Experiments

① interface I
{
default void show1()
{ s.o.p.ln ("Inside I inside show1");
}
}
class Main implements I
{
psvm(-)
{
new Main().show1();
I ii = new Main();
ii.show1();
}
}

Main would also work here

Output
Inside I ~~not~~ inside show1
Inside I inside show1

Conclusions
- default method in I does not need to be overridden or implemented in class Main
- Interface reference can be created to hold object of class.

~~That~~

② interface I
{ void show(); }
class Main implement I
{ public void show() { s.o.p.ln("show"); }
psvm(-)
{ I ii = new Main();
ii.show(); → show
}
}
- Interface reference can only call methods ~~are~~ related to interface or those methods overridden in class.

③ interface I → Illegal modifier
{ protected void hello();
}
// Compile error
// Only public, private, abstract
// default, static, strictfp are permitted.

④ interface I
{ private void hello();
}
// Compile error
// Method Requires a body instead of semi colon

⑤ interface I
{ private void hello()
{ s.o.u.t("hello in I..."); } }
}
class Main implements I
{ psvm(-)
{ I ii = new Main();
Main iii = new Main();
ii.hello(); iii.hello(); // Both can't access hello.
}
}

⑥ interface I → Illegal combination
{ abstract static void hi();
}
// only one of abstract, default, static permitted
// error Compiler

(7)
```
interface I
{
    public default void show()
    {
        s.o.p.t("Inside I");
    }
}
class Main implements I
{
    public void show()
    {
        s.o.p.t("Inside Main");
    }
    psvm(_)
    {
        I ii = new Main();
        Main ii1 = new Main();
        ii.show();   → Inside Main
        ii1.show();  → Inside Main
    }
}
```

not writing public decreases scope as it is overriding here.

Here is says This code was not there

then output → Inside I
Inside I

All method declarations in an interface including default methods, are implicitly public.

(8)
```
interface I
{
    public static void main(String args[])
    {
        s.o.p.ln("inside I inside main");
    }
}
class Main implements I
{
    psvm(_)
    {
        ~~I ii = new Main();~~
        I.main(args);
    }
}
```

↳ Call static methods by their name

9)
```
interface I
{
  public static void main(int arg)
  { s.o.ut(" inside I inside main");
  }
}
class Main implements I
{  public static void main(int arg)
   {
    s.o.p.ln(" inside class inside main");
    I.main (2);
   }
   ps vm ( String args[])
   { main(1);
   }
}
```

→ inside class inside main
→ inside I inside main

10) We can also call, void main (int arg) of Main from void main(int arg) of interface I by writing Main.main(2) in interface I method.

```
interface I
{  public static void main (int arg)
   { s.o.p.ln(" I 1");
     Main.main(1);
   }
}
class Main implements I
{  public static void main (int arg)
   { s.o.p.ln(" I 2");
   }
   public static void main(String args[])
   { I.main(1);
   }
}
```

→ I1
I2

11) Overloading is possible in interface too.

```
interface I
{  public static void main(int arg)
   {
   s.o.p.ln("inside I int arg inside main");
   I.main(2.0);
   }
   public static void main(double arg)
   {
   s.o.p.ln("inside I double arg inside main");
   Main.main(2);
   }
}
class Main implements I
{  public static void main(int arg)
   {
   s.o.p.ln("inside class inside main");
   }
   psvm(String args[])
   {
   I.main(1);
   }
}
```

Calling an interface method itself from inside an interface method.

Not writing anything over here, implying default or writing protected works over here.

→ inside I int arg inside main
inside I double arg inside main
inside class inside main

→ Reason we are able to use protected:-
static methods of interface cannot be overridden, so if we decrease scope from public in interface to protected in class, there for a function with same signature, there is no harm.

12) Overloading is possible for default methods in interface.

```
interface I
{   default void hello()
    {   s.o.p.ln("hello in I");
    }
    default void hello(int a)
    {   s.o.p.ln("hello in I int");
    }
}
class Main implements I
{   psvm(-)
    {   Main m = new Main();
        m.hello();
        m.hello(9);
    }
}
```

→ hello in I
hello in I int

13) Overridding is possible for static methods in interface

```
interface I
{   default void hello()
    {   s.o.p.ln("I1"); }
    default void hello(int a)
    {   s.o.p.ln("I2"); }
}
class Main implements I
{   public void hello()
    {   s.o.put("C1"); }
    public void hello(int a)
    {   s.o.ut("C2"); }
    psvm(-)
    {   Main m = new Main();
        m.hello(); m.hello(9);
    }
}
```

→ C1
C2

Only public works here as default methods are implicitly public.

