
}
```

```
public class Bar {  
    int field1;  
    float field2;  
    String field3;  
  
    void method() {  
        // body  
    }  
    void anotherMethod() {  
        // another body  
    }  
}
```

Inheritance. Idea

```
public class Foo {  
    int field1;  
    float field2;  
  
    void method() {  
        // body  
    }  
}
```

```
public class Bar {  
    int field1;  
    float field2;  
    String field3;  
  
    void method() {  
        // body  
    }  
    void anotherMethod() {  
        // another body  
    }  
}
```

Inheritance. Idea. Demo 1

```
public class Foo {  
    int field1;  
    float field2;  
  
    void method() {  
        // body  
    }  
}
```

```
public class Bar extends Foo{  
    String field3;  
  
    void anotherMethod() {  
        // another body  
    }  
}
```



```
public class DSC {  
  
    public String message = "Simple class";  
    public void displayMessage() {  
        System.out.println(message);  
    }  
  
}
```

```
class Tester {  
    public static void main (..) {  
        DSC class1 = new DSC();  
        class1.studentCount(100);  
    }  
}
```

```
public class DSC30 extends DSC {  
    public String ms = "DSC30";  
    public String message = "Best Class Ever";  
  
    public void studentCount(int num) {  
        System.out.println(ms + ":" + num);  
    }  
  
    @Override  
    public void displayMessage() {  
        System.out.println(message);  
    }  
}
```

A: "DSC30: 100"

B: "Best Class ever: 100"

C: Something else (no error)

D: Error

“Is - a” relationship

```
public class Foo {  
    int field1;  
    float field2;  
    void method() { }  
}
```

```
public class Bar extends Foo{  
    String field3;  
  
    void anotherMethod() {  
        // another body  
    }  
}
```

“Is - a” relationship

```
public class Foo {  
    int field1;  
    float field2;  
    void method() { }  
}
```

```
public class Bar extends Foo{  
    String field3;  
  
    void anotherMethod() {  
        // another body  
    }  
}
```



IS - A

“Is - a” relationship

