Important SCJP 6 Exam Notes v0.15

Written by Jonathan Giles, http://www.JonathanGiles.net, February 2009

Basics

- Remember that by default floating point numbers in Java are double, so when declaring a float, you must add an 'f' after it, otherwise compilation fails. Similarly, integer numbers are by default integers. When passing ints into methods that require data types such as short, it is necessary to cast.
- Whilst Integer can be boxed to int, Integer[] **cannot** be boxed into int[].
- Interface methods are implicitly public, regardless of the declaration. Therefore, classes implementing an interface must make all implemented methods public.
- Remember that when dealing with anonymous inner classes, any method local variables that are accessed must be declared final.
- Static methods cannot be overridden to be non-static, and vice versa. This will result in a compilation error.
- Any code after a 'throw' statement is unreachable, and so will result in a compilation error.
- Two objects that are equal (as determined by equals()) must return the same hashcode. The reverse however is not true – two objects that are not equal can return equal hashcodes.
- Static initialization blocks are called once per class. Instance initialization blocks are called once per constructor call – they are call directly after all super-constructors are called, and before the constructor runs.
- The finalize() method is only ever run by the **JVM** zero or once. If an object is saved from being GC'ed through the finalize() method, the next time the object can be GC'ed, finalize() won't be called. This does not count calls to finalize that weren't invoked through the GC process.
- The finalize() method does not implicitly call the super.finalize() method. This must be done explicitly if desired.
- It is **illegal** to compare a primitive type to an array type.
- It is **legal** for the left hand side of the instanceof operator to be null.
- It is **legal** to pass an int into a method which expects a float – Java will convert this automatically. Similar for long into double. In other words, Java will cast automatically into any type that is wider, even if it is floating point rather than integer.
- When concatenating a String, remember that **concatenation runs from left to right**. Therefore, if either operand is a String, the operands are concatenated. If both operands are numbers, they are added together. For example:
  - System.out.println(" " + 2 + 5 + " "); prints " 25 ".
  - System.out.println(2 + 5 + " "); prints "7 ".
- When evaluating boolean expressions, an assignment instruction returns the value of the assignment. Remember that subsequent references to that assigned Boolean need to use the new value, not the old one!
- Be wary of code that doesn't compile because it is unreachable. This is possible if there is a return statement and code after it, or alternatively if there is an exception that is thrown – all code after the thrown exception won't be executed.
- Remember that method-local variables are not initialized to a default value, so before being accessed, they must be initialized. Failure to do so leads to compilation errors.
- The instanceof operator must test an object in the same inheritance tree as the given type. For example, if s is a Short:
  - (s instanceof Number) is valid

    - (s instanceof String) is **NOT valid**, as Short isn't in the same inheritance tree as String.
- When comparing using == remember that the types have to be compatible operands, for example, it is not allowed to compare an enum type to a String. In the examples below, c is an enum of type Colors:

`if (c == Colors.GREEN) {`
`        System.out.print("green");`
`}`
	This is fine, comparing compatible (and equivalent) types.

`if (Colors.RED.equals(c)) {`
`        System.out.print("red");`
`}`
	This is fine, the equals method will compare any Object instance.

`if (c == "YELLOW") {`
`        System.out.print("yellow");`
`}`
	This is illegal: trying to test equality between incompatible types: Colors enum is incompatible with String.

`if (c.equals("BLUE")) {`
`        System.out.print("blue");`
`}`
	This is fine, the equals method will compare any Object instance.

`if (3 == "hi") {    }`	**This is illegal**: cannot compare int to String.

- Switch statements do not need to have a constant value in them, but the value given to a switch statement must be able to be cast to an int.
- Case statements on the other hand must be compile time constants. This means that they must be constants or final variables. This means it is also possible to also do calculations in a case statement (e.g. 'case 12+32:' or 'case CONST + 12'). Because of this, it is not legal to use any type of wrapper type, even if the variable is constant.
- Whilst a switch can take anything that can be cast to int, the case statements can only be within the range of the actual type given to the switch statement. Therefore, byte case values can only be from 0 to 127. Any value outside this range is a compile time error.
- It is illegal to have multiple case statements with the same value.
- Instance variables and objects are stored on the heap. Local variables are stored in the stack.
- When writing the basic for loop, it is allowable to declare multiple variables, but they must all be of the same type, and the type is only declared once, for example:
    - for (int i = 0, y = 10; I < 10; i++) { … }

Coupling and Cohesion

- Coupling is the degree to which one class knows about the other. **Loose coupling is when interaction is through an interface, whereas tight coupling is when interaction is between implementations.**
- Cohesion is all about how a single class is defined: it is used to indicate the degree to which a class has a single, well-focused purpose. The key benefits of high cohesion are that classes are easier to maintain and tend to be more reusable.