14) Calling various functions

```
interface I
{ default void hello()
  {
  s.o.p.ln("hello in I");
  Main m = new Main();
  m.display(); //Create object for calling default method
  m.hello(4);
  ~~m.k~~ I.hello("yoo");
  }
  default void hello(int a)
  {s.o.p.ln("hello in I int");
  }
  static void hello(String a)
  { kyu(a); //Calling static interface method
    new Main().show();
  }
  static void kyu(String a)
  {s.o.p.ln("kyu "+a);
  }
  default void display()
  {s.o.p.ln("display in I");
  }
  default void show()
  {s.o.p.ln("show in I");
  }
}
class Main implements I
{ public void display()
  {s.o.p.ln("display in C");
  }
  psvm(--)
  {Main m=new Main();
   m.hello();
  }
  static void kyu(String a)
  {s.o.p.ln("kyu in C"+a);
  }
}
```

overloading

```
hello in I
display in C
hello in ~~Int~~
kyu yoo
show in I
```

15)
```
interface J
{ static void hi()
  { s.o.p.ln("hi from J");
  }
}
class Test implements J
{ psvm(String args[])
  { hi(); } error
    super.hi();
  { Test t = new Test();
      t.hi(); // error
  }
}
```
Only J.hi() works.

16) Inside interface, we can use private in following ways:-

```
private void hi() { }
private static void hi() { }
private strictfp void hi() { }
private static strictfp void hi() { }
```
Only static, strictfp can be written with private.

17)
```
interface Foo
{ default void bar(){
  s.o.p.ln("bar in I");
  baz(); hi(); hello();
  }
  private static void bar/baz(){
  s.o.v.t("baz in I");
  new Main().hi();
  Main.baz();
  hello();
  new Main().this.hello(); } error
}
private void hello() { s.o.p.ln("hello in I"); }
default void hi() { s.o.p.ln("hi in I"); }
}
class Main implements Foo
{ psvm(-) { new Main().bar(); }
  public static void baz()
  { s.o.p.ln("baz in C"); }
}
```

bar in I baz in I
hi in I
baz in C
hello in I
hi in I

## Conclusions

- ~~Within a~~ From a default method, if we are calling default method inside interface we don't need to create object. If we create an object, then also no error, while calling default from default.
- From a default method, calling style of private, private static is same → just use function name.
- private static can call default methods using object of implementing class (of interface)
- But private static can in no way call private methods (with or without objects)
  - Same for only static method
- Private static can call another private static using its name inside interface.
- Private static can also call implementing class' static method using the class name.
- Private ~~class can~~ static can also call implementing class' non static method by creating an object of that class.
- private method in interface can call :-
  - ~~static~~ → private static method by name of method
  - default method of interface by its name
  - static method of ~~implenting~~ implementing class by class name
  - non static method of implementing class by its object creation.
  - another private method
  - another static ~~void~~ method
  - we can overload private methods
- private static can also call :-
  - static only methods
  - we can also overload private static.

Interface allows these methods :-

- default
- static
- private
- private static
- public abstract

---

## Module 2 of my experiments

1) protected interface I
{ }
error :- Illegal modifier for interface I ; only public abstract are permitted.

2) abstract interface I
{ }
No compile error

3) If we are writing any interface as public, file name should be interface name.

public interface hey
{ }
→ hey.java
filename

4) Inner Interface

interface I
{
interface J
{ }
}

→ public when written with outer interface becomes available to all packages else only in present package if we don't write default.
interface I
{
public interface J
{ }
}

only public is allowed.

5) strictfp interface I
{ }
→ warning → Floating point expressions are strictly evaluated from source level 17. Keyword strictfp is not required

static is also allowed.

interface I
{
public static interface J
{ }
static interface K
{ }
}

6)
```
interface I {
    void show();
    interface J
    {
        void hi();
    }
}
class Main implements I
{
    public void show()
    { s.o.p.ln("show"); }
    psvm(-)
    {
    }
}
```

No compile error if hi() is not there as Main does not implement inner interface J.

7)
```
interface I {
    void show();
    interface J
    {
        void hi();
    }
}
class Main implements I, I.J
{
    public void show()
    { s.o.p.ln("show"); }
    public void hi()
    { s.o.p.ln("hi"); }
    psvm(-)
    { }
}
```

8)
```
abstract interface I
{
    void show();
    static void display()
    { }
}
```

→ default, private, private static are also allowed.

9)
```
interface I {
    void show();
    interface J
    { void hi();
    }
}
class Main implements I.J
{   public void hi()
    {
    }
    psvm(-)
    {
    }
}
```

// If there is no show(), then also correct as Main does not implement I.

10) Inner interfaces are implicitly public & static. (within another class)

11)
```
interface I
{   interface J
    {   static void hello()
        { s.o.p.ln("hello in I.J"); }
        default void huhu()
        { s.o.p.ln("huhu in I.J"); }
    }
    static void hello()
    { s.o.p.ln("hello in I"); }
    default void ~~hello~~ huhu()
    { s.o.p.ln("huhu in I.J"); }
}
class Main implements I.J, I
{   psvm(-)
    { }
}
```

// Compile Error
// due to huhu
// Not due to hello.
So only default functions with same name in inner & outer interfaces generate error and static functions dont.

If we change method signature in any of default, error goes away.

