A Concentrated Guide

To

Sun Certified Programmer for Java 2 platform exam (310-025)

Collected, prepared & organized by:
Eng.\ Ashraf Fouad Ayoub
Eng.\ Ashraf Samir Helmy
Cairo, Egypt

Document version control

Date	Author	Version	Comment
12/09/2001	Ashraf Fouad	1.0 beta	Creation

Note To Holders:
If you receive an electronic copy of this document and print it out, please write your name on the equivalent of the cover page, for document control purposes.

If you receive a hard copy of this document, please write your name on the front cover, for document control purposes.

Introduction

To all who downloaded this document, and preparing for sun certification programmer exam, I think this document will help you a lot, because:

1- It collects its information mainly from the best 3 certification preparation books as mentioned by most certified individuals and previews in www.amazon.com :
   (a) "A programmer's guide to Java Certification"
   (b) "The complete Java 2 certification study guide"
   (c) "Java 2 exam cram"
2- Supplied with examples, diagrams.
3- Supplied with tips for most common exam mistakes.
4- Well organized as the exam objectives.
5- Concentrate on the certification material, so you can study from it, and if you need to know more about a topic or don't have previous experience in it, you don't have to buy a certification book, but any Java book will do the trick.
6- It was meant to be a complete document, no need for other, it target programmers with min. 2 to 4 month of experience in java.
7- Can be also used as a quick revision on the exam points as it is nearly 100 pages.
8- Can be used as a course material for teaching courses in Java certification as I intend to do in my company.

I (Ashraf Fouad) studied for the certification and passed the exam scoring 93%; nothing in the exam was outside the document, but I think I needed to solve more exams than I did. Also one of my friends studied from it and scored 84% and he told me that nothing was needed more than this document. I wish that as many can also contribute in supplying more tips than I mentioned, so I intend to keep receiving comments from you till the end of this year before making its final release.

This document is supplied in .pdf format with the name scpj2vX.pdf where X indicates document version number. It will come in a zip file named scpj2vX.zip in addition to a doc file named scpj2qavX.doc for my exam questions I met during preparation which I collected for the following reasons:

1- Hard or tricky question.
2- It came in several exams so you must have an exercise on such question type.
3- Type of question that you don't expect to be asked in exam, but they ask, such as asking about Object Oriented concepts.

Authors:

Eng.\Ashraf Fouad Ayoub
Web developer
Alexandria University
Faculty of Engineering
Department of computer science and automatic control
(1999)

Eng.\Ashraf Samir Helmy
Web developer
Ain Shams University
Faculty of Engineering
Department of computer and system engineering
(1999)

For feedback, please mention the following:

For correction:
Mail To*: ashraf_fouad76@yahoo.com
Mail subject*: SCPJ2 document correction
In the mail body:
Document version*: ...
Error in Tip number*: ...
Why error*: ...
Correction*: ...
Reference(If any): ...
If you wish your email is published among the contributors of the documents (Y/N): default (Y)

For commenting:
Mail To*: ashraf_fouad76@yahoo.com
Mail subject*: SCPJ2 document comment
In the mail body:
Document version*: ...
Comment*: ...
If you wish your email & opinion is published among the previews of the documents (Y/N): default (Y), this will occur only when I make my site & this document will be part of it.

For adding hints:
Mail To*: ashraf_fouad76@yahoo.com
Mail subject*: SCPJ2 document add hints
In the mail body:
Document version you have*: ...
Suggested chapter to add the hint*: ...
Hint*: ...
Reference*: ...
If you wish your email is published among the contributors of the documents (Y/N): default (Y)

For receiving update of the document:
Mail To*: ashraf_fouad76@yahoo.com
Mail subject*: SCPJ2 document subscribe.

The above are the minimum to send, send more and speak freely as you wish.

For the questions document, I don't want to receive more questions as the Internet is filled with enormous already, but I can receive comments to be added for further declaration, or if anyone doesn't know explanation of the answers provided and needs more. So please follow the following direction for sending your comments about questions document.

Mail To*: ashraf_fouad76@yahoo.com
Mail subject*: SCPJ2QA document
In the mail body:
Document version*: ...
Question number*: ...
What is your question*: ...
Anything you want to add(If any): ...
Reference(If any): ...
If you wish your email is published among the contributors of the documents (Y/N): default (Y)

Import Resources:
One of the most important resources to me was www.yahoogroups.com, it contains several groups for java certification, I encourge subscribing in what you are interested in, but my advice is to browse the mails through the web not to receive individual or digested mails, because they are very active groups. I mention hear their names and their description in yahoogroups, and I will keep always the newest version of this document on the files section of jcertification group mentioned below.

jcertification-subscribe@yahoogroups.com
Only e-Group for preparation of certification for java programmers. Experts for answering the questions and discussions. Feel free to post your questions and get the best answers. Ask anything regarding Java Certification. Register today and tell a friend about this.

javacertstudy-subscribe@yahoogroups.com
JavaCert.com is an online study community for Sun's Java certification exams.

sunjavacert-subscribe@yahoogroups.com

The list basically provides tips for Java aspirants. A list with information on Java Certification, SCJP, SCJA , SCJD , tips , books , tutorials, downloads, lists, mock and online exams. A list that can give u that extra niche to get a high score in Java Certification.
scjd-subscribe@yahoogroups.com
The main purpose of this list is to discuss issues/concerns/problems/solutions to your "Sun Certified Java Developer" assignment.

scjea-subscribe@yahoogroups.com
This mailing list will discuss the Sun enterprise architect exam using J2EE. This is a moderated list and any message not relevant to it would be deleted. Please check links page before you post questions on the list. I have about 8 years experience in IT ..I am a MCSD (old and new course) , SCJP 1.1 and SCJD 2.0 certified professional. Of all the exam I have given I found the SCJD 2.0 developer exam challenging and I am happy to inform you that I passed it with flying colors (149 / 155 marks).

java-dev-test-subscribe@yahoogroups.com
We are writing code to pass SCJD exam. Our members are from all over the world. We have the best Java developer certification website on earth. We all work together to make a difference!

URLs:
http://suned.sun.com/US/certification/java/java_progj2se.html
http://www.geocities.com/korayguclu/
http://javaquestion.tripod.com/
http://www.anilbachi.8m.com/
http://www.jttc.demon.co.uk/javacert.htm
http://www.jchq.net/
http://members.spree.com/education/javachina/Cert/FAQ_SCJP.htm
http://www.levteck.com/
http://www.software.u-net.com/javaexam/NotCovered.htm
http://www.jaworski.com/java/certification/
http://www.lanw.com/java/javacert/HardestTest.htm
http://www.lanw.com/java/javacert/TestApplet6.htm
http://www.angelfire.com/or/abhilash/Main.html
http://jquest.webjump.com/
http://www.go4java.20m.com/
http://indigo.ie/~dywalsh/certification/index.htm
http://www.michael-thomas.com/
http://www.acmerocket.com/skinny/
http://www.javacaps.com/

Chapter 1:

Language fundamentals

1.1) Java is **case-sensitive**. [3]

1.2) Source file may contain **ONLY ONE** public class or interface, and any numbers of default classes or interfaces. [3]

1.3) 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. [3]

1.4) The file name may begin with numbers if there is no `public` class on it. [3]

1.5) If no `package` is explicit declared, java places your classes into a default package (`Object`). [3]

1.6) Identifiers are composed of characters, where each character can be either letter (including Ŭ Ń ù Ă Ć Å), a connecting punctuation (underscore _) or any currency symbol (such as $ ¢ £ ¥) and **CANNOT** start with a digit. [1]

1.7) 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: [1] [3]

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 [8], but I didn't find it in any certification book.			

The following table shows three reserved predefined literals: [9]

null	true	false

The following table shows reserved keywords **NOT** currently in use: [9]

const	goto		
byvalue	generic	outer	I found this in reference [8], but I didn't find it in any certification book.
cast	inner	rest	
future	operator	var	

1.8) 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. [1]

1.9) Floating point literal by default is `double`, you can specify to be a `float` by appending "`F`" or "`f`" as suffix.[1]

1.10) Octal numbers are specified with "`0`" as prefix, Hexadecimal numbers are specified with "`0x`" or "`0X`" as prefix.[1]

1.11) Most important Unicode values: [1]

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

1.12) Escape sequences are used to define special character values and can be represented also in Unicode value, the following table shows them: [1]

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.

1.13) The single apostrophe ' need not to escaped in Strings, but it would be if specified as a character literal '\''. [1] [8]
Example:
```
String tx = "Delta values are labeled \"\u0394\" on the chart.";
```

1.14) Regardless of the type of comment, it can't be nested. [1]

1.15) Default values for member variables table: [1]

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

1.16) `static` variables in a class are initialized to default values when class is loaded if they are not explicitly initialized. [1]

1.17) Instance variables are initialized to default values when the class is instantiated if they are not explicitly initialized. [1]

1.18) Local variables (Automatic) are **NOT** initialized when they are instantiated at method invocation. The compiler javac reports use of uninitialized local variables. [1]

1.19) There can be **ONLY** one package declaration in a Java file, and if it appears, it must be the first non-comment statement. [8]

1.20) The JVM expects to find a method named main with the signature as follows: [8]
```
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

2.1) Operator precedence and associativity: [1]

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	`= += -= *= /= %= <<= >>= >>>= &= ^= |=`

2.2) Casting between primitive values and references **CANNOT** be applied. [1]

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

2.4) The modulo operator % give the value of the remainder of the division of the left operand (dividend) by the right operand (divisor). [3] [8]

2.5) 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). [3]
Example:

```
int x = -5 % 2;     // x = -1
int y = -5 % -2;    // y = -1
int z = 5 % -2;     // z = 1
```

2.6) `||` and `&&` work with boolean not with integers like `&` and `|`, and the result from `| - &` is `int`. **NOTE:** `&` and `|` are used with both boolean and integers. [3]

2.7) Unlike in C, integers in Java can **NEVER** be interpreted as boolean values, so expressions used for flow control **MUST** evaluate to boolean. [8]

2.8) The conditional assignment operator, the **ONLY** Java operator that takes three operands:? [1] [3] [8]
<condition>? <expression1>:<expression2>
Example:

```
String x = (salary < 1500)? "Poor": "Not poor";
String y = (salary > 1500)? "Poor": (salary1 < 10)? "poor1":"poor2";
```

2.9) Multiple assignment: [1]

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

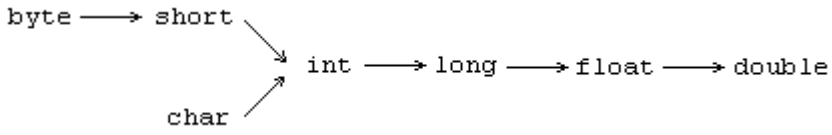
2.10) Boolean values **CANNOT** be casted to other data values, and vice versa, the same applies to the reference literal `null`, which is **NOT** of any type and therefore **CANNOT** be casted to any type. [1]

2.11) Conversion is done when: [8]
(a) Assigning a value to a primitive variable.
(b) Evaluating arithmatic 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)
```

2.12) 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. [1] [8]



2.13) 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. [8]

2.14) All six numbers types in Java are signed meaning they can be negative or positive. [3]

2.15) The **ONLY** integer primitive that is not treated as a signed number is `char`, which represents a Unicode character. [8]

2.16) All conversion of primitive's data types take place at compile time. [3]

2.17) Arithmetic operations:

(a) For unary operators if `byte` – `short` - `char` → converted to `int`.

(b) For Binary operands:

1. If one of operands is `double` the other operand is converted to a `double`.
2. If one of the operand is `float`, the other operand is converted to `float`.
3. If one of the operand is `long`, the other operand is converted to `long`.

(c) Else both the operands are converted to `int`.

2.18) Ranges of primitive data types: [1] [3]

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	1.4012984632481707e-45	3.4028234663852886e+38
double	64	8	4.94065645841246544e-324	1.79769313486231570e+308

2.19) Depending on the storing type of the arithmetic operation the precision is done. [3]

Example:

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

2.20) The compiler pays attention to the known range of primitives. [8]

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.

2.21) Important examples for arithmetic expression evaluation: [1]

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`

2.22) `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. [8]

2.23) `Float.NaN, Double.NaN` are considered non-ordinal for comparisons, this means all that are `false`: [3]
```
x < Float.NaN
x == Float.NaN
```
But you can test by `Float.isNaN(float f)`, `Double.isNan(double d)`.

2.24) 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. [8]

2.25) *<variable> <op>=<expression>*
is equivalent to
<variable> = (<variable type>) (<variable><operator>(<expression>)). [1]

2.26)
```
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: [1]
```
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.

2.27)
```
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). [1]

2.28) If your code conducts operations that overflow the bounds of 32-bit or 64-bit integer arithmatic, 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. [1] [8]

2.29) 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 … [1]
```
SomeObjRef.make().make().make();
```

2.30) To get the 2's complement: [3]
(a) Get the 1's complement by converting 1's to 0's and 0's to 1's.

(b) Add 1.

2.31) 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]

2.32) >> 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. [6]

2.33) When you shift a bit by a numeric value greater than the size in bits, Java does a modulas shift. [6]

2.34) To shift a negative number get the 2's complement and then shift it. [3]

2.35) Object reference equality (==, !=): [1]
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
```

2.36) The equals method: [8]
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 referes 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.

2.37) 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. [3] [8]

2.38) 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. [3]
Example:

```
Object x = new Button();
```

2.39) Object reference conversion take place at compile time because the compiler has all the information it needs to determine whether the conversion is legal or not. [3]

2.40) Short circuit evaluation: [1] [8]

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.