Primitive Wrappers

- When a method is overloaded, and it is called with an argument that could either be boxed or widened, the Java compiler will choose to **widen the argument over boxing it**. Similarly, **widening beats varargs**.
- Remember that Bytes won't widen to Integers, etc; only primitive types will widen automatically.
- The compiler **cannot** widen and autobox a parameter. The compiler **can** autobox and then widen a parameter (into an Object). For example:

`class widenAndBox{`
`    static void go(Long x) { …. }`
 	
`    public static void main(String[] a) {`
	This does not work. Java cannot widen this byte into a long and then box it into a Long.

`        byte b = 5;`
`        go(b);`
`    }`
`}`	

`class BoxAndWiden {`
`    static void go(Object o) {`
`        Byte b2 = (Byte) o;`
`    }`
 	
`    Public static void main(String[] a) {`
`        byte b = 5;`
`        go(b);`
`    }`
`}`
	This will work. The byte value is boxed into a Byte object, and then passed as an Object to go().

`class widenAndBox {`
`    static void go(final Integer x) {`
`       System.out.println("Integer");`
`    }`
 	
`    static void go(final long... x) {`
`       System.out.println("byte...  " + x.length);`
`    }`
 	
`    public static void main(final String[] a) {`
`       final byte b = 5;`
`       go(b);`
`        final int i = 5;`
`       go(i);`
`    }`
`}`
	Given the option of the byte being widened and boxed into an Integer or being widened and passed as a vararg into the long… method, it chooses the long option.
	
	Given the option of an integer being autoboxed into an Integer or widened and passed as a long, the JVM chooses to box it into an Integer.

`static void go(final long... x) {`
`    System.out.println("byte...  " + x.length);`
`}`
 	
`static void go(final Number n1, final Number n2) {`
`    System.out.println("Number, Number");`
`}`
 	
`public static void main(final String[] a) {`
`    final byte b1 = 5;`
`    final byte b2 = 7;`
`    go(b2, b2);`
`}`
	Given the option of either widening the bytes into longs, or boxing into Byte and passing as Number (which is superclass of Byte), the JVM chooses the second option.
	
	It is important to know that varargs methods are always chosen last.

- Three key methods:
    - **valueOf(String):** Takes a string and returns a **wrapper**.
    - **parseXxx(String):** Takes a string and returns a **primitive**.
    - **xxxValue():** Takes the current wrapper and returns its **primitive** value.

Garbage Collection

- Be careful when dealing with garbage collection questions when there is an ‘island of references’. For example:

`Class Lost {`
`    Lost e;`
`    Public static void main(String[] a) {`
`        Lost e1 = new Lost();`
`        Lost e2 = new Lost();`
`        Lost e3 = new Lost();`
`        e3.e = e2;`
`        e1.e = e3;`
`        e2.e = e1;`
`        e3 = null;`
`        e2 = null;`
`        e1 = null;`
	Even when e3 and e2 are set to be null, it is not possible to do any garbage collection, even though it feels like it should be possible to garbage collect e2.
	
	This is because e2 is still accessible from e1, indirectly. E1.e refers to the original e3, and e3.e still refers to the original e2.
	
	Therefore, no garbage collection is possible until all three objects are set to null, and they

`    }`
`}`
	become an island of references ready for GC.

Formatting

- When formatting using printf/format, be sure that the format string has the same number of statements as the output string. For example, the code below only has five Boolean statements, but six parameters. This is valid, but only five Booleans will be printed:
    - System.out.printf("%b %b %b %b %b", s1, s2, s3, b1, b2, s4);
- The important format characters include:

%b, %B	If the argument *arg* is `null`, then the result is "`false`". If *arg* is a `boolean` or `Boolean`, then the result is the string returned by `String.valueOf()`. Otherwise, the result is "true".
%s, %S	If the argument *arg* is `null`, then the result is "`null`". If *arg* implements `Formattable`, then `arg.formatTo` is invoked. Otherwise, the result is obtained by invoking `arg.toString()`.
%c, %c	The result is a Unicode character
%d	The result is formatted as a decimal integer
%f	The result is formatted as a floating-point number

