Chapter 1: Language Fundamentals

1. Java is case sensitive.
2. Source file may contain ONLY ONE public class or interface, and any numbers of default classes or interfaces.
3. If there is a public class or interface, the name of the source file must be the same as the name of the class or interface.
4. The file name may begin with numbers if there is no public class on it.
5. If no package is explicit declared, java places your classes into a default package (Object).
6. Identifiers are composed of characters, where each character can be any letter (including Ò ù Á ÆÅ), a connecting punctuation (underscore _) or any currency symbol (such as $ ¢ £ ¥) and CANNOT start with a digit.
7. Keywords are reserved identifiers that are predefined in the language, and CANNOT be used to denote other entities. NOTE: None of the keywords have a capital letter.

   The following table denotes the currently defined keywords:

abstract	***default if***	***package***	***synchronized***	***boolean***
do	***implements***	***private***	***this***	***break***
double	***import***	***protected***	***throw***	***byte***
else	***instanceof***	***public***	***throws***	***case***
extends	***int***	***return***	***transient***	***catch***
final	***interface***	***short***	***try***	***char***
finally	***long***	***static***	***void***	***class***
float	***native***	***super***	***volatile***	***continue***
for	***new***	***switch***	***while***	
strictfp	I found this in reference , but I didn't find it in any certification book.			

   The following table shows three reserved predefined literals:

null	***true***	***false***

   The following table shows reserved keywords NOT currently in use:

const	***goto***		
byvalue	***generic***	***outer***	I found this in reference , but
cast	***inner***	***rest***	I didn't find it in any certification
future	***operator***	***var***	book.

8. Integer literal by default is int, you can specify it as long by appending "L" or "l" as suffix, NOTE: THERE IS NO WAY to specify a short or byte literal.
9. Floating point literal by default is double, you can specify to be a float by appending "F" or "f" as suffix.
10. Octal numbers are specified with "0" as prefix, Hexadecimal numbers are specified with "0x" or "0X" as prefix.
11. Most important Unicode values:

Escape Sequence	Unicode Value	Character
' '	\u0020	Space
'0'	\u0030	0
'9'	\u0039	9
'A'	\u0041	A
'Z'	\u005a	Z
'a'	\u0061	a
'z'	\u007a	z

12. Escape sequences are used to define special character values and can be represented also in Unicode value, the following table shows them:

Escape Sequence	Unicode Value	Character
\b	\u0008	Backspace
\t	\u0009	Horizontal tabulation
\n	\u000a	Linefeed
\f	\u000c	Form feed
\r	\u000d	Carriage return
\’	\u0027	Apostrophe-quote
\”	\u0022	Quotation mark
\\	\u005c	Backslash
\xxx		A character in octal representation; xxx must range between 000 and 337
\uxxxx		A unicode character, where xxxx is a hexadecimal format number.

13. The single apostrophe ‘ need not to escaped in Strings, but it would be if specified as a character literal ‘\’’. Example:

    String tx = “Delta values are labeled \”\u0394\” on the chart.”;

14. Regardless of the type of comment, it can’t be nested.

15. Default values for member variables table:

Data type	Default value
boolean	false
char	‘\u0000’
Integer(byte, short, int, long)	0
Floating-point(float, double)	+0.0F or +0.0D
Object reference	null

16. Static variables in a class are initialized to default values when class is loaded if they are not explicitly initialized.

17. Instance variables are initialized to default values when the class is instantiated if they are not explicitly initialized.

18. Local variables (Automatic) are NOT initialized when they are instantiated at method invocation. The compiler javac reports use of un-initialized local variables.

19. There can be ONLY one package declaration in a Java file, and if it appears, it must be the first non-comment statement.

20. The JVM expects to find a method named main with the signature as follows:

    public static void main(String[] args)

    The array is typically named args, but it could be named anything.

    NOTE: You can have methods named main that have other signatures. The compiler will compile these without comment, but the JVM will not run an application that does not have the required signature.

Chapter 2: Operator and assignments

1) Operator precedence and associativity:

Postfix operators	**[] .** (parameters) expression++ expression--
Prefix unary operators	++ expression --expression +expression –expression ~ **!**
Object creation and cast	**new** (type)
Multiplication	*** / %**
Addition	**+ -**
Shift	<< >> >>>
Relational operators	< <= > >= **instanceof**
Equality operators	== **!=**
Bitwise/Logical AND	**&**
Bitwise/Logical XOR	^
Bitwise/Logical OR	|
Logical AND	**&&**
Logical OR	||
Conditional operator	**?:**
Assignment	= += -= ***=** /= **%=** <<= >>= >>>= **&=** ^= |=

2) Casting between primitive values and references **CANNOT** be applied.

3) ~ (Bit wise inversion) convert all 1's to 0's and vise versa.

4) The modulo operator % give the value of the remainder of the division of the left operand (dividend) by the right operand (divisor).