2.41) Parameter passing: All parameters are passed by value. [1]

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

2.42) 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. [1]

Example:

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

Chapter 3:

Declarations and Access Control

3.1) 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. [8]

3.2) 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. [1] [3]

3.3) When constructing multi-dimensional arrays with the `new` operator, the length of the deeply nested arrays may be omitted, these arrays are left unconstructed. [1]
Example:
```
double matrix[][] = new double[3][];
```

3.4) `length` of array object is a variable **NOT** a method. [3]

3.5) It is legal to specify the size of an array with a variable rather than a literal. [3]

3.6) The size of the array is fixed when it is created with the `new` operator or with special combined declaration and initialization. [8]

3.7) Anonymous arrays: [1]
new <elementType>[] {<initialization code>}
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];
        }
    }
}
```

3.8) 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:** Intialization by means of a bracketed list can be used only in the statement that declares the variable. [8]

3.9) 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. [1]

3.10) 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. [8]
```
int sizes[] = {4, 6, 8, 10}; //(1)
Object obj = sizes;          //(2)
int x = ((int[])obj)[2];     //(3)
```

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.[8]
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;
```

3.12) An `import` declaration does not recursively import sub-packages. [1]

3.13) The order of modifiers in class declaration: [3]

(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)

3.14) 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. [1]

3.15) The **ONLY** access modifier allowed to the top level class is `public` or friendly. [3]

3.16) `abstract` modifier implies that the class will be extended, but `abstract` class **CANNOT** be instantiated. [3]

3.17) 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`. [1]

3.18) it is **NOT** a **MUST** for an `abstract` class to have a `abstract` method. [10]

3.19) Interfaces as classes **CANNOT** be declared `protected, private, native, static, synchronized`. [3] [8]

3.20) 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. [8]
Example:

```
public interface RunObs extends Runnable, Observer
```

Any class implementing this interface will have to provide methods required by both `Runnable` and `Observer`.

3.21) The order of modifiers in method declaration: [3]

(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)

3.22) `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. [3]

3.23) The class must be declared `abstract` if: [3]
(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.

3.24) When abstract class implement interface there is no need to this class to implement all members of the interface. [3]

3.25) 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. [1]

3.26) `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`. [1]

3.27) The order of modifiers in variable declaration: [3]
(a) `public` or `private` or `protected`. (optional)
(b) `final` or `static` or `transient` or `volatile`. (optional)
(c) ***variable type***. (mandatory)
(d) ***variable name***. (mandatory)

3.28) **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. [1]
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);
    }
}
```

3.29) **ONLY** variables, methods and inner classes may be declared `protected`. [3]

3.30) `static` members can be called from the member objects. [3]
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.

3.31) Trying to use object class member before the constructor of the object class member called will compile fine but will give `NullPointerException`. [3]
Example:

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

3.32) 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. [3]

3.33) Summary of accessibility modifiers for members: [1] [3] [5]

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

3.34) `final` method **CANNOT** be `abstract` and vice versa. [1]

3.35) `final` method **CANNOT** be overridden. [3]

3.36) `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. [3]

3.37) You may not change a final object reference variable. [3]
Example:

```
final Date d  = new Date();
      Date d1 = new Date();
            d = d1;          // Illegal
```

3.38) You may change data owned by an object that is referred to by a final object reference variable. [3]
Example:

```
final walrus w1 = new walrus(1000);
w1.height = 1800;
```

3.39) 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. [3]

3.40) You can specify a block of code to be `static`. [3]
Example:

```
static { static int x = 1 }
```

3.41) Summary of other modifiers for members:[1] [8]

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`	*Not relevant.*	Declares that the method is implemented in another language.
`transient`	This variable's value will not be persistent (do not need to be saved) if its object is serialized.	*Not applicable.*
`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.*

3.42) When you declare a return primitive type from a method you can return less number of bits: [3]

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

3.43) Instance variables may not be accessed from static methods. [3]

3.44) The scope (visibility) of local variables is restricted to the code block in which they are declared. [8]

Chapter 4:

Flow Control and Exception handling

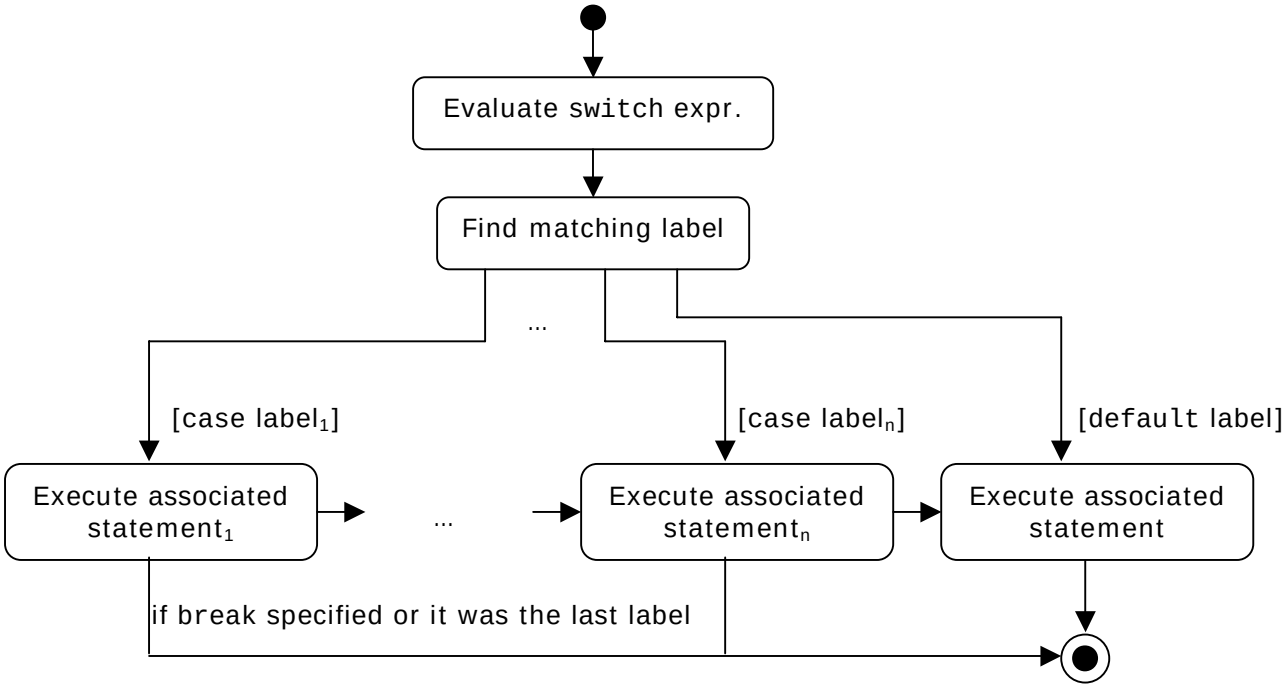
4.1) The rule of matching an `else` clause is that an `else` clause always refers to the nearest `if` which is not already associated with another `else` clause.[1]

4.2) The compiler always check for unreachable code, and give "`Statement not reachable`" error. [3]
Example:

```
for (int i = 0; i < 10; i++) {
    continue;
    System.out.println("Hello" + i); // Statement not reachable
}
```

4.3) The compiler always check for that all paths that will initialize local variables before they are used. [8]

4.4) State Diagram for `switch` statement: [1] [**I changed the diagram a little**]



4.5) In the `switch` statement: [1] [8]

(a) The case labels are **CONSTANT** expressions whose values must be **UNIQUE**.
(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.

4.6) Label rules: [8]
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.

4.7) `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. [1] [8]
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);
    }
}
```

4.8) `break` statement can be used in: [1]
(a) Labeled blocks.
(b) Loops (`for`, `while`, `do-while`).
(c) `switch` statement.

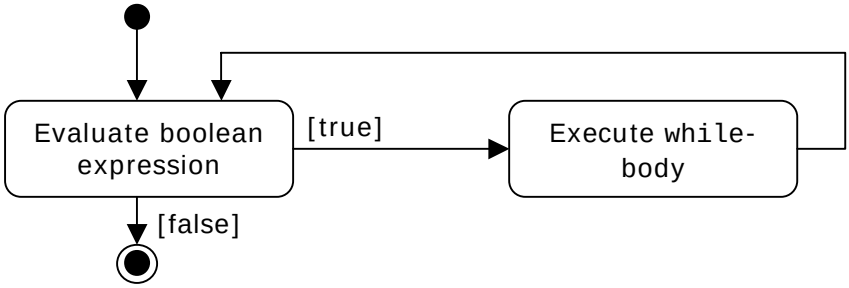
4.9) `continue` statement skips any remaining code in the block and continues with the next loop iteration, and can be used with an optional identifier which is the label of an arbitrary enclosing loop ⇒ Control is then transferred to the end of that enclosing labeled loop. [1] [8]
Example:

```
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);
        }
    }
```

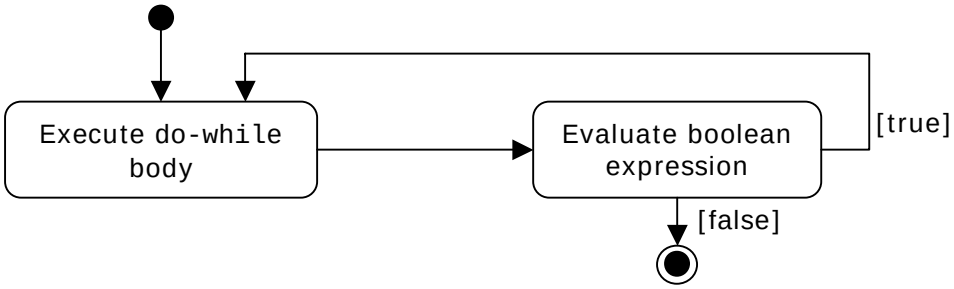
4.10) continue statement can be used **ONLY** in loops: [1]
for, while, do-while

4.11) while statement: [1]

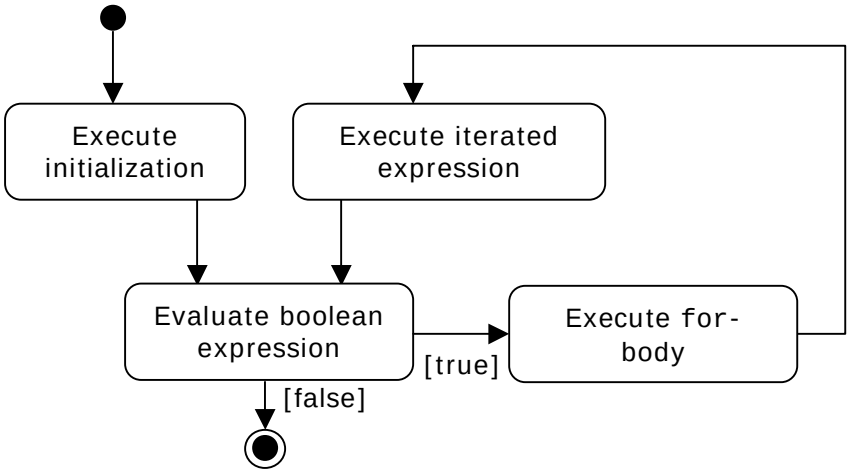


4.12) Any variable used in the expression of while loop must be declared before the expression evaluated. [3]

4.13) do-while statement: [1]



4.14) for statement: [1]



4.15) **None** of the for-loop sections are required for the code to compile, i.e. everything in a for loop is optional. [3] [8]

4.16) 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. [3]
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 (4.14) for tracing help.

4.17) In the `for` loop, it is legal to mix expression with variable declaration. [3]
Example:

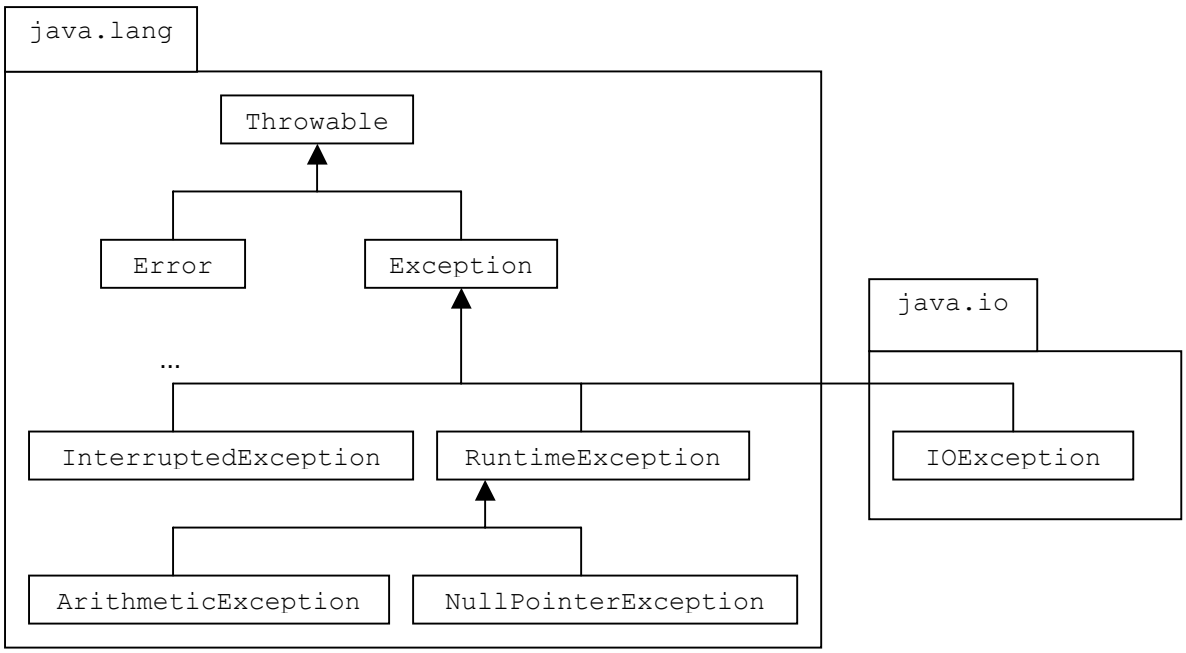
```
for (int x1 = 0, x2 = 0; x1 < 15; x1++) {}
```

4.18) 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. [3]

4.19) `return` statement: [1]

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

4.20) Partial Exception inheritance hierarchy: [1]



4.21) `try-catch-finally`: [1] [3] [8]
(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.