- In most cases, if the format character does not match the parameter type, an exception will be thrown. Examples include:

`System.out.printf("%b", 123)`	This is valid – Boolean 'true' will be printed.
`System.out.printf("%c", "s")`	This is **invalid** – a character is expected, but a string is given.
`System.out.printf("%d", 123)`	This is valid.
`System.out.printf("%f", 123)`	This is **invalid**; a decimal value cannot be formatted as a float.
`System.out.printf("%d", 123.45)`	This is **invalid**; a float cannot be formatted as a decimal.
`System.out.printf("%f", 123.45)`	This is valid.
`System.out.printf("%s", new Long("123"))`	This is valid, as the toString() method can be called on the Long object.
`System.out.printf("%s", 123L)`	This is valid – the long will be printed as a String.

- Be careful when the number of arguments is less than the number suggested in the format string – an exception will occur unless you specifically state the associations. For example:

`System.out.printf(`
`"Right now it is %tr on " + "%<tA, %<tB`
`%<te, %<tY.%n", Calendar.getInstance());`
	Good – each '<' says to use the previous argument. There is only one, and they all use it.

`Calendar cal = Calendar.getInstance();`
`System.out.printf(`
`"Right now it is %tr on " + "%tA, %tB`
`%<te, %<tY.%n",cal);`
	Bad – the second and third format string arguments are missing, and they no longer refer to the previous argument. This will lead to a runtime exception.

`Calendar cal = Calendar.getInstance();`
`System.out.printf(`
`"Right now it is %tr on " + "%tA, %tB`
`%<te, %<tY.%n",cal, cal, cal);`
	Good – the cal variable has been passed as an argument twice more, which satisfies the format string requirements.

- There are four DateFormat styles, and DEFAULT is equal to MEDIUM:
    - SHORT is completely numeric, such as 12.13.52 or 3:30pm
    - MEDIUM is longer, such as Jan 12, 1952
    - LONG is longer, such as January 12, 1952 or 3:30:32pm
    - FULL is pretty completely specified, such as Tuesday, April 12, 1952 AD or 3:30:42pm PST.

- DateFormat has three important formatters. They all return DateFormat instances, but format/parse differently:
  - DateFormat.getDateInstance()
  - DateFormat.getTimeInstance()
  - DataFormat.getDateTimeInstance()

Exceptions

- Two kinds of exceptions: JVM thrown and programmatic thrown. Main exceptions are:

JVM Exceptions	Programmatic Exceptions
NullPointerException	IllegalArgumentException
StackOverflow**Error**	IllegalStateException
ArrayIndexOutOfBoundsException	NumberFormatException
ClassCastException	AssertionError
ExceptionInInitializer**Error**	
StackOverflow**Error**	
NoClassDefFound**Error**	

- The main runtime and checked exceptions are:

Runtime Exceptions	Checked Exceptions
ArrayIndexOutOfBoundsException	ClassNotFoundException
ClassCastException	URISyntaxException
IllegalArgumentException	IOException
IllegalStateException	FileNotFoundException
NullPointerException	
NumberFormatException	

- When overriding a method:
  - **If the super method does not throw any exception, the sub method can throw any unchecked exception that it likes, but no checked exceptions.**
  - If the super method throws a checked exception (i.e. IOException), then the sub method can only throw IOException and subclasses (e.g. FileNotFoundException), as well as any unchecked exceptions.
- The following table shows what happens when exceptions occur, based on whether they are caught, etc:

Exception Thrown?	Try Exists?	Matching Catch Found?	Execution Control Flow Behaviour
No	N/A	N/A	Normal control flow
Yes	No	N/A	Method terminates
Yes	Yes	No	Method terminates. If a finally block exists, it is executed before method termination.
Yes	Yes	Yes	Execution jumps from the exception point in the try block to the matching catch block. The catch block executes and control jumps to the first line after the last catch block. Normal control flow returns.

Assertions

- Starting from Java 1.4, assert became a keyword, meaning that it cannot be used as a variable name.
- It is possible, when using Java 5, to tell it to compile code using an older compiler. This is useful when working with code that uses assert as an identifier (and not a keyword). This is done by using the '-source 1.3' compiler option. This allows the identifier to compile, but of course compilation will fail if assert is used as a keyword.

