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 3)

Interfaces:

- All **methods** in an interface are implicitly public, abstract, and **never static**.
- All variables in an interface are implicitly static, public, final. They cannot be transient or volatile. A class can shadow the variables it inherits from an interface, with its own variables.
- A top-level interface itself cannot be declared as static or final since it doesn't make sense.
- **Declaring parameters to be final is at method's discretion, this is not part of method signature.**
- Same case with final, synchronized, native. Classes can declare the methods to be final, synchronized or native whereas in an interface they cannot be specified like that. (These are implementation details, interface need not worry about this)
- But classes cannot implement an interface method with a static method.
- If an interface specifies an exception list for a method, then the class implementing the interface need not declare the method with the exception list. (Overriding methods can specify sub-set of overridden method's exceptions, here none is a sub-set). But if the interface didn't specify any exception list for a method, then the class cannot throw any exceptions.
- All interface methods should have public accessibility when implemented in class.
- Interfaces cannot be declared final, since they are implicitly abstract.
- A class can implement two interfaces that have a method with the same signature or variables with the same name.

Inner Classes

- A class can be declared in any scope. Classes defined inside of other classes are known as **nested classes**. There are four categories of nested classes.

1. Top-level nested classes / interfaces

    - Declared as a class member with static modifier.

    - Just like other static features of a class. Can be accessed / instantiated without an instance of the outer class. Can access only static members of outer class. Can't access instance variables or methods.

    - Very much like any-other package level class / interface. Provide an extension to packaging by the modified naming scheme at the top level.

    - Classes can declare both static and non-static members.

    - Any accessibility modifier can be specified.

    - Interfaces are implicitly static (static modifier also can be specified). They can have any accessibility modifier. There are no non-static inner, local or anonymous interfaces.

2. Non-static inner classes

    - Declared as a class member without static.

    - An instance of a non-static inner class can exist only with an instance of its enclosing class. So it always has to be created within a context of an outer instance.

    - Just like other non-static features of a class. Can access all the features (even private) of the enclosing outer class. Have an implicit reference to the enclosing instance.

    - Cannot have any static members.

    - Can have any access modifier.

3. Local classes

    - Defined inside a block (could be a method, a constructor, a local block, a static initializer or an instance initializer). Cannot be specified with static modifier.

    - Cannot have any access modifier (since they are effectively local to the block)

- Cannot declare any static members.(Even declared in a static context)

- Can access all the features of the enclosing class (because they are defined inside the method of the class) but can access only final variables defined inside the method (including method arguments). This is because the class can outlive the method, but the method local variables will go out of scope - in case of final variables, compiler makes a copy of those variables to be used by the class. (New meaning for final)

- Since the names of local classes are not visible outside the local context, references of these classes cannot be declared outside. So their functionality could be accessed only via super-class references (either interfaces or classes). Objects of those class types are created inside methods and returned as super-class type references to the outside world. This is the reason that they can only access final variables within the local block. That way, the value of the variable can be always made available to the objects returned from the local context to outside world.

- Cannot be specified with static modifier. But if they are declared inside a static context such as a static method or a static initializer, they become static classes. They can only access static members of the enclosing class and local final variables. But this doesn't mean they cannot access any non-static features inherited from super classes. These features are their own, obtained via the inheritance hierarchy. They can be accessed normally with 'this' or 'super'.

4. Anonymous classes

- Anonymous classes are defined where they are constructed. They can be created wherever a reference expression can be used.

- Anonymous classes cannot have explicit constructors. Instance initializers can be used to achieve the functionality of a constructor.

- Typically used for creating objects on the fly.

- Anonymous classes can implement an interface (implicit extension of Object) or explicitly extend a class. Cannot do both.

Syntax: new interface name() { } or new class name() { }

- Keywords implements and extends are not used in anonymous classes.

- Abstract classes can be specified in the creation of an anonymous class. The new class is a concrete class, which automatically extends the abstract class.

