Inheritance

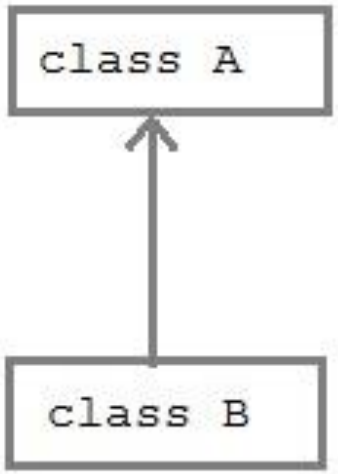
Index

Basics

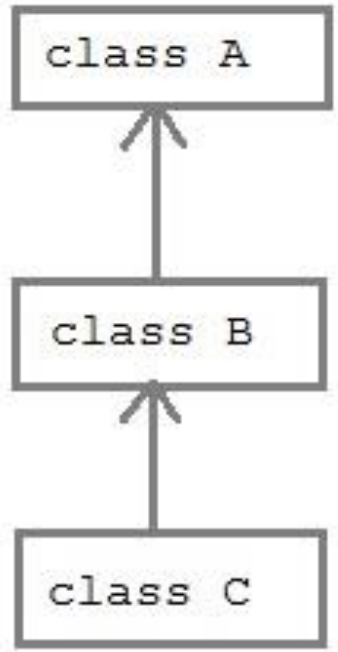
- Superclass
  - Class that is inherited
- Subclass
  - Class that does the inheritance.
  - Inherit all instance variables and method defined in superclass
  - Subclass is specialized version of Superclass

General form :

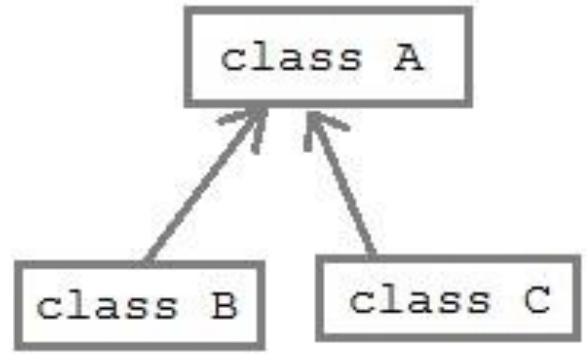
```
class subclass-name extends superclass-name {
 // body of class
}
```



Simple Inheritance



Multilevel inheritance



Heirarchical inheritance

```
class A {
int i, j;
void showij() {
System.out.println("i and j: " + i + " " +
j); }}

class B extends A {
int k;
void showk() {
System.out.println("k: " + k); }

void sum() {
System.out.println("i+j+k: " + (i+j+k)); }}

class sample_superkeyword {
public static void main(String args[]) {
A superOb = new A();
B subOb = new B();
superOb.i = 10;
superOb.j = 20;
System.out.println("Contents of
superOb: ");
superOb.showij();
System.out.println();
/* The subclass has access to all
public members of
its superclass. */
subOb.i = 7;
subOb.j = 8;
subOb.k = 9;
System.out.println("Contents of
subOb: ");
subOb.showij();
subOb.showk();
System.out.println();
System.out.println("Sum of i, j and k
in subOb:");
subOb.sum(); }}
```

```
Contents of superOb:
i and j: 10 20

Contents of subOb:
i and j: 7 8
k: 9

Sum of i, j and k in subOb:
i+j+k: 24
```

Reference of subclass is assigned to superclass object

- General Format:-
  - Superclass_classname obj_name = Subclass_object;
- It is important to understand that it is the type of the reference variable—not the type of the object that it refers to—that determines what members can be accessed.
- That is, when a reference to a subclass object is assigned to a superclass reference variable, we access only to those parts of the object defined by the superclass

```
class superclass{
private int a;
int b;
void display(){
    System.out.println("Value of b inside superclass "+b); }}

class subclass extends superclass{
int k;
void disp(){
  System.out.println("value of  b(inside disp)"+" "+b); } }

public class demo {
 public static void main(String args[]){
subclass obj=new subclass();
superclass obj_sup=obj;
obj_sup.display();
//obj_sup.disp(); will give error if uncommented
 }}
```

```
Value of b inside
  superclass 0
```

Using super

- 2 uses of super keyword
- (1)To call constructor of immediate super class
- General Format
  - super(*arg-list);*
- **super( )** must always be the first statement executed inside a subclass' constructor.
- Why super()???