- Assertions should not be used to validate arguments to public methods, including the main() method. This is because it cannot be guaranteed that assertions will be enabled, so the test should be written using other constructs.
- It is appropriate to use assertions to generate an alert when code is reached that should not be reachable.
- Assertions should not cause side-effects, so calling setter methods is a bad idea.
- To enable assertions, use either '-ea' or '-enableassertions' as command line switches.
- To disable assertions, use either '-da' or '-disableassertions' as command line switches.
- It is possible to selectively enable and disable assertions in different packages. Some examples include:

Java -ea -da:com.ga.Foo	Enables assertions for all classes except the com.ga.Foo class.
Java -ea -da:com.ga…	Enables assertions for all classes except those classes in com.ga **and all subpackages.**
Java –ea:com.foo…	Enable assertions in package com.foo and any of its subpackages.

Inheritance

- **Variable retrieval is always determined at compile-time**, and it depends on the declared type of the invoking object. For example, the following code will always return the x defined in c1, even if getObject returns a subtype of c1 which overrides x to be a different value:
  - ConvariantTest c1 = new SubCovariantTest();
    System.out.println(c1.getObject().x);
- When overriding a class that has a protected method or instance variable, it is not possible to access these with an instance of that class inside a subclass. It is however possible to access these with an instance of the subclass itself, or of course without an instance at all.
- Variables declared in interfaces are considered to be static and final. They can be accessed through static imports, but cannot be modified.
- The 'is-a' statement works for when a class extends another class, and also when a class implements an interface. In other words, it requires two different types, and always relies on polymorphism.

Inner Classes

- Whenever creating instances of inner classes (both normal and static), it is necessary that the enclosing classes name always be used when declaring the variable.

Collections

- A **TreeSet sorts its elements. By default, it will try to sort the elements in their natural order. For this to happen,** it is necessary that they implements Comparable. Therefore, all types in the set must either be comparable with each other, or the comparator given to the TreeSet be able to compare the elements. If a TreeSet<Number> already has one int inside it, all other numbers added must be of type int (or castable to int, e.g. short). This means long, float, etc cannot be added.
- Similarly, when using Collections.sort(), be certain that all the types in the collection are comparable. If even one element is of a different type, an exception will be thrown.
- When adding to a hashmap, if an items hashcode and equals are equal to another item already in the map, the new item will replace the old one. If the hashcodes are different (e.g. when hashcode is properly overridden or when hashcode is not overridden and the default one from Object is used), then the item will be able to be inserted without replacing the old one.
- Prior to using Collections.binarySearch, it is necessary to first ensure the collection is sorted, which can be achieved by calling Collections.sort(). If the collection isn't sorted,

the results are undefined. When using Collections.reverse(), it is not necessary that the collection be sorted first.

- If a collection is sorted using a comparator, it is critical that the binarySearch method also be called with the same comparator.
- **NavigableSet is an interface** which extends the SortedSet interface, and **is implemented by TreeSet**. The most important methods are lower(), floor(), ceiling() and higher(). All four methods take an element, and return the next element that is less than, less than or equal to, greater than or equal to, and greater than, respectively.
- NavigableSet has a function to return a set in descending order, as opposed to its usual ascending order sorting. This method is NavigableSet.descendingSet(). As with any set, it is possible to get an iterator from this set. Conversely, it is possible to get the descending set iterator by calling NavigableSet.descendingIterator().
- TreeSet and TreeMap have methods that return a collection in the reverse order. These are descendingSet() and descendingMap(), respectively.
- ‘Polling’ is the term used to mean retrieve and remove from the collection. The TreeSet interface has pollFirst() and pollLast() methods. Similarly, TreeMap has pollFirstEntry() and pollLastEntry().
- ‘Peeking’ is the term used to mean retrieve an object from a collection, without removing it.
- Backed collections are collections created from a set or map that refer back to the original map. When we add objects into either collection, it is in many cases expected that they turn up in the other collection also. This only happens however if the backed collection (i.e. the subset/submap) range includes the value of the inserted object.
- The Arrays.asList() method returns a list that is backed by the array. This means that changes (but not additions) will be reflected in the other.
- Remember that any set that implements SortedSet is sorted. This includes ConcurrentSkipListSet and TreeSet.
- The toArray() method of collections returns an Object array. To have this be cast to another, more relevant type, call the toArray(Object[]) method, which will cast the resulting array to the type of the array given as the argument.
- **HashMap is unsynchronized and can have null keys and values**. Conversely, **HashTable is synchronized and cannot have null keys and values.**