- Discussion for local classes on static/non-static context, accessing enclosing variables, and declaring static variables also holds good for anonymous classes. In other words, anonymous classes cannot be specified with static, but based on

the context, they could become static classes. In any case, anonymous classes are not allowed to declare static members. Based on the context, non-static/static features of outer classes are available to anonymous classes. Local final variables are always available to them.

- One enclosing class can have multiple instances of inner classes.

- Inner classes can have synchronous methods. But calling those methods obtains the lock for inner object only not the outer object. If you need to synchronize an inner class method based on outer object, outer object lock must be obtained explicitly. Locks on inner object and outer object are independent.

- Nested classes can extend any class or can implement any interface. No restrictions.

- All nested classes (except anonymous classes) can be abstract or final.

- Classes can be nested to any depth. Top-level static classes can be nested only within other static top-level classes or interfaces. Deeply nested classes also have access to all variables of the outer-most enclosing class (as well the immediate enclosing class's)

- Member inner classes can be forward referenced. Local inner classes cannot be.

- An inner class variable can shadow an outer class variable. In this case, an outer class variable can be referred as (outerclassname.this.variablename).

- Outer class variables are accessible within the inner class, but they are not inherited. They don't become members of the inner class. This is different from inheritance. (Outer class cannot be referred using 'super', and outer class variables cannot be accessed using 'this')

- An inner class variable can shadow an outer class variable. If the inner class is sub-classed within the same outer class, the variable has to be qualified explicitly in the sub-class. To fully qualify the variable, use classname.this.variablename. If we don't correctly qualify the variable, a compiler error will occur. (Note that this does not happen in multiple levels of inheritance where an upper-most super-class's variable is silently shadowed by the most recent super-class variable or in multiple levels of nested inner classes where an inner-most class's variable silently shadows an outer-most class's variable. Problem comes only when these two hierarchy chains (inheritance and containment) clash.)

- If the inner class is sub-classed outside of the outer class (only possible with top-level nested classes) explicit qualification is not needed (it becomes regular class inheritance)

Chapter 6 - Objects and Classes (Part 3)

Part 1 Part 2 Part 4

Interfaces:

- All **methods** in an interface are implicitly public, abstract, and **never static**.
- All variables in an interface are implicitly static, public, final. They cannot be transient or volatile. A class can shadow the variables it inherits from an interface, with its own variables.
- A top-level interface itself cannot be declared as static or final since it doesn't make sense.
- **Declaring parameters to be final is at method's discretion, this is not part of method signature.**
- Same case with final, synchronized, native. Classes can declare the methods to be final, synchronized or native whereas in an interface they cannot be specified like that. (These are implementation details, interface need not worry about this)
- But classes cannot implement an interface method with a static method.
- If an interface specifies an exception list for a method, then the class implementing the interface need not declare the method with the exception list. (Overriding methods can specify sub-set of overridden method's exceptions, here none is a sub-set). But if the interface didn't specify any exception list for a method, then the class cannot throw any exceptions.
- All interface methods should have public accessibility when implemented in class.
- Interfaces cannot be declared final, since they are implicitly abstract.
- A class can implement two interfaces that have a method with the same signature or variables with the same name.

Inner Classes

- A class can be declared in any scope. Classes defined inside of other classes are known as **nested classes**. There are four categories of nested classes.

1. Top-level nested classes / interfaces
    - Declared as a class member with static modifier.

- Just like other static features of a class. Can be accessed / instantiated without an instance of the outer class. Can access only static members of outer class. Can't access instance variables or methods.

- Very much like any-other package level class / interface. Provide an extension to packaging by the modified naming scheme at the top level.

- Classes can declare both static and non-static members.

- Any accessibility modifier can be specified.

- Interfaces are implicitly static (static modifier also can be specified). They can have any accessibility modifier. There are no non-static inner, local or anonymous interfaces.

2. Non-static inner classes

- Declared as a class member without static.

- An instance of a non-static inner class can exist only with an instance of its enclosing class. So it always has to be created within a context of an outer instance.