4.22) Subclasses of `Error` are used to signal errors that are usually fatal and are not caught by `catch` statements in the program. [8]

4.23) `throws` clause: [1]
The exception type specified in the `throws` clause in the method header can be a superclass type of the actual exceptions thrown.

4.24) 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. [1]

4.25) 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. [3] [8]

4.26) 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. [3]

4.27) 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. [3]

4.28) 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. [3]

4.29) 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. [3]

4.30) If class extends `Exception` ⇒ class represent checked exception, but if the class extends `RuntimeException` it mean it is unchecked. [3]

4.31) `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. [2]

4.32) `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: [2]

(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.

4.33) `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: [2]
Example:

```
java.lang.NullPointerException
         at MyClass.mash(MyClass.java:9)
         at MyClass.crunch(MyClass.java:6)
         at MyClass.main(MyClass.java:3)
```

4.34) `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. [2]
Example:

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

Chapter 5:

Object Oriented programming

5.1) `Object` is the highest-level java class and all classes are subclass of the `Object` class. [3]

5.2) One class may only be a subclass of another class or an implementation of interface(s) if this class of a relation (is a) of the superclass. [3]

5.3) Constructing a relationship of objects: [3]
(a) is a ⇒ superclass.
(b) has a ⇒ member variables.
Example:
A home is a house and has a family.

5.4) Methods are overloaded when there are multiple methods in the same class with the same name but with different unique parameter list. [3]

5.5) The number, type, and order of the input parameters plus the method name determines the signature. [8]

5.6) As return type, visibility, throws exceptions, keywords are **NOT** part of the signature, changing it is **NOT ENOUGH** to overload methods. [1] [3]

5.7) Java always chooses the overloaded method with the closest matching parameter list, the less to cast. [3]

5.8) Why cannot `final`, & `private` be overridden? [1]
`final` : because `final` prevents method overriding.
`private` : means that it is not accessible outside the class in which it is defined therefore a subclass cannot override it.

5.9) A subclass may override non-static methods inherited from the superclass, noting the following aspects: [1] [3]
(a) A new method definition **MUST** have the **SAME** method signature (method name and parameters types not essential the parameters name) and the **SAME** return type.
(b) The new method definition, in addition, **CANNOT** '**NARROW**' the accessibility of the method, but it can **'WIDEN**' it, i.e. can't replace with weaker access privileges.
(c) The new method definition in the subclass can only specify all or a subset of the exception classes (including their subclasses) specified in the `throws` clause of the overridden method in the superclass.
(d) Whether the parameters in the overriding method should be `final` is at the discretion of the subclass. A method's signature does not encompass the `final` modifier of parameters, only their types and order.
(e) An overridden method and a method in the superclass may declare their methods `synchronized` independently of one other.

5.10) When a *method* is invoked on an object using a reference, it is the class of the current object denoted by the reference, **NOT** the type of the reference.
When a *variable* of an object is accessed using a reference, it is the type of the reference, **NOT** the class of the current object denoted by the reference. [1]
References to member methods are resolved at runtime using the type of the object.
References to member variables are computed at compile time using the type of the reference. [8] [**NOTE, I found in one of the exams [ref. 13] that this differs for static methods, and the superclass method will be always executed**]
Example:

```
// Exceptions
class InvalidHoursException extends Exception {}
class NegativeHoursException extends InvalidHoursException {}
class ZeroHoursException extends InvalidHoursException {}
class Light {
    protected String billType = "Small bill";
    protected double getBill(int noOfHours)
```

```
        throws InvalidHoursException {
            double smallAmount = 10.0,
                   smallBill = smallAmount * noOfHours;
            System.out.println(billType + ": " + smallBill);
            return smallBill;
    }
}
class TubeLight extends Light {
    public String billType = "Large bill";
    public double getBill(final int noOfHours)
        throws ZeroHoursException {
        double largeAmount = 100.0,
                  largeBill = largeAmount * noOfHours;
        System.out.println(billType + ": " + largeBill);
        return largeBill;
    }
    public double getBill() {
        System.out.println("No bill");
        Return 0.0;
    }
}
public class Client {
    public static void main(String args[])
        throws InvalidHoursException {
        TubeLight tubeLightRef = new TubeLight();
        Light lightRef1 = tubeLightRef;
        Light lightRef2 = new Light();

        // Invoke overridden methods
        tubeLightRef.getBill(5);
        lightRef1.getBill(5);
        lightRef2.getBill(5);

        // Access shadowed variables
        System.out.println(tubeLightRef.billType);
        System.out.println(lightRef1.billType);
        System.out.println(lightRef2.billType);

        // Invoke overloaded method
        tubeLightRef.getBill();
    }
}
```

Output from the program:
Large bill: 500.0
Large bill: 500.0
Small bill: 50.0
Large bill
Small bill
Small bill
No bill

5.11) If no constructors are declared in a class, the compiler will create a default constructor that takes no parameters. [8]

5.12) **No** return type specified from the constructor. [3]

5.13) The constructor method **CANNOT** be `final`, `abstract`, `synchronized`, `native`, and `static`, and **CAN** be declared `public`, `protected`, `private`. [3] [8]

5.14) Constructors **CANNOT** be overridden; they can only be overloaded, but only in the same class, and they **CAN** have a list of thrown exceptions. [1] [8]

5.15) You **CANNOT** call constructor recursively. [3]

5.16) Java specifies that when using `this()` call & `super()`
(a) It **MUST** occur as the first statement in a constructor, followed by any other relevant statements.
(b) It **CAN ONLY** be used in a constructor ⇒ `this()` and `super()` calls **CANNOT** both occur in the same constructor. [1]

5.17) If a constructor at the end of such a `this()` - chain (which may not be a chain at all if no `this()` call is invoked) does not have explicit call to `super()`, then the call `super()` (without the parameters) is implicitly inserted to invoke the default constructor of the superclass ⇒ subclass without any default constructor will fail to compile if the superclass does not have default constructor (i.e. provides only non-default constructors). [1]

5.18) Interface is non-functional reference type class that contains constants and methods declarations but not functional methods. [3] [8]

5.19) We define a method within the interface but instead ending with `{}` end with `';'`. [3]

5.20) Interface can't implements another interface but it can extend another interface(s). [3]

5.21) The methods in an interface are all `abstract` by virtue of their declaration, and should not be declared `abstract`, an interface is `abstract` by definition & therefore **CANNOT** be instantiated. [1]

5.22) All interface methods will have `public` accessibility when implemented in the class (or its subclasses). [1]

5.23) A class can choose to implement only some of the methods of its interfaces, i.e. give a partial implementation of its interfaces ⇒ the class must then be declared as `abstract`. [8]

5.24) Interfaces methods **CANNOT** be declared `static`, because they comprise the contract fulfilled by the objects of the class implementing the interface and are therefore instance methods. [1]

5.25) There are three different inheritance relations at work when definning inheritance between classes and interfaces: [1]
(a) Linear implementation inheritance hierarchy between classes: a class extends another class.
(b) Multiple inheritance hierarchy between interfaces: an interface extends other interfaces.
(c) Multiple interface inheritance hierarchy between interfaces and classes: a class implements interfaces.

5.26) An interface can define constants, such constants are considered to be `public`, `static` and `final` regardless of whether these modifiers are specified, so interface may not however declare variables unless they are initialized (i.e. the assigning must be at the declaration statement). [1] [3]

5.27) The rule of thumb for reference values is that conversions up the hierarchy are permitted (called *upcasting*), but conversions down the hierarchy require explicit casting (called *downcasting*). In other words, conversions which preserve the inheritance is a relationship are allowed, other conversions require explicit cast or are illegal. [1]

5.28) Casting to a different type will compile fine but will throw `ClassCastException`.

5.29) What is reference type & object? [3]

Example:

```
Component c = new Button();
```

(a) `c` reference type is `Component` but as an object is `Button`.

(b) At compile time Java treat `c` as a `Component` for conversion and casting, at runtime the Java treat `c` as `Button`.

5.30) The rules for reference assignment are stated on the following code: [1]

```
SourceType srcref;
DestinationType destRef = srcRef;
```

(a) If the `SourceType` is a class type, the reference value `srcRef` may be assigned to the `destRef` reference, provided is one of the following:

- `DestinationType` is a superclass of the subclass `SourceType`.
- `DestinationType` is an interface type which is implemented by the class `SourceType`.

(b) If `SourceType` is an interface type, the reference value is `srcRef` may be assigned to `destRef` reference, provided `DestinationType` is one of the following:

- `DestinationType` is `Object`.
- `DestinationType` is a super interface of sub-interface `SourceType`.

(c) If the `SourceType` is an array type, the reference value in `srcRef` may be assigned to `destRef` reference, provided `DestiationType` is one of the following:

- `DestinationType` is `Object`.
- `DestinationType` is an array type, where the element type of the `SourceType` can be converted to the element type of `DestinationType`.

Note:

The above rules also apply for parameter passing conversions; this is reasonable, as parameters in java are passed by value, requiring that values of actual parameters must be assignable to formal parameters of compatible types.

5.31) The types of references variables can hold depened on the object hierarchy. [8]

Example:

```
Object anyRef;
String myString;
```

Because every class in Java descends from `Object`, `anyRef` could refer to any object. However, because `String` cannot be subclassed, `myString` can refer to a `String` object only.

Chapter 6:

Garbage Collection & Object Lifetime

6.1) JVMs typically run the garbage collection as a low priority Thread, which is activated when the JVM feels that it is running short of available memory. [8]

6.2) Local variable are set to `null` after use ⇒ this makes the objects denoted by the local variable eligible for garbage collection from this points onwards, rather than after the method terminates, this optimization technique should **ONLY** be used as a last resort when resources are scarce. [1]

6.3) The object is eligible for collecting after last reference that refers to it is dropped. [7]

6.4) The finalizer of the object is simply a method of an object that is called just before the object is deleted (i.e. the finalizer is the destructor of the object). [7]

6.5) All the processing in the virtual machine stops while the garbage collectors run (disadvantage). [7]

6.6) If the garbage collector didn't free enough memory then the request fails (`java.Lang.OutOfMemoryError`). [7]

6.7) The automatic garbage collection calls the `finalize()` method in an object which is eligible for garbage collection (i.e. an object is being out of scope or unreachable) before actually destroying the object, `finalize` method is an instance method which is defined in the class `Object` as `protected void finalize() throws Throwable`. [1] [3]
-A finalizer can be overridden in a subclass to take appropriate action before the object is destroyed.
-A finalizer can catch & throw exception, **HOWEVER**, any exception thrown but not caught by a `finalizer` when invoked by the garbage collection is ignored.
-In case of finalization failure, the object still remains eligible to be disposed of at the discretion of the garbage collection(unless it has been resurrected).
-The finalizer is only called **ONCE** on an object, regardless of being interrupted by any exception during its exception.