Generics

- Be careful when extending generic classes. For example, extending HashSet<Integer> means that you cannot have a ‘public boolean add(Object o)’ as the erasure of hashsets add method is the same as this. This will lead to a compiler error. To resolve this, the add method should accept an Integer.
- Unboxing from a collection only works when the collection is type-safe (i.e. has been generified). If it is not type-safe, calling get() will return an Object which cannot be cast to an int, for example.
- It **is possible** to pass a generic collection into a method as a non-generic collection. This **will result in an unchecked exception at compile-time, but the code will work**. Within this method, it is possible for it to add any type of object into the list. This will work successfully until the time comes to get() out of the collection if it isn’t handled properly.
- Conversely, **it is legal to pass a non-generic collection into a method expecting a generic collection.**
- Polymorphism does not apply to the generic types. For example:

`List<Object> myList = new ArrayList<JButton>();`	Wrong
`List<Number>numbers = new ArrayList<Integer>();`	Wrong
`List<JButton> myList = new ArrayList<JButton>();`	Ok
`List<Object> myList = new ArrayList<Object>();`	Ok

`List<Integer> myList = new ArrayList<Integer>();`	Ok

- It is possible to add instances of a subtype into a collection declared with the supertype. For example:
  - List<Animal> animals = new ArrayList<Animal>();
    animals.add(new Cat());
- It is important to understand how to declare and instantiate a generic collection:

`List<?> list = new ArrayList<Dog>();`	Valid
`List<? extends Animal> list = new ArrayList<Dog>();`	Valid
`List<?> list = new ArrayList<? extends Animal>();`	Invalid. Can't use wildcards in object creation.
`List<? extends Dog> list = new ArrayList<Integer>();`	Invalid. Can't assign an integer list to a reference that takes only a Dog and subtypes of Dog.
`List<? super Dog> list = new ArrayList<Animal>();`	Valid
`List<? super Animal> list = new ArrayList<Dog>();`	Invalid. Dog is 'too low' in the class hierarchy.

- It is possible to create generic methods that are individually generified. To do this, you must declare the type of the variable before the return type, e.g.:

`public <T extends Number> void makeArrayList(T t) { }`	This is valid
`void basket(List<? Super Apple> l) { l.add(new Apple()); }`	This is valid, even though it is not a generic method.

- Be careful when dealing with generic methods, for example:

`public <S extends CharSequence>`
`    S foo(S s) {`
`        // code here`
`}`
	Given the following choices, only the bold ones are valid:
	• **return s;**
	• return s.toString();
	• return new StringBuilder(s);
	• return (S) new StringBuilder(s);
	• **return null;**
	The other options are invalid as, even though they are CharSequences, they are not necessarily the same subtype as whatever is passed in S.

Strings, Formatting and Parsing (Regular Expressions)

Strings

- Spaces are sorted before characters, and uppercase letters are sorted before lowercase letters.
- String comparison is done on a char by char basis. As soon as one char is less than the other, the comparison is complete. For example:
  - Xml < xml (as 'X' < 'x')
  - Xml < java (as 'X' < 'j')
  - Java < XML (as 'J' < 'X')
  - '30' < '4' (as '3' < '4')

Searching

- The matcher.find() method returns a Boolean if a result is found.
- The matcher.start() method returns the starting position for a match.
- In general, a regex search runs from left to right, and once a source's character has been used in a match, it cannot be reused. For example, with the string 'abababa' and the expression 'aba', we will get two matches, not three.

- For the SCJP exam, it is important to know four metacharacters:

Meta Character	Description
\d	finds a single digit (i.e. 0-9).
\s	finds a single whitespace character.
\w	finds a single word character (letters, digits, or the underscore character).
'.'	The predefined dot metacharacter means ''any character can server here'. For example, using 'a.c' as the pattern and 'ac abc a c' as the source, we will get two results: 'abc' and 'a c' – the . metacharacter has matched the 'b' and the space.

- It is possible to use square brackets to specify sets and ranges of characters to search for. For example [abc] will search for 'a', 'b' and 'c'. Similarly, [a-f] searches for all characters between 'a' and 'f'. Finally, it is possible to put in multiple ranges into one set, for example, [a-fA-F] finds all lower and upper case characters between 'a' and 'f'.
- For the SCJP exam, there are three quantifiers that need to be understood: +, *, and ?.
    - + is used to state that there needs to be <u>one or more</u> characters matching.
    - * is used to state that there needs to be <u>zero or more</u> characters matching.
    - ? is used to state that there needs to be <u>zero or one</u> characters matching.

