Chapter 7
Java
Inheritance

The division of the chapter is as follows:

Inheritance:

What is inheritance?

Ans: it is mechanism of deriving one class from the other.

What is superclass?

Ans: The class from which the otherclass is derived is called as superclass

What is subclass?

Ans: The class which is derived from the superclass is called as subclass.

Note:
The subclass is a specialized version of the superclass or it can be said that the subclass is the extended version of the superclass.

Use of Inheritance:

- In Java, we can create inheritance relationships by extending a class.
- The most common reasons for using inheritance are:
  - To promote code reuse
  - To use Polymorphism.

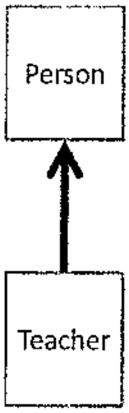
Types of Inheritance:

The different types of inheritance supported by Java are:

- **Single Inheritance**
- **Multilevel Inheritance**
- **Hierarchical Inheritance**

Q) WAJP to implement the following class diagram.

Program:



```
class Person {
private int id;
private String name;
private String address;
public int getId() {
return id;
}
public void setId(int id) {
this.id = id;
}
public String getName() {
return name;
}
public void setName(String name) {
this.name = name;
}
public String getAddress() {
return address;
}
public void setAddress(String address) {
this.address = address;
}}
```

```
class Teacher extends Person {
private double salary;
public double getSalary() {
return salary;
}
public void setSalary(double salary) {
this.salary = salary;
}
}
class TeacherTest {
public static void main(String args[]) {
Teacher t = new Teacher();
t.setId(101);
t.setName("Ravi");
t.setAddress("Mumbai");
t.setSalary(25000);
System.out.println("id = " + t.getId() );
System.out.println("name = " +
t.getName() );
System.out.println("address = " +
t.getAddress() );
System.out.println("salary = " +
t.getSalary() );
}
}
```

Output:

```
>javac TeacherTest.java
>java TeacherTest
 id = 101
  name = Ravi
  address = Mumbai
 Salary = 25000.0
```

What does JLS say?

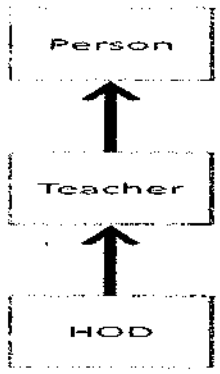
- A subclass **does not inherit the private members** of its parent class.
- However, if the superclass has public or protected methods for accessing its private fields, these can also be used by the subclass.
- Only members of a class that are declared protected or public are inherited by subclasses declared in a package other than the one in which the class is declared.

What do I say?

- When a class is inherited all the members of the super class are inherited by sub class.
- Now, the important thing to know is visibility. Private members are visible only in the class they are defined. Thus, the private members are not visible in the sub class.
- So, we may say as **private members are not inherited at all** because they are visible only in the class they are defined in.
- The only way private variables of base class can be altered is if there are public getters and setters.

Q) WAJP to implement the following class diagram.

Program:



```
class Person
{
protected int id;
protected String name;
protected String address;
public int getId()
{
return id;
}
public void setId(int id)
{
this.id = id;
}
public String getName()
{
return name;
}
public void setName(String name)
{
this.name = name;
}
public String getAddress()
{
return address;
}
public void setAddress(String
address)
{
this.address = address;
}}
```

```
class Teacher extends Person
{
protected double salary;
public double getSalary()
{
return salary;
}
public void setSalary(double salary)
{
this.salary = salary;
}
}

Class HOD~~{~~ extends Teacher{
protected String dept;

public String getDept(){
 return dept;

}
public void setDept(String dept).
 this.dept = dept.
}
}
```

```
class HODTest
{
public static void main(String args[])
{
HOD h= new HOD();
h.setId(102);
h.setName("Krishna");
h.setAddress("Mumbai");
h.setSalary(45000);
h.setDept("Computers");
System.out.println("id = " + h.getId() );
System.out.println("name = " + h.getName() );
System.out.println("address = " + h.getAddress() );
System.out.println("salary = " + h.getSalary() );
System.out.println("department = "+ h.getDept() );
}
}
```

Output:

```
>javac HODTest.java
>java HODTest
id = 102
name = Krishna
address = Mumbai
salary = 45000.00
department = Computers
```

IS-A and Has-A Relationship:

IS-A Relationship:

- In OOP, the **IS-A relationship** corresponds to the concept of inheritance.
- Consider the following code snippet:

```
class Person
{
// class implementation
}
class Student extends Person
{
// class implementation
}
```

Each object of a subclass is also the object of the superclass and not vice-versa

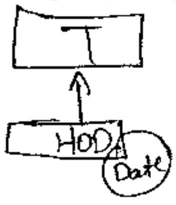
- From the above snippet the following conclusions can be drawn:
  - Student class is a Sub class of the Person class.
  - Student class is inherited from the Person class.
  - Student class is derived from the Person class.
  - Student class is a Sub type of the Person class.
- For eg: every dog is a animal but not every animal is a dog.
- Class will have IS-A relationship with the class, which is up in the inheritance structure and not vice-versa.