```
public class DSC {
```

```
    ...
```

```
}
```

```
public class DSC30 extends DSC{
```

```
    ...
```

```
}
```

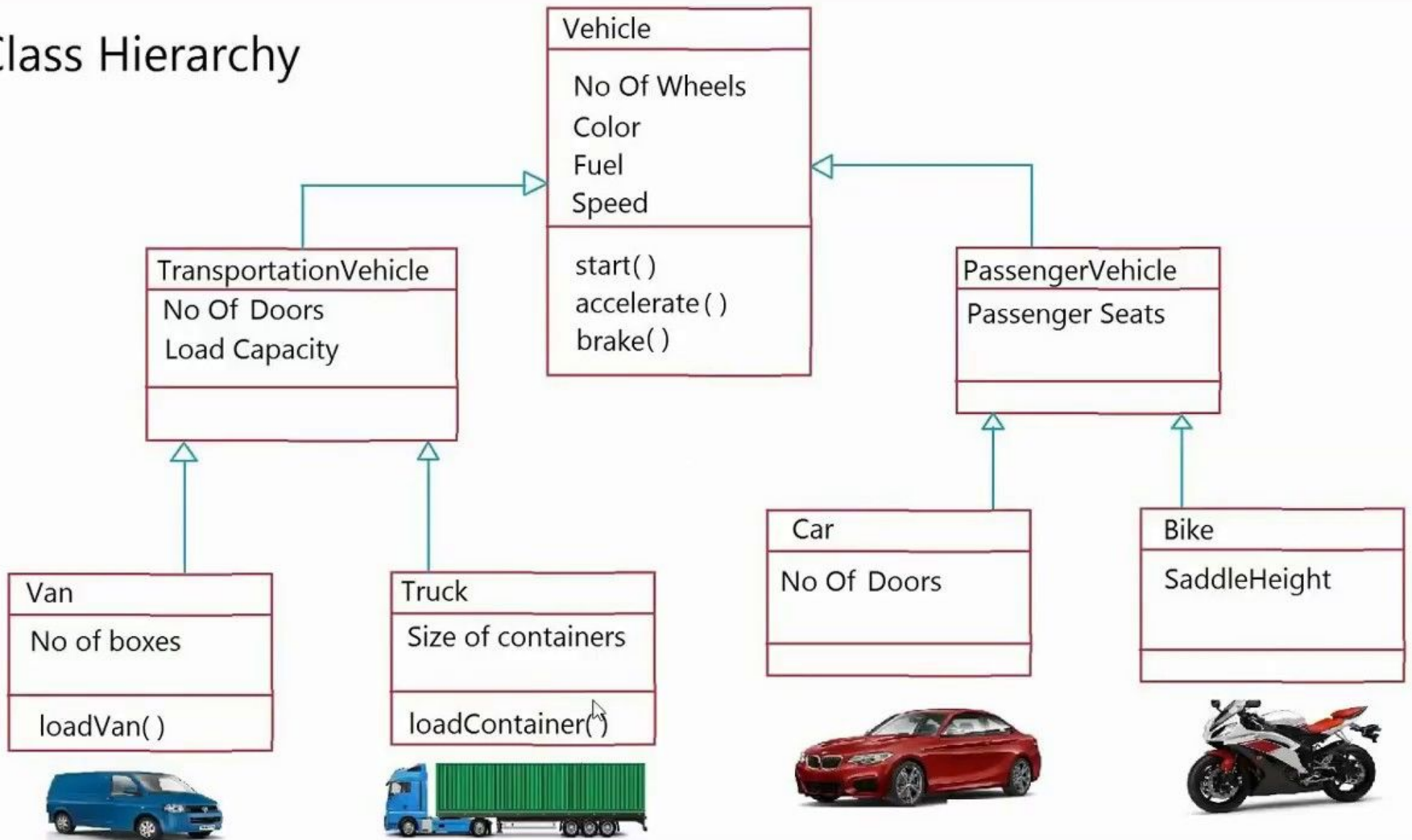
DSC



IS - A

DSC30

Class Hierarchy



Method overriding (demo 2)

- If a subclass (child class) provides the *specific* implementation of the method that has been declared by one of its parent class, it is known as method *overriding*.
- Method overriding is used to provide the specific implementation of a method which is already provided by its superclass.

Rules for Java Method Overriding

1. The method must have the *same* name as in the parent class.
2. The method must have the *same* parameter list as in the parent class.
3. There must be an IS-A relationship (inheritance).

How to find: closer scope first (just like variable names in Python)

```
public class DSC {  
  
    public String message = "Simple class";  
    public void displayMessage() {  
        System.out.println(message);  
    }  
  
}
```

```
public class DSC30 extends DSC {  
    public String ms = "DSC30";  
    public String message = "Best Class Ever";  
  
    public void studentCount(int num) {  
        System.out.println(ms + ":" + num);  
    }  
  
    @Override  
    public void displayMessage() {  
        System.out.println("In DSC30 Class");  
    }  
}
```

```
public static void main (String [] args) {  
  
    DSC class1 = new DSC();  
    DSC30 class30 = new DSC30();  
    class30.displayMessage();  
  
}
```

A: "Simple Class"

B: "Best Class ever"

C: "DSC30"

D: "In DSC30 class"

E: Error

Private fields: Visible by the class members only. Demo

Check time for question.

Protected fields: Visible by the subclasses only. Demo



ADAPTER PATTERN DESIGN



Stacks as Linked Lists

```
public class myList{  
    private class ListNode {  
        int element;  
        ListNode next;  
    }  
  
    private ListNode head;  
  
    public boolean addFirst(int elem)  
    {  
        ListNode n = new ListNode();  
        n.next = head;  
        n.element = elem;  
        head = n;  
        return true;  
    }  
}
```

Stacks as Linked Lists

```
public class myList {  
    private class ListNode {  
        int element;  
        ListNode next;  
    }  
  
    private ListNode head;  
  
    public boolean addFirst(int elem)  
    {  
        ListNode n = new ListNode();  
        n.next = head;  
        n.element = elem;  
        head = n;  
        return true;  
    }  
}
```

