Objectives 1

Declarations, Initialization and Scoping

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.
- A file can have only one package statement, but multiple imports.
- The package statement (if any) must be the first (non-comment) line in a source file.
- The import statements (if any) must come after the package and before the class declaration.

Declaration Rules (Objective 1.1) [contd.]

- If there is no package statement, import statements must be the first (non-comment) statements in the source file.
- package and import statements apply to all classes in the file.
- A file can have more than one non-public 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.

Class Access Modifiers (Objective 1.1) [contd.]

- 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 (Non-Access) (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 sub-classed.
- 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 non-abstract methods.
- The first concrete class to extend an abstract class must implement all of its abstract methods.

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 Implementation (Objective 1.2) [contd.]

- Interface methods are by default public and abstract—explicit declaration of these modifiers is optional.
- Interfaces can have constants, which are always implicitly public, static, and final.
- Interface constant declarations of public, static, and final are optional in any combination.

Interface Implementation (Objective 1.2) [contd.]

- A legal non-abstract 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).

Interface Implementation (Objective 1.2) [contd.]

- 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.

Interface Implementation (Objective 1.2) [contd.]

- Interfaces can extend one or more other interfaces.
- 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.

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.

Stack and Heap

- Local variables (method variables) live on the stack.
- Objects and their instance variables live on the heap.

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.

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.

Variable Declarations (Objective 1.3) [contd.]

- 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.

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.

Scope (Objectives 1.3 and 7.6)

- Scope refers to the lifetime of a variable.
- There are four basic scopes:
  - Static variables live basically as long as their class lives.
  - Instance variables live as long as their object lives.
  - Local variables live as long as their method is on the stack; however, if their method invokes another method, they are temporarily unavailable.
  - Block variables (e.g.., in a for or an if) live until the block completes.

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.

Other Modifiers—Members (Objective 1.3) [cont.]

- The first non-abstract (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.

Other Modifiers—Members (Objective 1.3) [cont.]

- 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.

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.

Statics (Objective 1.3) [contd.]

- 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.

Initialization Blocks (Objectives 1.3 and 7.6)

- Static initialization blocks run once, when the class is first loaded.
- Instance initialization blocks run every time a new instance is created. They run after all super-constructors and before the constructor's code has run.
- If multiple init blocks exist in a class, they follow the rules stated above, AND they run in the order in which they appear in the source file.

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.

Using a Variable or Array Element That Is Uninitialized and Unassigned (Objectives 1.3 and 7.6)

- When an array of objects is instantiated, objects within the array are not instantiated automatically, but all the references get the default value of null.
- When an array of primitives is instantiated, elements get default values.
- Instance variables are always initialized with a default value.
- Local/automatic/method variables are never given a default value. If you attempt to use one before initializing it, you'll get a compiler error.

Array Declaration, Construction, and Initialization (Objectives 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 left or right of the name.
- It is never legal to include the size of an array in the declaration.
- You must include the size of an array when you construct it (using new) unless you are creating an anonymous array.

Array Declaration, Construction, and Initialization (Objectives 1.3) [contd.]

- Elements in an array of objects are not automatically created, although primitive array elements are given default values.
- You'll get a NullPointerException if you try to use an array element in an object array, if that element does not refer to a real object.
- Arrays are indexed beginning with zero.

Array Declaration, Construction, and Initialization (Objectives 1.3) [contd.]

- An ArrayIndexOutOfBoundsException occurs if you use a bad index value.
- Arrays have a length variable whose value is the number of array elements.
- The last index you can access is always one less than the length of the array.
- Multidimensional arrays are just arrays of arrays.

Array Declaration, Construction, and Initialization (Objectives 1.3) [contd.]

- The dimensions in a multidimensional array can have different lengths.
- An array of primitives can accept any value that can be promoted implicitly to the array's declared type;. e.g., a byte variable can go in an int array.
- 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.

Array Declaration, Construction, and Initialization (Objectives 1.3) [contd.]

- If you assign an array to a previously declared array reference, the array you're assigning must be the same dimension as the reference you're assigning it to.
- You can assign an array of one type to a previously declared array reference of one of its supertypes. For example, a Honda array can be assigned to an array declared as type Car (assuming Honda extends Car).

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.

Enums (Objective 1.3)

- 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};
```

Literals and Primitive Casting (Objective 1.3)

- Integer literals can be decimal, octal (e.g. 013), or hexadecimal (e.g. 0x3d).
- Literals for longs end in L or l.
- Float literals end in F or f, double literals end in a digit or D or d.
- The boolean literals are true and false.
- Literals for chars are a single character inside single quotes: 'd'.

Basic Assignments (Objectives 1.3 and 7.6)

- Literal integers are implicitly ints.
- Integer expressions always result in an int-sized result, never smaller.
- Floating-point numbers are implicitly doubles (64 bits).
- Narrowing a primitive truncates the *high order* bits.
- Compound assignments (e.g. +=), perform an automatic cast.

Basic Assignments (Objectives 1.3 and 7.6)

- A reference variable holds the bits that are used to refer to an object.
- Reference variables can refer to subclasses of the declared type but not to superclasses.
- When creating a new object, e.g., Button b = new Button();, three things happen:
  - Make a reference variable named b, of type Button
  - Create a new Button object
  - Assign the Button object to the reference variable b

Member Access Modifiers (Objectives 1.3 and 1.4)

- Methods and instance (non-local) 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.

Member Access Modifiers (Objectives 1.3 and 1.4) [contd.]

- 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.

Member Access Modifiers (Objectives 1.3 and 1.4) [contd.]

- 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.

Member Access Modifiers (Objectives 1.3 and 1.4) [contd.]

- 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.

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.

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.

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.

Return Types (Objective 1.5)

- 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.

Advanced Overloading (Objectives 1.5 and 5.4)

- Primitive widening uses the "smallest" method argument possible.
- Used individually, boxing and var-args are compatible with overloading.
- You CANNOT widen from one wrapper type to another. (IS-A fails.)
- You CANNOT widen and then box. (An int can't become a Long.)
- You can box and then widen. (An int can become an Object, via an Integer.)
- You can combine var-args with either widening or boxing.

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 and Instantiation (Objectives 1.6 and 5.4) [contd.]

- 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!).

Constructors and Instantiation (Objectives 1.6 and 5.4) [contd.]

- 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().

Constructors and Instantiation (Objectives 1.6 and 5.4) [contd.]

- 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 and Instantiation (Objectives 1.6 and 5.4) [contd.]

- 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.