Subclass inherit member variables of superclass, so subclass initialize it.so super class used to initialize member variables .Also it allows member variables to be declared private in superclass

```
class Box1{
    double width;
    double height;
    double depth;Box1(){//code here}
       Box1(double w,double h,double d){
        width=w;
        height=h;
        depth=d;} }
class BoxWeight1 extends Box1 {
double weight;
BoxWeight1(double w, double h, double d, double m) {
//    width=w;
//    height=h;
//    depth=d;
              super(w, h, d); // call superclass constructor

weight = m;}

void display(){
    System.out.println("result is "+(width*height*depth*weight));  }}
```

```
public class sample_super {
    public static void
main(String[] args) {
        BoxWeight1 obj=new
BoxWeight1(2, 3, 4,5);
        obj.display(); }}
```

Result is 120

```
class Box1{
   private double width;
 private  double height;
 private double depth;Box1(){//code here}
       Box1(double w,double h,double d){
        width=w;
        height=h;
        depth=d;} }
class BoxWeight1 extends Box1 {
double weight;
BoxWeight1(double w, double h, double d, double m) {
    '
    |
                super(w, h, d);
    (



weight = m;}

void display(){
    //System.out.println(“value of weight  is "+weight);  }}
```

We can even use super to pass object also.

```
// construct clone of an object
BoxWeight(BoxWeight ob) { // pass object to constructor
super(ob);
weight = ob.weight;
}
```

Note:- super(ob) invoke constructor of Box(Box box)
That is possible because…

a superclass variable can be used to reference any object derived from that class.

Second use of super

- refers to the superclass of the subclass in which it is used
- general form:
  - super.*member* // *member* can be either a method or an instance variable.
- applicable to situations in which member names of a subclass hide members by the same name in the superclass.
- super can be used to call methods that are hidden by a subclass.

```
class A {
int i;
}
// Create a subclass by extending class A.
class B extends A {
int i;  // this i hides the i in A
B(int a, int b) {
super.i = a; // i in A
i = b; // i in B
}
void show() {
System.out.println("i in superclass: " + super.i);
System.out.println("i in subclass: " + i);
}
}
class sample_superkeyword {
public static void main(String args[]) {
B subOb = new B(1, 2);
subOb.show();
}
}
```

Output:-

```
i in superclass: 1
i in subclass: 2
```

Creating a Multilevel Hierarchy

- Earlier used are simple class hierarchies that consist of only a superclass and a subclass.
- use a subclass as a superclass of another
- given three classes called **A**, **B**, and **C**,
  - C→B→A
  - **C** can be a subclass of **B**, which is a subclass of **A**

When Constructors Are Called

- In class hierarchy ,what order constructors are called???
  - Eg:- we have subclass B and superclass A which constructor gets called first???

    Constructors are called in order of derivation,
    from superclass to subclass
- Super() must be the first statement used in subclass constructor, so even if we don't call super(),default or parameterless constructor will get called of superclass.

```
class A {
A() {
System.out.println("Inside A's constructor."); }}

// Create a subclass by extending class A.
class B extends A {
B() {
System.out.println("Inside B's constructor.");  }}

// Create another subclass by extending B.
class C extends B {
C() {
System.out.println("Inside C's constructor."); }}

class sample_superkeyword {
public static void main(String args[]) {
C c = new C();
}
}
```

```
Inside A's constructor.
Inside B's constructor.
Inside C's constructor.
```

Method Overriding

If a super class method logic is not fulfilling sub class business requirements, sub class should override that method with required business logic. Usually in super class, methods are defined with generic logic which is common for all sub classes.

Eg:- Shape classes like Rectangle,Square,Circle etc...should have methods area() and perimeter().
Shape with generic logic but in sub classes Rectangle,Square,Circle etc.
These methods must have logic based on their specific logic

- Why method overriding???

```
class A {
int i, j;
A(int a, int b) {
i = a;
j = b;}
void show() {
System.out.println("i and j: " + i + " " + j); } }

class B extends A {
int k;

B(int a, int b, int c) {
super(a, b);
k = c; }

void show() {
System.out.println("k: " + k); }}

class sample_superkeyword {
public static void main(String args[]) {
B subOb = new B(1, 2, 3);
subOb.show(); // this calls show() in B
}}
```

k: 3

To call overridden method in class A

```
class B extends A {
int k;
B(int a, int b, int c) {
super(a, b);
k = c;
}

void show() {
super.show(); // this calls A's show()
System.out.println("k: " + k);
}
}
```

```
i and j: 1 2
k: 3
```

```
class A {
int i, j;
A(int a, int b) {
i = a;
j = b;
}
void show() {
System.out.println("i and j: " + i + " " + j);
}
}

class B extends A {
int k;
B(int a, int b, int c) {
super(a, b);
k = c;
}
// overload show()
void show(String msg) {
System.out.println(msg + k);
}
}
```

```
i and j: 1 2
    k: 3
```

```
class Override {
public static void main(String args[]) {
B subOb = new B(1, 2, 3);
subOb.show("This is k: "); // this calls show()
in B
subOb.show(); // this calls show() in A
}
}
```