```
public class myStack {  
    private class StackNode {  
        int element;  
        StackNode next;  
    }  
  
    private StackNode top;  
  
    public boolean push(int elem){  
        StackNode n = new StackNode();  
        n.next = top;  
        n.element = elem;  
        top = n;  
        return true;  
    }  
}
```

Why redo all the work???



ADAPTER PATTERN DESIGN



Adapter Pattern

We need to implement the Stack interface with the following methods:

- `push (int element)` - add element on the stack
- `int pop()` – remove element from top of the stack
- `int peek()` – return the element at the top of the stack

We would like to find a way to **reduce** our work

Adapter Pattern

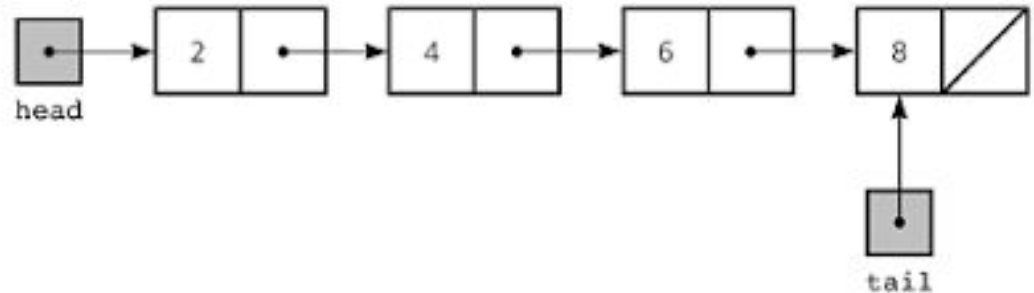
We know that there is a nice implementation of a *Double-ended List* (head and tail) interface **DList** which provides the following methods:

- `addFront (int element)`
- `int removeFront()`
- `int peekFront()`

Adapter Pattern

We know that there is a nice implementation of a *Double-ended List* (head and tail) interface **DList** which provides the following methods:

- addFront (int element)
- int removeFront()
- int peekFront()
- addBack(int element)
- int removeBack()
- int peekBack()



Idea: Inheritance!



We could just make **Stack** *extend* the **DList** and write the additional methods by using other existing methods.

```
public class Stack extends DList{  
    ...  
  
    public int pop(){  
        return removeFront();  
        // a call to the removeFront  
        // inherited from DList  
    }  
    ...  
}
```

Prons? Cons?

It is possible just make Stack extend the DList:

```
public class Stack extends Dlist
```

Which of the following is the **biggest** drawback of this approach:

- A:** It is inefficient from a *running-time* perspective: A double-ended linked list is not a good choice for a *Stack* implementation.
- B:** It is incorrect. There is no way to implement all the methods in the *Stack* interface with the methods in the double-ended linked list.
- C:** It exposes methods that are *not supposed* to be part of the Stack interface.

Inheritance is not always the right answer

We know that there is a nice implementation of a *Double-ended List* (head and tail) interface **DList** which provides the following methods:

- `addFront (int element)`
- `int removeFront()`
- `int peekFront()`

- `addBack(int element)`
- `int removeBack()`
- `int peekBack()`

Inheritance is not always the right answer

We know that there is a nice implementation of a *Double-ended List* (head and tail) interface **DList** which provides the following methods:

- `addFront (int element)`
- `int removeFront()`
- `int peekFront()`

The other methods in the *DList* are public and accessible by anyone.

- `addBack(int element)`
- `int removeBack()`
- `int peekBack()`

But a *Stack* does not expose such methods!

(Ex: `addBack`, `removeBack()` etc)

Adapter Design Pattern

- In software engineering, one of the classic “design patterns” is the *adapter*.
- An *adapter* is a class that “maps” from the interface of one ADT -- the one we’re trying to implement -- into the interface of another ADT that already exists.
- If we adapt an **ADT B** to implement another **ADT A**, then every method of A must be “converted” into a related call of B.

Adapter Design Pattern

- I can use a **private** *DList* variable
- And use its methods within a *Stack* class

Adapter Design Pattern

