ENCAPSULTAION

Encapsulation in Java is a mechanism of wrapping the data (variables) and code acting on the data (methods) together as a single unit.

(Or)

Encapsulation is a process of providing security to most important component of a object.(i.e. data)

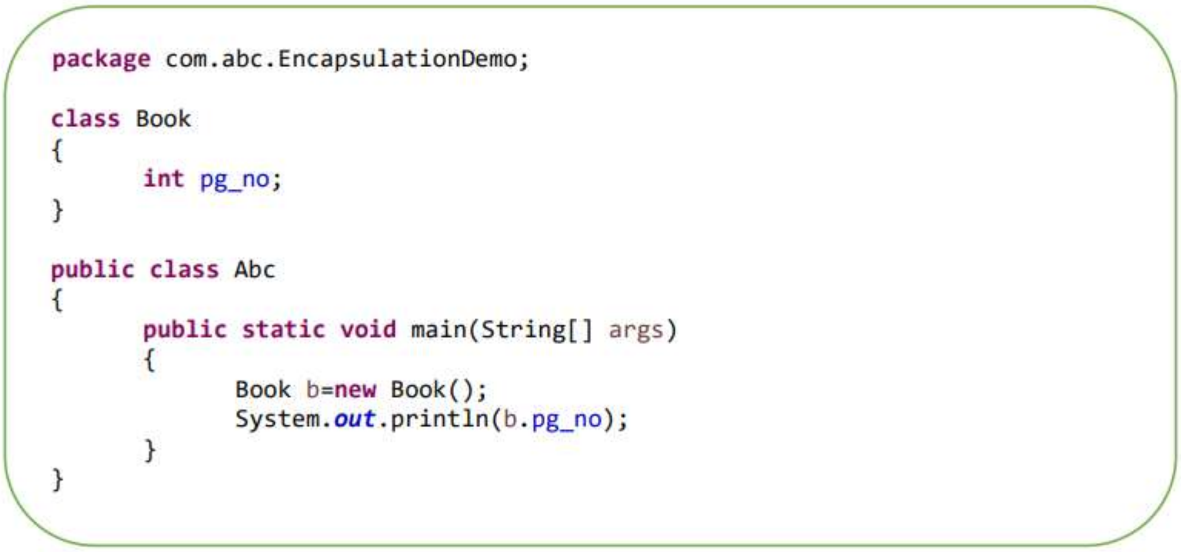
The most important component of a object is the data members(variables).

Access to data members can be prevented using private access modifier.

Controlled access to data member can be given using Accessors(getters) and mutators (setters).

Note: encapsulation does not mean preventing access. Encapsulation means providing controlled access.