5) A useful rule to calculate the modulo: Drop any negative sign from the operands, calculate the modulo, then the result sign is relative to the left operand (dividend).
   Example:
   int x = -5 % 2; // x = -1
   int y = -5 % -2; // y = -1
   int z = 5 % -2; // z = 1

6) || and *&&* work with boolean not with integers like *&* and |, and the result from | - *&* is *int*.

   NOTE: *&* and | are used with both boolean and integers.

7) Unlike in C, integers in Java can **NEVER** be interpreted as boolean values, so expressions used for flow control **MUST** evaluate to boolean.

8) The conditional assignment operator, the ONLY Java operator that takes three operands:?
   <condition>? <expression1>:<expression2>
   Example:
   String x = (salary < 1500)? "Poor": "Not poor";
   String y = (salary > 1500)? "Poor": (salary1 < 10)? "poor1":"poor2";

9) Multiple assignment:

   k = j = 10; //   (k = (j = 10))

10) Boolean values **CANNOT** be casted to other data values, and vice versa, the same applies to the reference literal null, which is **NOT** of any type and therefore **CANNOT** be casted to any type.

11) Conversion is done when:
    (a) Assigning a value to a primitive variable.
    (b) Evaluating arithmetic expressions.
    (c) Matching the signature of methods.

    Example:

The *Math* class has *max* and *min* methods in several versions, one for each of *int*, *long*, *float*, *double*. Therefore, in the following code that calls *Math.max* with one *int* and one *long* in line 3, the compiler converts the *int* primitive to a *long* value. The alternative in line 4 forces the compiler to cast the value n to an *int* and calls the version of *max* that uses two *int* primitives.

```
int m = 93;                    //(1)
long n = 91;                          //(2)
long x = Math.max(m, n);       //(3)
int y = Math.max(m, (int) n);  //(4)
```

12) Widening primitive conversion is as follows (it doesn't lose information about the magnitude of a value), and any other conversion is called narrowing primitive conversion and may cause lose of information. At runtime, casts that lose information do not cause a runtime excpetion, and it is up to the programmer to think through all the implications of the cast.

$$\begin{tikzcd}
\text{Byte} \arrow[r] & \text{short} \arrow[dr] & & & \\
& & \text{int} \arrow[r] & \text{long} \arrow[r] & \text{float} \arrow[r] & \text{double} \\
& \text{Char} \arrow[ur] & & &
\end{tikzcd}$$

13) Integers of int (32-bit) or smaller can be converted to floating-point representation, but because a float also uses only 32 bits and must include exponent information, there can be a loss of precision.

14) All six numbers types in Java are signed meaning they can be negative or positive.

15) The ONLY integer primitive that is not treated as a signed number is char, which represents a Unicode character.

16) All conversion of primitive's data types take place at compile time.

17) Arithmetic operations:
   a) For unary operators if byte – short - char Æ converted to int.
   b) For Binary operands:
      i) If one of operands is double the other operand is converted to a double.
      ii) If one of the operand is float, the other operand is converted to float.
      iii) If one of the operand is long, the other operand is converted to long.
   c) Else both the operands are converted to int.

18) Ranges of primitive data types:

Type	Bits	Bytes	Minimum range	Maximum range
byte	8	1	-2^{7}	$2^{7} - 1$
short	16	2	-2^{15}	$2^{15} - 1$
char	16	2	\u0000	\uFFFF
int	32	4	-2^{31}	$2^{31} - 1$
long	64	8	-2^{63}	$2^{63} - 1$
float	32	4	1.40129846432481707e-45	3.40282346638528860e+38
double	64	8	4.94065645841246544e-324	1.79769313486231570e+308

19) Depending on the storing type of the arithmetic operation the precision is done.
Example:

```
int  x = 7/3;                  // x = 2
byte b = 64; b *= 4;  // b = 0
```

20) The compiler pays attention to the known range of primitives.
Example:

```
int n2 = 4096L;    // (1) would require a specific (int) cast
short s1 = 32000;  // (2) OK
short s2 = 33000;  // (3) out of range for short primitive
```

In spite the fact that 4096 would fit in an *int* primitive, the compiler will object on the first line because the literal is in *long* format. You could force the compiler to accept line 3 with a specific (*short*) cast, but the result would be a negative number due to the high bit being set.

21) Important examples for arithmetic expression evaluation:

Arithmetic Expression	Evaluation	Result when printed
4/0	Arithmetic Exception	
4.0/0.0	(4.0/0.0)	POSITIVE_INFINITY
-4.0/0.0	((-4.0)/0.0)	NEGATIVE_INFINITY
0.0/0.0	(0.0/0.0)	NaN

22) *NaN* can result from mathematical functions that are undefined, such as taking the square root of a negative number. In float to double conversion, if the float has one of the special values, *NaN*, *POSITIVE_INFINITY*, or *NEGATIVE_INFINITY*, the double ends up with the corresponding double special values.

23) *Float.NaN*, *Double.NaN* are considered non-ordinal for comparisons, this means all that are *false*:
x < Float.NaN
x == Float.NaN
But you can test by *Float.isNaN(float f), Double.isNan(double d).*

24) While casting special floating-point values, such as *NaN*, *POSITIVE_INFINITY* to integer values, they will be casted without any complaint from the compiler or an exception.

