30/07/23

Access Control, In-built Packages, Object Class {Encapsulation}

Access Control

How a member can be accessed is determined by the access modifier attached to its declaration. Usually, you will want to restrict access to the data members of a class - allowing access only through methods. Also, there will be times when you will want to define methods that are private to a class.

Java's access modifiers are public, private and protected. Java also defines a default access level. protected applies only when inheritance is involved.

When no access modifier is used, then by default the member of a class is public within its own package, but cannot be accessed outside of its package.

	Class	Package	Subclass (same pkg)	Subclass (diff pkg)	World (diff pkg & not subclass)
public	+	+	+	+	+
protected	+	+	+	+	
no modifier	+	+	+		
private	+				

+ : accessible

blank : not accessible

```
package packageOne;
public class Base
{
   protected void display(){
        Sout ("in Base");
   }
}
package packageTwo;
public class Derived extends packageOne.Base {
     public void show(){
            new Base().display(); //this is not working
            new Derived().display(); //is working
            display(); //is working
     }
}
```

protected allows access from subclasses and from other classes in the same packag.
We can use child class to use protected member outside the package but only child class object can access it. That's why any Derived class instance can access the protected method in Base.
The other line creates a Base instance (not a Derived instance!) And access to protected methods of that instance is only allowed from object of the same package.

display();
→allowed, because the caller, an instance of Derived has access to protected members and fields of its subclases, even if they're in different packages.

new Derived().display();
→allowed, because you call the method on an instance

of Derived and that instance has access to the protected methods of its subclasses.

new Base(). display();
→ not allowed because the caller's (the this instance) class is not defined in the same package like the Base class, so this can't access the protected method. And it doesn't matter as we see that the current subclasses a class from that package. That backdoor is closed.

Remember that any time talks about a subclass having access to a superclass member, we could be talking about the subclass inheriting the member, not simple accessing the member through a reference to an instance of the superclass.

```
⇒ class C
      protected member;

//in a different package
class S extends C
      obj.member; //only allowed if type of obj is S or
                        subclass of S.
```

The motivation is probably as following. If obj is an S, class S has sufficient knowledge of its internals, If obj is not an S, it's probably another subclass S2 of C, which S has No idea of. S2 may have not even been born when S is written. For S to manipulate S2's protected internals is quite dangerous, If this is allowed, from S2's point of view, it doesn't know who will tamper with its protected internals and now, this makes S2 job very hard to reason about its own state.

Now if obj is D, and D extends S, is it dangerous for S to access obj.member? Not really.
Now S's member is a shared knowledge of S and all its subclasses, including D. S as the superclass has the right to define behaviour, and D as the subclass has the obligation to accept and confirm.

For easier understanding, the rule should really be simplified to require obj's (static) type to be exactly S. After all, it's very unusual and inappropriate for subclass D to appear in S. And even if it happens, that the static type of obj is D, our simplified rule can deal with it easily by upcasting: ((S)obj).member.

More about Packages

$$\begin{tikzcd}
 & & & \text{Packages} \arrow[dll] \arrow[d] & & \\
 & \text{user-defined} & & \text{in-built} \arrow[dlll] \arrow[dll] \arrow[dl] \arrow[d] \arrow[dr] \arrow[drr] & & \\
\text{lang} & \text{io} & \text{util} & \text{applet} & \text{awt} & \text{net}
\end{tikzcd}$$

Abstract Classes

Such a class that determines the nature of method that the subclass will implement.
i.e. when superclass only defines generalised from the child classes will follow, but subclasses fill in details on their own i.e. make body of their own such class are abstract classes

- Methods used by child class to make changes/ overrides parent class is ~~cla~~ abstract method.

Syntax:

· abstract data_type name/funcⁿ (String name);

Any class that contains 1 or more abstract methods must also be declared abstract.

We cannot create abstract constructor and static abstract method and final abstract class (cause it ~~could~~ won't be able to inherit that)
Cause u won't be able to override

⇒ We cannot do multiple inheritance in abstract.

⇒ Interface
They contain abstract class where funcⁿ are public and abstract and variable are static and final.

Allow multiple inheritance.