Chapter 6. Overloaded and Overridden Methods

1) Overridden Methods :

The rules for overriding a method are as follows:

- The *argument list must exactly match* that of the overridden method.
- The *return type must exactly match* that of the overridden method.
- The *access level must not be more restrictive* than that of the overridden method.
- The *access level can be less restrictive* than that of the overridden method.
- The overriding method *must not throw new or broader checked exceptions* than those declared by the overridden method. For example, a method that declares a FileNotFoundException cannot be overridden by a method that declares a SQLException, Exception, or any other non-runtime exception unless it's a subclass of FileNotFoundException.
- The overriding method *can throw narrower or fewer exceptions*. Just because an overridden method "takes risks" doesn't mean that the overriding subclass' exception takes the same risks. Bottom line: An overriding method doesn't have to declare any exceptions that it will never throw, regardless of what the overridden method declares.
- You *cannot override a method marked* final.
- *If a method can't be inherited, you cannot override it*.

2) Example for overriding methods :

Let's take a look at overriding the eat() method of Animal:

```
public class Animal {
public void eat() { }
}
```

Below table shows the illegal overrides of the method eat() in a subclass.

Illegal Override Code	Problem with the Code
`private void eat() { }`	Access modifier is more restrictive
`public void eat() throws IOException { }`	Declares a checked exception not declared by superclass version
`public void eat(String food) { }`	A legal overload, not an override, because the argument list changed
`public String eat() { }`	Not an override because of the return type, but not an overload either because there's no change in the argument list

3) Overloaded Methods :

Rules for overloading a methods are as below

- Overloaded methods *must* change the argument list.
- Overloaded methods *can* change the return type.
- Overloaded methods *can* change the access modifier.
- Overloaded methods *can* declare new or broader checked exceptions.
- A method *can* be overloaded in the same class or in a subclass.

4) Example for Overloading methods

Example 1:

```
public class Animal {
public void eat() {
System.out.println("Generic Animal Eating Generically");
}
}
public class Horse extends Animal {
public void eat() {
System.out.println("Horse eating hay ");
}
public void eat(String s) {
System.out.println("Horse eating " + s);
}
}
```

Notice that the Horse class has both overloaded *and* overridden the eat() method.

Table below shows which version of the three eat() methods will run depending on how they are invoked.

Method Invocation Code	Result
`Animal a = new Animal(); a.eat();`	`Generic Animal Eating Generically`
`Horse h = new Horse(); h.eat();`	`Horse eating hay`
`Animal ah = new Horse(); ah.eat();`	`Horse eating hay` Polymorphism works—the actual object type (Horse), not the reference type (Animal), is used to determine which `eat()` is called.
`Horse he = new Horse(); he.eat("Apples");`	`Horse eating Apples` The overloaded `eat(String s)` method is invoked.
`Animal a2 = new Animal(); a2.eat("treats");`	Compiler error! Compiler sees that Animal class doesn't have an `eat()` method that takes a String.
`Animal ah2 = new Horse(); ah2.eat("Carrots");`	Compiler error! Compiler *still* looks only at the reference type, and sees that Animal doesn't have an `eat()` method that takes a string. Compiler doesn't care that the actual object might be a Horse at runtime.

Example 2:

```
package OverloadingnArrays;

public class OverloadedCls {

	void meth(int i)
	{
		System.out.println("Integer value is "+i);
	}
	char [] meth(String str)
	{
		return str.toCharArray();
	}
	private void meth()
	{
		System.out.println("private meth");
	}
	void meth(int i,int j)
	{
		System.out.println("two args "+i+" & "+j);
	}

	public static void main(String[] args)
	{
		OverloadedCls oc = new OverloadedCls();
		oc.meth(10);
		System.out.println(oc.meth("java"));
		oc.meth(5, 15);
		oc.meth();
	}
}
Output:
Integer value is 10
java
two args 5 & 15
private meth
```

In the above example we can see how method arguments, return type, access control changes works with overloaded methods.

