Class Access Modifiers

- 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.
- 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
- 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 Modifiers (nonaccess)

- Classes can also be modified with final, abstract, or strictfp.
- A class cannot be both final and abstract.
- A final class cannot be subclassed.
- An abstract class cannot be instantiated.
- 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
- abstract methods.

Member Access Modifiers

- 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 different 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 in when subclasses are involved:

- Default members can be accessed only by other 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 an instance of the superclass (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, in practice, private to all other classes (in other words, no other classes from the subclass' package or any other package will have access to the protected member from the subclass).

Local Variables

- Local (method, automatic, 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

- Final methods cannot be overridden in a subclass.
- Abstract methods have been declared, with a signature and return type, but have not been implemented.
- Abstract methods end in a semicolon—no curly braces.
- Three ways to spot a nonabstract 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.

- 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 synchronized modifier applies only to methods.
- Synchronized methods can have any access control and can also be marked final.
- Synchronized methods cannot be abstract.
- The native modifier applies only to methods.
- The strictfp modifier applies only to classes and methods.
- Instance variables can
- Have any access control
- Be marked final or transient
- Instance variables cannot be declared 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.

Static variables and methods

- They are not tied to any particular instance of a class.
- An instance of a class does not need to exist in order to use static members of the class.
- There is only one copy of a static variable per class and all instances share it.
- Static variables get the same default values as instance variables.
- A static method (such as main()) cannot access a nonstatic (instance) variable.
- Static members are accessed using the class name: ClassName.theStaticMethodName()

- Static members can also be accessed using an instance reference variable, someObj.theStaticMethodName(), but that’s just a syntax trick; the static method won’t know anything about the instance referred to by the variable used to invoke the method. The compiler uses the class type of the reference variable to determine which static method to invoke.

- Static methods cannot be overridden, although they can be redeclared/ redefined by a subclass. So although static methods can sometimes appear to be overridden, polymorphism will not apply.

Declaration Rules

- A source code file can have only one public class.
- If the source file contains a public class, the file name should match the public class name.
- A file can have only one package statement, but can have multiple import statements.
- The package statement (if any) must be the first line in a source file.
- The import statements (if any) must come after the package and before the class declaration.
- If there is no package statement, import statements must be the first 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.
- In a file, classes can be listed in any order (there is no forward referencing problem).
- Import statements only provide a typing shortcut to a class’ fully qualified name.
- Import statements cause no performance hits and do not increase the size of your code.
- If you use a class from a different package, but do not import the class, you must use the fully qualified name of the class everywhere the class is used in code.
- Import statements can coexist with fully qualified class names in a source file.
- Imports ending in ‘.*;’ are importing all classes within a package.
- Imports ending in ‘;’ are importing a single class.
- You must use fully qualified names when you have different classes from different packages, with the same class name; an import statement will not be explicit enough.

Encapsulation, IS-A, HAS-A

- The goal of encapsulation is to 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,” “is derived 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.

Overriding and Overloading

- 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

- 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.

- A subclass uses super.overriddenMethodName to call the superclass version of an overridden method.
- Overloading means reusing the same method name, but with different arguments.
- Overloaded methods Must have different argument lists
- May have different return types, as long as the 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 determines which overridden method is used at runtime.
- Reference type determines which overloaded method will be used a compile time.

Instantiation and Constructors

- Objects are constructed:
- 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 abstract classes, has at least one constructor.
- Constructors must have the same name as the class.
- Constructors do not have a return type. If there *is* a return type, then it is simply a method with the same name as the class, and not a constructor.
- 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 methods and variables are only accessible *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.

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

Return Types

- Overloaded methods can change return types; overridden methods cannot.

- 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.

- For methods with an object reference return type, a subclass of that type can be returned.

- For methods with an interface return type, any implementer of that interface can be returned.