**CHAPTER 1: DECLARATIONS AND ACCESS CONTROL**

Remember that in this chapter, when we talk about classes, we're referring to non-inner classes, or top-level classes. We'll devote all of Chapter 8 to inner classes.

**Identifiers (Objective 1.3)**

- Identifiers can begin with a letter, an underscore, or a currency character.

- After the first character, identifiers can also include digits.

- Identifiers can be of any length.

- JavaBeans methods must be named using camelCase, and depending on the method's purpose, must start with set, get, is, add, or remove.

**Declaration Rules (Objective 1.1)**

- A source code file can have only one public class.

- If the source file contains a public class, the filename must match the public class name.

- If there is no package statement, import statements must be the first (noncomment) statements in the source file.

- package and import statements apply to all classes in the file.

- A file can have more than one nonpublic class.

- Files with no public classes have no naming restrictions.

**Class Access Modifiers (Objective 1.1)**

- There are three access modifiers: public, protected, and private.

- There are four access levels: public, protected, default, and private.

- Classes can have only public or default access.

- A class with default access can be seen only by classes within the same package.

- A class with public access can be seen by all classes from all packages.

- Class visibility revolves around whether code in one class can

Create an instance of another class

Extend (or subclass), another class

Access methods and variables of another class

**Class Modifiers (Nonaccess) (Objective 1.2)**

- Classes can also be modified with final, abstract, or strictfp.

- A class cannot be both final and abstract.

- A final class cannot be subclassed.

Eg :

public final class A{

public void b() {...}

}

- An abstract class cannot be instantiated.

Eg:

abstract class A {

public abstract void absMethod();

public void nonAbsMethod() {...}

}

- A single abstract method in a class means the whole class must be abstract.

- An abstract class can have both abstract and nonabstract methods.

- The first concrete class to extend an abstract class must implement all of its abstract methods.

- An abstract class with no abstract method is legal.

- Methods marked with abstract end with semicolon and not curly braces.

**Interface Implementation (Objective 1.2)**

- Interfaces are contracts for what a class can do, but they say nothing about the way in which the class must do it.

- Interfaces can be implemented by any class, from any inheritance tree.

- An interface is like a 100-percent abstract class, and is implicitly abstract whether you type the abstract modifier in the declaration or not.

- An interface can have only abstract methods, no concrete methods allowed.

- Interface methods are by default public and abstract .explicit declaration of these modifiers is optional.

Eg:

interface A1 {

void a();

void b(int I);

}

- Interfaces can have constants, which are always implicitly public, static, and final.

Eg:

Interface A2 {

int AA=21;

void go();

}

- Interface constant declarations of public, static, and final are optional in any combination.

- A legal nonabstract implementing class has the following properties:

It provides concrete implementations for the interface's methods.

It must follow all legal override rules for the methods it implements.

It must not declare any new checked exceptions for an implementation method.

It must not declare any checked exceptions that are broader than the exceptions declared in the interface method.

It may declare runtime exceptions on any interface method implementation regardless of the interface declaration.

It must maintain the exact signature (allowing for covariant returns) and return type of the methods it implements (but does not have to declare the exceptions of the interface).

- A class implementing an interface can itself be abstract.

- An abstract implementing class does not have to implement the interface methods (but the first concrete subclass must).

- A class can extend only one class (no multiple inheritance), but it can implement many interfaces.

- Interfaces can extend one or more other interfaces.

Eg :

interface A3 extends A1, A2 {

void c();

}

- Interfaces cannot extend a class, or implement a class or interface.

- When taking the exam, verify that interface and class declarations are legal before verifying other code logic.

**Member Access Modifiers (Objectives 1.3 and 1.4)**

- Methods and instance (nonlocal) variables are known as "members".

- Members can use all four access levels: public, protected, default, private.

- Member access comes in two forms:

- Code in one class can access a member of another class.

- A subclass can inherit a member of its superclass.

- If a class cannot be accessed, its members cannot be accessed.

- Determine class visibility before determining member visibility.

- public members can be accessed by all other classes, even in other packages.

- If a superclass member is public, the subclass inherits it—regardless of package.

- Members accessed without the dot operator (.) must belong to the same class.

- this. always refers to the currently executing object.

- this.aMethod() is the same as just invoking aMethod().

