**CHAPTER 3: ASSIGNMENTS**

**Stack and Heap**

- Local variables (method variables) live on the stack.

- Objects and their instance variables live on the heap.

**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'.

**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.

**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.

- 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

**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.

**Passing Variables into Methods (Objective 7.3)**

- Methods can take primitives and/or object references as arguments.

- Method arguments are always copies.

- Method arguments are never actual objects (they can be references to objects).

- A primitive argument is an unattached copy of the original primitive.

- A reference argument is another copy of a reference to the original object.

- Shadowing occurs when two variables with different scopes share the same name. This leads to hard-to-find bugs, and hard-to-answer exam questions.

**Array Declaration, Construction, and Initialization (Obj. 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.

- 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.

- 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.

- 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.

- 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).

**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.

**Using Wrappers (Objective 3.1)**

- The wrapper classes correlate to the primitive types.

- Wrappers have two main functions:

- To wrap primitives so that they can be handled like objects

- To provide utility methods for primitives (usually conversions)

- The three most important method families are

- xxxValue() Takes no arguments, returns a primitive. Its an instance method.

Eg

Integer ii=10;

int i= ii.intValue();

- parseXxx() Takes a String, returns a primitive, throws NFE

Eg : int x = Integer.parseInt("1234");

- valueOf() Takes a String, returns a wrapped object, throws NFE. This is a static method.

- Wrapper constructors can take a String or a primitive, except for Character, which can only take a char.

- Radix refers to bases (typically) other than 10; octal is radix = 8, hex = 16.

**Boxing (Objective 3.1)**

- As of Java 5, boxing allows you to convert primitives to wrappers or to convert wrappers to primitives automatically.

- Using == with wrappers is tricky; wrappers with the same small values (typically lower than 127), will be ==, larger values will not be ==.

**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.

- (wide & var-arg), (box & var-arg), (box & wide)

-If exact match not found jvm will try to match the method in the following order

1) widening between primitives, (2)Boxing, (3) var-arg

**Garbage Collection (Objective 7.4)**

- In Java, garbage collection (GC) provides automated memory management.

- The purpose of GC is to delete objects that can't be reached.

- Only the JVM decides when to run the GC, you can only suggest it.

- Objects must be considered eligible before they can be garbage collected.

- An object is eligible when no live thread can reach it.

- To reach an object, you must have a live, reachable reference to that object.

- Java applications can run out of memory.

- Islands of objects can be GCed, even though they refer to each other.

- Request garbage collection with System.gc(); (recommended).

- Class Object has a finalize() method. “protected void finalize() throws Throwable{…}”

- The finalize() method is guaranteed to run once and only once before the garbage collector deletes an object.

- The garbage collector makes no guarantees, finalize() may never run.

- You can uneligibilize an object for GC from within finalize().