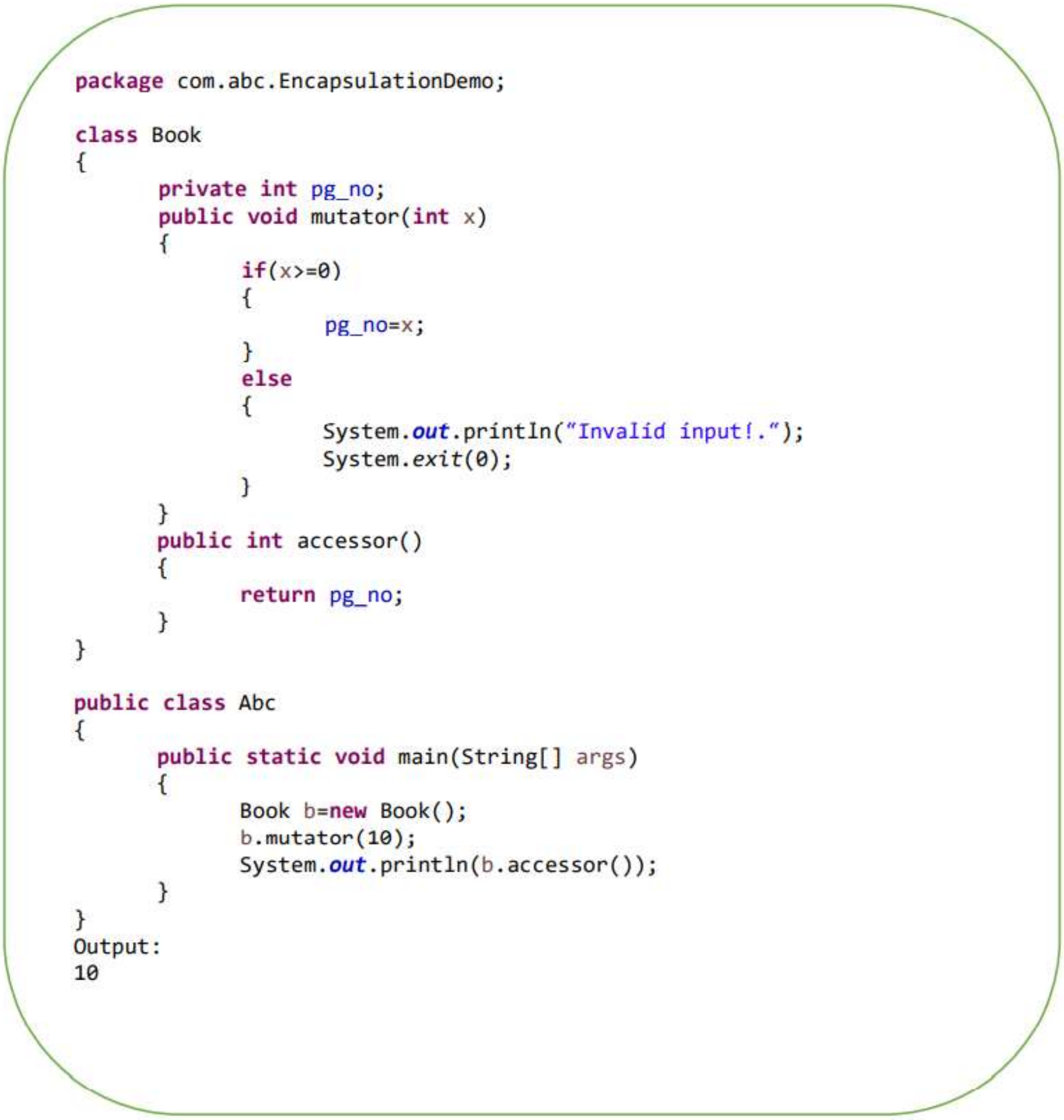
Program 1:



```
package com.abc.EncapsulationDemo;

class Book
{
        int pg_no;
}

public class Abc
{
        public static void main(String[] args)
        {
                Book b=new Book();
                System.out.println(b.pg_no);
        }
}
```

In the above program there is no security provided to the important component i.e., pg_no so external code can access and modify the value of pg_no.

Encapsulation means providing controlled access using accessor and mutators as shown below.



```
package com.abc.EncapsulationDemo;

class Book
{
	private int pg_no;
	public void mutator(int x)
	{
		if(x>=0)
		{
			pg_no=x;
		}
		else
		{
			System.out.println("Invalid input!.");
			System.exit(0);
		}
	}
	public int accessor()
	{
		return pg_no;
	}
}

public class Abc
{
	public static void main(String[] args)
	{
		Book b=new Book();
		b.mutator(10);
		System.out.println(b.accessor());
	}
}
Output:
10
```

In the program controlled access has been provided using accessor and mutators.