12)
```
interface J
{ void show(); }
interface I extends J
{ }
class Test implements I
{ public void show(){ } → not writing show definition will give error
  psvm(-)
  {
  }
}
```

13)
```
interface I
{ }
interface J extends I
{ }
class Test implements I, J → warning :- redundant super interface I for type Test already defined by J
{ psvm(-) { } }
```

14)
```
interface I
{ void show(); }
interface J extends I
{ default void show()
  { s.o.p.ln("show in J"); }
}
class Main implements J
{ psvm(-) → We already gave show() implementation in J so we don't have to give implementation in Main
  { }
}
```

- Following give error:-
  - Static
  - private
  - private static
  - public
- Only default or default public works

15)
```
interface J
{ default void show(){ s.o.p.ln("show in J"); } }
interface I extends J { }
class hey implements I
{ psvm(-){
    I i = new hey(); i.show(); → show in J printed (since I show() was searched in hey, if found then hey's show() printed. Then show was searched in I, then J)
    hey h = new hey(); h.show();
    J j = new hey(); j.show();
  }
}
```

16)
```
interface J
{  void help();
}
interface I extends J
{   interface K
    {  default void help()
       {  s.o.p.ln("help in I.K");
       }
    }
}
class hey implements I, I.K
{   psvm(-)
    {}
}
```

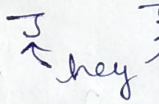
// This is compile error.
// default method help() inherited from I.K conflicts with another method inherited from J.

Above and below are same program implemented in different way. Both give compile error

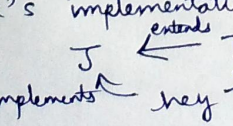
```
interface J
{   void help();
}
interface I extends J
{   interface K
    {  default void help()
       {  s.o.p.ln("help in I.K");
       }
    }
}
class hey implements J, I.K
{   psvm(-){}
}
```

We might think that implementation of help() that J requires can be given by I.K but that doesn't happen.

What happening?




From diagram, we see there's no relation between J, I.K, hence I.K is not able to transfer its hey() implementation to J. So to ~~est~~ give solve this error I.K must extend J in above program, then I.K's implementation of hey() will satisfy J.




(warning may occur as I.K's ~~super~~ super interface is J)

## some conclusions

- If class extends an interface (lets say J), it must provide implementation for all abstract methods.
- If class does not provide implementation for any method which is abstract, then it must implement some other interface I which is extending J and also providing implementation inside I itself for that abstract method.

17)
```
interface J extends I.K
{
    void help();
    default void done()
    { s.o.p.ln("done in J"); }
}
interface I
{
    interface K extends J
    {
        default void help()
        { s.o.p.ln("help in I.K"); }
        void done();
    }
}
```

→ Compile error → cycle detected (a cycle exists in type hierarchy between I.K and J)

18)
```
interface J
{ void show(); }
interface I
{
    void show();
    interface K extends I, J
    {
        default void show()
        { s.o.p.ln("show in I.K"); }
        default void help()
        { s.o.p.ln("help in I.K"); }
    }
}
class hey implements I, I.K
{
    psvm(-)
    { }
}
```

inner interface can extend outer interface.

19)
```
class Test
{ interface Yes
  { void help(); }
}
class Y extends Test
{ }
```
↳ ~~{}~~ even if we don't provide help()'s implementation, we will not get error as we explicitly don't implement the interface.

20)
```
class Test
{ interface Yes
  { void show(); }
}
class T implements Test.Yes
{ public void show()
  { s.o.p.ln("show of interface"); }
}
```
✓ Since we implemented interface explicitly, we need to provide its implementation.

21)
```
class Test implements Test.Yes
{ interface Yes
  { void show(); }
  void show() {}
}
```
→ compile error
cycle detected
we can't implement/extend ~~same class~~ itself.

22)
```
class Test
{ interface Yes
  { void show(); }
  class tt extends Test implements Test.Yes
  { }
  public void show()
  { s.o.p.ln("show"); }
}
```
✓ Inner interface method implementation provided by outer class.

23) Interface inside a class can be protected but interface inside interface cannot be protected.

private, public too

24) interface J
{ public static void main(String[] args)
{ s.o.p.ln("hi from J");
}
}

javac J.java
java J
→ hi from J

class inside an interface will always be static

25) interface J
{ class A implements J
{ psvm(—)
{ s.o.w.t("main from J.A");
}
}
}

javac J.java
java J
Runtime error → main not found

26)
```
interface J
{   psvm(-)
    {  }
}
class A implements J
{  }
```
→ If class was written here, program would run fine and A would extend J

```
javac J.java
java A
Runtime error
Main not found
```

27)
```
interface J
{   void hi();
    class A implements J
    {  public void hi()
       {  s.o.p.ln("hi from J.A");
       }
    }
    psvm(-)
    {  s.o.p.ln("main from J");
       new A().hi();
    }
    void hi();
}
```

```
javac J.java
java J
→ main from J
→ hi from J.A
```

28)
```
interface J
{   class A
    {  public void hi()
       {  s.o.u.t("hi from J.A");
       }
    }
    void hi();
}
class Main extends J.A implements J
{   psvm(-)
    {   J.A obj = new J.A();
        obj.hi();  → hi from J.A
    }
}
```