Example 2

```
class BaseClass
{
	int a=5,b=5;
	int sumInt()
	{
		int sum = 0;
		sum = a+b;
		return sum;
	}

	void printSum(BaseClass obj)
	{
		System.out.println("this is base class method "+obj.sumInt());
	}
}
```

```
class DerivedClass extends BaseClass
{
	int i=10,j=10;
	int sumInt()
	{
		int sum = 0;
		sum = i+j;
		return sum;
	}
}

public class PolymrphClass
{
	public static void main(String[] args)
	{
		// TODO Auto-generated method stub
		BaseClass bc = new BaseClass();
		DerivedClass dc = new DerivedClass();
		System.out.println(bc.sumInt());
		System.out.println(dc.sumInt());

		bc.printSum(dc);

		bc = dc;
		System.out.println(bc.sumInt());
	}
}
Output:
10
20
this is base class method 20
20
```

5) Difference between overloaded and overridden methods are as follows

	Overloaded Method	Overridden Method
argument list	Must change	Must not change
return type	Can change	Must not change
exceptions	Can change	Can reduce or eliminate. Must not throw new or broader checked exceptions
access	Can change	Must not make more restrictive (can be less restrictive)
invocation	*Reference* type determines which overloaded version (based on declared argument types) is selected. Happens at *compile* time. The actual *method* that's invoked is still a virtual method invocation that happens at runtime, but the compiler will already know the *signature* of the method to be invoked. So at runtime, the argument match will already have been nailed down, just not the actual *class* in which the method lives.	*Object* type (in other words, *the type of the actual instance on the heap*) determines which method is selected. Happens at *runtime*.

Overloaded	Can change	Can change		Must change		Can change
Method	{Access Specifier}	{Return type}	Method Name	({Argument List})	throws	{Exception}
Overridden	Must Not Make More Restrictive	Must Not Change		Must Not Change		Must Not Throw New And Broader Exceptions

Access Modifiers :

1) There are three access modifiers *public*, *private* and *protected* but there are four access control *default* being the fourth.

2) A class could have only 2 out of 4 access control. They are *public* and *default.*

Default access : Its a package level access and any class A in package x cannot be a super class of a class B in package Y. i.e. default modifier makes class invisible outside the package. When no modifier is used for a class then it considered as *default* modifier.

Ex: `class Beverage {}` has default access.

Public access : When a class is marked with *public* modifier then it is available to all the classes virtually in the java universe.

Ex: `public class Beverage {}`

3) Member Access :

- All the four access controls will be used by class members.
- It is always better to check class modifier before member modifiers for faster understanding.
- public : When a member variable or method is declared public then it means all other classes regardless of package have access to it(assuming the class is *visible* outside package).

  Ex :

```
package book;
import cert.*; // Import all classes in the cert package
class Goo {
public static void main(String [] args) {
Sludge o = new Sludge();
o.testIt();
}
}
```

Now look at the second file

```
package cert;
public class Sludge {
public void testIt() {
System.out.println("sludge");
}
}
```

Also when a member of a super class is declared *public* then subclass will inherits that member regardless of both classes are in the same package. Hence (.) operator i.e. reference is not needed to use those members. In above example testIt() method could be directly called.

Example

```
class InheritorMethCheck1
{
	void meth1()
	{
		System.out.println("Inside meth1");
	}
}

class InheritorMethCheck2 extends InheritorMethCheck1
{
	void meth2()
	{
		meth1();
		System.out.println("Inside meth2");
	}
}

public class InheritorMethCheck {

	public static void main(String[] args)
	{
		InheritorMethCheck2 imc = new InheritorMethCheck2();
		imc.meth2();
	}
}
Output:
Inside meth1
Inside meth2
```

- Private :

  Members marked *private* could be used by only the class in which they are defined, for other classes they are invisible.