HAS-A Relationship:

- In OOP, the HAS-A relationship is a relationship where one object is a member of another object.
- In other words, the HAS-A relationship is based on containment rather than on inheritance.
- Consider the following code snippet:

```
class Teacher
{
...
}
class Hod extends Teacher
{
  private Date doj;
...
}
```

- In the above code, Teacher and Hod Class are in an is-A relationship whereas , the HOD and Date class have a HAS-A relationship.



Polymorphism:

- The term polymorphism comes from two Greek words, poly means many and morphs means forms.
- Polymorphism allows us to perform various operations by using the same method.
- In Java it is possible to use a single method to perform different functions by changing the implementation of the method.
- Polymorphism can be static or dynamic:
  - In static, also known as early binding, the binding is Performed during compilation time.
  - In dynamic, also known as late binding, the binding occurs during run time, depending on the type of object.

Static Polymorphism:

Static polymorphism is achieved by overloading methods.

Method Overloading:

- In Java, we can declare two or more methods, with the same name in the class, provided their parameter declarations vary.
- The Java compiler observes the signature of the methods (i.e. method name, number of parameters and type of parameters) and is able to differentiate amongst the methods of the same names by the difference in their method signatures.
- The difference can be:
  - **In the number of parameters.**
  - **Sequence of order of parameters.**
  - **Data types of parameters.**
- The difference in method signatures helps the Java compiler to bind the appropriate method.

Rules for Method Overloading:

Rule1: We MUST Change the argument list (number, type and/or sequence of Parameters).

Rule2: we Can change the return type.

Rule3: We Can Change the access modifier

Rule4: We Can make an overloaded method from static to non-static

Rule5: we Can overload a method in the Same Class Or in a subclass.

Rule6: We can declare new checked exceptions for the overloaded method.

Program to illustrate Method Overloading:

```
class A
{

protected void display()
{
System.out.println("welcome");
}

}

class B extends A
{

protected String display(String s)
{
return "welcome " + s;
}

public void display(String s, int n)
{
System.out.println("welcome " + s + "@ " + n);
}

}

class ABTest
{

public static void main(String args[])
{
A a = new A();
a.display();
B b = new B();
b.display();
String str1 = b.display("Vimal");
System.out.println(str1);
b.display("Kunal" , 200);
}

}
```

Output:

```
Welcome
Welcome
Welcome Vimal
Welcome Kunal@200.
```

Dynamic Polymorphism:

- Polymorphism exhibited at runtime is called dynamic polymorphism.
- The Java compiler is not aware of the method to be invoked during compilation; therefore JVM invokes the relevant method during runtime.
- This happens because methods are called by using objects type and objects are created at runtime.
- Hence Dynamic polymorphism is also called runtime polymorphism or dynamic binding.

Method Overriding:

- In method overriding, the Java compiler does not decide which method is called by the user, since it has to wait till the object of the sub class is created.
- After creating the object, JVM has to bind the method call to an appropriate method.
- But the methods in the super and sub classes have the same name and same method signatures. Then how will JVM decide which method is called?
- JVM calls the method depending on the reference type of the object which is used to call the method.

```
class A
{
void calculate(double x)
{
   System.out.println("Square of the give number " + x * x);
}
}

class B extends A
{
void calculate(double x)
{
System.out.println("Square root of given number   " + Math.sqrt(x) );
}
}

class CalTest
{
public static void main(String args[])
{
A a = new A();
a.calculate(5);
B b = new B();
b.calculate(36);
}
}
```

Method Overriding is a technique of re-implementing or re-writing a method of a superclass in its subclass.

Output:

Square of the give no. 25.0
Square root of given no. 6.0

Rules for Method Overriding:

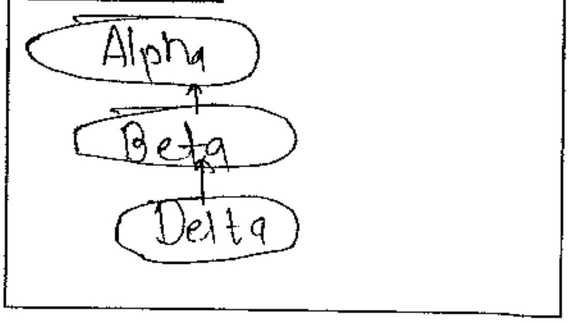
Rule1: The method signature must be same

Rule2: The return type must be same (from Java 5 covariant return type is also allowed).

