# innerclasses

Sometimes we can declare a class inside another class, such type of class is known as innerclass.

class{

class{

….

}

}

Inner class concept introduced in 1.1V to resolve of GUI(awt) bugs, As a part of event handling.

But innerclass concept had powerful features, and benefits. So programmers started use this in regular programming.

Inheritance: Parent contains common methods.

Child contains specific methods, for code reusability.

InnerClass: Without existing one type of object, there is no chance of existing another type of object. Then we should go for inner classes.

class University { // outer class

class Department { // inner class

….

}

}

University consists of several departments. Without existing university, there is no chance of existing Department. hence we have to declare Department class inside university class.

Like that without car\_obj, there is no engine\_obj

class Car {

class Engine {

….

}

}

Without map\_obj, there is no entry\_obj.

|  |  |
| --- | --- |
| key | value |
| 101 | tharsi |
| 102 | New va |
| 104 | lucky |

entry-obj map-obj

interface Map {

interface Entry {

….

}

}

Without exacting outerclass obj, there is no chance of excising innerclass obj.

Inner class is a “Has-A” relationship.

Has-A: (composition/aggregation)

## Type of innerclasses

## Normal/ regular innerclass

class Outer {

class Inner {

….

}

}

Javac Outer.java 🡺 **Outer.class**, and **Outer$Inner.class** will be generated

.jar is collection of .class file

If we are declaring name class (no anonymous inner) directly inside a class (no method inner) without static modifier (no static nested) is called regular innerclass.

Case 1

class Outer {

class Inner {

}

}

javac Outer.java 🡺 Outer.class, Inner.class

java Outer 🡺 RE: no Main method

java Outer$Inner 🡺 RE: no Main method

Case 2

class Outer {

public static void main(String ard[]){

}

class Inner {

}

}

java Outer 🡺 run fine

java Outer$Inner 🡺 RE: no Main method

Case 3

innerclass always talks about an object that is can’t live without outerclass\_obj.

class Outer {

class Inner {

public static void main(String ard[]){

}

}

}

javac Outer.java 🡺 CE: In inner class modifier 'static' is only allowed in constant variable declarations

Inside innerclass we can’t declare any static members; hence we can’t declare main method.

We can’t run inner class directly form CMD.

Case 4

class Outer {

class Inner {

public void m1(){

}

}

public static void main(String ard[]){

Outer o = new Outer();

Outer.Inner i = o. new Inner();

i.m1();

// new Outer(). new Inner(). m1();

}

Accessing Innerclass code, from Static area of Outer class (main method).

Need to create Outer Obj, from that create Inner Obj, then call method.

Case 5

class Outer {

class Inner {

public void m1(){

}

}

public void m2(){

Inner i = new Inner();

i.m1();

}

public static void main(String ard[]){

Outer o = new Outer();

o.m2();

}

Accessing inner class code from instance area of outer class.

It’s easy. By creating inner Obj then call method.

Case 6

class Outer {

class Inner {

public void m1(){

}

}

}

Class Test{

public static void main(String ard[]){ or public void m3(){

Outer o = new Outer();

Outer.Inner i = o. new Inner();

i.m1();

// new Outer(). new Inner(). m1();

}

}

Accessing Inner class code

From **Outer class** Static area (main()) From Instance Area of **Outer class**

Or Inner i = new Inner();

From Outside of **Outer class** i.m1();

Outer o = new Outer();

Outer.Inner i = o. new Inner();

i.m1();

Case 7

class Outer {

int x = 10;

static int y = 20;

class Inner {

sop(x);

sop(y);

}

}

From inner class, we can access Outer class’s static or non-static members directly.

this keyword

class Test {

int x;

int y;

public void setData(int x, int y){

this.x = x; // runtime object-handler change this 🡺 obj this.y = y; // so this.y 🡺 obj.y (this refers current object)

}

}

Main(){

Test obj = new Test();

obj.setData(4,3);

}

Class 8

class Outer {

int x = 10;

class Inner {

int x = 100;

public void m2(){

int x = 1000;

sop(x); // 1000

sop(this.x); // 100

sop(Outer.x); // 10

}

}

}

Within Innerclass

this. 🡺 Refers current innerclass obj

Outer.this. 🡺 Refers current outer class

### Modifiers

public

<default>

final

abstract

strictfp

class Outer {

class Inner {

public

<default>

final

abstract

strictfp

+

private

protected

static

}

}

### Nesting of innerclasses

Nesting of inner-classes is possible.

class A {

class B {

class C {

class D {

Sop(“inner most class”);

}

}

}

}

class test{

main(){

A a = new A();

A.B b = a. new B();

A.B.C c = b. new C();

c.m1(); // new A(). new B(). new C(). m1();

}

}

## Method local innerclass

Sometimes we can declare classes inside a method such type of innerclass are called method local innerclass.

The main purpose of Method-local-innerclass is to **define method specific repeatedly required functionality**.

Method-local-innerclass is best suitable to meet nested-method (not in java) requirements.

class Outer {

class Inner {

public void m1(){ we can **declare this method** outside

...... but this code not need outside entire class.