  - Ex :

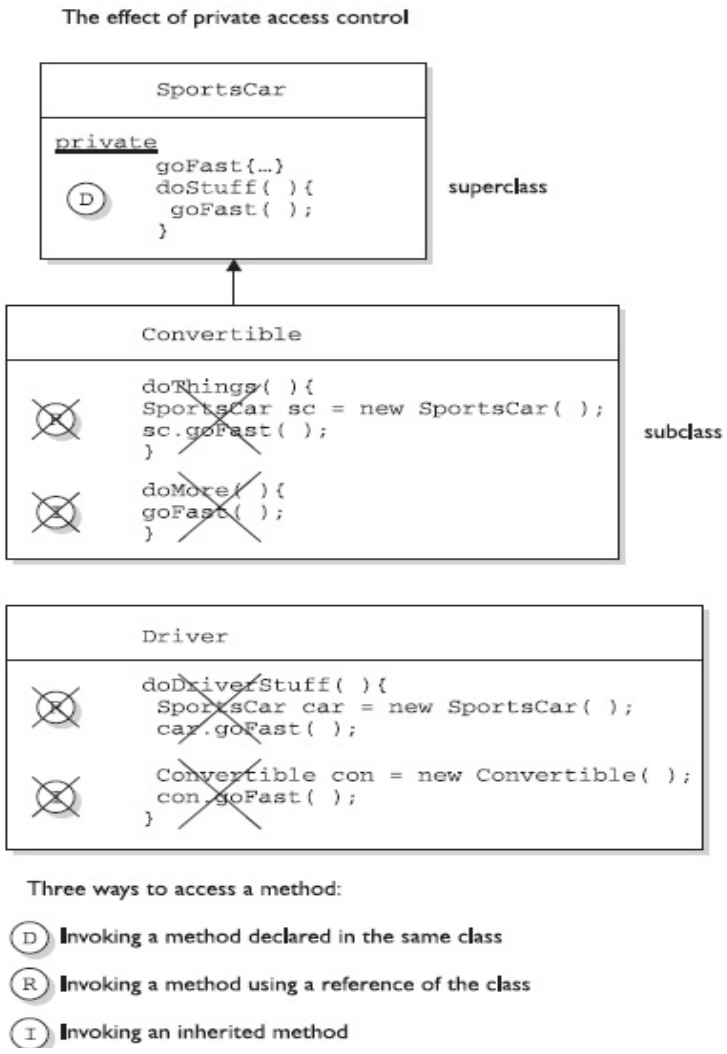
```
package chapter2;

class PrivateMemberClass1
{
  private int a=10, b=10;
  int c = 20;
  private int addint()
  {
		int sum = 0;
```

```
            sum = a+b;
            return sum;
      }
      int mulint()
      {
            int mul;
            mul=a*b;
            return mul;
      }
}
public class PrivateMemberClass {
      /*Private members cannot be used outside the same class */
      public static void main(String[] args) {
            PrivateMemberClass1 prcls = new PrivateMemberClass1();
            //System.out.println(prcls.addint());//dont compile
            //System.out.println(prcls.a); //dont compile
            System.out.println(prcls.c);
            System.out.println(prcls.mulint());
      }
}
```

If we try to remove comment line for method addint() then compiler will give following error.

```
Exception in thread "main" java.lang.Error: Unresolved compilation problem:
      The method addint() from the type PrivateMemberClass1 is not visible
      at chapter2.PrivateMemberClass.main(PrivateMemberClass.java:24)
```