Covariant Return Types:

```
class Alpha {
Alpha doStuff(char c) {
return new Alpha();
}}

class Beta extends Alpha {
Alpha doStuff(char c) {
return new Beta();
}}

class Delta extends Beta {
Alpha doStuff(char c) {
return new Beta();
}}
```

Class Diagram:



Rule3: We can change the access modifier in following way:
(default → protected → public.

Rule4: We can not override a static method to make it non-static.
(static → static, Nonstatic → Nonstatic)

Rule5: We cannot override private methods.
[becoz it is not visible].

Rule6: We cannot override the final method.

Rule7:
If the overriding method has a throws clause it its declaration, then the overridden method must also have a throws clause.

Impor

Overloading V/S Overriding:

Points of Difference	Overloading	Overriding
Arguments:	We must change the type, number or sequence of arguments of overloaded methods.	We must not change the type, number or sequence of arguments in the argument list.
Access Modifier:	We can change the access modifier of an overloaded method.	We can change the access modifier of the overridden method that is less restrictive than the superclass version of the method. (default → protected → public)
Return Type:	We can change the return type of an overloaded method.	We cannot change the return type of overridden method (except the covariant returns.)
Declaration Context:	A method can be overloaded in the same class or in a subclass.	A method can only be overridden in a subclass.
Method call resolution:	At compile time, the *declared type of the reference is used* to determine which method will be executed at runtime.	The runtime type *of the* reference, i.e., the type of the object referenced at *runtime,* determines which method is selected for execution.
Exceptions:	We can change the exceptions thrown by an overloaded method.	We can reduce or eliminate the exceptions of an overridden method, but the exception thrown must not be new.

Example of Overloading and Overriding:

```
class A
{
void calculate(int x)
{
System.out.println("Square of the given number " + x * x);
}
}

class B extends A
{
void calculate(int x)
{
System.out.println("Cube of the given number " + x * x * x);
}
void calculate(float x)
{
System.out.println("Square root of given number f " + Math.sqrt(x) );
}
}

class CalTest10b
{
public static void main(String args[])
{
A a = new A();
a.calculate(5);
B b = new B();
b.calculate(6);
b.calculate(36f);
}
}
```

Output: ~~8~~ Square Of the given no.25

cube " ~~3648~~ 216

Square root " 6.0

36
6
98

Constructors of Superclasses and Subclasses:

- When the instance of a subclass is created, the superclass constructor is invoked first and then the subclass constructor is invoked.
- In other words, every constructor invokes the constructor of its superclass with an implicit call to the super() method, before invoking the subclass constructor.

Rules for Constructors of Superclasses and Subclasses:

1) Every constructor invokes the DC of the super class. (unless it is called by super(par) or this()).

2) If the subclass constructor wants to call the superclass constructor, then it should use:

- ❖ super() in case of DC and
- ❖ super(parameter) in case of PC
- ❖ The super() or super(parameter) should be the first statement.

3) If the subclass method wants to call the superclass method, then use

- ❖ super.methodname() in case of no parameters.
- ❖ super.methodname(parameters) in case of parameters.
- ❖ The super.methodname() statement can be anywhere in the subclass method.

P1

```
class A {
A() {
System.out.println("A() SupCC");
}
}
class B extends A {
B() { super();
System.out.println("B() SubCC");
}
}
class ST30 {
public static void main(String args[]) {
A a = new A();
B b = new B();
}
}
```

Output:

```
A() SupCC
A() SupCC
B() SubCC
```

∀ Each Subclass Constructor Calls the DC Of Superclass

P2

```
class A {
A() {
System.out.println("A() SupCC");
}
}
class B extends A {
B() { ← Super();
System.out.println("B() SubCC");
}
B(int d) { ← super();
System.out.println("B(d) SubCC");
}
}
class ST31 {
public static void main(String args[]) {
A a = new A();
B b = new B();
B b1 = new B(5);
}
}
```

Output:

```
A() SupCC
A() SupCC
B() SubCC
B(d) SubCC
```

this(); ← Under 1st
Super(); Outer class

P3

```
class A {
A() {
System.out.println("A() SupCC");
}
}
class B extends A {
B() { ← Super();
System.out.println("B() SubCC");
}
B(int d) {
this();
System.out.println("B(d) SubCC");
}
}
class ST32 {
public static void main(String args[]) {
A a = new A();
B b = new B();
B b1 = new B(5);
}
}
```

Output:

```
A() SupCC
A() SupCC
B() SubCC
~~B(d) SubCC~~
```

```
A() SupCC
B() SubCC
B(d) SubCC
```