Dynamic Method Dispatch

- Method overriding forms the basis for one of Java's most powerful concepts: *dynamic method dispatch*.
- Dynamic method dispatch is important because this is how Java implements run-time polymorphism.
- a superclass reference variable can refer to a subclass object. Java uses this fact to resolve calls to overridden methods at run time.
- How????

When an overridden method is called through a superclass reference, Java determines which version of that method to execute based upon the type of the object being referred to at the time the call occurs. Thus, this determination is made at run time.

```
class A {
void callme() {
System.out.println("Inside A's callme method"); }}

class B extends A {
void callme() {
System.out.println("Inside B's callme method"); }}

class C extends A {
void callme() {
System.out.println("Inside C's callme method"); }}

class sample_superkeyword  {
public static void main(String args[]) {
A a = new A(); // object of type A
B b = new B(); // object of type B
C c = new C(); // object of type C
A r;
r = a; // r refers to an A object
r.callme(); // calls A's version of callme
r = b; // r refers to a B object
r.callme(); // calls B's version of callme
r = c; // r refers to a C object
r.callme(); // calls C's version of callme   }}
```

Inside A’s callme method
Inside B’s callme method
Inside C’s callme method

Why overriding is so important

- Java to support run-time polymorphism
  - Java implements the “one interface, multiple methods”
- run-time polymorphism allows
  - code reuse

Why overloading

```
public class DataArtist {
    ...
    public void draw(String s) {
        ...
    }
    public void draw(int i) {
        ...
    }
    public void draw(double f) {
        ...
    }
    public void draw(int i, double f) {
        ...
    }
}
```

Basic difference between Overloading and Overriding

Overloading	Overriding
Differs in no. or type of arguments	Signature should be same
Occurs within a class	Occurs in inheritance concept
Compile time polymorphism	Run time polymorphism
Return type can be same or different	Return type must be same
It increase readability of the program	Used to provide specific implementation of method defined in superclass

Using Abstract Classes

- It’s a class that contains only a generalized form that’s shared by all subclasses, leaving to each subclass to fill in the details.
- Used
  - when superclass not able to create a meaningful implementation for a method.
  - When Superclass makes it mandatory for subclass to implement all the abstract methods
- General Format:-
  - abstract *type name(parameter-list)*;

Rules

- Any class that contains one or more abstract methods must also be declared abstract
- Abstract classes can have no objects i.e. it cannot be directly instantiated with new keyword.
- We cannot declare abstract constructors or abstract static methods .
- Any subclass of an abstract class must either implement all the abstract methods or declare itself as an **abstract**.

```
// A Simple demonstration of abstract.
abstract class A {
    abstract void callme();
// concrete methods are still allowed in abstract classes
void callmetoo() {
System.out.println("This is a concrete method.");
}
}
class B extends A {
void callme() {
System.out.println("B's implementation of callme.");
}
}

class AbstractDemo {
public static void main(String args[]) {
B b = new B();
b.callme();
b.callmetoo();
}
}
```

```
B's implementation of callme.
  This is a concrete method.
```

Using final with Inheritance

- 3 uses of final keyword
  - To create variable having constant values
  - To prevent overriding
  - To prevent inheritance

Using final to prevent Overriding

```
class A {
final void meth() {
System.out.println("This is a final method.");
}
}
class B extends A {
void meth() { // ERROR! Can't override.
System.out.println("Illegal!");
}
}
```

Using final to prevent Inheritance

Can we declare class as both final and abstract???

Eg:-import java.lang.Math;

```
public final class Math {

    public static final double E = 2.718281828459045;
    public static final double PI = 3.141592653589793;
```

The Object Class

- Its special class defined by Java.
- All classes are subclasses of **Object**
- Reference of type **Object** can refer to object of subclasses
- Few functions provided by Object class which can be overridden
- String toString( ):- Returns a string that describes the object
- boolean equals(Object object):- Determines whether one object is equal to another.