Program-2

```
package com.abc.EncapsulationDemo;

class Dog
{
	private String name;
	private String  breed;
	private int cost;
	public String getName() {
		return name;
	}
	public void setName(String x) {
		name = x;
	}
	public String getBreed() {
		return breed;
	}
	public void setBreed(String y) {
		breed = y;
	}
	public int getCost() {
		return cost;
	}
	public void setCost(int z) {
		cost = z;
	}
}
public class Launch
{
	public static void main(String[] args)
	{
		Dog d = new Dog();
		d.setName("Lobo");
		d.setBreed("pug");
		d.setCost(15000);
		System.out.println(d.getName());
		System.out.println(d.getBreed());
		System.out.println(d.getCost());
	}
}
Output:
Lobo
Pug
15000
```

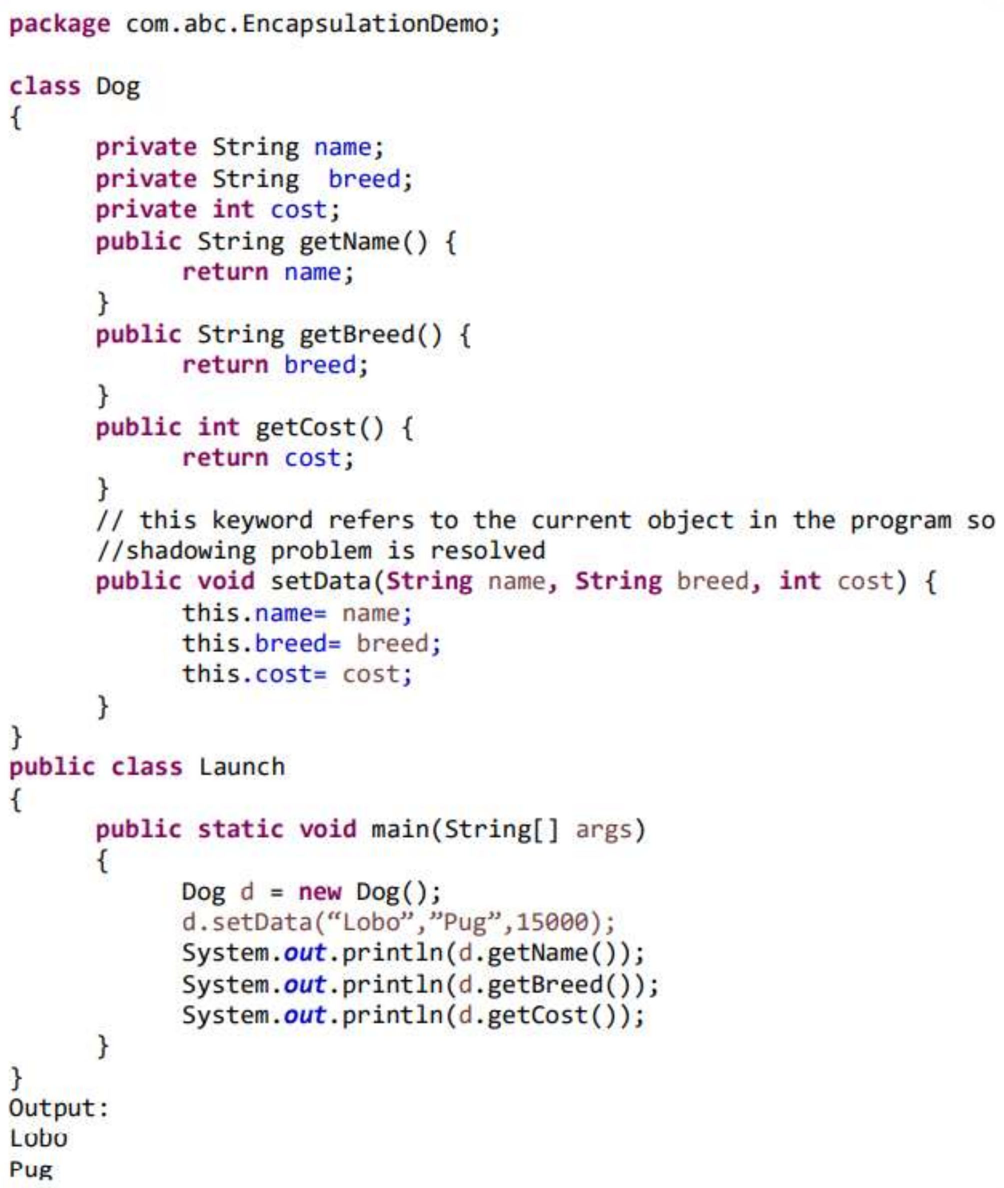
Accessor can return a single value at any given time but mutator can set multiple values at a time as shown below.

```
package com.abc.EncapsulationDemo;

class Dog
{
	private String name;
	private String  breed;
	private int cost;
	public String getName() {
		return name;
	}
	public String getBreed() {
		return breed;
	}
	public int getCost() {
		return cost;
	}
	public void setData(String x, String y, int z) {
		name= x;
		breed= y;
		cost= z;
	}
}
public class Launch
{
	public static void main(String[] args)
	{
		Dog d = new Dog();
		d.setData(“Lobo”,”Pug”,15000);
		System.out.println(d.getName());
		System.out.println(d.getBreed());
		System.out.println(d.getCost());
	}
}
Output:
Lobo
Pug
15000
```