- Just like other non-static features of a class. Can access all the features (even private) of the enclosing outer class. Have an implicit reference to the enclosing instance.

- Cannot have any static members.

- Can have any access modifier.

3. Local classes

- Defined inside a block (could be a method, a constructor, a local block, a static initializer or an instance initializer). Cannot be specified with static modifier.

- Cannot have any access modifier (since they are effectively local to the block)

- Cannot declare any static members.(Even declared in a static context)

- Can access all the features of the enclosing class (because they are defined inside the method of the class) but can access only final variables defined inside the method (including method arguments). This is because the class can outlive the method, but the method local variables will go out of scope - in case of final variables, compiler makes a copy of those variables to be used by the class. (New meaning for final)

- Since the names of local classes are not visible outside the local context, references of these classes cannot be declared outside. So their functionality could be accessed only via super-class references (either interfaces or classes). Objects of those class types are created inside methods and returned as super-class type references to the outside world. This is the reason that they can only access final variables within the local block. That way, the value of the variable can be always made available to the objects returned from the local context to outside world.

- Cannot be specified with static modifier. But if they are declared inside a static context such as a static method or a static initializer, they become static classes. They can only access static members of the enclosing class and local final variables. But this doesn't mean they cannot access any non-static features inherited from super classes. These features are their own, obtained via the inheritance hierarchy. They can be accessed normally with 'this' or 'super'.

4. Anonymous classes

- Anonymous classes are defined where they are constructed. They can be created wherever a reference expression can be used.

- Anonymous classes cannot have explicit constructors. Instance initializers can be used to achieve the functionality of a constructor.

- Typically used for creating objects on the fly.

- Anonymous classes can implement an interface (implicit extension of Object) or explicitly extend a class. Cannot do both.

Syntax: new interface name() { } or new class name() { }

- Keywords implements and extends are not used in anonymous classes.

- Abstract classes can be specified in the creation of an anonymous class. The new class is a concrete class, which automatically extends the abstract class.

- Discussion for local classes on static/non-static context, accessing enclosing variables, and declaring static variables also holds good for anonymous classes. In other words, anonymous classes cannot be specified with static, but based on the context, they could become static classes. In any case, anonymous classes are not allowed to declare static members. Based on the context, non-static/static features of outer classes are available to anonymous classes. Local final variables are always available to them.

- One enclosing class can have multiple instances of inner classes.

- Inner classes can have synchronous methods. But calling those methods obtains the lock for inner object only not the outer object. If you need to synchronize an inner

class method based on outer object, outer object lock must be obtained explicitly. Locks on inner object and outer object are independent.

- Nested classes can extend any class or can implement any interface. No restrictions.

- All nested classes (except anonymous classes) can be abstract or final.

- Classes can be nested to any depth. Top-level static classes can be nested only within other static top-level classes or interfaces. Deeply nested classes also have access to all variables of the outer-most enclosing class (as well the immediate enclosing class's)

- Member inner classes can be forward referenced. Local inner classes cannot be.

- An inner class variable can shadow an outer class variable. In this case, an outer class variable can be referred as (outerclassname.this.variablename).

- Outer class variables are accessible within the inner class, but they are not inherited. They don't become members of the inner class. This is different from inheritance. (Outer class cannot be referred using 'super', and outer class variables cannot be accessed using 'this')

- An inner class variable can shadow an outer class variable. If the inner class is sub-classed within the same outer class, the variable has to be qualified explicitly in the sub-class. To fully qualify the variable, use classname.this.variablename. If we don't correctly qualify the variable, a compiler error will occur. (Note that this does not happen in multiple levels of inheritance where an upper-most super-class's variable is silently shadowed by the most recent super-class variable or in multiple levels of nested inner classes where an inner-most class's variable silently shadows an outer-most class's variable. Problem comes only when these two hierarchy chains (inheritance and containment) clash.)

- If the inner class is sub-classed outside of the outer class (only possible with top-level nested classes) explicit qualification is not needed (it becomes regular class inheritance)