25) <variable> <op>=<expression>
is equivalent to
<variable> = (<variable type>) (<variable><operator>(<expression>)).

26)
```
short h = 40;        // OK, within range
h       = h + 2;     // Error can't assign int to short
```
Solution for the above situation, choose one of the following:
```
h = (short) (h+2);
h += 2;
```
NOTE:
```
h = h + (short)2;    // Requires additional casting
```
Will not work because binary numeric promotion leads to an int values as result of evaluating the expression on the right-hand side.

27) *System.out.println("We put two and two together and get " + 2 + 2);*
Prints: We put two and two together and get 22
NOT: We put two and two together and get 4
Declaration: ((("We put two and two together and get ") + 2) + 2).

28) If your code conducts operations that overflow the bounds of 32-bit or 64-bit integer arithmetic, that's your problem, i.e. Adding 1 to the maximum int value 2147483647 results in the minimum value –2147483648, i.e. the values "wrap-around" for integers, and no over or underflow is indicated.

29) The dot operator has left associativity, in the following example the first call of the *make()* returns an object reference that indicates the object to execute the next call, and so on …
SomeObjRef.make().make().make();

30) To get the 2's complement:
a) Get the 1's complement by converting 1's to 0's and 0's to 1's.
b) Add 1.

31) In the << left shift operator all bytes are moved to the left the number of places you specify, and zero is padded from the right. [6]

32) >> Signed right shift and >>> unsigned right shift work identically for positive numbers, in >>> operator zeros

fill the left most bits, but in >> will propagate the left most one through the rest of bits.

33) When you shift a bit by a numeric value greater than the size in bits, Java does a modulus shift.

34) To shift a negative number get the 2's complement and then shift it.

35) Object reference equality (==, !=):
The equality operator == and the inequality operator != can be applied to object references to test if they denote the same object. The operands must be type compatible, i.e. it must be possible to cast one into the other's type, otherwise it is a compile time error.
Example:
Pizza pizza_A = new Pizza("Sweat & Sour");
Pizza pizza_B = new Pizza("Hot & Spicy");
Pizza pizza_C = pizza_A;

String banner = "Come and get it";

boolean test = banner == pizza_A;   // Compile time error
boolean test1 = pizza_A == pizza_B; // false
boolean test2 = pizza_A == pizza_C; // true

36) The *equals* method:
In the Java standard library classes, the method that compares content is always named *equals* and takes an *Object* reference as input. It does not look at the value of the other object until it has been determined that the other object reference is not *null* and that it refers to the same type, else it will return *false*.
NOTE:
The *equals* method in the *Object* class returns true **ONLY IF**
this == obj
so in the absence of an overriding *equals* method, the == operator and *equals* method are equivalent.

37) Using *instanceof*: if the right-hand operand that **MUST** name a reference type may equally will be an interface; In such case the test determines if the object at the left-hand argument that **MUST** be a name of a reference variable implements the specified interface.

38) If we compare an object using *instanceof* and the class we compare with is not in the hierarchy this will cause compiler error, (Must be on the hierarchy above or bottom the class). We can overwrite the problem of compiler error caused by *instanceof* by declare the class from the *Object* class.
Example:
Object x = new Button();

39) Object reference conversion takes place at compile time because the compiler has all the information it needs to determine whether the conversion is legal or not.

40) Short circuit evaluation:
In evaluation of *boolean* expression involving conditional AND *&&* or OR ||, the left operand is evaluated before the right one, and the evaluation is short circuited, i.e.: if the result of the boolean expression can be determined from the left-operand, the right-hand operand is not evaluated.
NOTE:
when bitwise AND & or bitwise OR | are used in a boolean expression, both operands are evaluated, and **NO** short circuit evaluation is applied.

41) Parameter passing: All parameters are passed by value.

Data type of the formal parameter	Value passed
Primitive data types	Primitive data value
Class type	Reference value
Array type	Reference value

42) A formal parameter can be declared with the keyword final preceding the parameter declaration. A final

parameter is also known as blank final variable, i.e. it is blank (uninitialized) until a value is assigned to it at method invocation.

Example:

```
public static void bake(final Pizza pizzaToBeBaked) {
   pizzaToBeBaked.meat = “chicken”;   // Allowed
   pizzaToBeBaked = null;             // Not Allowed
}
```

Chapter 3: Declarations and Access Control

1) Arrays are a special kind of reference type that does not fit in the class hierarchy but can always be cast to an *Object* reference. Arrays also implement the *Cloneable* interface and inherit the *clone* method from the *Object* class, so an array reference can be cast to a *Cloneable* interface.

2) Array declaration and constructor:
   <elementType1> <arrayName>[] = new <elementType2> [numberofelements];
   Note:
   <elementType2> must be assignable to *<elementType1>*, i.e.: class or subclass of *<elementType1>*, and when the array is constructed, all its elements are initialized to the default value *for <elementType2>*, **WHATEVER** the array is automatic variable or member variable.