Note: It is a convention in java that within a setter the local variable names should be same as that of the instance variables name.

Because of this convention a name clash occurs between the local variables and the instance variables. This name clash is referred as **shadowing problem**.

To overcome this problem in java we will use "**this**" keyword. this keyword would discriminate between local and instance variable because **this keyword refers to the current object in program.**



```
package com.abc.EncapsulationDemo;

class Dog
{
	private String name;
	private String  breed;
	private int cost;
	public String getName() {
		return name;
	}
	public String getBreed() {
		return breed;
	}
	public int getCost() {
		return cost;
	}
	// this keyword refers to the current object in the program so
	//shadowing problem is resolved
	public void setData(String name, String breed, int cost) {
		this.name= name;
		this.breed= breed;
		this.cost= cost;
	}
}
public class Launch
{
	public static void main(String[] args)
	{
		Dog d = new Dog();
		d.setData("Lobo","Pug",15000);
		System.out.println(d.getName());
		System.out.println(d.getBreed());
		System.out.println(d.getCost());
	}
}
Output:
Lobo
Pug
```

Constructor

A constructor is a specialized setter which has same name as class name. A constructor does not have a return type and it is called during object creation.

```
package com.abc.EncapsulationDemo;

class Dog
{
	private String name;
	private String  breed;
	private int cost;
	public String getName() {
		return name;
	}
	public String getBreed() {
		return breed;
	}
	public int getCost() {
		return cost;
	}
	public Dog(String name, String breed, int cost) {
		this.name= name;
		this.breed= breed;
		this.cost= cost;
	}
}
public class Launch
{
	public static void main(String[] args)
	{
		Dog d = new Dog("Lobo","Pug",15000);
		System.out.println(d.getName());
		System.out.println(d.getBreed());
		System.out.println(d.getCost());
	}
}
Output:
Lobo
Pug
15000
```

