Using the java.lang.String Class

- String objects are immutable, and String reference variables are not.
- If you create a new String without assigning it, it will be lost to your program.
- If you redirect a String reference to a new String, the old String can be lost.
- String methods use zero-based indexes, except for the second argument of substring().
- The String class is final—its methods can't be overridden.
- When a String literal is encountered by the VM, it is added to the pool.
- Strings have a method named length(), arrays have an attribute named *length*.
- StringBuffers are mutable—they can change without creating a new object.
- StringBuffer methods act on the invoking object, but objects can chang without an explicit assignment in the statement.
- StringBuffer equals() is not overridden; it doesn't compare values.
- In all sections, remember that *chained* methods are evaluated from left to right.

Using the java.lang.Math Class (Exam Objective 8.1)

- The abs() method is overloaded to take an *int*, a *long*, a *float*, or a *double*.
- The abs() method can return a negative if the argument is the minimum *int* or *long* value equal to the value of Integer.MIN_VALUE or Long.MIN_VALUE, respectively.
- The max() method is overloaded to take *int*, *long*, *float*, or *double* arguments.
- The min() method is overloaded to take *int*, *long*, *float*, or *double* arguments.
- The random() method returns a *double* greater than or equal to 0.0 and less than 1.0.
- The random() does not take any arguments.
- The methods ceil(), floor(), and round() all return integer equivalent floating-point numbers, ceil() and floor() return *doubles*, round() returns a *float* if it was passed an *int*, or it returns a *double* if it was passed a *long*.

- The round() method is overloaded to take a *float* or a *double*.
- The methods sin(), cos(), and tan() take a *double* angle in radians.
- The method sqrt() can return NaN if the argument is NaN or less than zero.
- Floating-point numbers can be divided by 0.0 without error; the result is either positive or negative infinity.
- NaN is not equal to anything, not even itself.

Using Wrappers

- 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)
- Other than Character and Integer, wrapper class names are the primitive's name, capitalized.
- Wrapper constructors can take a String or a primitive, except for Character, which can only take a *char*.
- A Boolean object can't be used like a *boolean* primitive.
- The three most important method families are
- xxxValue() Takes no arguments, returns a primitive
- parseXxx() Takes a String, returns a primitive, is static, throws NFE
- valueOf() Takes a String, returns a wrapped object, is static, throws NFE
- Radix refers to bases (typically) other than 10; binary is radix 2, octal = 8, hex = 16.

Using equals()

- Use == to compare primitive variables.
- Use == to determine if two reference variables refer to the *same object*.
- == compares bit patterns, either primitive bits or reference bits.
- Use equals() to determine if two objects are *meaningfully equivalent*.
- The String and Wrapper classes override equals() to check for values.
- The StringBuffer class equals() is *not* overridden; it uses == under the covers.
- The compiler will not allow == if the classes are not in the same hierarchy.
- Wrappers won't pass equals() if they are in different classes.

Overriding hashCode() and equals()

- The critical methods in class Object are equals(), finalize(), hashCode(), and toString().
- equals(), hashCode(), and toString() are public (finalize()is protected).
- Fun facts about toString():
    - Override toString() so that System.out.println() or other methods can see something useful.
    - Override toString() to return the essence of your object's state.
- Use == to determine if two reference variables refer to the same object.
- Use equals() to determine if two objects are *meaningfully equivalent*.
- If you don't override equals(), your objects *won't* be useful hashtable/hashmap keys.
- If you don't override equals(), two different objects can't be considered *the same*.
- Strings and wrappers override equals() and make good hashtable/hashmap keys.

- When overriding equals(), use the instanceof operator to be sure you're evaluating an appropriate class.
- When overriding equals(), compare the objects' *significant* attributes.
- Highlights of the equals() contract:
    - *Reflexive:* x.equals(x) is true.
    - *Symmetric:* If x.equals(y) is true, then y.equals(x) must be true.
    - *Transitive:* If x.equals(y) is true, and y.equals(z) is true, then z.equals(x) is true.
    - *Consistent:* Multiple calls to x.equals(y) will return the same result.
    - *Null:* If x is not null, then x.equals(null) is false.
- If x.equals(y) is true, then x.hashCode() == y.hashCode()must be true.
- If you override equals(), override hashCode().
- Classes HashMap, Hashtable, LinkedHashMap, and LinkedHashSet use hashing.
- A *legal* hashCode() override compiles and runs.
- An *appropriate* hashCode() override sticks to the contract.
- An *efficient* hashCode() override distributes keys randomly across a wide range of buckets.
- To reiterate: if two objects are equal, their hashcodes must be equal.
- It's *legal* for a hashCode() method to return the same value for all instances (although in practice it's very inefficient).
- Highlights of the hashCode() contract:
    - *Consistent:* Multiple calls to x.hashCode() return the same integer.
    - If x.equals(y) is true, then x.hashCode() == y.hashCode()must be true.

- If x.equals(y) is false, then x.hashCode() == y.hashCode() can be either true or false, but false will tend to create better efficiency.

- Transient variables aren't appropriate for equals() and hashCode().