3) When constructing multi-dimensional arrays with the new operator, the length of the deeply nested arrays may be omitted, these arrays are left unconstructed.
   Example:
   double matrix[][] = new double[3][];

4) Length of array object is a variable **NOT** a method.

5) It is legal to specify the size of an array with a variable rather than a literal.

6) The size of the array is fixed when it is created with the *new* operator or with special combined declaration and initialization.

7) Anonymous arrays:
   new <elementType>[] {<initialization code>}
   NOTE: No array size is mentioned in the above syntax. Example of usage:

```
class AnonArray {
   public static void main(String[] args) {
      System.out.println("Minimum value = " + findMin(new int[] {3,5,2}));
   }
   public static int findMin(int[] dataSeq) {
      int min = dataSeq[0];
      for (int index=1; index<dataSeq.length; index++) {
         if (min >= dataSeq[index])
            min = dataSeq[index];
      }
   }
}
```

8) There is **NO** way to `bulk' initialize an array, if you want to initialize array to certain value during declaration => you **MUST** iterate with the value you want. **NOTE**: Initialization by means of a bracketed list can be used only in the statement that declares the variable.

9) It is possible to create arrays of zero length of any type, a common natural occurrence of an array of zero length is the array given as an argument to the *main()* method when a Java program is run without any program arguments.

10) Primitive arrays have no hierarchy, and you can cast a primitive array reference **ONLY** to and from an Object reference. Converting and casting array elements follow the same rules as primitive data types. Look to the strange **LEGAL** syntax for casting an array type as shown in line 3 in the following example.

```
int sizes[] = {4, 6, 8, 10};          //(1)
Object obj = sizes;                   //(2)
int x = ((int[])obj)[2];              //(3)
```

11) Casting of arrays of reference types follows the same rules as casting single references.
   NOTE that an array reference can be converted independantly of whether or not the array has been populated

with references to real objects. Example:
Suppose you have a class named *Extend* that extends a class named *Base*. You could then use the following code to manipulate a reference to an array of *Extend* references:

```
Extend[] exArray = new Extend[20];
Object[] obj = exArray;
Base[]bArray = exArray;
Extend[] temp = (Extend[])bArray;
```

12) An *import* declaration does not recursively import sub-packages.

13) The order of modifiers in class declaration:

a)	public.	(optional)
b)	final or abstract.	(CANNOT appear together)
c)	class.	(mandatory)
d)	classname.	(mandatory)
e)	extends.	(optional)
f)	superclassname.	(mandatory if extends specified)
g)	implements.	(optional)
h)	interfacelist.	(mandatory if implements specified)
i)	{}.	(mandatory)

14) If the access modifier is omitted => (package or default accessibility), in which case they are only accessible in the package but not in any sub-packages.

15) The **ONLY** access modifier allowed to the top level class is *public* or friendly.

16) *abstract* modifier implies that the class will be extended, but *abstract* class **CANNOT** be instantiated.

17) The compiler insists that a class that has an *abstract* method must be declared *abstract*, and this forces its subclasses to provide implementation for this method, and if a subclass does not provide an implementation of its inherited methods must be declared *abstract*.

18) It is **NOT** a **MUST** for an *abstract* class to have an *abstract* method.

19) Interfaces as classes **CANNOT** be declared *protected, private, native, static, synchronized.*

20) An interface is different from a class in also it can extend **MORE** than one interface, this follows from the fact that a class can implement more than one interface.
Example:
public interface RunObs extends Runnable, Observer
Any class implementing this interface will have to provide methods required by both *Runnable* and *Observer*.

21) The order of modifiers in method declaration:

a)	public or private or protected.	(optional for package declaration)
b)	abstract or final or native or static or synchronized.	(optional)
c)	returntype.	(mandatory)
d)	methodname.	(mandatory)
e)	throws clause.	(optional)
f)	{}.	(mandatory)

22) *abstract* methods or methods defined in an interface must end with ';'. (i.e. abstract method is non-functional methods that haven't body), and abstract methods declared **ONLY** on interface or abstract classes.

23) The class must be declared *abstract* if:
a) The class has one or more *abstract* methods.
b) The class inherits one or more *abstract* methods (from an abstract parent) for which it doesn't provide implementation for one or more of the *abstract* methods of the parent class.
c) The class declares that it implements an interface but doesn't provide implementation for **EVERY** method of that interface.

24) When an abstract class implements interface there is no need to this class to implement all members of that interface.

25) Interfaces just specify the method prototypes and not the implementation; they are by their nature, implicitly *abstract*, i.e. they **CANNOT** be instantiated. Thus specifying an interface with the keyword *abstract* is not appropriate, and should be omitted, but it won't give compile error if specified.

26) *final* classes **CANNOT** be extended. Only a class whose definition is complete (i.e. has implementation of all the methods) can be specified to be *final*.