```
import java.util.*;

public class MyStack {
    private List stack = new LinkedList();
    public int pop() {
        return stack.remove(0);
    }

    public void push(int element) {
        stack.add(0, element);
    }
}
```

```
public static void main(...){
    MyStack st = new MyStack();
    st.push(4);
    st.push(5);
    st.pop();
    System.out.print(st.pop());
}
```

TODO!

Think, how to use methods from DLL class to implement a queue!



Nested Classes



Nested classes: briefly

- In Java, class can have *another class* as its member.
- The class written within is called the **nested class**, and the class that holds the inner class is called the **outer class**.

Syntax:

```
class Outer_Demo {  
    class Inner_Demo {  
  
    }  
}
```

Nested classes

- Inner classes are a *security* mechanism in Java.
- In general, a class **cannot** be associated with the access modifier *private*, but if we have the class as a member of other class, then the *inner* class can be made *private*.

```
class OuterDemo {  
    int num;  
    // inner class  
    private class InnerDemo {  
        public void print() {  
            System.out.println("This is an inner class");  
        }  
    }  
}
```

```
class OuterDemo {  
    int num;  
    // inner class  
    private class InnerDemo {  
        public void print() {  
            System.out.println("This is an inner class");  
        }  
    }  
}  
// Accessing the inner class from the method within  
void displayInner() {  
    InnerDemo inner = new InnerDemo();  
    inner.print();  
}  
}
```

```
class OuterDemo {
```

```
    int num;
```

```
    // inner class
```

```
    private class InnerDemo {
```

```
        public void print() {
```

```
            System.out.println("This is an inner class");
```

```
        }
```

```
    }
```

```
    // Accessing the inner class from the method within
```

```
    public void displayInner() {
```

```
        InnerDemo inner = new InnerDemo();
```

```
        inner.print();
```

```
    }
```

```
}
```

```
Outer_Demo outer = new Outer_Demo();
```

```
// Accessing the displayInner() method.
```

```
outer.displayInner();
```

```
This is an inner class.
```

Question

```
public class ExceptTest {  
  
    public int num = 10;  
    private int num2 = 20;  
  
    public class inner{  
        public int test() {  
            int num = 100;  
            return num2 + num;  
        }  
    }  
}
```

```
public class b {  
  
    public static void main (String[] args) {  
        ExceptTest t = new ExceptTest();  
        System.out.println(t.test());  
    }  
}
```

What is the output?

A: 30

B: 120

C: Error (due to privacy)

D: Error (due to inner class)

Question

```
public class ExceptTest {  
  
    public int num = 10;  
    private int num2 = 20;  
  
    public int test() {  
        return num + num2;  
    }  
  
}  
  
public class b {  
  
    public static void main (String[] args) {  
        ExceptTest t = new ExceptTest();  
        System.out.println(t.num + t.num2);  
    }  
  
}
```

What is the output?

- A: 10
- B: 20
- C: 30
- D: Error

Question

```
public class ExceptTest {  
  
    public int num = 10;  
    private int num2 = 20;  
  
    public int test() {  
        return num + num2;  
    }  
  
}
```

```
public class b {  
  
    public static void main (String[] args) {  
        ExceptTest t = new ExceptTest();  
        System.out.println(t.test());  
    }  
  