Tokenizing

- It is possible to tokenize a String by calling the split() method. The problem with this is that it returns a fully processed array.
- The scanner class allows for the tokenization to be done in a loop, allowing for early exits, etc. Important Scanner methods include:
    - hasNext() method returns true when there is another token.
    - hasNextXxx() method returns true when the next token is of the Xxx type. If the next token is not of this type, it returns false, even if there are more tokens.
    - nextXxx() returns the next token in the string as the specified type. **This will result in an exception if the next token isn't of this type.**

Formatting

- Format strings have the following structure:
  %[arg_index$][flags][width][.precision]conversion char
- It is illegal to use a conversion character that does not match the given argument. There are exceptions however:
    - Integer **cannot** be formatted as %f.
    - Float **cannot** be formatted as %d.
    - Long (obviously) can be formatted as %d.
    - Double (obviously) can be formatted as %f.
    - Anything (?) can be formatted as %s.
- The relevant flags for the SCJP exam are:

"-"	Left justify the argument
"+"	Include a sign (+ or -) with this argument
"0"	Pad this argument with zeroes
","	Use locale-specific grouping separators (i.e. the comma is 123,456)
"("	Enclose negative numbers in parentheses

- Examples include:

%1$(7d	First argument, use brackets for negative numbers, minimum width is 7, Format as integer.
%0,7d	Zero-positioned argument, Use locale-specific formatting, Minimum width is 7, Format as integer.

%+-7d	Take single argument, Use a + or – sign as necessary, Align to the left Minimum width is 7, Format as integer.
%2$b + %1$5d	Take second argument, and format as a Boolean. Print “ + “. Take first argument, minimum width is 5, format as integer.

Console

- The method to get the System console is System.**console**(), not System.getConsole().

File I/O

- When using FileWriter, it will automatically create a file if one does not exist. If the file already exists, it will overwrite the contents unless it is set to append.
- PrintWriter has a constructor which takes a File object, whereas BufferedWriter does not.
- File.delete() only deletes directories if they are empty.

Serialization

- When unserializing an object, it is important that either the parent class also implement serializable, or have a public constructor.
    - If a parent class does not implement Serializable, it (and all super classes), must have a public constructor.
- Remember that when deserializing, constructors are not called, so it does not matter if the constructor is public or not.
- When deserializing, all transient variables are initialized to their default values. This is because no constructors or initializers are called. Therefore, even if a transient variable is declared as ‘transient int var = 3’, when deserialized, its value will be 0.
- When serializing, remember that all instance variables need to be serializable too. If they aren’t an exception is thrown at runtime.
- When serializing an object, all instance variables are serialized, as well as the instance itself. For example:

```
import java.io.*;
class Animal implements Serializable { }
class Cow extends Animal {
    Milk m = new Milk();
}
class Milk implements Serializable {
    SaturatedFat sf1 = new SaturatedFat();
    SaturatedFat sf2 = new SaturatedFat();
    SaturatedFat sf3 = new SaturatedFat();
}
class SaturatedFat implements Serializable {
}
```

When serializing an instance of Cow, we serialize the instance of Cow itself, the instance of Milk, and the three instances of SaturatedFat.

This means in total we are serializing 5 objects.

We do not consider the serialization of Animal as a separate object.

- There are two special methods that can be implemented to give more control over the serialization/deserialization process. They are the following (note that they are private):
    - private void writeObject(ObjectOutputStream oos) { … }
    - private void readObject(ObjectInputStream ois) { … }

Threads

- The synchronized keyword when used on methods synchronizes on the current instance. Therefore, if two threads are to synchronize a method call so one occurs after the other, it is important that they synchronize on the same lock/instance.