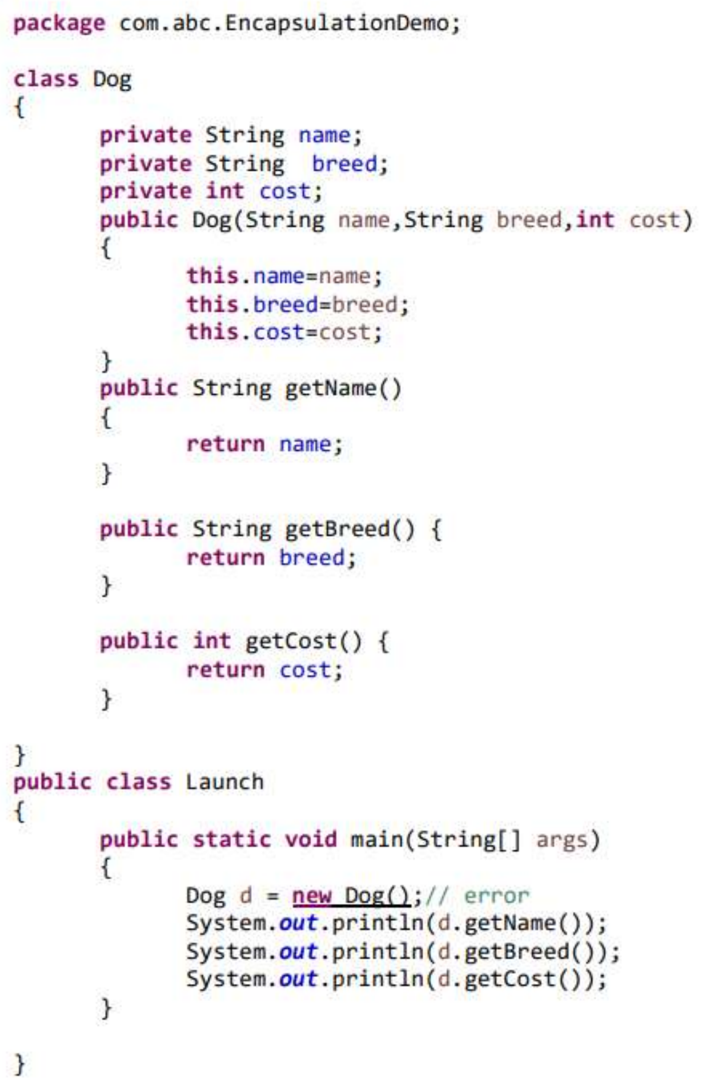
Note:

1. within a class if the programmer does not provide a single constructor then the java compiler would automatically insert a zero parameterized constructor call as "**<u>the default constructor</u>**".

2. The first line within a constructor is "super() method call".

```java
package com.abc.EncapsulationDemo;

class Dog
{
        private String name;
        private String  breed;
        private int cost;
        public String getName() {
                return name;
        }

        public String getBreed() {
                return breed;
        }

        public int getCost() {
                return cost;
        }

}
public class Launch
{
        public static void main(String[] args)
        {
                Dog d = new Dog();// default constructor is executed during
object creation
                System.out.println(d.getName());
                System.out.println(d.getBreed());
                System.out.println(d.getCost());
        }

}
```