P4

```
class A {
A() {
System.out.println("A() SupCC");
}
A(int d) {
System.out.println("A(d) SupCC");
}
}
class B extends A {
B() {
System.out.println("B() SubCC");
}
B(int d) {
super(40);
System.out.println("B(d) SubCC");
}
}
class ST33 {
public static void main(String args[]) {
A a = new A();
B b = new B();
B b1 = new B(5);
}
}
```

Output:

A() SupCC
A() SupCC
B() SubCC
A(d) SupCC
B(d) SubCC

P5

```
class A {
A() {
System.out.println("A() SupCC");
}
A(int d) {
System.out.println("A(d) SupCC");
}
}
class B extends A {
B() {
System.out.println("B() SubCC");
}
B(int d) {
System.out.println("B(d) SubCC");
super(40);
}
}
class ST34 {
public static void main(String args[]) {
A a = new A();
B b = new B();
B b1 = new B(5);
}
}
```

Output:

CE, Bcoz call to Super must be the 1st statement in constructor.

P6

```
class A {
A() {
System.out.println("A() SupCC");
}
void displayA() {
System.out.println("A method");
}}
class B extends A {
B() {          super
System.out.println("B() SubCC");
}
void displayB() {
super.displayA();
System.out.println("B method");
super.displayA();
}}
class ST35 {
public static void main(String args[]) {
A a = new A();
a.displayA();
B b = new B();
b.displayB();
}
}
```

Output:

A () Sup CC
A method.
A()Sup CC
B() Sub CC
A method
B method
A method.

P7

```
class A {
A(int d) {
System.out.println("A(d) SupCC");
}
void displayA() {
System.out.println("A method");
}}

class B extends A {
void displayB() {
System.out.println("B method");
}}

class ST36 {
public static void main(String args[]) {
A a = new A(20);
a.displayA();
B b = new B();
b.displayB();
}}
```

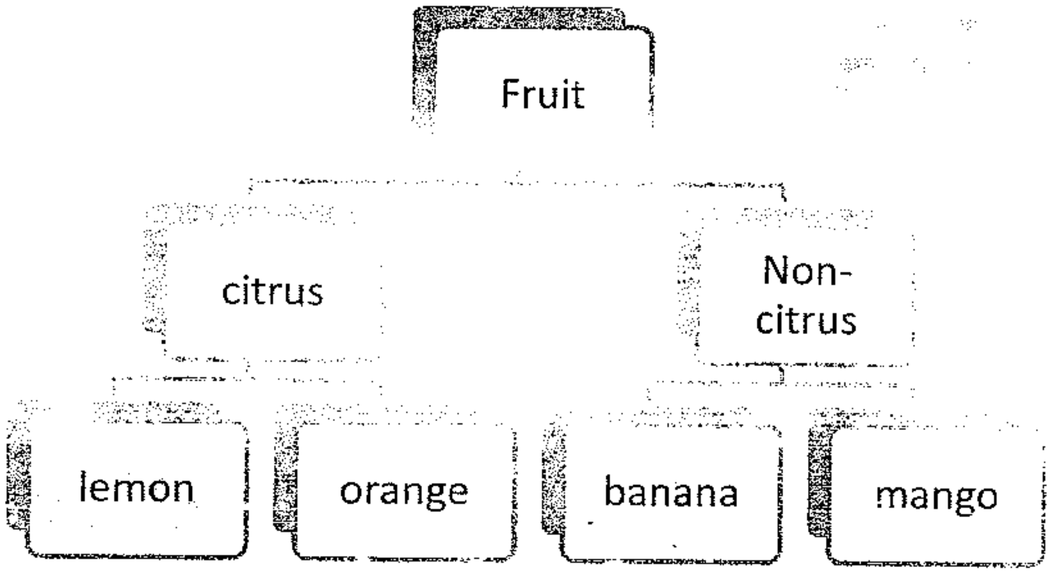
B() {
Super();
}

Output:

CE, Bcoz class A does not have a DC.

Reference Variable Casting:

- Converting a class type into another type is possible through casting.
- But the classes should have some relationship between them by the way of inheritance.
- When we come down from super class to sub classes, we are becoming more and more specific.
- This is called Specialization which needs narrowing or down-casting.
- When we go back from sub classes to super class, we are becoming more general.
- This is called Generalization which needs widening or upcasting.