6.8) Finalizers are **NOT** implicitly chained like constructors for subclasses ⇒ a finalizer in a subclass should explicitly call the finalizer in its superclass as its **LAST** action (in a `finally` block). [1]

6.9) The `finalize` method is called directly from the system and never called by the programmer directly. If called by the programmer it acts as an ordinary method and it don't count by the JVM ⇒ so the JVM can call it when applying garbage collection. [7]

6.10) Static members are considered always to be lived objects. [7]

6.11) A `finalize` method can make object accessible again, 'resurrect' it, thus avoiding it being garbage collected, one simple technique is to assign its `this` reference to a static variable, from which it can be retrieved later. [1]

6.12) A finalizer is called **ONLY ONCE** on an object before being garbage collected ⇒ an object can only resurrected **ONCE**. [1]

6.13) Method `System.gc()` or `Runtime.getRuntime().gc()` in the `java.lang` package can be used to **SUGGEST** to the JVM that now is a good time to run the garbage collection but you **CANNOT** guarantee that all objects eligible for garbage collection will be collected). Method `System.runFinalization()` can be used to **SUGGEST** that the JVM expend effort toward running the finalizers (which have not been executed before) for objects eligible for garbage collection. [1] [8]

6.14) There are **NO** guarantees that the objects no longer in use will be garbage collected and their finalizers executed at all. Garbage collection **MIGHT NOT** even be run if the

program execution might remain allocated after program termination, unless reclaimed by the operating system or by other means. [1]

6.15) There are also **NO** guarantees on the order in which the objects will be garbage collected, or on the order in which the finalizers will be executed. Therefore, the program should not make any decisions based on these assumptions. [1] [8]

6.16) You should directly invoke the garbage collection system before entering a time critical section of code. [7]

6.17) Instance initializer expressions are executed in the order in which the instance member variable are defined in the class, the same is true for static initializer expressions ⇒ if a constant expression is used in the initialization of a member variable then all its operands must be defined before they can be used in the expression. [1]

6.18) A **LOGICAL** error can occur if the order of the initializer expressions is not correct. [1]
Example:

```
Class Hotel {
    private int NO_OF_ROOMS = 12;
    private int MAX_NO_OF_GUEST = initMaxGuests();
    private int OCCUPANCY_PER_ROOM = initOccupancy();

    private int initMaxGuests() {
        System.out.println("Occupancy_PER_ROOM: " + OCCUPANCY_PER_ROOM );
        System.out.println("MAX_NO_OF_GUEST: " + NO_OF_ROOMS *
                                                 OCCUPANCY_PER_ROOM);
        return NO_OF_ROOMS * OCCUPANCY_PER_ROOM;
    }

    public int getMaxGuests() {
        return MAX_NO_OF_GUEST;
    }

    public int initOccupancy() {
        return 2;
    }

    public int getOccupancy() {
        return OCCUPANCY_PER_ROOM;
    }
}
public class TestOrder {
    public static void main(String args[]) {
        Hotel objRef = new Hotel();
        System.out.println("After object creation: ");
        System.out.println("OCCUPANCY_PER_ROOM: " + objRef.getOccupancy());
        System.out.println("MAX_NO_OF_GUEST: " + objRef.getMaxGuests());
    }
}
```

Output:
OCCUPANCY_PER_ROOM: 0
MAX_NO_OF_GUEST: 0
After object creation:
OCCUPANCY_PER_ROOM: 2
MAX_NO_OF_GUEST: 0

6.19) Initializer expressions **CANNOT** pass on the checked exceptions, only unchecked ones. If any checked exception is thrown during execution of an initializer expression, it must be caught and handled within the initializer expression. [1]

6.20) Static initializer blocks: [1]

- Can include arbitrary code.
- Code is executed **ONLY ONCE** when the class is initialized.
- Is **NOT** contained in any method.
- A class can have more than one static initializer block.
- A typical use of static initializer in a class is to load any external libraries that class needs, for example, to execute native methods.
- Must catch and handle the checked exception in the block as no constructor is involved in class initialization.

Example:

```
class StaticInitializers {
    final static int ROWS = 12, COLUMNS = 10;
    static long[][] matrix = new long[ROWS][COLUMNS];
    // …
    static {
        for (int i = 0; i < matrix.length; i++)
            for (int j = 0; j < matrix[i].length; j++)
                matrix[i][j] = 2*i + j;
    }
    // …
}
```

6.21) Instance initializer blocks: [1]

- Code is executed every time an instance of the class is created.
- Is **NOT** contained in any method.
- A typical use of an instance initializer block is an anonymous classes, which cannot have constructors.
- A class can have more than one instance initializer block.
- Exception handling differs from that in static initializer blocks in the following respect: if an instance initializer block does not catch a checked exception that can occur during its execution, then the exception must be declared in the `throws` clause of every constructor in the class.

Example:

```
class InstanceInitializers {
    long[] squares = new long[10];
    // …
    {
        for(int i = 0; i < squares.length; i++)
            squares[i] = i*i;
    }
    // …
}
```

6.22) The action of creating a new object is the biggest use of memory in a Java application. [7]

6.23) Constructing initial object state:(when invoking new operator): [1]

- Instance variables are initialized to their default values.
- Constructor is invoked which can lead to local chaining of constructors, the invocation of the constructor at the end of the local chain of constructor invocations results in the following actions, **BEFORE** the constructor's execution resumes:

(a) Invocation of superclass's constructor implicitly or explicitly. Constructor chaining ensures that the inherited state of the object is constructed first.

(b) Initialization of instance member variables by executing their instance initializer expressions and any instance initializer blocks in the order they are specified in the class definition.

Example:

```
class SuperclassA {

    Public SuperclassA() {
        System.out.println("Constructor in SuperclassA");
```

```
    }
}
class SubclassB extends SuperclassA {

    public SubclassB() {
        this(3);
        System.out.println("Default constructor in SubclassB");
    }

    public SubclassB(int i) {
        System.out.println("Non-default constructor in SubclassB");
        value = i;
    }
    {
        System.out.println("Instance initializer block in SubclassB");
        // value = 2; // Not Ok
    }

    private int value = initializerExpression();

    private int initializerExpression() {
        System.out.println(
               "Instance initializer expression in SubclassB");
        return 1;
    }
}
public class ObjectConstruction {
    public static void main(String args[]) {
        new SubclassB();
    }
}
```

Output:

Constructor in SuperclassA
Instance initializer block in SubclassB
Instance initializer expression in SubclassB
Non-default constructor in SubclassB
Default constructor in SubclassB

Chapter 7:

Inner classes

7.1) Inner classes are classes defined at a scope smaller than a package, i.e. you can define an inner class inside another class, inside a method and even as part of an exception. [3]

7.2) Why make a nested class? [8]

(a) You might need to make use of the special functionality of class `A` from within class `B` without complicating the inheritance hierarchy of either class.

(b) From the point of view of programming philosophy, if class `A` exists solely to help work with class `B`, you might as well make class `A` a member of class `B`. This helps to keep all of the code related to class `B` in a single source code file.

7.3) Just as member methods in a class have unlimited access to all `private` and other variables and methods in the class, nested classes also have unlimited access. [8]

7.4) Notes on nested top level classes: [1] [3]

- Is considered **NOT** included in inner classes as mentioned in [1], but considered included in inner classes as mentioned in [8] & also while solving question, **STILL** confusing till **NOW**. When I took my exam, I had several question in it that makes it included in the inner classes.
- They behave much like top-level classes except that they are defined within the scope of another class.
- Interfaces are implicitly `static`, nested interfaces can optionally be prefixed with the keyword `static` and have `public` accessibility.
- **CANNOT** have the same name as an enclosing class or package.
- `static` methods do **NOT** have a `this` reference & can therefore access other `static` methods & variables directly in the class, this also applies to methods in a nested top-level class.
- Top-level nested class can define both static & non-static & instance members, however the code can **ONLY** directly access `static` members in the enclosing context regardless of their accessibility.
- **CAN** implement any arbitrary interface.

Example:

```
// Filename: TopLevelClass.java
public class TopLevelClass {                          // (1)
    // …
    static class NestedTopLevelClass {                // (2)
        // …
        interface NestedTopLevelInterface1 {          // (3)
            // …
        }
        static class NestedTopLevelClass1             // (4)
            implements NestedTopLevelInterface1 {
            // …
        }
    }
}
```

The full name of nested top-level class at (4) is:

```
TopLevelClass.NestedTopLevelClass.NestedTopLevelClass1
```

When compiled:

```
TopLevelClass$NestedTopLevelClass$NestedTopLevelClass1.class
TopLevelClass$NestedTopLevelClass$NestedTopLevelInterface1.class
TopLevelClass$NestedTopLevelClass.class
TopLevelClass.class
```

- `import` statement can be used by clients to provide shortcut for the names of nested top-level classes and interfaces.

Example:

```
// Filename: Client1.java
import TopLevelClass.*;
public class Client1 {
   NestedTopLevelClass.NestedTopLevelClass1 objRef1 = new
 NestedTopLevelClass.NestedTopLevelClass1();
```

```
}

// Filename: Client2.java
import TopLevelClass. NestedTopLevelClass.*;
public class Client2 {
    NestedTopLevelClass1 = objRef1 = new NestedTopLevelClass1();
}
```

7.5) Notes on Non-static Inner classes: [1] [3]

- Are defined without the keyword `static`, as members of an enclosing class, and can also be nested to any depth.
- An instance of a non-static inner class can **ONLY** exist with an instance of its enclosing class. This means that an instance of a non-static inner class must be created in the context of an instance of the enclosing class.
- **CANNOT** have static members, i.e. the class does not provide any services, only instances of the class do.
- **CANNOT** have the same name as an enclosing class.
- Methods of a non-static inner class can directly refer to any member (including classes) of any enclosing class, including private members. No explicit reference is required.
- It can have any accessibility including `abstract`.
- To create an instance of the non-static inner class:
  (a) If you inside the scope of the enclosing class use:
  `new` ***`Memberclass()`***
  or
  `this.new` ***`Memberclass()`***
  (b) If you are outside the scope of the enclosing class you need to use an instance of the enclosing class to create the member object, you can use a specific form of the `new` operator as follows:
  <*enclosing object reference*>.new <*Non static inner class name*>
- Referring to members in the enclosing class is possible, but if you want to use this reference to refer to it, you can use it in the form:
  <*enclosing class name*>.this.<*enclosing class member name*>
- Example (About referencing shadowed members):

```
class TLClassA {                              // (1) Top-level class
    private String msg = "TLClassA object ";
    private TLClassA(String objNo) {
        msg = msg + objNo;
    }
    public void printMessage() {
        System.out.println(msg);
    }
    class InnerB {                            // (2) Non-static Inner class
        private String msg = "InnerB object ";
        public InnerB(String objNo) {
            msg = msg + objNo;
        }
        public void printMessage() {
            System.out.println(msg);
        }
        class InnerC {                        // (3) Non-static Inner class
            private String msg = "InnerC object ";
            public InnerC(String objNo) {
                msg = msg + objNo;
            }
            public void printMessage() {
                System.out.println(msg);
                System.out.println(this.msg);
                System.out.println(InnerC.this.msg);
                System.out.println(InnerB.this.msg);
                InnerB.this.printMessage();
                System.out.println(TLClassA.this.msg);
```

```
                TLClassA.this.printMessage();
            }
        }
    }
}
public class Client2 {
    public static void main(String args[]) {
        TLClassA a = new TLClassA("1");
        TLClassA.InnerB b = a.new InnerB ("1");
        TLClassA.InnerB.InnerC c = b.new InnerC ("1");
        c.printMessage();
        TLClassA.InnerB bb = new TLClassA("2").new InnerB ("2");
        TLClassA.InnerB.InnerC cc = bb.new InnerC ("2");
        cc.printMessage();
        TLClassA.InnerB.InnerC ccc = new TLClassA("3").new InnerB ("3").new
InnerC("3");
    }
}
```

Output of the program:
InnerC object 1
InnerC object 1
InnerC object 1
InnerB object 1
InnerB object 1
TLClassA object 1
TLClassA object 1
InnerC object 2
InnerC object 2
InnerC object 2
InnerB object 2
InnerB object 2
TLClassA object 2
TLClassA object 2
When compiled:

```
TLClassA$InnerB$InnerC.class
TLClassA$InnerB.class
TLClassA.class
Client2.class
```

- **CAN** extend other classes and can themselves be extended. **BUT** if a name conflict arises, the inherited member shadows the member with the same name in the enclosing class. The compiler, however, **REQUIRES** that explicit references be used. Example:

```
class B {
    protected double x = 2.17;
}
class A {
    private double x = 3.14;

    class C extends B {
        // private double w = x;            // Compile time error
        private double y = this.x;          // x from the superclass
        private double u = super.x;         // x from the superclass
        private double z = A.this.x;        // x from the enclosing class
    }
}
```

7.6) Notes on Local classes: [1]

- Is a class that is defined in a block. This could be a method body, a constructor, a local block, a static initializer or an instance initializer.

- Is **ONLY** visible within the context of the block, i.e. the name of the class is only valid in the context of the block in which it is defined. Clients outside the context of a local class **CANNOT** create or access these classes directly, because they are all local.
- **CANNOT** be specified with the keyword `static`. However, if the context is static (i.e. a static method or a static initializer) then the local class is implicitly static. Otherwise, the local class is non-static.
- **CANNOT** have static members as they can't provide class specific services.
- **CANNOT** have any accessibility. This restriction applies to local variables, and is also enforced for local classes.
- Can access members defined within the class.
- Can access `final` local variables, `final` method parameters and `final catch`-block parameters in the scope of the local context.
- Can access members inherited from its superclass in the usual way, also the standard `this` reference or the `super` keyword can be used.
- A non-static local class can access members defined in the enclosing class explicitly, or the special form of the `this` can be used.
- A non-static local class can directly access members inherited by the enclosing class, or the special form of the `this` can be used.

Example:

```
class SuperB {
    protected double x;
    protected static int n;
}
class SuperC {
    protected double y;
    protected static int m;
}
class TopLevelA extends SuperC {         // Top level class
    private double z;
    private static int p;

    void nonStaticMethod(final int i) {  // Non-static method
        final int j = 10;
        int k;
        class NonStaticLocalD extends SuperB { // Non-static local class
        // static double d;    // Not ok, Only non-static members allowed
           int ii = i;         // final from enclosing method
           int jj = j;         // final from enclosing method
        // double kk = k;      // Not ok, Only finals from enclosing method
           double zz = z;      // non-static from enclosing class
           int pp = p;         // static from enclosing class
           double yy = y;      // inherited by the enclosing class
           int mm = m;         // static from enclosing class
           double xx = x;      // non-static inherited from superclass
           int nn = n;         // static from superclass
        }
    }

    static void staticMethod(final int i) {
        final int j = 10;
        int k;
        class StaticLocalE extends SuperB {
        // static double d;    // Not ok, Only non-static members allowed
            int ii = i;        // final from enclosing method
            int jj = j;        // final from enclosing method
        // double kk = k;      // Not ok, Only finals from enclosing method
        // double zz = z;      // Not ok, Non-static member
            int pp = p;        // static from enclosing class
        // double yy = y;      // Not ok, Non-static member
            int mm = m;        // static from enclosing class
            double xx = x;     // non-static inherited from superclass
```

```
            int nn = n;         // static from superclass
        }
    }
}
```

When compiled: [as in reference [1]]

```
TopLevelA$1$NonStaticLocalD.class
TopLevelA$1$StaticLocalE.class
TopLevelA.class
SuperB.class
SuperC.class
```

When compiled: [as I tried with **Oracle jdeveloper** 3.1, I don't know which is right]

```
TopLevelA$1$NonStaticLocalD.class
TopLevelA$2$StaticLocalE.class
TopLevelA.class
SuperB.class
SuperC.class
```

- A method can return an instance of the local class. The local class must be assignable to the return type of the method, often supertype of the local class is specified as the return type.