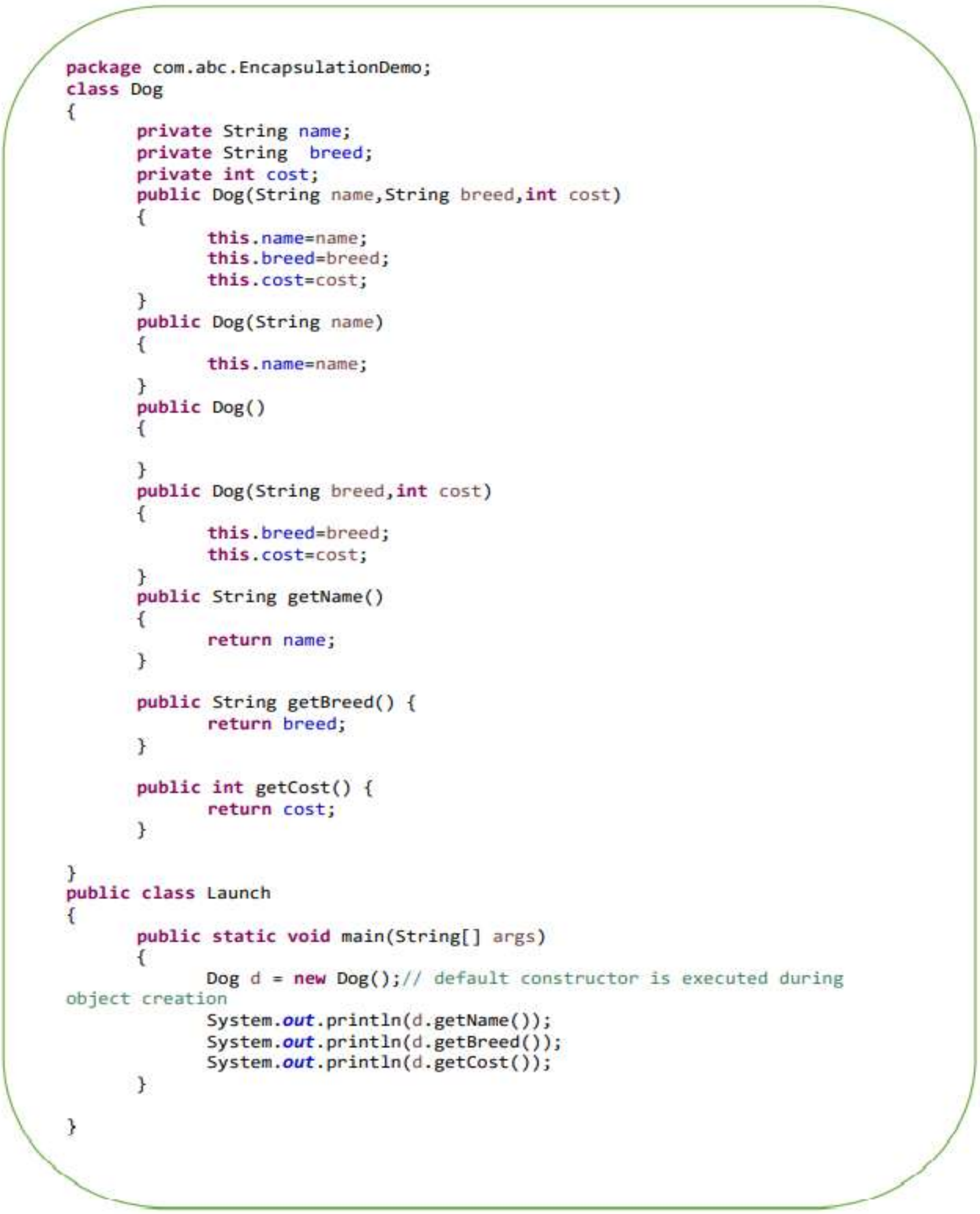
If the programmer has inserted the constructor then the java compiler will not insert a constructor



```java
package com.abc.EncapsulationDemo;

class Dog
{
	private String name;
	private String  breed;
	private int cost;
	public Dog(String name,String breed,int cost)
	{
		this.name=name;
		this.breed=breed;
		this.cost=cost;
	}
	public String getName()
	{
		return name;
	}

	public String getBreed() {
		return breed;
	}

	public int getCost() {
		return cost;
	}

}
public class Launch
{
	public static void main(String[] args)
	{
		Dog d = new Dog();// error
		System.out.println(d.getName());
		System.out.println(d.getBreed());
		System.out.println(d.getCost());
	}

}
```

The above program will give a compile time error because constructor is inserted in class Dog by programmer which is excepting parameters, but we are calling zero parameterized constructor

The above program error can be overcome using constructor overloading.