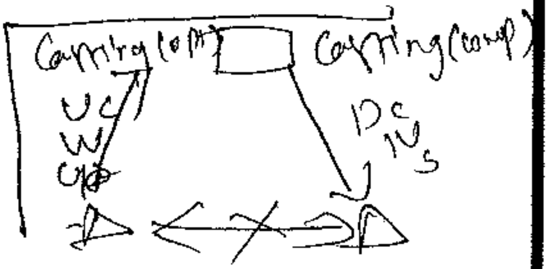


- **Generalization (Widening/Upcasting)** is safe because the classes will become more general.
- For example, if we say lemon is a fruit, there will be no objection.
- Hence Java compiler
- It will do implicit casting.
- <u>**Eg:**</u> Fruit f = new Fruit();
  Lemon l = new Lemon();
  f = l;
  f = (Fruit) l; (Optional Casting bcoz down to up);

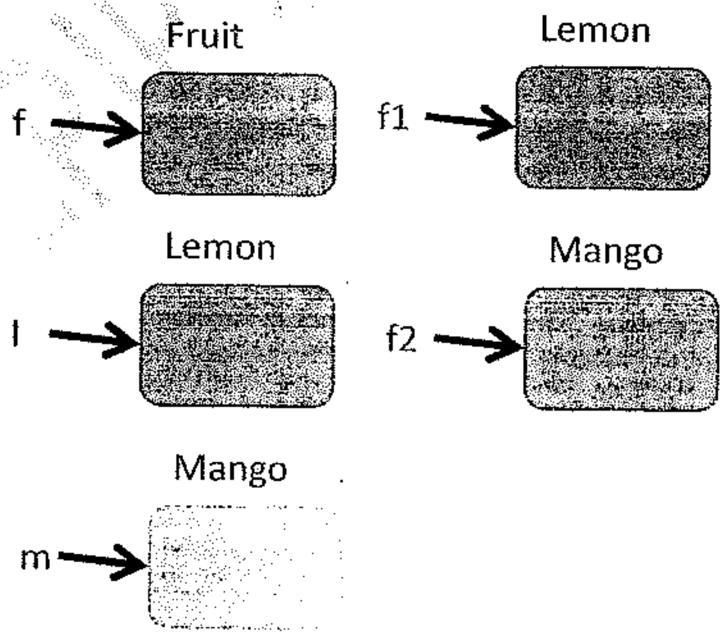
- **Specialization (Narrowing/Down-casting)** is not safe because the classes will become more and more specific.
- If we say fruit is a citrus fruit, we should show proof for the statement.
- Hence Java compiler will ask the programmer to use cast operator.
- <u>**Eg:**</u> Fruit f = new Fruit();
  Citrus c = new Citrus();
  c = f; → Incompatible types
  c = (Citrus) f;

Example:

```
class Fruit
{       }
class Lemon extends Fruit
{       }
class Mango extends Fruit
{       }

class ABC40Test {
public static void main(String args[] )
{
Fruit f = new Fruit();
Lemon l = new Lemon();
Mango m = new Mango();
Fruit f1 = new Lemon();
Fruit f2 = new Mango();

// Line1

}
}
```



Reference Variable Casting Example



No	Line1	Result	Note
1	f = l;	CAR	G
2	f = (Fruit)l;	CAR	G
3	f = m;	CAR	G
4	f = (Fruit)m;	CAR	G
5	l = f;	CF	S (incompatible types).
6***	l = (Lemon)f;	Compiles. But Jvm ~~CAR~~ ClassCastException raises java.lang.Class	Specialization Using object of Superclass does not works.
7	l = m;	CF	Incompatible type.
8	l = (Lemon)m;	CF	Incompatible types.
9	f = f1;	CAR	
10	f = f2;	CAR	
11	f1 = l;	CAR	G
12	f2 = m;	CAR	G
13	l = f1;	CF	Incompatible types.
14***	l = (Lemon)f1;	CAR	Specilization Using Subclass object works.
15	f1 = f;	CAR	
16	f2 = f;	CAR	

Test Paper

Q1) Select the correct options from the following in the context of the superclass Fruit and the subclass Apple. (Select two)

Class Diagram:

Options:

A. Fruit is derived from Apple.

[✓ B. Apple inherits from Fruit.]

C. Fruit HAS-A Apple.

[✓ D. Apple IS-A Fruit.]

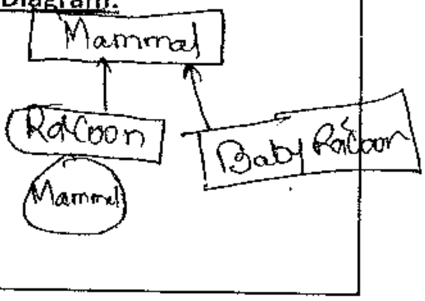
Solution: B & D

Q2) Given:

```
11. class Mammal { }
12.
13. class Raccoon extends Mammal {
14. Mammal m = new Mammal();
15. }
16.
17. class BabyRaccoon extends Mammal { }
```

Which four statements are true? (Choose four.)

Class Diagram:



Options:

[✓ A. Raccoon is-a Mammal.]