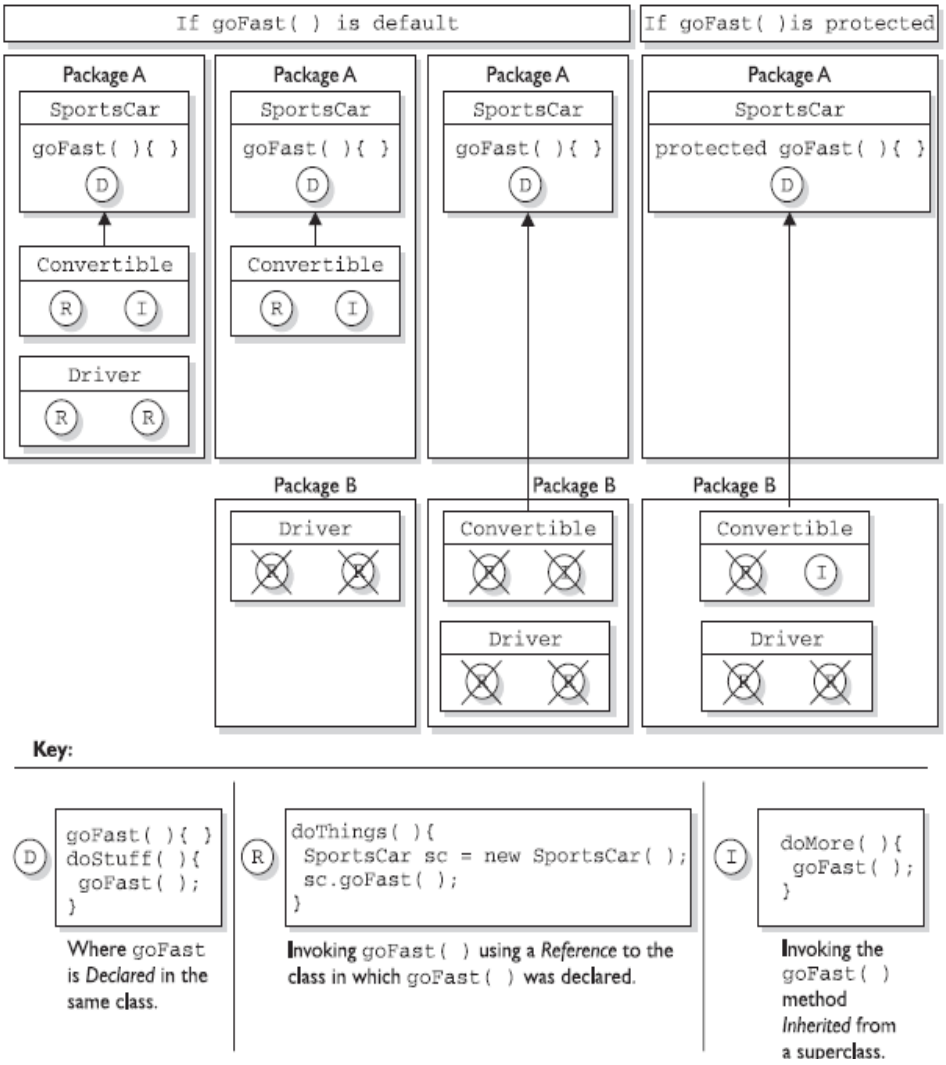
- Even for a subclass, superclass private members are invisible. When subclass declares a method of the same name following all the rules of overriding even then the method wont considered overridden as superclass method is invisible.
- It is always better to mark member variables as private and give access to them through public methods. For example if cat class have variable weight then it could take a negative value hence giving access through methods is a better solution where one can make a check for valid values.
- Effect of public and private modifiers on classes from same or diff packages is shown below.



- Protected and Default :

  protected and *default* modifiers are same with only one difference. *default* modifier is only for package level and no class outside package access those members by reference or inheritance. But with *protected* modifier a class in one package can inherit(that means only through subclass and not be reference i.e. (.) operator) protected members of another package. Hence one can say default is only for package level but protected is for package+kids.
- If Class A of package 1 has protected members and if it has been extended by Class B of package 2 then Class B will have access to protected members of Class A. But what if Class C in package 2 extends Class B of same package. Does Class C will also have access to protected members of Class A. NO. Because protected members in subclass becomes private members of it and wont be visible to other subclasses.
- The effect of default and protected access is shown below

The effects of protected access



- Summary :

Determining Access to Class Members

Visibility	Public	Protected	*Default*	Private
From the same class	Yes	Yes	Yes	Yes
From any class in the same package	Yes	Yes	Yes	No
From any non-subclass class outside the package	Yes	No	No	No
From a subclass in the same package	Yes	Yes	Yes	No
From a subclass outside the same package	Yes	Yes	No	No