Feel free to contact santhosh_arena@yahoo.co.in

SCJP java Programmer certification Exam And Training. This site is entirely independent of Sun Microsystems Inc, The Sun Certified Java Programmers Exam (SCJP) is the internationally recognized java certification exam. I am offering free mock Exam preparation questions, practice tests, tutorials, certification faq and sample code. This page contains a very detailed tutorial organized by topic. At the end of the tutorial you can learn from your mistakes and apply your concepts on the free mock exams java certification practice tests.

Chapter 6 - Objects and Classes (Part 1)

Implementing OO relationships

a) "is a" relationship is implemented by inheritance (extends keyword)

b) "has a" relationship is implemented by providing the class with member variables.

Overloading and Overriding

a) Overloading is an example of polymorphism. (operational / parametric)

b) Overriding is an example of runtime polymorphism (inclusive)

c) A method can have the same name as another method in the same class, provided it forms either a valid overload or override

Overloading	Overriding
Signature has to be different. Just a difference in return type is not enough.	Signature has to be the same. (including the return type)
Accessibility may vary freely.	Overriding methods cannot be more private than the overridden methods.
Exception list may vary freely.	Overriding methods may not throw more checked exceptions than the overridden methods.
Just the name is reused. Methods are independent methods. Resolved at compile-time based on method signature.	Related directly to sub-classing. Overrides the parent class method. Resolved at run-time based on type of the object.
Can call each other by providing appropriate argument list.	Overriding method can call overridden method by super.methodName(), this can be used only to access the immediate super-class's method. super.super won't work. Also, a class outside the inheritance hierarchy can't use this technique.

Methods can be static or non-static. Since the methods are independent, it doesn't matter. But if two methods have the same signature, declaring one as static and another as non-static does not provide a valid overload. It's a compile time error.	static methods don't participate in overriding, since they are resolved at compile time based on the type of reference variable. A static method in a sub-class can't use 'super' (for the same reason that it can't use 'this' for) Remember that a static method can't be overridden to be non-static and a non-static method can't be overridden to be static. In other words, a static method and a non-static method cannot have the same name and signature (if signatures are different, it would have formed a valid overload)
There's no limit on number of overloaded methods a class can have.	Each parent class method may be overridden at most once in any sub-class. (That is, you cannot have two identical methods in the same class)

- Variables can also be overridden, it's known as shadowing or hiding. But, member variable references are resolved at compile-time. So at the runtime, if the class of the object referred by a parent class reference variable, is in fact a sub-class having a shadowing member variable, only the parent class variable is accessed, since it's already resolved at compile time based on the reference variable type. Only methods are resolved at run-time.

```
public class Shadow {

  public static void main(String s[]) {

        S1 s1 = new S1();

        S2 s2 = new S2();

        System.out.println(s1.s); // prints S1

        System.out.println(s1.getS()); // prints S1

        System.out.println(s2.s); // prints S2

        System.out.println(s2.getS()); // prints S2

        s1 = s2;

        System.out.println(s1.s); // prints S1, not S2 -

                        // since variable is resolved at compile time

        System.out.println(s1.getS()); // prints S2 -

                        // since method is resolved at run time

}
```

```
}

class S1 {
  public String s = "S1";

  public String getS() {
        return s;
  }
}
 class S2 extends S1{
  public String s = "S2";

  public String getS() {
        return s;
  }
}
```

In the above code, if we didn't have the overriding getS() method in the sub-class and if we call the method from sub-class reference variable, the method will return only the super-class member variable value. For explanation, see the following points.

- Also, methods access variables only in context of the class of the object they belong to. If a sub-class method calls explicitly a super class method, the super class method always will access the super-class variable. Super class methods will not access the shadowing variables declared in subclasses because they don't know about them. (When an object is created, instances of all its super-classes are also created.) But the method accessed will be again subject to dynamic lookup. It is always decided at runtime which implementation is called. (Only static methods are resolved at compile-time)

```
public class Shadow2 {
  String s = "main";
  public static void main(String s[]) {
        S2 s2 = new S2();
```

```
        s2.display();  // Produces an output - S1, S2
        S1 s1 = new S1();
        System.out.println(s1.getS()); // prints S1
        System.out.println(s2.getS()); // prints S1 - since super-class method
                                       // always accesses super-class variable
    }
}
class S1 {
    String s = "S1";
    public String getS() {
        return s;
    }
    void display() {
        System.out.println(s);
    }
}
class S2 extends S1{
    String s = "S2";
    void display() {
        super.display();   // Prints S1
        System.out.println(s); // prints S2
    }
}
```

- With OO languages, the class of the object may not be known at compile-time (by virtue of inheritance). JVM from the start is designed to support OO. So, the JVM insures that the method called will be from the

real class of the object (not with the variable type declared). This is accomplished by virtual method invocation (late binding). Compiler will form the argument list and produce one method invocation instruction - its job is over. The job of identifying and calling the proper target code is performed by JVM.

