**CHAPTER 2: OBJECT ORIENTATION**

Here are some of the key points from each certification objective in this chapter.

**Encapsulation, IS-A, HAS-A (Objective 5.1)**

- Encapsulation helps hide implementation behind an interface (or API).

- Encapsulated code has two features:

Instance variables are kept protected (usually with the private modifier).

Getter and setter methods provide access to instance variables.

- IS-A refers to inheritance.

- IS-A is expressed with the keyword extends.

- IS-A, "inherits from," and "is a subtype of" are all equivalent expressions.

- HAS-A means an instance of one class "has a" reference to an instance of another class.

**Inheritance (Objective 5.5)**

- Inheritance is a mechanism that allows a class to be a subclass of a superclass, and thereby inherit variables and methods of the superclass.

- Inheritance is a key concept that underlies IS-A, polymorphism, overriding, overloading, and casting.

- All classes (except class Object), are subclasses of type Object, and therefore they inherit Object's methods.

**Polymorphism (Objective 5.2)**

- Polymorphism means ‘many forms'.

- A reference variable is always of a single, unchangeable type, but it can refer to a subtype object.

- A single object can be referred to by reference variables of many different types— as long as they are the same type or a supertype of the object.

- The reference variable's type (not the object's type), determines which methods can be called!

- Polymorphic method invocations apply only to overridden instance methods.

**Overriding and Overloading (Objectives 1.5 and 5.4)**

- Methods can be overridden or overloaded; constructors can be overloaded but not overridden.

- Abstract methods must be overridden by the first concrete (nonabstract) subclass.

- With respect to the method it overrides, the overriding method:

Must have the same argument list

Must have the same return type, except that as of Java 5, the return type can be a subclass—this is known as a covariant return.

Must not have a more restrictive access modifier

May have a less restrictive access modifier

Must not throw new or broader checked exceptions

May throw fewer or narrower checked exceptions, or any unchecked exception.

- final methods cannot be overridden.

- Only inherited methods may be overridden, and remember that private methods are not inherited.

- A subclass uses super.overriddenMethodName() to call the superclass version of an overridden method.

- Overloading means reusing a method name, but with different arguments.

- Overloaded methods:

Must have different argument lists

May have different return types, if argument lists are also different

May have different access modifiers

May throw different exceptions

- Methods from a superclass can be overloaded in a subclass.

- Polymorphism applies to overriding, not to overloading

- Object type (not the reference variable's type), determines which overridden method is used at runtime.

- Reference type determines which overloaded method will be used at compile time.

**Reference Variable Casting (Objective 5.2)**

- There are two types of reference variable casting: downcasting and upcasting.

- Downcasting: If you have a reference variable that refers to a subtype object, you can assign it to a reference variable of the subtype. You must make an explicit cast to do this, and the result is that you can access the subtype's members with this new reference variable.

Eg:

Object o = getSomeObject(),

String s = (String) o;

- Upcasting: You can assign a reference variable to a supertype reference variable explicitly or implicitly. This is an inherently safe operation because the assignment restricts the access capabilities of the new variable.

**Implementing an Interface (Objective 1.2)**

- When you implement an interface, you are fulfilling its contract.

- You implement an interface by properly and concretely overriding all of the methods defined by the interface.

- A single class can implement many interfaces.

**Return Types (Objective 1.5)**

- Overloaded methods can change return types; overridden methods cannot, except in the case of covariant returns.

- Object reference return types can accept null as a return value.

- An array is a legal return type, both to declare and return as a value.

- For methods with primitive return types, any value that can be implicitly converted to the return type can be returned.

- Nothing can be returned from a void, but you can return nothing. You're allowed to simply say return, in any method with a void return type, to bust out of a method early. But you can't return nothing from a method with a non-void return type.

- Methods with an object reference return type, can return a subtype.

- Methods with an interface return type, can return any implementer.

**Constructors and Instantiation (Objectives 1.6 and 5.4)**

- You cannot create a new object without invoking a constructor.

- Each superclass in an object's inheritance tree will have a constructor called.

- Every class, even an abstract class, has at least one constructor.

- Constructors must have the same name as the class.

- Constructors don't have a return type. If the code you're looking at has a return type, it's a method with the same name as the class, and a constructor.

- Typical constructor execution occurs as follows:

- The constructor calls its superclass constructor, which calls its superclass constructor, and so on all the way up to the Object constructor.

- The Object constructor executes and then returns to the calling constructor, which runs to completion and then returns to its calling constructor, and so on back down to the completion of the constructor of the actual instance being created.

- Constructors can use any access modifier (even private!).

- The compiler will create a default constructor if you don't create any constructors in your class.

- The default constructor is a no-arg constructor with a no-arg call to super().

- The first statement of every constructor must be a call to either this(), (an overloaded constructor), or super().

- The compiler will add a call to super() if you do not, unless you have already put in a call to this().

- Instance members are accessible only after the super constructor runs.

- Abstract classes have constructors that are called when a concrete subclass is instantiated.

- Interfaces do not have constructors.

- If your superclass does not have a no-arg constructor, you must create a constructor and insert a call to super() with arguments matching those of the superclass constructor.

- Constructors are never inherited, thus they cannot be overridden.

- A constructor can be directly invoked only by another constructor (using a call to super() or this()).

- Issues with calls to this():

- May appear only as the first statement in a constructor.

- The argument list determines which overloaded constructor is called.

- Constructors can call constructors can call constructors, and so on, but sooner or later one of them better call super() or the stack will explode.

- Calls to this() and super() cannot be in the same constructor. You can have one or the other, but never both.

**Statics (Objective 1.3)**

- Use static methods to implement behaviors that are not affected by the state of any instances.

- Use static variables to hold data that is class specific as opposed to instance specific—there will be only one copy of a static variable.

- All static members belong to the class, not to any instance.

- A static method can't access an instance variable directly.

- Use the dot operator to access static members, but remember that using a reference variable with the dot operator is really a syntax trick, and the compiler will substitute the class name for the reference variable, for instance: d.doStuff(); becomes: Dog.doStuff();

- static methods can't be overridden, but they can be redefined.

**Coupling and Cohesion (Objective 5.1)**

- Coupling refers to the degree to which one class knows about or uses members of another class.

- Loose coupling is the desirable state of having classes that are well encapsulated, minimize references to each other, and limit the breadth of API usage.

- Tight coupling is the undesirable state of having classes that break the rules of loose coupling.

- Cohesion refers to the degree in which a class has a single, well-defined role or responsibility.

- High cohesion is the desirable state of a class whose members support a single, well-focused role or responsibility.

- Low cohesion is the undesirable state of a class whose members support multiple, unfocused roles or responsibilities.