27) The order of modifiers in variable declaration:

a)	*public* or *private* or *protected*.	(optional)
b)	*final* or *static* or *transient* or *volatile*.	(optional)
c)	***variable type.***	(mandatory)
d)	***variable name.***	(mandatory)

28) **Within a class definition**, reference variables of this class's type can be used to access all **NOT INHERITED** members regardless of their accessibility modifiers.
Example:

```
Class Light {
   // Instance variables
   private int noOfWatts;
   private boolean indicator;
   private String location;

   public void switchOn() {indicator = true;}
   public void switchOff() {indicator = false;}
   public boolean is On() {return indicator;}

   public static Light duplicate (Light oldLight) {
      Light newLight = new Light();
      newLight.noOfWatts = oldLight.noOfWatts;
      newLight.indicator = oldLight.indicator;
      newLight.location = new String(oldLight.location);
   }
}
```

29) **ONLY** variables, methods and inner classes may be declared *protected*.

30) *static* members can be called from the member objects.
Example:
this.xyz
NOTE:
The use of *this* **MUST** be from a non-static method, or "this cannot be referenced from a static context" compiler error will be thrown.

31) Trying to use object class member before the constructor of the object class member called will compile fine but will give *NullPointerException*.
Example:

```
public class Trial {
   static Date d ;
   public static void main (String args[]) {
      System.out.println( d.getYear() );
   }
}
```

32) In local object if u try to check *null* of a local object before the initialize of it is called => will cause compilation

error that variable might not have been initialized, you can overwrite this problem by initializing an object with *null* value.

33) Summary of accessibility modifiers for members:

Modifiers	Members
public	Accessible everywhere.
Protected	Accessible by any class in the same package as its class, and accessible only by subclasses of its class in other packages.
default(no modifier)	Only accessible by classes, including subclasses, in the same package as its class(package accessibility).
private	Only accessible in its own class and not anywhere else.

More restrictive ↓

Access Modifier	Its own class	Class in Same Package	Subclass in Same Package	Subclass in Different Package	Class in Different Package
public	Yes	Yes	Yes	Yes	Yes
protected	Yes	Yes	Yes	Yes	No
default	Yes	Yes	Yes	No	No
private	Yes	No	No	No	No

34) *final* method **CANNOT** be abstract and vice versa.

35) *final* method **CANNOT** be overridden.

36) *final* variables must be initialized before being used even it is member variable (i.e. take the default value), no default value applied for local final variables.

37) You may not change a final object reference variable.
Example:
```
final Date d  = new Date();
       Date d1 = new Date();
                d = d1;                    // Illegal
```

38) You may change data owned by an object that is referred to by a final object reference variable.
Example:
```
final walrus w1 = new walrus(1000);
w1.height = 1800;
```

39) Static method may not be overridden to be non-static and vice versa, i.e. overriding static methods **MUST** remain static & non-static **MUST** also remain non-static.

40) You can specify a block of code to be *static*.
Example:
```
static { static int x = 1 }
```

41) Summary of other modifiers for members:

Modifiers	Variables	Methods
static	Defines a class variable.	Defines a class method.
Final	Defines a constant.	The method cannot be overridden.
abstract	Not relevant.	No method body is defined; its class is then implicitly abstract.
synchronized	Not relevant.	Methods can only be executed by one thread at a time.
native another	Not relevant.	Declares that the method is implemented in language.
transient	This variable's value will not be	Not applicable.

	persistent (do not need to be saved) if its object is serialized.	
volatile	The variable's value can change asynchronously; the compiler should not attempt to optimize it, i.e. signal the compiler that the designated variable may be changed by multiple threads and that it cannot take any shortcuts when retrieving the value in this variable.	Not applicable.

42) When you declare a return primitive type from a method you can return less number of bits:

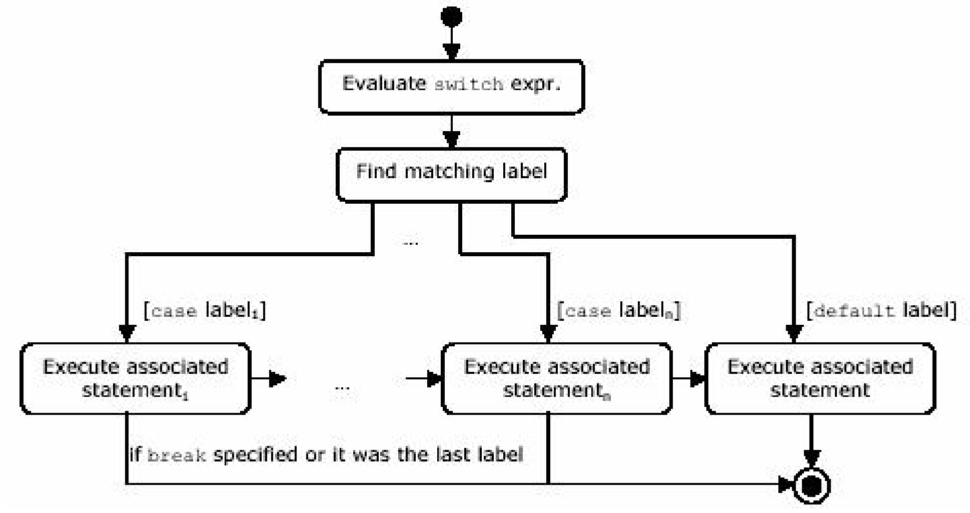
Return type	Can return
short	byte – short
int	byte – short – int
float	byte – short – int – long – float
double	byte – short – int – long – float – double

43) Instance variables may not be accessed from static methods.