[✓ B. Raccoon has-a Mammal.]

[✓ C. BabyRaccoon is-a Mammal.]

D. BabyRaccoon is-a Raccoon.

E. BabyRaccoon has-a Mammal.

[✓ F. BabyRaccoon is-a BabyRaccoon.]

Solution: A B C F ,

Q3) Which four statements are true? (Choose four.)

Options:

~~A. Has-a relationships should never be encapsulated.~~

~~B. Has-a relationships should be implemented using inheritance.~~

[✓ C. Has-a relationships can be implemented using instance variables.]

[✓ D. Is-a relationships can be implemented using the extends keyword.]

[✓ E. Is-a relationships can be implemented using the implements keyword.]

F. The relationship between Movie and Actress is an example of an is-a relationship.

[✓ G. An array or a collection can be used to implement a one-to-many has-a relationship.]

Solution: C D E G

Q4) Given:

```
1. public class A {
2. public void doit() {
3. }
4. public String doit() {
5. return "a";
6. }
7. public double doit(int x) {
8. return 1.0;
9. }
10. }
```

What is the result?

Options

A. An exception is thrown at runtime.
B. Compilation fails because of an error in line 7
[✓ C. Compilation fails because of an error in line 4] [In Method overloading argument list must change]
D. Compilation succeeds and no runtime errors with class A occur.

Solution: C

Q5) Given the following class definitions:

```
1. class Parent {
2. public void printResults(String... results) {
3. System.out.println("In Parent");
4. }
5. }
6.
7. class Child extends Parent {
8. public int printResults(int id) {
9. System.out.println("In Child");
10. return 0;
11. }
12.}
```

what is the result of the following statement?

```
new Child().printResults(0);
```

Options:

A. In Parent
[✓ B. In Child]
C. 0
D. Line 2 generates a compiler error.
E. Line 8 generates a compiler error.

Solution:

Q6) What is the result of the following program?

```
class Parent {
public float computePay(double d) {
System.out.println("In Parent");
return 0.0F;
}
}

public class Child extends Parent {
public double computePay(double d) {
System.out.println("In Child");
return 0.0;
}

public static void main(String [ ] args) {
new Child().computePay(0.0);
}
}
```

Method overriding return type should be Same.

Options:

A. In Parent

B. In Child

C. 0.0

D. null

E. The code does not compile.

Solution: E

Q7) Given the following class definitions:

```
1. class Parent {
2. public void print(double d) {
3. System.out.print("Parent");
4. }
5. }
6.
7. class Child extends Parent {
8. public void print(int i) {
9. System.out.print("Child");
10. }
11.}
```

what is the result of the following code?

```
15. Child child = new Child();
16. child.print(10);
17. child.print(3.14);
```

Options:

A. ChildParent

B. ChildChild

C. ParentParent

D. Line 8 generates a compiler error.

E. Line 17 generates a compiler error

Solution: A

Q8) Given:

```
21. class Money {
22. private String country = "Canada";
23. public String getC() { return country; }
24. }
25. class Yen extends Money {
26. public String getC() { return super.country; }
27. }
28. public class Euro extends Money {
29. public String getC(int x) { return super.getC(); }
30. public static void main(String[ ] args) {
31. System.out.print(new Yen().getC() + " " + new Euro().getC());
32. }
33. }
```

Class Diagram:

What is the result?

Private members are not visible outside the Class

Options:

A. Canada

B. nullCanada

C. Canada null

D. Canada Canada

[✓ E. Compilation fails due to an error on line 26.]

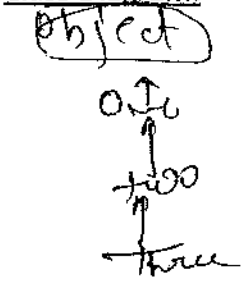
F. Compilation fails due to an error on line 29.

Solution:

Q9) Given:

```
class One {
public One foo() { return this; }
}
class Two extends One {
public One foo() { return this; }
}
class Three extends Two {
// insert method here
}
```

Class Diagram:



Which two methods, inserted individually, correctly complete the Three class? (Choose two.)