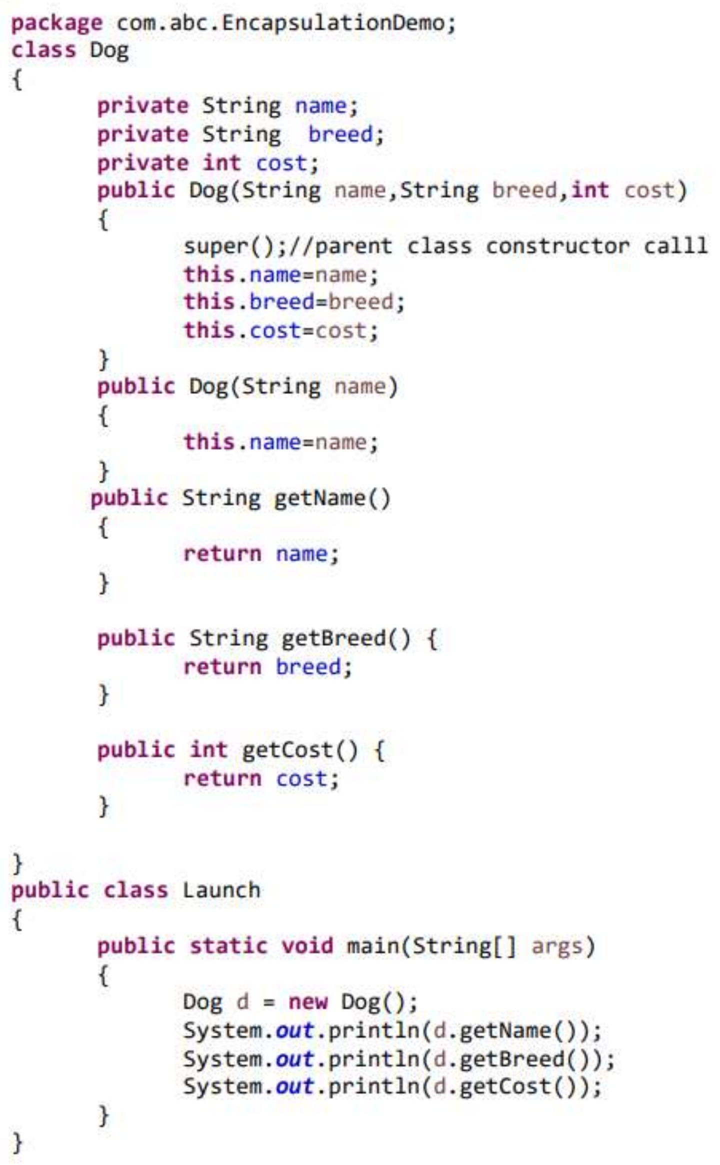


```
package com.abc.EncapsulationDemo;
class Dog
{
	private String name;
	private String  breed;
	private int cost;
	public Dog(String name,String breed,int cost)
	{
		this.name=name;
		this.breed=breed;
		this.cost=cost;
	}
	public Dog(String name)
	{
		this.name=name;
	}
	public Dog()
	{

	}
	public Dog(String breed,int cost)
	{
		this.breed=breed;
		this.cost=cost;
	}
	public String getName()
	{
		return name;
	}

	public String getBreed() {
		return breed;
	}

	public int getCost() {
		return cost;
	}

}
public class Launch
{
	public static void main(String[] args)
	{
		Dog d = new Dog();// default constructor is executed during
object creation
		System.out.println(d.getName());
		System.out.println(d.getBreed());
		System.out.println(d.getCost());
	}

}
```

Note: constructor overloading is a process having multiple constructors in the same class