7.7) Notes on Anonymous classes: [1]

- It combines the process of definition and instantiation into a single step, as they are defined at the location they are instantiated, using additional syntax with the new operator.
- The context determines whether the anonymous class is static, and the keyword `static` is **NOT** used explicitly.
- It **CANNOT** define constructors (as it does not have name).
- It implicitly extends the `Object` class.
- Syntax for defining and instantiation of anonymous classes:
  `new` <*superclass*> (<*optional argument list*>) {<*class declaration*>}
- It provides a **SINGLE** interface implementation, and **NO** arguments are passed.
- Syntax for defining and instantiating anonymous class that implements an interface:
  `new` <*interface name*> () {<*class declaration*>}

Example:

```
interface IDrawable {
    void draw();
}
class Shape implements IDrawable {
    public void draw() {
        System.out.println("Drawing a Shape.");
    }
}
class Painter {                         // Top level class
    public Shape createShape() {        // Non-static method
        return new Shape() {            // Extends superclass Shape
            public void draw() {
                System.out.println("Drawing a new Shape.");
            }
        };
    }
    public static IDrawable createIDrawable() { // static method
        return new IDrawable() {                // implements interface
            public void draw() {
                System.out.println("Drawing a new IDrawable.");
            }
        }
    }
}
public class Client {
    public static void main(String args[]) {
```

```
        IDrawable[] drawables = {
            new Painter().createShape(),
            Painter.createIDrawable(),
            newPainter().createIDrawable()
        };
        for (int I = 0; I < drawables.length; I++)
            drawables[I].draw();
        System.out.println("Anonymous Class Names:");
        System.out.println(drawables[0].getClass());
        System.out.println(drawables[1].getClass());
    }
}
```

When compiled:

```
class Painter$1
class Painter$2
```

7.8) All inner classes can define **ONLY** non-static members, and can have `static final` members.

7.9) Summary of classes and interfaces: [1]

Entity	Declaration Context	Accessibility modifiers	Require outer instance	Direct Access to Enclosing Context	Defines static or Non-static members
Package level class	As package member	public or default	No	N/A	Both static and non-static
Top-level nested class (static)	As static class member	All	No	Static members in enclosing context	Both static and non-static
Non-static inner class	As non-static class member	All	Yes	All members in enclosing context	Only non-static + static final ONLY
Local class (non-static)	In block with non-static context	None	Yes	All members in enclosing context + local final variables	Only non-static + static final ONLY
Local class (static)	In block with static context	None	No	Static members in enclosing context + local final variables	Only non-static + static final ONLY
Anonymous class (non-static)	As expression in non-static context	None	Yes	All members in enclosing context + local final variables	Only non-static + static final ONLY
Anonymous class (static)	As expression in static context	None	No	Static members in enclosing context + local final variables	Only non-static + static final ONLY
Interface	As package member or	Public only	N/A	N/A	Static variables +

	static class member	non-static method prototypes

Chapter 8:

Threads

8.1) At most **ONE** object is executing per CPU, while others might be waiting for resources or waiting for chances to execute or sleeping or dead. [3]

8.2) The `main()` method can be finished, but the program will keep running until all the user threads are done, i.e. Program terminates when the last non-deamon thread ends. [1]

8.3) Daemon threads run in the background and do not prevent a program from terminating. For example, the garbage collector is a daemon thread, a daemon thread is at the mercy of the runtime system, it is stopped if there are no more user threads running. [1]

8.4) The `Thread` class provide: [2]

`public final void setDaemon(boolean on)`

Marks this thread as either a daemon thread or a user thread.

`public final boolean isDaemon()`

Tests if this thread is a daemon thread.

8.5) If the code spawning threads is not put within `try – catch` block in the `main()` method, the main thread would finish executing before the child thread; however, the program would run until the child thread completes. [1]

8.6) Implementing threads is achieved in one of two ways: [1] [8]

Implementing `java.lang.Runnable`	Extending `java.lang.Thread`
1. A class implements the `Runnable` interface providing the `run()` method, which will be executed by the thread.	1. A class extending the `Thread` class overrides the `run()` method from the `Thread` class to define the code executed by the thread.

2. An object of the `Thread` class is created. An object of a class implementing the `Runnable` interface is passed as an argument to a constructor of the `Thread` class.
`Thread(Runnable threadTarget)`
`Thread(Runnable threadTarget, String threadName)`
	2. This subclass may call a `Thread` constructor explicitly in its constructors to initialize the thread.

3. The `start()` method is invoked on the `Thread` object created in 2. However, the `start()` method returns immediately after a thread has been spawned. This method is defined in the `Thread` class. 4. It is possible to attach more than one `Thread` to a `Runnable` object.	3. The `start()` method inherited from the `Thread` class is invoked on the object of the class to make the thread eligible for running.

8.7) `run` methods **CANNOT** throw exceptions so all checked exception **MUST** be caught using `try – catch`. [3]

8.8) When a thread is started, it gets connected to the JVM scheduling mechanism and executes a method declared as follows: [8]

```
public void run()
```

NOTE:

The `run` method in the `Thread` class is empty.

8.9) Calling your thread's `start()` method doesn't immediately cause the thread to run, it just make it eligible to run, and the instructions after the `start()` call is executed before or after the thread is **NOT** determined. The JVM sets up some resources and puts the Thread in the list of runnable Threads. Exactly when a thread gets to execute depends on its

priority, the activity of the other threads, and the characteristics of the particular JVM. [3] [8]

8.10) When creating threads, implementing the `Runnable` interface is usually preferred to extending the `Thread` class for two main reasons: [1]
(a) Extending the `Thread` class means that the subclass cannot extend any other class, whereas by implementing the `Runnable` interface it has this option.
(b) A class might only be interested in being `Runnable`, and therefore inheriting the full overhead of the `Thread` class would be excessive.

8.11) You can simulaneously create and start a thread as in the following method:

```
public void startThread() {
    new Thread(this).start();
}
```

NOTE: You might think that the new thread in the preceding example would be garbage-collected because no reference to it is being kept, but the JVM created a reference in its list of threads. [8]

8.12) A thread might be halted in mid-calculation and another allowed to use the same data, resulting in a disaster ⇒ that's why we need synchronization. [8]

8.13) Java provides the foundation for solving synchroniztion in the `Object` class. Each object has an associated lock variable that can be manipulated only by the JVM. This lock provides a monitor mechanism that can be used to allow only one thread at a time to have access to an object (mutually exclusive lock). [1] [8]

8.14) Because it would take additional time for the JVM to check the lock condition of an object every time it is accessed, the lock is ignored by default. The keyword `synchronized` is used to indicate a method or block of code that needs to be guarded by the lock mechanism. [8]

8.15) When `synchronized` is used as a statement, it requires a reference to the object to be locked. For convenience, `synchronized` can be used as a method modifier, in which case the entire method is the block of code, and `this` is automatically the object reference. [8]

8.16) The monitor mechanism enforces the following rules of synchronization: [1]
- **NO** other thread can enter a monitor if a thread has already acquire the monitor. Threads wishing to acquire the monitor will wait for the monitor to become available.
- When a thread exits a monitor, a waiting thread is given the monitor, and can proceed the shared resource associated with the monitor.

8.17) There are two ways in which code can be synchronized: [1] [8]

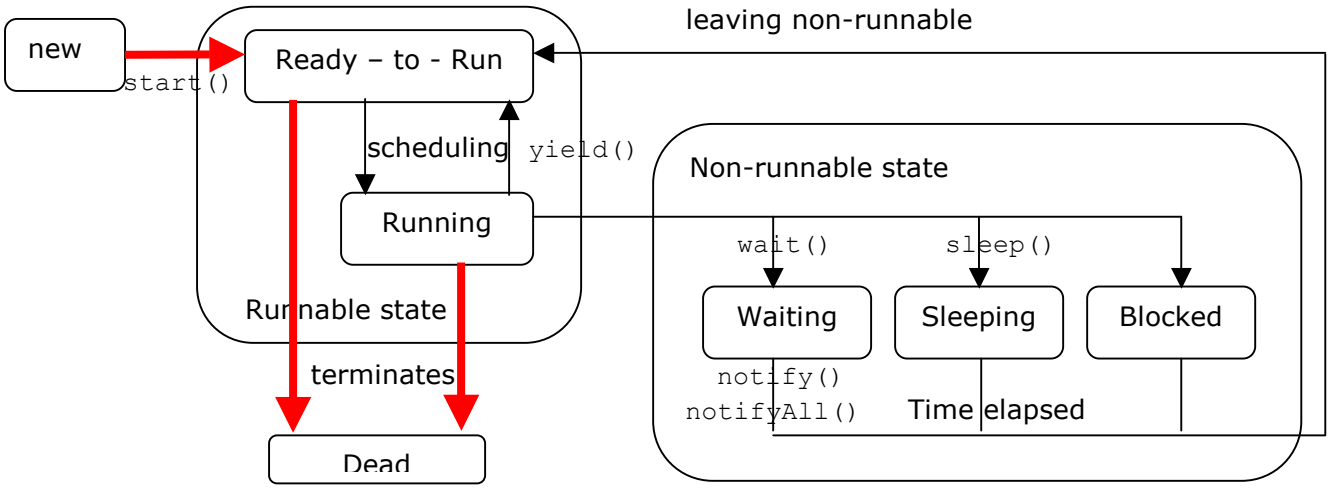
Synchronized methods	Synchronized blocks
1. Are useful in situations where methods can manipulate the state of an object in ways that can corrupt the state if executed concurrently.	1. Allows arbitrary code to be synchronized on the monitor of an arbitrary object; the code block is usually related to the object on which the synchronization is being done. `syncronized (<objectreference>) {…}`
2. While a thread is inside a synchronized method of an object, all other threads that wish to execute this synchronized method or any other synchronized method of the object will have to wait.	2. The braces of the block **CANNOT** be left out even if the code block has just one statement.
3. The non-synchronized methods of the object can of course be called at any time by any thread.	3. Once a thread has entered the code block after acquiring the monitor of the specified object, no other thread will be able to

execute the code block or another code requiring monitor until the monitor is released by the object.

4. Synchronized methods can be `static`.
4. Inner classes can access data in their enclosing context; An inner object might need to synchronize on its associated outer object, in order to ensure integrity of data in the latter.
5. Classes also have a class-specific monitor, which is analogous (similar) to the object monitor.
5. **DO NOT** synchronize on a local variable, because it will accomplish nothing because every thread has its own copy of local variables, so you should use an instance object reference variable instead.
6. Synchronization of static methods in a class is independent from the synchronization of instance methods on objects of the class (i.e. A thread acquiring the monitor on any object of the class to execute a `static synchronized` method has no bearing on any thread acquiring the monitor on any object of the class to execute a synchronized instance method.
6. Be sure that the object chosen really protects the data you want to protect.
7. A subclass decides whether the new definition of an inherited synchronized method will remain synchronized in the subclass.
8. Consumes extra CPU cycles on entry and exit, so you should not synchronize without good cause.

8.18) Thread objects have a distinct life cycle with four basic states: [8]
(a) new.
(b) Runnable.
(c) Blocked (Non-runnable).
(d) Dead.
The transitions from new to runnable and from runnable to dead are simple and permanant, the transitions between runnable and blocked occupy most of the Java programmer's attention.

8.19) Thread states: [1] [2] [8] [**I modified the graph a little**]



Ready-to-run: Means thread is eligible for running. A call to static method `yield()` will cause current running thread to move to ready-to-run. Here the thread awaits its turn to get the CPU time, if the thread is carrying out a complex computation, you should insert an occasional call to `yield()` in the code to ensure that other threads get a chance to run.

Running: The CPU is currently executing the thread, the "*thread schedular*" decides which thread is in the running state.

Non Runnable states: A thread can go from the running state into one of the non-runnable states, depending on the transition, & remains their till a special transition moves it to ready-to-run state.

(a) **waiting**: A thread can call `wait()` method, it must be notified by another thread, in order to move to 'ready-to-run' state.

In the `java.lang.Object` class:

```
public final void wait()throws InterruptedException
```

- The calling thread gives up the CPU.
- The calling thread gives up the lock.
- The calling thread goes into the waiting state.
- The thread that calls the `wait` must be the owner of the object's monitor.
- The following two methods in the `Object` class causes current thread to wait until either another thread invokes the `notify()` method or the `notifyAll()` method for this object, or a specified amount of time has elapsed.

```
void wait(long timeout) throws InterruptedException
void wait(long timeout,int nanos)throws InterruptedException
```

(b) **sleeping**: Can call `sleep()`, wakesup after specified amount of time elapsed.

```
public static void sleep(long millis) throws
InterruptedException
public static void sleep(long millis, int nanos) throws
InterruptedException
```

NOTE:
Sleeping is not a high-precision timing operation because it depends on the clock of the underlying operating system. In addition, there is no guarantee that the thread will immediately begin to execute after the time delay is up; that is up to the JVM thread scheduler.

(c) **blocked state**: A running thread on executing a blocking operation requiring resource(like I/O method), and also a thread is blocked if it fails to acquire the monitor on an object; the blocking opertaion must complete before the thread can proceed to ready-to-run state.