}
```

What is the output?

- A: 10
- B: 20
- C: 30
- D: Error

Mapping Attributes

- Before deciding on what methods to use, one needs to **map** the corresponding attributes.
- **For example:** To use the List as a Stack, we need to map the **top** of the stack to some position in the list (front or back — our choice, but how to choose?)

Mapping methods

- Once this is done, we can map the methods on top of the stack to methods operating on the head of the List.
- If we choose the front....
 - push -> addFront
 - pop -> removeFront
 - peek -> peekFront

Mapping

Mapping between single linked list and Stack



- What is the time cost of adding or removing an element at the head or at the tail of an N-element List...
 - If List is implemented using a singly linked list?
Head: _____ Tail (no direct pointer): _____
- A. $O(1)$, $O(1)$
- B. $O(1)$, $O(n)$
- C. $O(n)$, $O(n)$
- D. $O(n^2)$, $O(n^2)$
- E. Other/none/more

Mapping between single linked list and Stack



- What is the time cost of **removing** an element at the head or at the tail of an N-element List...
 - If List is implemented using a singly linked list?
Head: _____ Tail (*with* direct pointer): _____
- A. $O(1), O(1)$
- B. $O(1), O(n)$
- C. $O(n), O(n)$
- D. $O(n^2), O(n^2)$
- E. Other/none/more

Move to DLL slides for running time questions

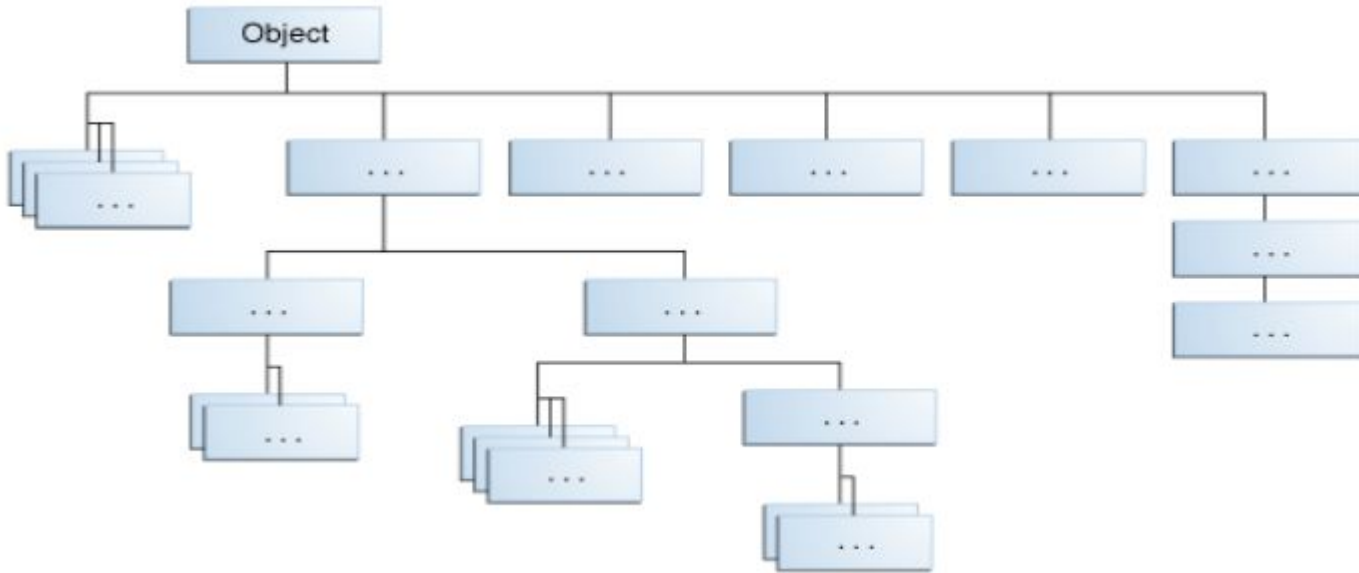


Object class



The Java Platform Class Hierarchy

The Object class, defined in the java.lang package, defines and implements behavior common to all classes



```
public class Bicycle {
```

```
    Instance variables;
```

```
    Constructor(s)
```

```
    public void setHight(int newValue){}
```

```
    public void speedUp(int increment) {}
```

```
}
```

```
public class Bicycle {  
  
    Instance variables;  
    Constructor(s)  
    public void setHight(int newValue){}  
    public void speedUp(int increment) {}  
}
```

```
public class MountainBike extends Bicycle {  
  
    Its own instance variables;  
    Its own constructor(s)  
    public void setSpeed(int newValue) {}  
}
```