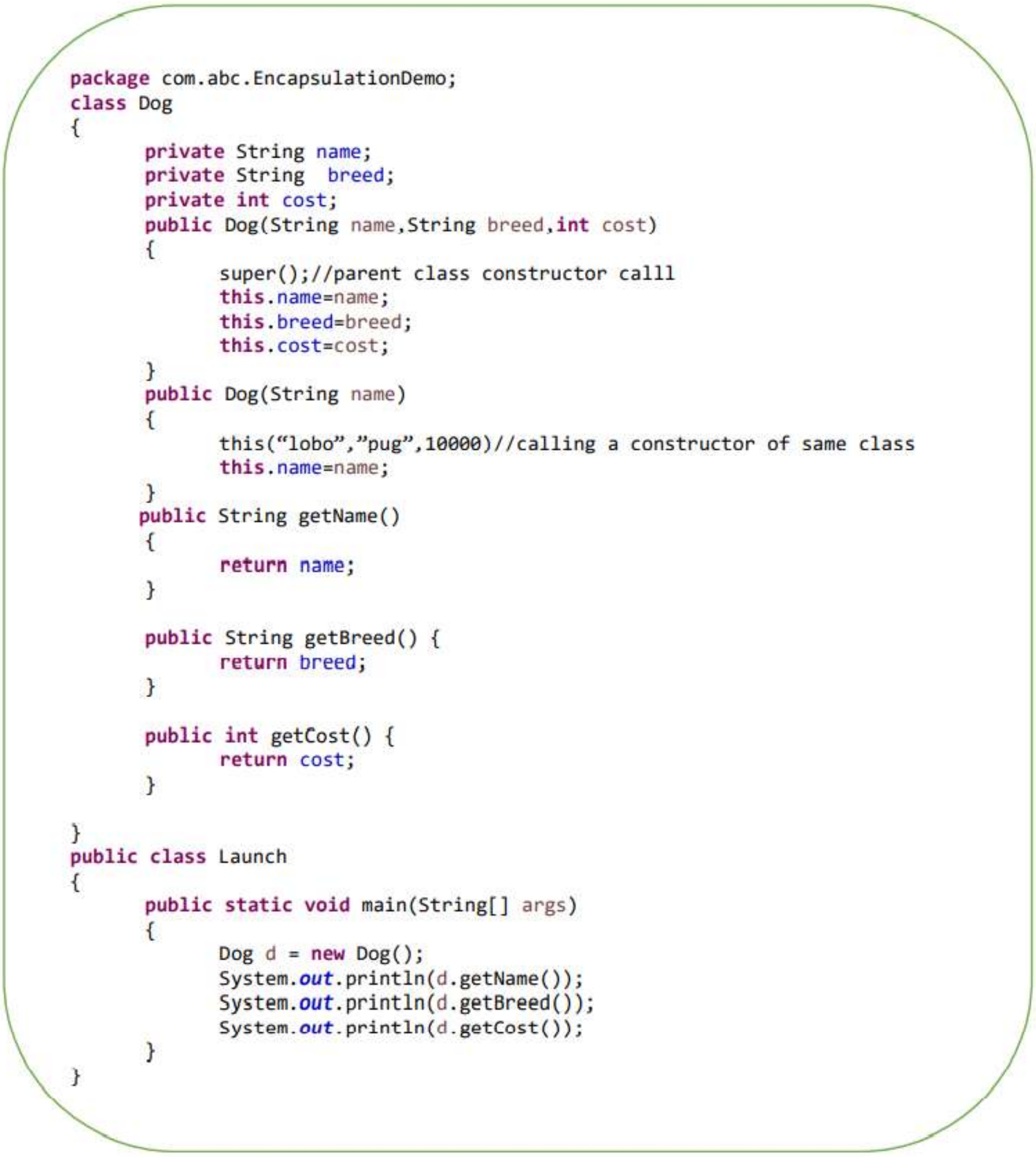
Constructor chaining and local chaining

Constructor chaining is a process of calling a constructor of parent class using super()



```java
package com.abc.EncapsulationDemo;
class Dog
{
        private String name;
        private String  breed;
        private int cost;
        public Dog(String name,String breed,int cost)
        {
                super();//parent class constructor calll
                this.name=name;
                this.breed=breed;
                this.cost=cost;
        }
        public Dog(String name)
        {
                this.name=name;
        }
       public String getName()
        {
                return name;
        }

        public String getBreed() {
                return breed;
        }

        public int getCost() {
                return cost;
        }

}
public class Launch
{
        public static void main(String[] args)
        {
                Dog d = new Dog();
                System.out.println(d.getName());
                System.out.println(d.getBreed());
                System.out.println(d.getCost());
        }
}
```

Local chaining is a process of calling constructor within in a constructor of same class using this().



```
package com.abc.EncapsulationDemo;
class Dog
{
	private String name;
	private String  breed;
	private int cost;
	public Dog(String name,String breed,int cost)
	{
		super();//parent class constructor calll
		this.name=name;
		this.breed=breed;
		this.cost=cost;
	}
	public Dog(String name)
	{
		this(“lobo”,”pug”,10000)//calling a constructor of same class
		this.name=name;
	}
       public String getName()
	{
		return name;
	}

	public String getBreed() {
		return breed;
	}

	public int getCost() {
		return cost;
	}

}
public class Launch
{
	public static void main(String[] args)
	{
		Dog d = new Dog();
		System.out.println(d.getName());
		System.out.println(d.getBreed());
		System.out.println(d.getCost());
	}
}
```