- Whenever wait(), notify() or notifyAll() are used, they must be used within synchronized blocks. If wait(), notify() or notifyAll() are run on an object, then the synchronization must be on that object, for example:

Bad Synchronization	Good Synchronization

`public synchronized void method()`
`{`
`    Thread t = new Thread();`
`    t.start();`
`    System.out.println(“X”);`
`    t.wait(1000);`
`    System.out.println(“Y”);`
`}`
	`public synchronized void method()`
	`{`
	`    Thread t = new Thread();`
	`    Synchronized(t) {`
	`        t.start();`
	`        System.out.println(“X”);`
	`        t.wait(1000);`
	`        System.out.println(“Y”);`
	`    }`
	`}`

- When wait() is called, all locks are released.
- When a thread goes to sleep, it does not release the locks that it holds.
- Be mindful that even in synchronized blocks it is possible to change the object that the synchronization is occurring on. This means that the synchronization remains on the old object, and so calling notify() or notifyAll() will result in an IllegalMonitorStateException.
- The thread sleep() and yield() methods are static, and as such operate on the calling thread, not on the thread object on which they may be called.
- The Thread.yield() method is intended to be used to have a thread give up its running state so that another thread may have a chance to run, but there is no obligation when yield() is called for the thread to give up its running state.
- The non-static join() method of class Thread lets one thread “join onto the end” of another thread. It throws an InterrupedException that needs to be handled.

Java and Javac commands

Javac.exe

- The javac -d command tells the compiler where to put the .class files it generates when compiling source code (d is for destination).
- When compiling a class that is within a package, use forward slashes between the package folder names, not dots. For example:
  - Javac -d ../classes com/wickedlysmart/MyClass.java.
- When compiling, javac will automatically create folders to represent the package structure, but it will not create the base destination folder, in the above example the ‘../classes’ folder. If this destination folder doesn’t exist, the compiler fails.
- Javac looks in the current directory by default, so this does not have to be manually specified (unlike java).

Java.exe

- When executing a java class using java.exe, do not include the .class extension.
- It is possible to set properties by using the -D command. Examples include the following. In both cases, the key is ‘cmdProp’. In the second example, because there are spaces in the value, we use quotes. Note that there is no space between the -D and the key/value:
  - java -DcmdProp=cmdValue TestProps
  - java -DcmdProp=”cmdVal take 2” TestProps
- It is necessary to manually specify the current directory by including the ‘.’ period in the classpath declaration, if the current directory has relevant class files.

Locale, Dates, Numbers and Currency

- Locales are created through the following constructors. **Note that the language is always first, and the country is always second.** An example language is ‘en’, whereas an example country is ‘US’ or ‘UK’.
  - Locale(String language)
  - Locale(String language, String country)
  - Locale(String language, String country, String variant)
- Calendar does not have a constructor; to get an instance, call Calendar.getInstance().
- To get and set the default locale, call Locale.getDefault() and Locale.setDefault(Locale).
- Calendar.getTime() returns a Date, but Date.getTime() returns a long. For a Calendar to return the time in milliseconds, call Calendar.getTimeInMillis().

Things to watch out for

- If an exception occurs in a try { .. } catch block and it isn’t caught, then execution does not continue – it throws the exception to the calling method.
- Be careful when dealing with Maps – they don’t have add() methods.
- Be careful with static constants, make sure that all constants needed are imported properly.
- If a static constant is in a different package, and is not marked public, it is invisible to any class trying to statically import it.
- Similarly, if an interface or class is in a different package and is not marked public, it will not be visible to classes and interfaces outside this package.
- Be careful not to call a non-static method from a static method. There always needs to be an instance variable used.
- When referring to JARs in the classpath, it is important to remember that the file must specifically be included, including the .jar extension.
- Be careful when using API methods that they exist. String does not have an ‘insert’ method, but it does have a ‘concat’ method. All String methods return a new String.
- Be careful and check to ensure all brackets, braces and semi-colons are in place, particularly when defining arrays and anonymous classes.
- When code looks funny, try to think whether it actually could make sense if rearranged, rather than immediately jump to the conclusion that compilation has failed, for example, the following code is equivalent, and both compile:

```
int I = 1;
do while (I < 1)
    System.out.println(“I is “ + I);
while (I > 1);
```

```
int I = 1;
do {
    while (I < 1) {
        System.out.println(“I is “ + I);
    }
} while (I > 1);
```

- Be wary of enumerations – if each enum has a constructor call associated with its definition, a matching constructor must exist.
- The finalize method must be overridden as a **public** or **protected** method.
- The serialization special methods must be overridden precisely as follows:
  - **private** void writeObject(ObjectOutputStream oos) { … }
  - **private** void readObject(ObjectInputStream ois) { … }
- Remember, and check, which thread-related methods throw exceptions. In summary, the following thread-related methods throw checked exceptions and must be handled:
  - Object.wait()
  - Thread.join()
  - Thread.sleep()
- Be careful to ensure that all necessary imports are performed.