- private members can be accessed only by code in the same class.

- private members are not visible to subclasses, so private members cannot be inherited.

- Default and protected members differ only when subclasses are involved:

- Default members can be accessed only by classes in the same package.

- protected members can be accessed by other classes in the same package, plus subclasses regardless of package.

- protected = package plus kids (kids meaning subclasses).

- For subclasses outside the package, the protected member can be accessed only through inheritance; a subclass outside the package cannot access a protected member by using a reference to a superclass instance (in other words, inheritance is the only mechanism for a subclass outside the package to access a protected member of its superclass).

- A protected member inherited by a subclass from another package is not accessible to any other class in the subclass package, except for the subclass' own subclasses.

**Local Variables (Objective 1.3)**

- Local (method, automatic, or stack) variable declarations cannot have access modifiers.

- final is the only modifier available to local variables.

- Local variables don't get default values, so they must be initialized before use.

**Other Modifiers—Members (Objective 1.3)**

- final methods cannot be overridden in a subclass.

- abstract methods are declared, with a signature, a return type, and an optional throws clause, but are not implemented.

- abstract methods end in a semicolon—no curly braces.

- Three ways to spot a non-abstract method:

- The method is not marked abstract.

- The method has curly braces.

- The method has code between the curly braces.

- The first nonabstract (concrete) class to extend an abstract class must implement all of the abstract class' abstract methods.

- The synchronized modifier applies only to methods and code blocks.

- synchronized methods can have any access control and can also be marked final.

- abstract methods must be implemented by a subclass, so they must be inheritable. For that reason:

- abstract methods cannot be private.

- abstract methods cannot be final.

- The native modifier applies only to methods.

- The strictfp modifier applies only to classes and methods.

**Methods with var-args (Objective 1.4)**

- As of Java 5, methods can declare a parameter that accepts from zero to many arguments, a so-called var-arg method.

- A var-arg parameter is declared with the syntax type... name; for instance: doStuff(int... x) { }

- A var-arg method can have only one var-arg parameter.

- In methods with normal parameters and a var-arg, the var-arg must come last.

**Variable Declarations (Objective 1.3)**

- Instance variables can

- Have any access control

- Be marked final or transient

- Instance variables can't be abstract, synchronized, native, or strictfp.

- It is legal to declare a local variable with the same name as an instance variable; this is called "shadowing."

- final variables have the following properties:

final variables cannot be reinitialized once assigned a value.

final reference variables cannot refer to a different object once the object has been assigned to the final variable.

final reference variables must be initialized before the constructor completes.

- There is no such thing as a final object. An object reference marked final does not mean the object itself is immutable.

- The transient modifier applies only to instance variables.

- The volatile modifier applies only to instance variables.

**Array Declarations (Objective 1.3)**

- Arrays can hold primitives or objects, but the array itself is always an object.

- When you declare an array, the brackets can be to the left or right of the variable name.

- It is never legal to include the size of an array in the declaration.

- An array of objects can hold any object that passes the IS-A (or instanceof) test for the declared type of the array. For example, if Horse extends Animal, then a Horse object can go into an Animal array.

**Static Variables and Methods (Objective 1.4)**

- They are not tied to any particular instance of a class.

- No classes instances are needed in order to use static members of the class.

- There is only one copy of a static variable / class and all instances share it.

- static methods do not have direct access to non-static members.

**Enums (Objective 1.3)**

- An enum specifies a list of constant values that can be assigned to a particular type.

- An enum is NOT a String or an int; an enum constant's type is the enum type. For example, WINTER, SPRING, SUMMER, and FALL are of the enum type Season.

- An enum can be declared outside or inside a class, but NOT in a method.

- An enum declared outside a class must NOT be marked static, final, abstract, protected, or private.

- Enums can contain constructors, methods, variables, and constant class bodies.

- enum constants can send arguments to the enum constructor, using the syntax BIG(8), where the int literal 8 is passed to the enum constructor.

- enum constructors can have arguments, and can be overloaded.

- enum constructors can NEVER be invoked directly in code. They are always called automatically when an enum is initialized.

- The semicolon at the end of an enum declaration is optional. These are legal: enum Foo { ONE, TWO, THREE}

enum Foo { ONE, TWO, THREE};