- JVM knows about the variable's real type at any time since when it allocates memory for an object, it also marks the type with it. Objects always know 'who they are'. This is the basis of instanceof operator.

- Sub-classes can use super keyword to access the shadowed variables in super-classes. This technique allows for accessing only the immediate super-class. super.super is not valid. But casting the 'this' reference to classes up above the hierarchy will do the trick. By this way, *variables in super-classes above any level can be accessed from a sub-class, since variables are resolved at compile time, when we cast the 'this' reference to a super-super-class, the compiler binds the super-super-class variable*. But this technique is not possible with methods since methods are resolved always at runtime, and the method gets called depends on the type of object, not the type of reference variable. *So it is not at all possible to access a method in a super-super-class from a subclass.*

```
public class ShadowTest {

        public static void main(String s[]){

        new STChild().demo();

        }

}

class STGrandParent {

        double wealth = 50000.00;

        public double getWealth() {

        System.out.println("GrandParent-" + wealth);

        return wealth;

        }

}

class STParent extends STGrandParent {

        double wealth = 100000.00;

        public double getWealth() {

        System.out.println("Parent-" + wealth);

        return wealth;

        }
```

```
}

class STChild extends STParent {

        double wealth = 200000.00;



        public double getWealth() {

        System.out.println("Child-" + wealth);

        return wealth;

        }

        public void demo() {

        getWealth(); / / Calls Child method

        super.getWealth(); / / Calls Parent method

                / / Compiler error, GrandParent method cannot be accessed

        / /super.super.getWealth();

        / / Calls Child method, due to dynamic method lookup

                ((STParent)this).getWealth();

                / / Calls Child method, due to dynamic method lookup

                ((STGrandParent)this).getWealth();

        System.out.println(wealth); / / Prints Child wealth

        System.out.println(super.wealth); / / Prints Parent wealth

                / / Prints Parent wealth

                System.out.println(((STParent)(this)).wealth);

                / / Prints GrandParent wealth

        System.out.println(((STGrandParent)(this)).wealth);

        }
```

```
}
```

- An inherited method, which was not abstract on the super-class, can be declared abstract in a sub-class (thereby making the sub-class abstract). There is no restriction.

- In the same token, a subclass can be declared abstract regardless of whether the super-class was abstract or not.

- Private members are not inherited, but they do exist in the sub-classes. Since the private methods are not inherited, they cannot be overridden. A method in a subclass with the same signature as a private method in the super-class is essentially a new method, independent from super-class, since the private method in the super-class is not visible in the sub-class.

```
public class PrivateTest {

                public static void main(String s[]){

                new PTSuper().hi(); // Prints always Super

                new PTSub().hi(); // Prints Super when subclass doesn't have hi method

                        // Prints Sub when subclass has hi method

                PTSuper sup;

                sup = new PTSub();

                sup.hi(); // Prints Super when subclass doesn't have hi method

                // Prints Sub when subclass has hi method

                }

        }

         class PTSuper {

                public void hi() { // Super-class implementation always calls superclass hello

                hello();

                }

                private void hello() { // This method is not inherited by subclasses, but exists in them.

                // Commenting out both the methods in the subclass show this.
```

```
        // The test will then print "hello-Super" for all three calls
        // i.e. Always the super-class implementations are called
        System.out.println("hello-Super");
        }
    }
    class PTSub extends PTSuper {
        public void hi() { // This method overrides super-class hi, calls subclass hello
        try {
        hello();
        }
        catch(Exception e) {}
        }
        void hello() throws Exception { // This method is independent from super-class hello
        // Evident from, it's allowed to throw Exception
        System.out.println("hello-Sub");
        }
    }
```

- Private methods are not overridden, so calls to private methods are resolved at compile time and not subject to dynamic method lookup. See the following example.

```
    public class Poly {
        public static void main(String args[]) {
        PolyA ref1 = new PolyC();
        PolyB ref2 = (PolyB)ref1;
        System.out.println(ref2.g()); // This prints 1
```

```
                // If f() is not private in PolyB, then prints 2
            }
    }
    class PolyA {
            private int f() { return 0; }
            public int g() { return 3; }
    }
    class PolyB extends PolyA {
            private int f() { return 1; }
            public int g() { return f(); }
    }
    class PolyC extends PolyB {
            public int f() { return 2; }
    }
```