44) The scope (visibility) of local variables is restricted to the code block in which they are declared.

Chapter 4: Flow Control and Exception handling

1. The rule of matching an else clause is that an else clause always refers to the nearest if which is not already associated with another else clause.
2. The compiler always checks for unreachable code, and give "Statement not reachable" error.
   Example:
   ```
   for (int i = 0; i < 10; i++) {
       continue;
       System.out.println("Hello" + i); // Statement not reachable
   }
   ```
3. The compiler always checks for that all paths that will initialize local variables before they are used.
4. State Diagram for switch statement: [I changed the diagram a little]


Evaluate switch expr.
Find matching label
...
[case label₁]
[case labelₙ]
[default label]
Execute associated statement₁
...
Execute associated statementₙ
Execute associated statement
if break specified or it was the last label

5. In the switch statement:
   a. The *case* labels are **CONSTANT** expressions whose values must be **UNIQUE**. [When one friend tried identical case labels it compiled !!!! but several books says that they must be unique, if you find any declaration please send]
   b. Constants in case statements can be integer literals, or they can be variables defined as *static* and *final*.
   c. The type of the integral expression must be *char*, *byte*, *short* or *int* (All primitives that implicit cast to *int*).
   d. The type of the case label **CANNOT** be *boolean*, *long* or floating point.
   e. The compiler **CHECKS** that the constant is in the range of the integer type in the *switch* statement, i.e. if you are using a *byte* variable in the *switch* statement, the compiler will object if it finds *case* statement constants outside –128 to 127 range that a *byte* primitive can have.
   f. The associated statement of the case label can be a list of statements which need not be a statement block.
   g. The labels (including the *default* label) can be specified in any order in the *switch* body.
   h. If it doesn't have a *break* statement during execution when we reach the condition all cases after it is executed.
   i. If the condition matches a case value it will perform all code in the *switch* following the matching *case* statement until a *break* statement or the end of the *switch* statement is encountered.
   j. If there is no *default* statement and no exact match, execution resumes after the *switch* block of code.
   k. The code block can have another *switch* statement, i.e. *switch* statement can be nested.
   l. The code block associated with a *case* **MUST** be complete within the *case*, i.e. you can't have an *if-else* or loop structure that spreads across multiple *case* statements.
6. Label rules:
   Identifiers used for labels on statements do not share the same namespace as the variables, classes, and methods

of the rest of a Java program. The naming rules, as far as legal characters, are the same as for variables except that labels are always terminated with a colon (there can be a space between the name and the colon). You can reuse the same label name multiple points in a method as long as one usage is not nested inside another. Labels cannot be freestanding, i.e. they must be associated with a statement.

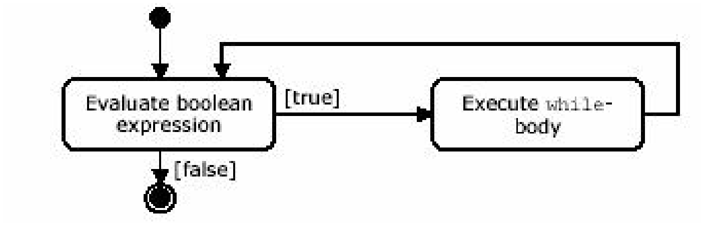
7. *break* statement immediately terminates the loop code block, and can be used with an optional identifier which is the label of an enclosing statement => control is then transferred to the statement following this enclosing labeled statement.
   Example:
```
class LabeledBreakOut {
   public static void main(String args[]) {
      int[][] squareMatrix = {{4, 3, 5},{2, 1, 6},{9, 7, 8}};
      int sum = 0;

      outer: // label
      for (int i = 0; i < squareMatrix.length; i++ ) {  // (1)
         for ( int j = 0; j < squareMatrix[i].length; j++) { // (2)
            if ( j == i )
               break; // (3) Terminate this loop control to (5)
            System.out.println( "Element[" + i + ", " + j + "]:" +  squareMatrix[i][j]);
            sum += squareMatrix[i][j];
            if (sum > 10)
               break outer; // (4) Terminate both loops control to (6)
         } // (5) Continue with the outer loop
      } // end outer loop
      // (6) Continue here
      System.out.println("sum: " + sum);
   }
}
```

8. *break* statement can be used in:
   a. Labeled blocks.
   b. Loops (*for, while, do-while*).
   c. *switch* statement.