Public void m2(){

For(i=0 to 100){ this code(functionality) need

Sop(“100 hello”); **repeatedly** with in this m1()

} method. So we write code

} within method m2()…

But nested method is not possible java.

class Inner(){

public void m2(){

for(i=0 to 100){

Sop(“100hello”);

}

}

Inner i = new Inner(); **Only able within this class**

i.m2(); Now we can call n times that code..

}

}

We can access this method-local-innerclasses only within a method. Outside a method we can’t access (method level).

Because of its scope method-local-innerclass is most rarely used innerclass.

class Outer {

public void myBig() {

class Inner {

public int sum(int n1, int n2){

return n1+n2;

}

}

Inner i = new Inner();

Sop(i.sum(233,44));

//some code

Sop(i.sum(2,45));

Sop(i.sum(299,75));

}

}

class test{

main(){

Outer o = new Outer();

o.myBig();

}

}

javac Outer.java 🡺 Outer.class, Outer$1Inner.class

**myBig()** method require **sum(num1, num2)** functionality repeatedly.

We can declare method-local-innerclass within instance/ static

methods.

|  |  |
| --- | --- |
| **Instance method’s**  **local-innerclass** | **static method’s**  **local-innerclass** |
| class Outer {  int x =10;  Static int y =20;  **public void m1() {**  class Inner {  p v m2(){  Sop(x);  Sop(y);  }  }  }  }  Run… we can access all instance or static members here | class Outer {  int x =10;  Static int y =20;  **public static void m1() {**  class Inner {  p v m2(){  Static Sop(x);  Sop(y);  }  }  }  }  CE: non-static variable x cannot be referenced from a static context. (class level) |

class Outer {

int x =10; // instance variable

Static int y =20; // static variable

public void m1() {

int a = 10; // local variable

**final** int b = 20;

class Inner {

p v m2(){

Sop(a); // cant access local variable of method

Sop(b);

}

}

}

}

From method-local-innerclass we can’t access local variable of the methods.

If the local variable declared as final then we can access.

Heap

Stack

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  |  |

Local variable stored Objects stored

Final variable value will be replaced CompileTime only.

Whenever calling methods, It’s local variable will be created. Once method completes automatically local variable will be destroyed.

At above case within method-local-innerclass we can access x,y,b variables.

Final variable b will be repaced by value at compile time.

public void m1() {

class Inner {

p v m2(){

Sop(20); // variable will be replaced by value at compilation

}

}

}

So after stack (local variable) gone, we can call i.m2()

class Outer {

public void m1() {

int a = 10;



class Inner {

a=10

p v m2(){ stack

Sop(a);

}

} .m2()

Inner i = new Inner();

i.m2(); heap

}

}

class Test {

main(){

Outer o = new Outer();

o.m1();

}

}

In above case after m2() completion: variable a =10 will be removed form stack area.

i.m2(); can be called any time… but a is not in the stack.

That’s why local variable can’t be referenced from method-local innerclasses.

## Anonymous innerclass

We can declare Innerclass without name; such type innerclasses are called anonymous innerclass

There is no use of knowing this person’s name. We are not going to meet this person again. Just for instance use.

The main purpose of anonymous innerclass is just for instance use. (**One time usage**)

### Three type of innerclasses

* Extends a class
* Implements an interface
* Defined inside arguments m1(…)

#### Anonymous innerclass that extends a class

|  |  |
| --- | --- |
| public class PopCorn {  // + 100 methods  public void taste(){  Sop("Salt");  }  }  Parent Reference  Child Object  (no name) | public class Test {  main(String[] args) {  PopCorn p1 = new PopCorn();  PopCorn p2 = new PopCorn(){  public void taste(){  Sop("sugar");  }  };  p1.taste();  p2.taste();  Sop(p1.getClass().getName());  Sop(p2.getClass().getName());  }  } |

javac Test.java 🡺 PopCorn.class, Test.class, Test$1.class

output 🡺 salt, suger, PopCorn, Test$1

1. Popcorn p = new Popcorn();

Just creating object for Popcorn class

1. Popcorn p = new Popcorn(){

…….

}

We are declaring a class that extends Popcorn without name

(Anonymous innerclass)

For that child class we are creating object with parent reference

1. Popcorn p = new Popcorn(){

public void taste(){

Sop(“sugar”);

}

}

We are declaring a class that extends Popcorn without name

In that child class Override taste() method

For that child class we are creating object with parent reference

Defining a thread by extending Thread class

|  |  |
| --- | --- |
| class MyThread extends Thread {  public void run(){  for(i=0 to 10)  Sop(“childT”);  }  }  class Test{  main(String[] args) {  MyThread t = new MyThread ();  t.start();  }  } | class Test {  main(String[] args) {  Thread t = new Thread(){  public void run(){  for(i=0 to 10)  Sop(“childT”);  }  };  t.start();  }  } |

At both examples defining class extends Thread, override run() method. Then creating child class object. From that thread-object calling start()

We use 2nd approach when one time use.

#### Anonymous innerclass that Implements an interface

## Static nested class