Dead: when the thread is completed as it exits the run method to which it is attached, you **CANNOT** restart a dead thread.

8.20) JVM also dies when the `System.exit` or `exit` method of `Runtime` is called.

8.21) Priorities: Are Integer values from 1 (`Thread.MIN_PRIORITY`) to 10 (`Thread.MAX_PRIORITY`), if no explicit thread priority is specified for a thread, it is given default priority(`Thread.NORM_PRIORITY`). [1]

8.22) A thread inherits priority of parent thread not `Thread.NORM_PRIORITY` & can be explicitly set or read using methods in the `Thread` class:

```
public final void setPriority(int newPriority)
public final int getPriority()
```

8.23) Thread schedules are implementation and platform dependent. [1]

8.24) If a thread that does not have a lock on an object attempts to call the object's `wait` or `notify` method, an `IllegalMonitorStateException` is thrown – typically with the

message "current thread not owner". To ensure that is never happens, the `wait()`, `notify()`, and `notifyAll()` must be executed in `synchronized` code. [1] [8]

8.25) When `notify` is called, a thread is removed from the wait set and returned to the list of runnable threads. If more than one thread is waiting, you **CANNOT** control or predict which one it will be, a call to `notify()` has no consequence if there are not any threads waiting. If there is a chance that more than one thread is waiting, you can use `notifyAll()`, it removes all waiting threads from the wait list, **ONLY** one of these will actually get a lock on the object and be allowed to execute the synchronized method, the others will run and find that the object is still locked. [3] [8]

8.26) It is a good thing to put the `wait()` in a loop that test the waiting condition to guarantee that the connection for waiting is fulfilled when this thread is notified. [1]

8.27) A thread becomes the owner of the object's monitor in one of three ways: [3]
(a) By executing a synchronized instance method of that object (i.e. by executing a `synchronized` method inside this method you can call the wait method)
(b) By executing the body of a synchronized block that synchronizes on the object.
(c) For objects of type Class, by executing a `synchronized static` method of that class.

8.28) Automatic variables **CANNOT** be shared between threads each thread has it's copy and can't modify the value of the other thread. [3]

8.29) `public final boolean` **`isAlive`**`()` [11]
Tests if this thread is alive. A thread is alive if it has been started and has not yet died.
Example:
Parent thread finds if any child threads are alive before terminating itself
`isAlive()` will return `true` at all states (including suspended) except when the thread is in new or dead state.

8.30) `public final void join() throws InterruptedException`
Waits for this thread to die. A call to this method invoked in a thread will wait and not return until thread has completed. A parent thread can use this method to wait for its child thread to compkete before continuing.[1] [2]

8.31) The programmer is ultimately responsible for avoiding deadlocks. [8]

8.32) To avoid common mistakes in the exam, here is a summary of methods used with threads: [8]

Class	Method	Type	Needs	Timeout Form
Thread	`yield()`	static		no
Thread	`sleep(#)`	static	`try-catch`	always
Thread	`start()`	instance		no
Thread	`run()`	instance		no
Thread	`interrupt()`	instance		no
Object	`wait()`	instance	`synchronized, try-catch`	optional
Object	`notify()`	instance	`synchronized`	no
Object	`notifyAll()`	instance	`synchronized`	no

Chapter 9:

Fundamental classes

9.1) `Object` class: [1] [2]

Is the mother class of all classes; A class definition, without the `extends` clause, implicitly extends the `Object` class.

Method	Notes
`public int hashCode()`	Returns a hash code value for the object. If two objects are equal according to the `equals` method, then calling the `hashCode` method on each of the two objects must produce the same integer result.
`public final Class getClass()`	Returns the runtime class of an object.
`public boolean equals(Object obj)`	This method returns `true` if and only if `x` and `y` refer to the same object (`x==y` has the value `true`). Usually overridden to provide semantics of the object value equality. Expression `obj.equals(null)` is always `false`.
`protected Object clone() throws CloneNotSupportedException.`	Creates and returns a copy of this object.
`public String toString()`	Returns a string representation of the object. if the subclass does not override this method, it returns a textual representation of the object, which has the following fomat: *< getClass().getName()>* @ *<Integer.toHexString(hashCode())>* The `println()` method in the `PrintStream` will convert its argument to a textual representation using this method.
`protected void finalize() throws Throwable`	Called just before an object is garbage collected, so that cleanup can be done. The default `finalize()` method in the `Object` class does nothing.

`public final void wait() throws InterruptedException`
`public final void wait(long timeout) throws InterruptedException`
`public final void wait(long timeout, int nanos) throws InterruptedException`
	Throws `IllegalMonitorStateException` if the current thread is not the owner of this object's monitor. Throws `InterruptedException` if this thread is interrupted by another thread.

`public final void notify()`	Wakes up a single thread that is waiting on this object's monitor. If any threads are waiting on this object, one of them is chosen to be awakened. The choice is arbitrary and occurs at the discretion of the implementation.
`public final void notifyAll()`	Wakes up all threads that are waiting on this object's monitor.

9.2) The wrapper classes: [1]

- The objects of all wrapper classes that can be instantiated are immutable.
- The `Void` class is not instantiable.
- Common wrapper class constructors:
  Each wrapper class (except `Character` class has only one constructor) has the following two constructors:
  i. A constructor that takes a primitive value and returns an object of the corresponding wrapper class.

```
Character charObj = new Character('\n');
```

ii. A constructor that takes a `String` object representing the primitive value, and returns an object of the corresponding wrapper class; these constructor throw `NumberFormatException` if the `String` parameter is not valid.

```
Boolean b1Obj = new Boolean("True"); // case ignored : true
Boolean b2Obj = new Boolean("XX");   // false
Double b3Obj  = new Double("3.142");
```

- Common wrapper class utility methods:
  i. Each wrapper class (except `Character`) defines `static` method `valueOf(String s)` that returns the wrapper new object corresponding to the primitive value represented by the `String` object passed as argument; throws `NumberFormatException` if the `String` parameter is not valid.

```
Boolean bObj = Boolean.valueOf("false");
Integer intObj = Integer.valueOf("2010");
```

  ii. Each overrides the `equals()` comparing two wrapper objects for object value equality.
  iii. Each overrides the `toString()` returning a `String` representing the primitive value.
  iv. Each overrides the `hashCode()` returning a hash value based on the primitive value in the wrapper object.
- Numeric wrapper classes:
  i. Each is a subclass of the abstract `java.lang.Number` class.
  ii. Each defines *type*`Value()` methods for converting primitive value in the wrapper object to a value of any numeric primitive datatype.

```
Byte byteObj = new Byte ((byte)16);
Integer intObj = new Integer(42030);
short s = intObj.shortValue();
long l = byteObj.longValue();
```

  Can cause potiential loss of information when primitive value in wrapper object is converted to a narrower primitive datatype.
  iii. Each numeric wrapper class defines a static method `parse`*Type*`(String s)` that returns the primitive numeric value represented by `String` represented by `String` Object passed as argument; these methods throws `NumberFormatException` if the `String` parameter is not a valid argument.
  iv. Most important constants: <*wrapper class*>.`MIN_VALUE`, <*wrapper class*>.`MAX_VALUE`

- `Void` class does not wrap any primitive value, it only denotes the class object representing the primitive type `void`.

9.3) The `Math` class: [1] [2] [11] [12]

- Is a `final` class ⇒ **CANNOT** be subclassed.
- constructor is `private` ⇒ **CANNOT** be instantiated.
- All constants and methods are `public` and `static` ⇒ just access using class name.

Category	Methods	Example & declaration
Constants	`Math.PI`	
	`Math.E`	
Random	`public static double random()`	0.0 ≤ random number < 1.0 Returned values are chosen pseudorandomly with (approximately) uniform distribution from that range.

Absolute	`public static <type> abs( <type> a)` Overloaded methods for `int, long, float, double` versions.	Returns the absolute value of a *type* value. If the argument is not negative, the argument is returned. **NOTE:** that if the argument is equal to the value of `Integer.MIN_VALUE`, or `Long.MIN_VALUE` the most negative representable `int/long` value, the result is that same value, which is negative.
Comparing	`public static <type> max ( <type> a, <type> b)` Overloaded methods for `int, long, float, double` versions.	Returns the greater of two *type* values. **NOTE:** If either value is `NaN`, then the result is `NaN`. `Math.max(-0.0, +0.0)` returns `+0.0`
	`public static <type> min( <type> a, <type> b)` Overloaded methods for `int, long, float, double` versions.	Returns the smaller of two *type* values. **NOTE:** If either value is `NaN`, then the result is `NaN`. `Math.min(-0.0, +0.0)` returns `-0.0`

Rounding	`public static double ceil(double a)`	Smallest integer greater than this number.
		double x = 0;
		x = Math.ceil(8.4); // x = 9.0
		x = Math.ceil(8.9); // x = 9.0
		x = Math.ceil(-9.4); // x = -9.0
		x = Math.ceil(-9.8); // x = -9.0

	`public static double floor(double a)`	Greatest integer smaller than this number.
		double x = 0;
		x = Math.floor(8.4); // x = 8.0
		x = Math.floor(8.9); // x = 8.0
		x = Math.floor(-9.4); // x = -10.0
		x = Math.floor(-9.8); // x = -10.0

	`public static long round(double a)`
	`public static int round(float a)`
		Returns the closest `long/int` to the argument.
		If the argument is negative infinity or any value less than or equal to the value of *`WrapperClass`*`.MIN_VALUE`, the result is equal to the value of *`WrapperClass`*`.MIN_VALUE`.
		If the argument is positive infinity or any value greater than or equal to the value of *`WrapperClass`*`.MAX_VALUE`, the result is equal to the value of *`WrapperClass`*`.MAX_VALUE`.
		Its algorithm works by `+0.5/-0.5` to the argument if it is positive/negative and truncate to the nearest integer equivalent.
		int x = 0;
		x = Math.round(8.4); // x = 8
		x = Math.round(8.9); // x = 9
		x = Math.round(-9.4); // x = -9
		x = Math.round(-9.8); // x = -10

	`public static double rint (double a)`	Returns the closest `double` value to `a` that is equal to a mathematical integer. If two `double` values that are mathematical integers are equally close to the value of the argument, the result is the integer value that is even.
		double x = 0;
		x = Math.rint(12.9); // x = 13.0
		x = Math.rint(12.5); // x = 12.0
		x = Math.rint(11.5); // x = 12.0

Exponential	`public static double exp(double a)`	Returns the exponential number *e* (i.e., 2.718...) raised to the power of a `double` value.
	`public static double pow(double a, double b)`	Returns of value of the first argument raised to the power of the second argument. If (`a == 0.0`), then `b` must be greater than `0.0`; otherwise an exception is thrown. An exception also will occur if (`a <= 0.0`) and `b` is not equal to a whole number.
	`public static double log(double a)`	Returns the natural logarithm (base *e*) of a `double` value.
	`public static double sqrt(double a)`	Returns the square root of a `double` value. If the argument is `NaN` or less than zero, the result is `NaN`.
Trigonometric (angle input in radians)	`public static double sin(double a)`	Returns the trigonometric sine of an angle.
	`public static double cos(double a)`	Returns the trigonometric cosine of an angle.
	`public static double tan(double a)`	Returns the trigonometric tangent of an angle.

9.4) The `String` class: [1] [2] [3] [11] [12]

- Is a `final` class implements immutable character stings, which are read only once the string has been created and initialized.
- Characters are represented as Unicode.
- There is no limitation in java on the length of a `String`, however the operating system may impose limitations.
- A `String` literal is implemented as an anonymous `String` object; Java optimizes handling of string literals: **ONLY ONE** anonymous `String` object is shared by all `String` literals with the same contents **EVEN** across classes, but a `String` created by the `new` operator is always a different new object, **EVEN** if it's created from a previously predefined literal.

Example:

```
public class AnonStrings {
    static String str1 = "You cannot touch me!";
    public static void main(String args[]) {

        String emptyStr = new String();
        System.out.println("0: " + emptyStr);

        String str2 = "You cannot touch me!";
        String str3 = new String(str2);
```

```
        System.out.println("1: " + (str1 == str2));
        System.out.println("2: " + str1.equals(str2));

        System.out.println("3: " + (str2 == str3));
        System.out.println("4: " + str2.equals(str3));

        System.out.println("5: " + (str1 == Auxiliary.str1));
        System.out.println("6: " + str1.equals(Auxiliary.str1));
    }
}
class Auxiliary {
    static String str1 = "You cannot touch me!";
}
```

Output:
0:
1: true
2: true
3: false
4: true
5: true
6: true

- Note a difference between a string & array; `String` has a method `length()` to get its length, but array has a member variable `length` whose value is the number of elements in the array.
- Strings are compared lexicographically.