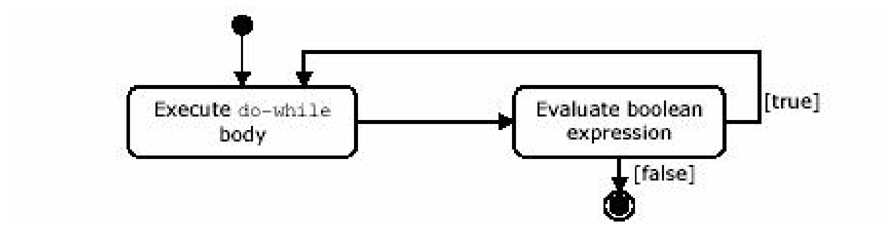
9. *continue* statement skips any remaining code in the block and continues with the next loop iteration, and can be used with an optional identifier which is the label of an arbitrary enclosing loop => Control is then transferred to the end of that enclosing labeled loop.
   Example:
```
class LabeledSkip {
   public static void main(String args[]) {
      int[][] squareMatrix = {{4, 3, 5},{2, 1, 6},{9, 7, 8}};
      int sum = 0;

      outer: // label
      for (int i = 0; i < squareMatrix.length; i++ ) {  // (1)
         for ( int j = 0; j < squareMatrix[i].length; j++) { // (2)
            if ( j == i )
               continue; // (3) Control to (5)
            System.out.println( "Element[" + i + ", " + j + "]:" +    squareMatrix[i][j]);
            sum += squareMatrix[i][j];
            if (sum > 10)
               continue outer; // (4) Control to (6)
         } // (5) Continue with the outer loop
      } // end outer loop
```

```
        // (6) Continue here
        System.out.println("sum: " + sum);
    }
}
```

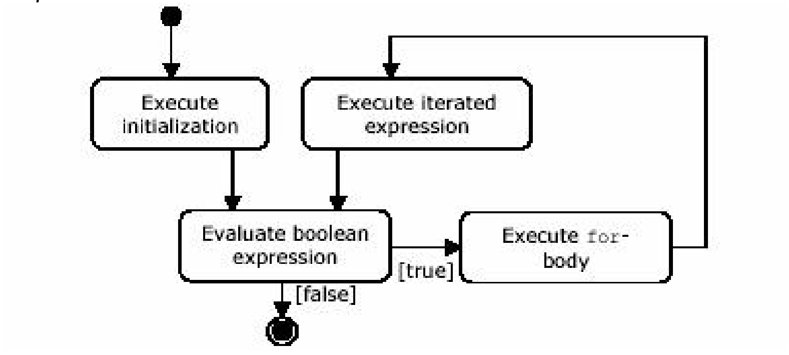
10. *continue* statement can be used ONLY in loops:
    for, while, do-while
11. *while* statement:



12. Any variable used in the expression of while loop must be declared before the expression evaluated.
13. *do-while* statement:



14. *for* statement:



15. **None** of the for-loop sections are required for the code to compile, i.e. everything in a *for* loop is optional.
16. All the sections of the for loop are independent of each other. The three expressions in the for statement doesn't need to operate on the same variables. In fact, the iterator expression does not even need to iterate a variable; it could be a separate Java command.
    Example:
    ```
    for (int x1 = 0; x1 < 6; System.out.println("iterate" + x1))
        x1 += 2;
    ```

    Output:
    ```
    iterate2
    iterate4
    iterate6
    ```
    NOTE: Most of who study for the certification solved the above example wrong, and say it just iterate till iterate4 only, in fact this is wrong, look to (14) for tracing help.

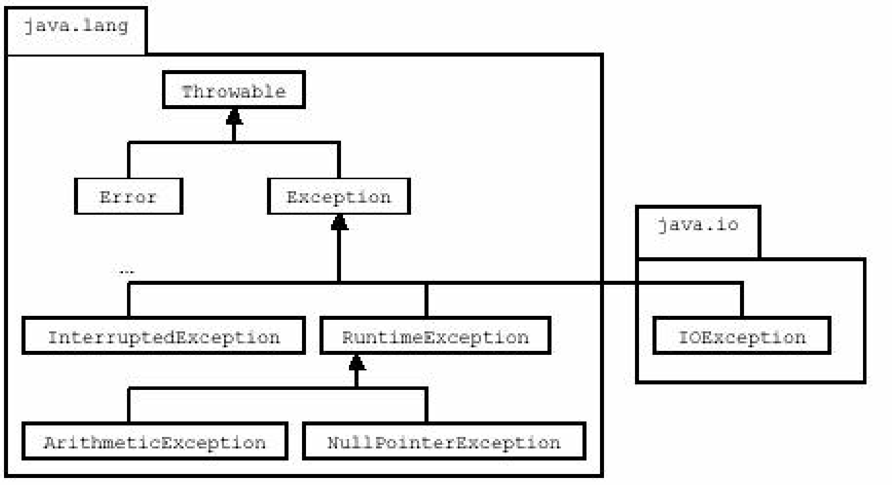
17. In the initialization part in the for loop, it is legal to mix expression with expressions, or variable declaration with variable declaration, BUT it is illegal to mix variable declaration with expressions.
    Example:
    for (int x1 = 0, x2 = 0; x1 < 15; x1++) {}   // valid

    int k = 0;
    for (System.out.println("Initial"), k = 1; k < 10; k++) {} // valid

    for (int j = 0, System.out.println("Begin"); j < 10; j++) {} // NOT VALID
18. It doesn't matter whether you pre-increment or post-increment the variable in the iterated expression in the for loop. It is always incremented after the loop executes and before the expression is evaluated.
19. *return* statement:

Form of return statement	In void method	In non-void method
return;	Optional	Not allowed
return <expression>;	Not allowed	Mandatory

20. Partial Exception inheritance hierarchy:



21. *try-catch*-finally:
    a. Block notation is MANDATORY.
    b. When an exception or error is thrown, the JVM works back through the chain of method calls that led to the error, looking for an appropriate handler to catch the object, if no handler is found, the Thread that created the error or exception dies.
    c. For each try block there can be zero or more catch blocks but only one finally block.
    d. The catch blocks & finally block must appear in conjunction with a try block, and in the above order.
    e. A try block must be followed by either at least one catch block or one finally block.
    f. Each catch block defines an exception handler, and the header takes exactly one argument, which is the exception, its block willing to handle.
    g. The exception must be of the Throwable class or one of its subclasses.
    h. When an exception is thrown, java will try to find a catch clause for the exception type. If it doesn't found one, it will search for a handler for a super type for the exception.
    i. The compiler complains if a *catch* block for a superclass exception shadows the catch block for a subclass exception, as the *catch* block of the subclass exception will never be executed, so the order of the *catch* clauses must reflect the exception hierarchy, with the most specific exception first.
    j. The *finally* block encloses code that is always executed at some time after the *try* block, regardless of whether an exception was thrown, it executed after the *try* block in case of no *catch* block or after the *catch*

block if found, EXCEPT in the case of exiting the program with *System.exit(0);* .

k. Even if there is a *return* statement on the try block, the finally block will be *executed* after the *return* statement.

l. If a method doesn't handle an exception the finally block is *executed* before the exception is propagated.

22. Subclasses of Error are used to signal errors that are usually fatal and are not caught by catch statements in the program.

23. *throws* clause:

The exception type specified in the *throws* clause in the method header can be a superclass type of the actual exceptions thrown.

24. A subclass can override a method defined in its superclass by providing a new implementation, but the method definition in the subclass can only specify all or subset of the exception classes (including their subclass) specified in the *throws* clause of the overridden method in the superclass else it will give compilation error.

25. Runtime exceptions are referred to as unchecked exceptions because the compiler does not require explicit provision in the code for catching them. All other exceptions, meaning all those that **DO NOT** derive from *java.lang.RuntimeException*, are checked exceptions because the compiler will insist on provisions in the code for cathing them. A checked exception must be caught somewhere in your code. If you use a method that throws a checked exception but do not catch this checked exception somewhere, your code will not compile.

26. Each method must however either handle all checked exceptions by supplying a *catch* clause or list each unhandled exceptions as a thrown exceptions in the *throws* clause.

27. To throw your exception you just use the *throw* keyword with an instance of an exception object to throw, and you must caught this thrown exception if this exception is checked but if it is runtime exception or unchecked exception you needn't to catch.

28. If you want to handle an exception in more than one handler you can re-throw the exception, and the *throw* statement must be the **LAST** line of your block because any line under it is unreachable.

29. Runtime exceptions are a special case in Java. Because they have a special purpose of signaling events that happen at runtime, usually as the result of a programming error, or bug, they do not have to be caught. If not handled they terminate the application.

30. If class extends *Exception* => class represent checked exception, but if the class extends *RuntimeException* it mean it is unchecked.

31. *getMessage()* method in the *Throwable* class prints the error message string of this *Throwable* object if it was created with an error message string; or *null* if it was created with no error message.

32. *toString()* method returns a short description of this *Throwable* object. If this *Throwable* object was created with an error message string, then the result is the concatenation of three strings:
    a. The name of the actual class of this object.
    b. ": " (a colon and a space).
    c. The result of the *getMessage()* method for this object.

    If this *Throwable* object was created with no error message string, then the name of the actual class of this object is returned.

33. *printStackTrace()* method in the *Throwable* class prints *this Throwable* and its backtrace to the standard error stream. This method prints a stack trace for this *Throwable* object on the error output stream that is the value of the field *System.err*. The first line of output contains the result of the *toString()* method for this object. Remaining lines represent data previously recorded by the method *fillInStackTrace()*. The format of this information depends on the implementation, but the following example may be regarded as typical:

    Example:

    ```
    java.lang.NullPointerException
    at MyClass.mash(MyClass.java:9)
    at MyClass.crunch(MyClass.java:6)
    ```

at MyClass.main(MyClass.java:3)

34. fillInStackTrace() method fills in the execution stack trace. This method records within this Throwable object information about the current state of the stack frames for the current thread. This method is useful when an application is re-throwing an error or exception.
    Example:

```
try {
   a = b / c;
} catch(ArithmeticThrowable e) {
   a = Number.MAX_VALUE;
   throw e.fillInStackTrace();
}
```