```
public class Bicycle {  
  
    Instance variables;  
    Constructor(s)  
    public void setHeight(int newValue){}  
    public void speedUp(int increment) {}  
}
```

```
MountainBike myBike = new MountainBike();
```

```
public class MountainBike extends Bicycle {  
  
    Its own instance variables;  
    Its own constructor(s)  
    public void setSpeed(int newValue) {}  
}
```

```
public class Bicycle {  
  
    Instance variables;  
    Constructor(s)  
    public void setHeight(int newValue){}  
    public void speedUp(int increment) {}  
}
```

```
public class MountainBike extends Bicycle {  
  
    Its own instance variables;  
    Its own constructor(s)  
    public void setSpeed(int newValue) {}  
}
```

`MountainBike myBike = new MountainBike();`

`MountainBike` is descended from `Bicycle` and `Object`.

```
public class Bicycle {  
  
    Instance variables;  
    Constructor(s)  
    public void setHeight(int newValue){}  
    public void speedUp(int increment) {}  
}
```

```
public class MountainBike extends Bicycle {  
  
    Its own instance variables;  
    Its own constructor(s)  
    public void setSpeed(int newValue) {}  
}
```

```
MountainBike myBike = new MountainBike();
```

MountainBike is descended from Bicycle and Object.

Therefore, a MountainBike:

- “Is - a” Bicycle and
- “Is - a” Object,

and it can be used wherever Bicycle or Object objects are called for.

```
public class Bicycle {  
  
    Instance variables;  
    Constructor(s)  
    public void setHeight(int newValue){}  
    public void speedUp(int increment) {}  
}
```

```
public class MountainBike extends Bicycle {  
  
    Its own instance variables;  
    Its own constructor(s)  
    public void setSpeed(int newValue) {}  
}
```

```
MountainBike myBike = new MountainBike();
```

Therefore, a **MountainBike**:

- “Is - a” **Bicycle** and
- “Is - a” **Object**,

and it can be used wherever **Bicycle** or **Object** objects are called for.

Casting:

```
Object temp = new MountainBike();
```

then temp is both an **Object** and a **MountainBike**

Implicit casting: a subclass can be used in place of a superclass

```
public class Bicycle {  
  
    Instance variables;  
    Constructor(s)  
    public void setHeight(int newValue){}  
    public void speedUp(int increment) {}  
}
```

```
public class MountainBike extends Bicycle {  
  
    Its own instance variables;  
    Its own constructor(s)  
    public void setSpeed(int newValue) {}  
}
```

```
MountainBike myBike = new MountainBike();
```

Therefore, a **MountainBike**:

- “Is - a” **Bicycle** and
- “Is - a” **Object**,

and it can be used wherever **Bicycle** or **Object** objects are called for.

Casting:

```
Object temp = new MountainBike();
```

then temp is both an **Object** and a **MountainBike**

Implicit casting: a subclass can be used in place of a superclass

```
MountainBike myBike = temp; ← Compile error
```



```
public class Bicycle {  
  
    Instance variables;  
    Constructor(s)  
    public void setHeight(int newValue){}  
    public void speedUp(int increment) {}  
}
```

```
public class MountainBike extends Bicycle {  
  
    Its own instance variables;  
    Its own constructor(s)  
    public void setSpeed(int newValue) {}  
}
```

```
MountainBike myBike = new MountainBike();
```

Therefore, a **MountainBike**:

- “Is - a” **Bicycle** and
- “Is - a” **Object**,

and it can be used wherever **Bicycle** or **Object** objects are called for.

Casting:

```
Object temp = new MountainBike();
```

then temp is both an **Object** and a **MountainBike**
Implicit casting

```
MountainBike myBike = temp; ← Compile error
```

```
MountainBike myBike = (MountainBike)temp;  
Explicit casting
```

Casting:

```
Object temp = new MountainBike();
```

then temp is both an **Object** and a **MountainBike**
Implicit casting

```
MountainBike myBike = temp; ← Compile error
```

```
MountainBike myBike = (MountainBike)temp;
```

Explicit casting