Options:

A. public void foo() {}

B. public int foo() { return 3; }

[✓ C. public Two foo() { return this; }]

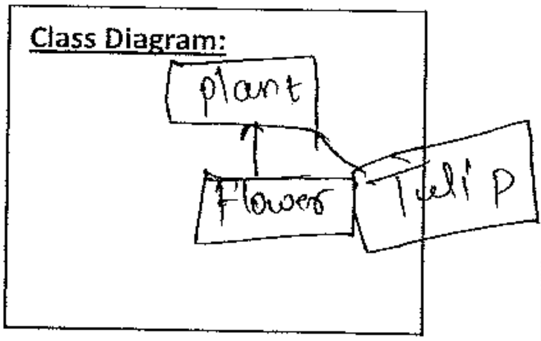
[✓ D. public One foo() { return this; }]

E. public Object foo() { return this; }

Solution:

Q10) Given:

```
1. class Plant {
2. String getName() { return "plant"; }
3. Plant getType() { return this; }
4. }
5. class Flower extends Plant {
6. // insert code here
7. }
8. class Tulip extends Plant { }
```



Class Diagram:

Which statement(s), inserted at line 6, will compile? (Choose three)

Options:

A. Flower getType() { return this; }

B. String getType() { return "this"; }

C. Plant getType() { return this; }

D. Tulip getType() { return new Tulip(); }

Solution: A, C, D'

Q11) Given:

```
10. class SuperCalc {
11. public static int multiply(int a, int b) { return a * b;}
12. }
```

and:

```
20. class SubCalc extends SuperCalc{
21. public static int multiply(int a, int b) {
22. int c = super.multiply(a, b); // SuperCalc.multiply.
23. return c;
24. }   25. }
```

and:

```
30. SubCalc sc = new SubCalc ();
31. System.out.println(sc.multiply(3,4));
32. System.out.println(SubCalc.multiply(2,2));
```

What is the result?

Options:

A. 12
4

B. The code runs with no output.

C. An exception is thrown at runtime.

D. Compilation fails because of an error in line 21.

[✓ E. Compilation fails because of an error in line 22.]

F. Compilation fails because of an error in line 31.

Super Can't be used for static Members.

Solution:

Q12) Given:

```
1. public class Blip {
2. protected int blipvert(int x) { return 0; }
3. }
4. class Vert extends Blip {
5. // insert code here
6. }
```

Which five methods, inserted independently at line 5, will compile? (Choose five.)

Options:

[✓ A. public int blipvert(int x) { return 0; }] OR

B. private int blipvert(int x) { return 0; } OR

[✓ C. private int blipvert(long x) { return 0; }] OL

D. protected long blipvert(int x) { return 0; } OR

[✓ E. protected int blipvert(long x) { return 0; }]

[✓ F. protected long blipvert(long x) { return 0; }] OL

[✓ G. protected long blipvert(int x, int y) { return 0; }]

Solution: A, C, E, F, G.

Q13) Given:

```
10. class One {
11. void foo() { }
12. }
13. class Two extends One {
14. //insert method here
15. }
```

default

Which three methods, inserted individually at line 14, will correctly complete class Two? (Choose three.)

Options:

A. int foo() { /* more code here */ }

[✓ B. void foo() { /* more code here */ }] B

[✓ C. public void foo() { /* more code here */ }]

D. private void foo() { /* more code here */ }

Default → Prot → public

[✓ E. protected void foo() { /* more code here */ }]

Solution: B, C, E

Q14) Given:

```
     2. public class Hi {
del  3. void m1() { }
     4. protected void() m2 { }
     5. }
     6. class Lois extends Hi {
     7. // insert code here
     8. }
```

Which four code fragments, inserted independently at line 7, will compile? (Choose four.)

Options:

A. public void m1() { } ✓

B. protected void m1() { } ✓

C. private void m1() { } ✗

D. void m2() { } ✗

E. public void m2() { } ✗ ✓

F. protected void m2() { } ✓

G. private void m2() { } ✗

Solution: A, B, E, F

Q15) Given:

```
     1. class ClassA {
     2. public int numberOfInstances;
     3. protected ClassA(int numberOfInstances) {
pc   4. this.numberOfInstances = numberOfInstances;
     5. }
     6. }
     7. public class ExtendedA extends ClassA {
     8. private ExtendedA(int numberOfInstances) {
pc   9. super(numberOfInstances);
    10. }
    11. public static void main(String[] args) {
    12. ExtendedA ext = new ExtendedA(420);
    13. System.out.print(ext.numberOfInstances);
    14. }   15. }
```

Options: ✓

✓ A. 420 is the output.

B. An exception is thrown at runtime.

C. All constructors must be declared public.

D. Constructors CANNOT use the private modifier.

E. Constructors CANNOT use the protected modifier.

Solution:

Q16)

```
class A {
A() {
System.out.println("A");
}}
class B105 extends A {
public static void main(String args[ ]) {   /* super should be in constructor, the 1st line)
~~super();~~
B105 b1 = new B105();
}
B105() {
System.out.println("B");
}}
```

What is the result?

Options:

A. Compiles and output: A

B. Compiles and output: B

[✓ C. Compilation fails.]