Category	Methods	Example & declaration

Constructors	Created from:
	(a) literal.
	(b) `byte` array.
	(c) `char` array.
	(d) `StringBuffer`.
		`String str3 = new String("Hi");`
		`//new String() creates empty string`
		`byte[] b = {97, 98, 98, 97} ;`
		`String s = new String(b); // stores "abba"`
		`StringBuffer strBuf = new StringBuffer("axe");`
		`String s = new String(strBuf);`

Reading	`public int length()`	Returns the length of this string.
	`public char charAt(int index)`	Returns the character at the specified index. An index ranges from `0` to `length() - 1`. if index is not valid `IndexOutOfBoundsException` is thrown. (case sensitive)
Search	`public int indexOf(int ch)`	Returns the index within this string of the first occurrence of the specified character. (case sensitive)
	`public int indexOf(int ch, int fromIndex)`	Returns the index within this string of the first occurrence of the specified character, starting the search at the specified index. (case sensitive)
	`public int indexOf(String str)`	Returns the index within this string of the first occurrence of the specified substring. (case sensitive)
	`public int indexOf(String str, int fromIndex)`	Returns the index within this string of the first occurrence of the specified substring, starting at the specified index. (case sensitive)
	`public int lastIndexOf(int ch)`	Returns the index within this string of the last occurrence of the specified character. (case sensitive)

	`public int lastIndexOf(int ch, int fromIndex)`	Returns the index within this string of the last occurrence of the specified character, searching backward starting at the specified index. (case sensitive)
	`public int lastIndexOf(String str)`	Returns the index within this string of the rightmost occurrence of the specified substring. (case sensitive)
	`public int lastIndexOf(String str, int fromIndex)`	Returns the index within this string of the last occurrence of the specified substring. (case sensitive)
Comparing	`public boolean equals(Object anObject)`	Compares this string to the specified object. The result is `true` if and only if the argument is not `null` and is a `String` object that represents the same sequence of characters as this object. (case sensitive)
	`public boolean equalsIgnoreCase(String anotherString)`	Compares this `String` to another `String`, ignoring case considerations.
	`public int compareTo(Object o)`	Compares this `String` to another Object. If the Object is a `String`, this function behaves like `compareTo(String)`. Otherwise, it throws a `ClassCastException` (as Strings are comparable only to other Strings).
	`public int compareTo(String anotherString)`	Compares two strings lexicographically. The comparison is based on the Unicode value of each character in the strings. The character sequence represented by this `String` object is compared lexicographically to the character sequence represented by the argument string. `0` strings are equals > `0` the string is lexicographically greater than the argument < `0` the string is lexicographically less than the argument.
	`public int compareToIgnoreCase(String str)`	Compares two strings lexicographically, ignoring case considerations.
Case conversion	`public String toLowerCase()`	Converts all of the characters in this `String` to lower case using the rules of the default locale, which is returned by `Locale.getDefault()`. If no character in the string has a different lowercase version, based on calling the `toLowerCase` method defined by `Character`, then the original string is returned.

	`public String toUpperCase()`	Converts all of the characters in this `String` to upper case using the rules of the default locale, which is returned by `Locale.getDefault()`. If no character in this string has a different uppercase version, based on calling the `toUpperCase` method defined by `Character`, then the original string is returned.
Concatenation	`public String concat(String str)`	Concatenates the specified string to the end of this string.
Extracting	`public String trim()`	Removes whitespaces from both ends of this string. If this `String` object represents an empty character sequence, or the first and last characters of character sequence represented by this `String` object both have codes greater than `'\u0020'` (the space character), then a reference to this `String` object is returned. It trims all ASCII control characters as well. Whitespaces include: '\t' \u0009 Horizontal tabulation '\n' \u000A new line '\f' \u000C form feed '\r' \u000D carriage return ' ' \u0020 space
	`public String substring(int beginIndex)`	Returns a new string that is a substring of this string. The substring begins with the character at the specified index and extends to the end of this string.
	`public String substring(int beginIndex, int endIndex)`	Returns a new string that is a substring of this string. The substring begins at the specified `beginIndex` and extends to the character at index `endIndex - 1`. Thus the length of the substring is `endIndex-beginIndex`.
Getting string representation for various types	`public static String valueOf(Object obj)`	if the argument is `null`, then a string equal to `"null"`; otherwise, the value of `obj.toString()` is returned.
	`public static String valueOf(char[] data)`	Returns the string representation of the `char` array argument.
	`public static String valueOf(char[] data, int offset, int count)`	Returns the string representation of a specific subarray of the `char` array argument. The `offset` argument is the index of the first character of the subarray. The `count` argument specifies the length of the subarray.
	`public static String valueOf(<`*`primitive type`*`> value)`	Returns the string representation of the <*`primitive type`*> argument.

- Passing `null` to `indexOf` or `lastIndexOf` will throw `NullPointerException`, passing empty string returns `0`, passing a string that's not in the target string returns `-1`.
- `+` and `+=` operators are overloaded for Strings.

9.5) The `StringBuffer` class: [1]

- Is a `final` thread-safe class, implements mutable character strings.
- The capacity of a `StringBuffer` is the maximum number of characters that a `String` class can accommodate, before its size is automatically augmented.

Category	Methods	Example & declaration
Constructors	`public StringBuffer()`	Constructs a string buffer with no characters in it and an initial capacity of 16 characters
	`public StringBuffer (int length)`	Constructs a string buffer with no characters in it and an initial capacity specified by the length argument.
	`public StringBuffer (String str)`	The initial contents of the string buffer is a copy of the argument string. The initial capacity of the string buffer is 16 plus the length of the string argument.
Changing & Reading characters	`public int length()`	Returns the length of this string.
	`public char charAt(int index)`	Returns the character at the specified index. An index ranges from `0` to `length() - 1`. if index is not valid `IndexOutOfBoundsException` is thrown. (case sensitive)
	`public void setCharAt(int index, char ch)`	The character at the specified index of this string buffer is set to `ch`.
Appending, (increase length)	`public StringBuffer append(<primitive type> b)`	Appends the string representation of the *<primitive type>* argument to the string buffer.
	`public StringBuffer append(char[] str)`	Appends the string representation of the `char` array argument to this string buffer.
	`public StringBuffer append(char[] str, int offset, int len)`	Appends the string representation of a subarray of the `char` array argument to this string buffer. Characters of the character array `str`, starting at index `offset`, are appended, in order, to the contents of this string buffer. The length of this string buffer increases by the value of `len`.
	`public StringBuffer append(Object obj)`	Appends the string representation of the `Object` argument to this string buffer.
	`public StringBuffer append(String str)`	Appends the string to this string buffer
Inserting (increase length)	`public StringBuffer insert(int offset, <primitive type> b)`	Inserts the string representation of the *<primitive type>* argument into this string buffer.
	`public StringBuffer insert(int offset, char[] str)`	Inserts the string representation of the `char` array argument into this string buffer.

	`public StringBuffer insert(int index, char[] str, int offset, int len)`	Inserts the string representation of a subarray of the `str` array argument into this string buffer. The subarray begins at the specified `offset` and extends `len` characters. The characters of the subarray are inserted into this string buffer at the position indicated by `index`. The length of this `StringBuffer` increases by `len` characters.
	`public StringBuffer insert(int offset, Object obj)`	Inserts the string representation of the `Object` argument into this string buffer.
	`public StringBuffer insert(int offset, String str)`	Inserts the string into this string buffer.
Deleting (decrease length)	`public StringBuffer delete(int start, int end)`	Removes the characters in a substring of this `StringBuffer`. The substring begins at the specified `start` and extends to the character at index `end - 1` or to the end of the `StringBuffer` if no such character exists. If `start` is equal to `end`, no changes are made.
	`public StringBuffer deleteCharAt(int index)`	Removes the character at the specified position in this `StringBuffer` (shortening the `StringBuffer` by one character).
Reversing (maintain the same length)	`public StringBuffer reverse()`	The character sequence contained in this string buffer is replaced by the reverse of the sequence.
Manipulating capacity	`public int capacity()`	Returns the current capacity of the String buffer. The capacity is the amount of storage available for newly inserted characters; beyond which an allocation will occur.
	`public void ensureCapacity(int minimumCapacity)`	Ensures that the capacity of the buffer is at least equal to the specified minimum. If the current capacity of this string buffer is less than the argument, then a new internal buffer is allocated with greater capacity. The new capacity is the larger of: - The `minimumCapacity` argument. - Twice the old capacity, plus `2`. If the `minimumCapacity` argument is nonpositive, this method takes no action and simply returns.

	`public void setLength(int newLength)`	Sets the length of this String buffer. This string buffer is altered to represent a new character sequence whose length is specified by the argument. if the `newLength` argument is less than the current length of the string buffer, the string buffer is truncated to contain exactly the number of characters given by the `newLength` argument.

- The compiler uses `StringBuffer` to implement the `String` concatenation operator + Example:

```
String str = 4 + "U" + "Only";
```

is equivalent to:

```
String str =
   new StringBuffer().append(4).append("U").append("Only").toString();
```

The code does not create any temporary `String` object when concatenating several things, where a single `StringBuffer` object is modified and finally converted to a `String`.

- **NOTES (VERY COMMON ERRORS IN THE EXAM)**:
  (a) `+` and `+=` operators are **NOT** overloaded for `StringBuffer` class, and it will produce compile time error.
  (b) `equals` method is **NOT** overriden for `StringBuffer` class, and it works just as == operator.
  (c) Comparing or assigning `StringBuffer` to `String` will cause compile time error.
  (d) `trim()` method is not a `StringBuffer` method.

Chapter 10:

Collections

10.1) A collection allows a group of objects to be treated as a single unit. Arbitrary objects can be stored, retrieved and manipulated as elements of these collections. [12]

10.2) Collections Framework presents a set of standard utility classes to manage such collections. [12]

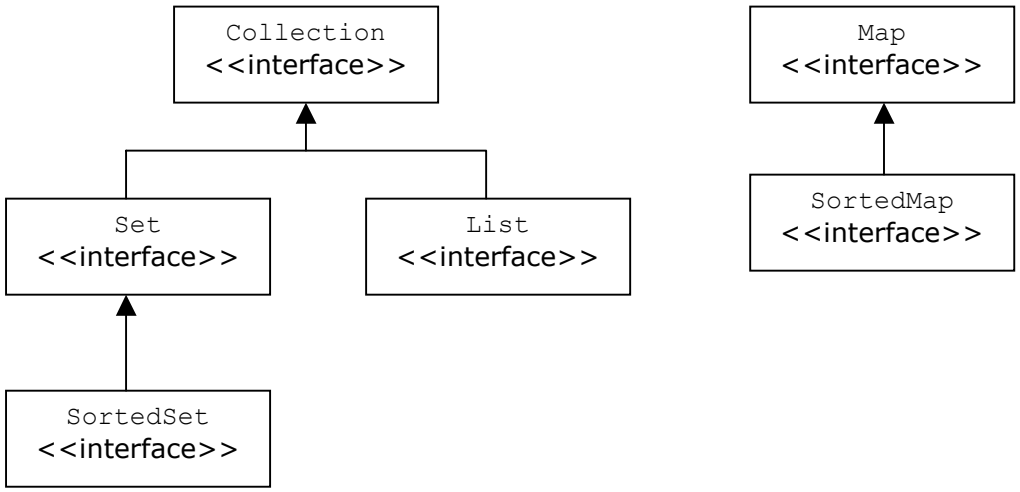
(a) It contains core interfaces which allow collections to be manipulated independent of their implementations. These interfaces define the common functionality exhibited by collections and facilitate data exchange between collections.

(b) A small set of implementations that are concrete implementations of the core interfaces, providing data structures that a program can use.

(c) A variety of algorithms to perform various operations such as, sorting and searching.

10.3) Collections framework is interface based, collections are implemented according to their interface type, rather than by implementation types. By using the interfaces whenever collections of objects need to be handled, interoperability and interchangeability are achieved. [12]

10.4) Core Interfaces in the Collection Framework: [1]



Interface	Description
`Collection`	A basic interface that defines the operations that all classes that maintain collections of objects typically implement.
`Set`	Extends the `Collection` interface for sets that maintain unique elements.
`SortedSet`	Augments the `Set` interface for sets that maintain their elements in a sorted order.
`List`	Extends the `Collection` interface for lists that maintain their elements in a sequence and need **NOT** be unique, i.e. the elements are in order.
`Map`	A basic interface that defines operations that classes that represent mappings of keys to values typically implement.
`SortedMap`	Extends the `Map` interface for maps that maintain their mappings in key order.

10.5) There is **NO** direct implementation of the `Collection` interface. [1]

10.6) Implementations of the core interfaces: [1] [12]

	Interfaces				
Data structure	Set	SortedSet	List	Map	SortedMap
Hash table	HashSet (√ null)			HashMap (√ null) Hashtable (× null)	
Resizable array			ArrayList (√ null) Vector (√ null)		
Balanced tree		TreeSet			TreeMap
Linked list			LinkedList (√ null)		

10.7) Classes that implement the interfaces use different storage mechanisms. [12]

(a) Arrays: Indexed access is faster. Makes insertion, deletion and growing the store more difficult.

(b) Linked List: Supports insertion, deletion and growing the store. But indexed access is slower.

(c) Tree: Supports insertion, deletion and growing the store. Indexed access is slower. But searching is faster.

(d) Hashing: Supports insertion, deletion and growing the store. Indexed access is slower. But searching is faster. However, requires the use of unique keys for storing data elements.

