Inner classes should be used in following conditions

1. A class can be public however; its constructor needs to be private.
2. A class needs to perform complex logic. The inner class can help to reduce the complexity.
3. It is required only within the boundary of outer class.

Inner classes can help in building objects using builder pattern. An outer class has an inner class that helps to build the outer class.

However, if the outer classes have hierarchy then the inner classes can also have the hierarchy. In other words, if the outer classes have extends relationship then their respective inner classes can also have extends relationship.

Example: -

**public** **class** Employee {

**private** String name;

**private** String dept;

**private** String designation;

**private** **float** salary;

**private** **int** peopleLeadingNum;

//Getters are removed for brevity.

**protected** String print() {

**return** "{name="+name+"}{dept="+dept+"}{designation="+designation+"}{salary="+salary+"}{peopleLeadingNum="+peopleLeadingNum+"}";

}

@Override

**public** String toString() {

**return** "[" + **this**.print() + "]";

}

**public** **static** EmployeeBuilder getEmployeeBuilder() {

**return** **new** EmployeeBuilder();

}

**public** **static** **class** EmployeeBuilder {

**protected** Employee obj;

**protected** EmployeeBuilder() {

**this**.obj = **new** Employee();

}

//Setters are removed for brevity.

**public** Employee build() {

**return** obj;

}

}

}

**public** **class** Manager **extends** Employee{

**private** **int** peopleReportingNum;

//Getters removed for brevity.

**protected** String print() {

**return** "{name="+name+"}{dept="+dept+"}{designation="+designation+"}{salary="+salary+"}{peopleLeadingNum="+peopleLeadingNum+"}";

}

@Override

**public** String toString() {

String str = **super**.print();

**return** "[" + str + "{peopleReportingNum="+peopleReportingNum+"}]";

}

**public** **static** ManagerBuilder getManagerBuilder() {

**return** **new** ManagerBuilder();

}

**public** **static** **class** ManagerBuilder **extends** EmployeeBuilder {

**private** Manager obj;

**private** ManagerBuilder() {

**this**.obj = **new** Manager();

**super**.obj = **this**.obj;

}

//Setters removed for brevity.

**public** Manager build() {

**return** obj;

}

}

}

**public** **class** TestBuilder {

**public** **static** **void** main(String[] args) {

System.***out***.println("\*\*\* Building Developer");

EmployeeBuilder eBuilder = Employee.*getEmployeeBuilder*();

eBuilder.name("Emp 1");

eBuilder.dept("Dept 1");

eBuilder.designation("Developer");

eBuilder.salary(1111.11f);

eBuilder.peopleLeadingNum(0);

Employee emp = eBuilder.build();

System.***out***.println(emp.toString());

System.***out***.println("\*\*\* Building Technical Leader - Leads Three Employees");

EmployeeBuilder leadBuilder = Employee.*getEmployeeBuilder*();

leadBuilder.name("Emp 1");

leadBuilder.dept("Dept 1");

leadBuilder.designation("Technical Leader");

leadBuilder.salary(11111.11f);

leadBuilder.peopleLeadingNum(3);

Employee leadEmp = leadBuilder.build();

System.***out***.println(leadEmp.toString());

System.***out***.println("\*\*\* Building Manager - Reports 10 People");

ManagerBuilder mBuilder = Manager.*getManagerBuilder*();

mBuilder.name("Manager Emp 2");

mBuilder.dept("Dept 1");

mBuilder.designation("Manager");

mBuilder.salary(222222.22f);

mBuilder.peopleLeadingNum(0);

mBuilder.peopleReportingNum(10);

Manager manager = mBuilder.build();

System.***out***.println(manager.toString());

System.***out***.println("\*\*\* Building Senior Manager - Leads Two Managers and Reports by 2 People");

ManagerBuilder smBuilder = Manager.*getManagerBuilder*();

smBuilder.name("Manager Emp 3");

smBuilder.dept("Dept 1");

smBuilder.designation("Senior Manager");

smBuilder.salary(333333.33f);

smBuilder.peopleLeadingNum(2);

smBuilder.peopleReportingNum(10);

Manager srManager = smBuilder.build();

System.***out***.println(srManager.toString());

}

}

\*\*\* Building Developer

[{name=Emp 1}{dept=Dept 1}{designation=Developer}{salary=1111.11}{peopleLeadingNum=0}]

\*\*\* Building Technical Leader - Leads Three Employees

[{name=Emp 1}{dept=Dept 1}{designation=Technical Leader}{salary=11111.11}{peopleLeadingNum=3}]

\*\*\* Building Manager - Reports 10 People

[{name=Manager Emp 2}{dept=Dept 1}{designation=Manager}{salary=222222.22}{peopleLeadingNum=0}{peopleReportingNum=10}]

\*\*\* Building Senior Manager - Leads Two Managers and Reports by 2 People

[{name=Manager Emp 3}{dept=Dept 1}{designation=Senior Manager}{salary=333333.34}{peopleLeadingNum=2}{peopleReportingNum=10}]

Inner class can help to share complex operations with outer class. For example, I had to process huge amount of data that has multiple parent child relationships. Using java.util.Map would have introduced lot of iterations. Hence, I created an Inner Class that had parent child relationship and was able to process data with minimum iterations. This improved performance by marginal percentage.

Example: -

* Below is a very small part from a program. The entire example is not provided.
* Data class is required within the boundary of ProcessData class.

**public** **class** ProcessData {

**public** **static** **void** main(String[] args) {

}

**static** **class** Data {

Data subData = **null**;

Data parentData = **null**;

Data(/\*Constructor Arguments\*/) {

//Constructor Code

}

Data createSubData(/\*Method Arguments\*/) {

Data data = new Data(/\*Constructor Arguments\*/);

data.parentData = this;

this.subData = data;

return data;

}

}

}

Example: - Say, in an application, classes in a particular layer follow specific method definition. (This scenario may occur in auto generated code). These methods are parsed by a class and executed with command design pattern.

Note: - Below program is not a complete example.

**public** **class** Class1 {

**public** **static** **void** method(String str, Integer integer, Bean1 object) **throws** Exception{

}

}

**public** **class** Class2 {

**public** **static** **void** method(String str, Integer integer, Bean2 object) **throws** Exception{

}

}

**import** java.lang.reflect.InvocationTargetException;

**import** java.lang.reflect.Method;

**import** java.util.ArrayList;

**import** java.util.List;

**public** **class** CommandPattern {

**private** List<MethodHolder> methodHolders = **new** ArrayList<CommandPattern.MethodHolder>();

**public** **static** **void** parseClassMethods(Class<?> clazz) {

}

**public** **static** **class** MethodHolder {

**private** Method method;

**public** **void** execute(String str, Integer integer, Object obj, Object classObject) **throws** IllegalAccessException, IllegalArgumentException, InvocationTargetException {

Object[] objArray = **new** Object[3];

objArray[0] = str;

objArray[1] = integer;

objArray[2] = obj;

method.invoke(methodHolder, objArray);

}

}

}

CommandPattern class parse Class1, Class2, … ClassN classes and create MethodHolder inner class objects those hold method objects. The MethodHolder object execute method on classObject.