D. An exception is thrown at runtime.

Solution: C

Q17) Given:

```
class X {
X()(8) { System.out.print(1); }
(6) X(int x) {
this()(7); System.out.print(2);
}
}
public class Y extends X {
Y()(4) { super(6)(5); System.out.print(3); }
(2) Y(int y) {
this()(3); System.out.println(4);
}
public static void main(String[] a) { new Y(5)(1); }
}
```

What is the result?

this() :- within class
Super :- outside class

1 2 3 4

Options:

A. 13

B. 134

[✓ C. 1234]

D. 2134

E. 2143

F. 4321

Solution: C

Q18) Given:

```
class Atom {
Atom() { System.out.print("atom "); }
}
class Rock extends Atom {
Rock(String type) { System.out.print(type); }
}
public class Mountain extends Rock {
Mountain() {
super("granite ");
new Rock("granite ");
}
public static void main(String[] a) { new Mountain(); }
}
```

super

What is the result?

Options:

A. Compilation fails.

B. atom granite

C. granite granite

D. atom granite granite

E. atom granite atom granite ✓

Solution: B.

Q19) Given:

```
class MySuper {
public MySuper(String str) {
System.out.println("Hello");
}}
class MySub102 extends MySuper {
public MySub102(String str) {
System.out.println("Hi");
}
public static void main(String args[ ]) {
new MySub102("Bye");
}}
```

PC

PC

super

Sub102 class MySuper does not have a dc

What is the result?

Options:

A. Compiles and output is "HelloHi"

B. Compiles and output is "HelloHiBye".

C. Compiles and output is "Hi"

D. Compilation Error.

Solution:

Q20) Given:

```
1. class Super {
2. private int a;
3. protected Super(int a) { this.a = a; }
4. }
...
11. class Sub extends Super {
12. public Sub(int a) { super(a); }
13. public Sub() { this.a = 5; }
14. }
```

super or

Which two, independently, will allow Sub to compile? (Choose two.)

Options:

A. Change line 2 to: public int a;

B. Change line 2 to: protected int a;

[✓ C. Change line 13 to: public Sub() { this(5); }]

[✓ D. Change line 13 to: public Sub() { super(5); }]

E. Change line 13 to: public Sub() { super(a); }

Solution: C & D

Q21) Given:

```
3. class Employee {
4. String name; double baseSalary;
5. Employee(String name, double baseSalary) {
6. this.name = name;
7. this.baseSalary = baseSalary;
8. }
9. }
10. public class SalesPerson extends Employee {
11. double commission;
12. public SalesPerson(String name, double baseSalary, double commission) {
13. // insert code here
14. } 15. }
```

super

Which two code fragments, inserted independently at line 13, will compile? (Choose two.)

Options:

[✓ A. super(name, baseSalary);]

B. this.commission = commission;

C. super(); this.commission = commission;

D. this.commission = commission; super();

[✓ E. super(name, baseSalary); this.commission = commission;]

F. this.commission = commission; super(name, baseSalary);

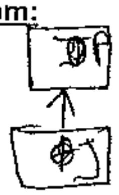
G. super(name, baseSalary, commission);

Solution: A & E

Q22) Suppose class Jasmine extends class Flower. Which of the following choices will compile correctly, given the following code? Select three choices.

```
Jasmine j=new Jasmine();
Flower f=new Flower();
```

Class Diagram:



Options:

A. f=j; ✓ G1

B. f=(Flower)j; ✓ M

C. j=(Jasmine)f; ✓ S

D. j=f;

E. j=(Flower)f;

Solution: A, B, C

Q23) Given:

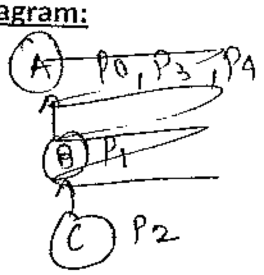
```
11. class ClassA {}
12. class ClassB extends ClassA {}
13. class ClassC extends ClassA {}
```

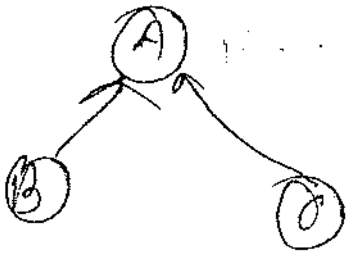
and:

```
21. ClassA p0 = new ClassA();
22. ClassB p1 = new ClassB();
23. ClassC p2 = new ClassC();
24. ClassA p3 = new ClassB();
25. ClassA p4 = new ClassC();
```

Which three are valid? (Choose three.)

Class Diagram:





Options:

A. p0 = p1; ✓ G1

B. p1 = p2;

C. p2 = p4; ✗ S Casting

D. p2 = (ClassC)p1; ✗

E. p1 = (ClassB)p3; ✓ S

F. p2 = (ClassC)p4; ✓ S

Solution: A, E, F