10.8) By convention each of the `Collection` implementation classes provide a constructor to create a collection based on the elements in the `Collection` object passed as argument. By the same token, `Map` implementations provide a constructor that accepts a `Map` argument. This allows the implementation of a collection (`Collection`/`Map`) to be changed. **BUT** Collections and Maps are **NOT** interchangeable. [12]

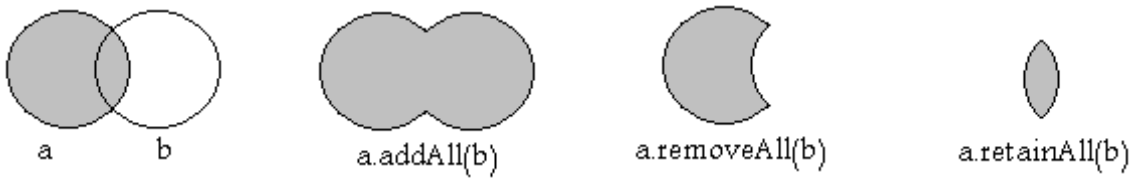
10.9) Some of the operations in the `Collection` interface are optional, meaning that a collection may choose not to provide a proper implementationof such an operation; in such case, an `UnsupportedOperationException` is thrown when the optional operation is invoked. The implementation of collections in the `java.util` package support all the optional operations in the `Collection` Interface. [1]

10.10) `Collection` interface operations: [1] [2]

Category	Methods	Example & declaration
Basic	`public int size()`	Returns the number of elements in this collection.
	`public boolean isEmpty()`	Returns `true` if this collection contains no elements.
	`public boolean contains(Object o)`	Returns `true` if this collection contains the specified element.
	`public boolean add(Object o)`	Ensures that this collection contains the specified element (optional operation). Returns `true` if this collection changed as a result of the call. (Returns `false` if this collection does not permit duplicates and already contains the specified element.)

	`public boolean remove(Object o)`	Removes a single instance of the specified element from this collection, if it is present (optional operation). Returns true if this collection contained the specified element (or equivalently, if this collection changed as a result of the call).
Bulk	`public boolean containsAll(Collection c)`	Returns `true` if this collection contains all of the elements in the specified collection.
	`public boolean addAll(Collection c)` (optional operation)	Adds all of the elements in the specified collection to this collection.
	`public boolean removeAll(Collection c)` (optional operation)	Removes all this collection's elements that are also contained in the specified collection.
	`public boolean retainAll(Collection c)` (optional operation).	Retains only the elements in this collection that are contained in the specified collection. In other words, removes from this collection all of its elements that are not contained in the specified collection.
	`public void clear()` (optional operation).	Removes all of the elements from this collection.
Array	`public Object[] toArray()`	Returns an array containing all of the elements in this collection. If the collection makes any guarantees as to what order its elements are returned by its iterator, this method must return the elements in the same order.
	`public Object[] toArray(Object[] a)`	Returns an array containing all of the elements in this collection whose runtime type is that of the specified array.
Iterators	`public Iterator iterator()` Interface has the following methods: `boolean hasNext();` `Object next();` `void remove();`	Returns an iterator over the elements in this collection. There are **NO** guarantees concerning the order in which the elements are returned (unless this collection is an instance of some class that provides a guarantee).

10.11) The operations performed by the `addAll()`, `removeAll()`, `retainAll()` methods can be visualized by Venn diagrams. [1]



10.12) The class `java.util.Collections` class (**NOT** to be confused with the `Collection` interface) provides static methods which implement polymorphic algorithms including sorting, searching and shuffling elements. Operates on the collection passed as first argument of the method;most methods accept a `List` object while a few operate on arbitrary `Collection` objects. [1]

Methods	Example & declaration
`public static int binarySearch(List list , Object key)`	Searches the specified list for the specified object using the binary search algorithm.

`public static void fill(List list, Object o)`	Replaces all of the elements of the specified list with the specified element.
`public static void shuffle(List list)`	Randomly permutes the specified list using a default source of randomness.
`public static void sort(List list)`	Sorts the specified list into ascending order, according to the *natural ordering* of its elements. All elements in the list must implement the `Comparable` interface. Furthermore, all elements in the list must be *mutually comparable* (that is, `e1.compareTo(e2)` must not throw a `ClassCastException` for any elements `e1` and `e2` in the list).

10.13) Sets: [1]

- Does not define any new method, but adds the restriction that duplicated are prohibited.
- It models a mathematical set.

Set methods	Corresponding mathematical operations
`a.containsAll(b)`	$b \subseteq a$? (subset)
`a.addAll(b)`	$a = a \cup b$ (union)
`a.removeAll(b)`	$a = a - b$ (difference)
`a.retainAll(b)`	$a = a \cap b$ (intersection)
`a.clear()`	$a = \varnothing$ (empty set)

- `HashSet` class as an example of sets: [1]

`HashSet()`
: Constructs a new, empty set.

`HashSet(Collection c)`
: Constructs a new set containing the elements in the specified collection, but it will contain no duplicates.

`HashSet(int initialCapacity)`
: Constructs a new, empty set with the specified initial capacity.

`HashSet(int initialCapacity, float loadFactor)`
: Constructs a new, empty set with the specified initial capacity and the specified load factor(the ratio of number of elements stored to its current capacity).

Example:

```
Import java.util.*;
public class CharacterSets {
    int nArgs = args.length;

    // A set keeping track of all characters prevoiusly encountred
    Set encountered = new HashSet();

    // For each command line argument
    for ( int i = 0; i < nArgs; i++) {
        String argument = args[1];

        // Convert string to a set of characters
        Set characters = new HashSet();
        int size = argument.length();

        // For each character in the argument
        for ( int j = 0; j < size; j++)
            // append character
            characters.add( new Character(argument.charAt(j)) );

        // Determine if there exists a common subset
        Set commonSubset = new HashSet( encountered );
        CommonSubset.retainAll( characters );
```

```
            boolean areDisjunct = commonSubset.size() == 0;

            if (areDisjunct)
                System.out.println(characters + " and " + encountered +
                                   " are disjunct");
            else {
                // Determine superset and subset relations
                boolean isSubset = encountered.containsAll(characters);
                boolean isSuperset = characters.containsAll(encountered);
                if (isSubset && isSuperSet)
                    System.out.println(characters + " is equivalent to " +
                                       encountered);
                else if (isSubset)
                    System.out.println(characters + " is a subset of " +
                                       encountered);
                else if (isSuperset)
                    System.out.println(characters + " is a superset of " +
                                       encountered);
                else
                    System.out.println(characters + " and " + encountered + "
                                       have " + commonSubset + " in common");
            }
            // Remember the characters
            encountered.addAll( characters );
        }
    }
```

Running the program with the following arguments:
Java CharacterSets i said i am maids

Results in the following output:
[i] and [] are disjunct.
[d, a, s, i] is a superset of [i]
[i] is a subset of [d, a, s, i]
[a, m] and [d, a, s, i] have [a] in common.
[d, a, s, m, i] is equivalent to [d, a, s, m, i]

10.14) Lists: [1] [2]

- Lists are collections which maintain their elements in order (also called sequence), and can contain duplicates.
- In addition to operations inherited from the `Collection` interface, the first interface also defines operations that operate specially on lists: access by numerical position, search in list, customized iterators, operations on parts of a list(called open range-view operations).
- `Vector` and `ArrayList` classes implement dynamically resizable arrays, unlike the `ArrayList` class, the `Vector` class is thread-safe.

Category	Methods	Example & declaration
Reading	`Object get(int index)`	Returns the element at the specified position in this list.
Inserting	`void add(int index, Object element)`	Inserts the specified element at the specified position in this list (optional operation). Shifts the element currently at that position (if any) and any subsequent elements to the right (adds one to their indices).
	`void add(Object element)`	Appends the specified element to the end of this list (optional operation).

	`boolean addAll(Collection c)`	Appends all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's iterator (optional operation).
	`boolean addAll(int index, Collection c)`	Inserts all of the elements in the specified collection into this list at the specified position (optional operation). Shifts the element currently at that position (if any) and any subsequent elements to the right (increases their indices).
Replacing	`Object set(int index, Object element)`	Replaces the element at the specified position in this list with the specified element (optional operation). Returns the element previously at the specified position.
Deleting	`boolean remove(int index)`	Removes the first occurrence in this list of the specified element (optional operation). If this list does not contain the element, it is unchanged.
Searching	`int indexOf(Object element)`	Returns the index in this list of the first occurrence of the specified element, or -1 if this list does not contain this element.
	`int lastIndexOf(Object element)`	Returns the index in this list of the last occurrence of the specified element, or -1 if this list does not contain this element.
Iteration	`ListIterator listIterator()`	Returns a list iterator of the elements in this list (in proper sequence).
	`ListIterator listIterator(int index)`	Returns a list iterator of the elements in this list (in proper sequence), starting at the specified position in this list.
Open range view	`List subList(int fromIndex, int toIndex)`	Returns a view of the portion of this list between the specified `fromIndex`, inclusive, and `toIndex`, exclusive.

```
interface ListIterator extends Iterator {
    boolean hasNext();
    boolean hasPrevious();

    Object next();
    Object previous();

    int nextIndex();
    int previousIndex();

    void remove();
    void add(Object o);
    void set(Object o);
}
```

10.15) Maps: [1] [2]

- Defines mappings from keys to values **NOT** allowing duplicate keys, each key maps to at most one value.
- `HashMap` class is not thread-safe, the `Hashtable` class is.

Category	Methods	Example & declaration

Basic	`Object put(Object key, Object value)`	Associates the specified value with the specified key in this map (optional operation). If the map previously contained a mapping for this key, the old value is replaced.
	`Object get(Object key)`	Returns the value to which this map maps the specified key.
	`Object remove(Object key)`	Removes the mapping for this key from this map if present (optional operation).
	`boolean containsKey(Object key)`	Returns `true` if this map contains a mapping for the specified key.
	`boolean containsValue(Object value)`	Returns `true` if this map maps one or more keys to the specified value.
	`int size()`	Returns the number of key-value mappings in this map. If the map contains more than `Integer.MAX_VALUE` elements, returns `Integer.MAX_VALUE`.
	`boolean isEmpty()`	Returns `true` if this map contains no key-value mappings.
Bulk	`void putAll(Map t)`	Copies all of the mappings from the specified map to this map (optional operation). These mappings will replace any mappings that this map had for any of the keys currently in the specified map.
	`void clear()`	Removes all mappings from this map (optional operation).
Collection views	`Set keySet()`	Returns a set view of the keys contained in this map. The set is backed by the map, so changes to the map are reflected in the set, and vice-versa.
	`Collection values()`	Returns a collection view of the values contained in this map. The collection is backed by the map, so changes to the map are reflected in the collection, and vice-versa.
	`Set entrySet()`	Returns a set view of the mappings contained in this map. Each element in the returned set is a `Map.Entry`. The set is backed by the map, so changes to the map are reflected in the set, and vice-versa.

```
interface Entry {
    Object getKey();
    Object getValue();
    Object setValue(Object value);
}
```

10.16) Sorted Sets and Sorted Maps: [1] [2]

- Objects can specify their natural order by implementing the `Comparable` interface, or be dictated a total order by a comparator which implements the `Comparator` interface.
- All comparators implement the `Comparator` interface, which has the following single method:

```
int compare(Object obj1, Object obj2)
```

The `compare()` method returns a negative integer, zero or a positive integer if the first object is less than, equal to or greater than the second object, according to the total order.

- Objects can specify their natural order by implementing `Comparable` interface. Many of the standard classes in Java API, such as wrapper classes, `String`, `Date` and `File` implement this interface. This interface defines a single method:

  `int compareTo(Object o)`

  returns negative, zero, positive if the current object is less than, equal to or greater than the specified object.

In this case a natural comparator queries objects implementing `Comparable` about their natural order. Objects implementing this interface can be used:

- As elements in a sorted set.
- As keys in sorted map.
- In lists which can be sorted automatically by the `Collections.sort()` method.

SortedSet interface methods	SortedMap interface method
`SortedSet headSet(Object toElement)` returns a view of a portion of this sorted set, whose elements are strictly less than the specified element.	`SortedMap headMap(Object toElement)`
`SortedSet tailSet(Object toElement)` returns a view of a portion of this sorted set, whose elements are greater than or equal the specified element.	`SortedMap tailMap(Object toElement)`
`SortedSet subSet(Object fromElement, Object toElement)` returns a view of a portion of this sorted set, whose elements range from `fromElement` inclusive to `toElement` exclusive.	`SortedMap subMap(Object fromElement, Object toElement)`
`Object first()` returns the first (minimum) element currently in this sorted set.	`Object firstKey()`
`Object last()` returns the last (maximum) element currently in this sorted set.	`Object lastKey()`
`Comparator comparator()` returns the comparator associated with this sorted set, or null if the natural ordering is used.	`Comparator comparator()`

TreeSet	TreeMap
TreeSet()	TreeMap()
TreeSet(Comparator c)	TreeMap(Comparator c)
TreeSet(Collection c)	TreeMap(Map m)
TreeSet(SortedSet s)	TreeMap(SortedMap m)

FINAL NOTES: [12]

- `Vector(5,10)` means initial capacity 5, additional allocation (capacity increment) by 10.
- `Stack` extends `Vector` and implements a LIFO stack. With the usual `push()` and `pop()` methods, there is a `peek()` method to look at the object at the top of the stack without removing it from the stack.
- `BitSet` class implements a `Vector` of bits that grows as needed. Each component of the bit set has a `boolean` value. The bits of a `BitSet` are indexed by nonnegative integers. Individual indexed bits can be examined, set, or cleared. One `BitSet` may be used to modify the contents of another `BitSet` through logical AND, logical inclusive OR, and logical exclusive OR operations. By default, all bits in the set initially have the value `false`. A `BitSet` has a size of 64, when created without specifying any size.

- `ConcurrentModificationException` exception (extends `RuntimeException`) may be thrown by methods that have detected concurrent modification of a backing object when such modification is not permissible. For example, it is not permssible for one thread to modify a Collection while another thread is iterating over it. In general, the results of the iteration are **UNDEFINED** under these circumstances. Some Iterator implementations (including those of all the collection implementations provided by the JDK) may choose to throw this exception if this behavior is detected. Iterators that do this are known as *fail-fast* iterators, as they fail quickly and cleanly, rather that risking arbitrary, non-deterministic behavior at an undetermined time in the future.

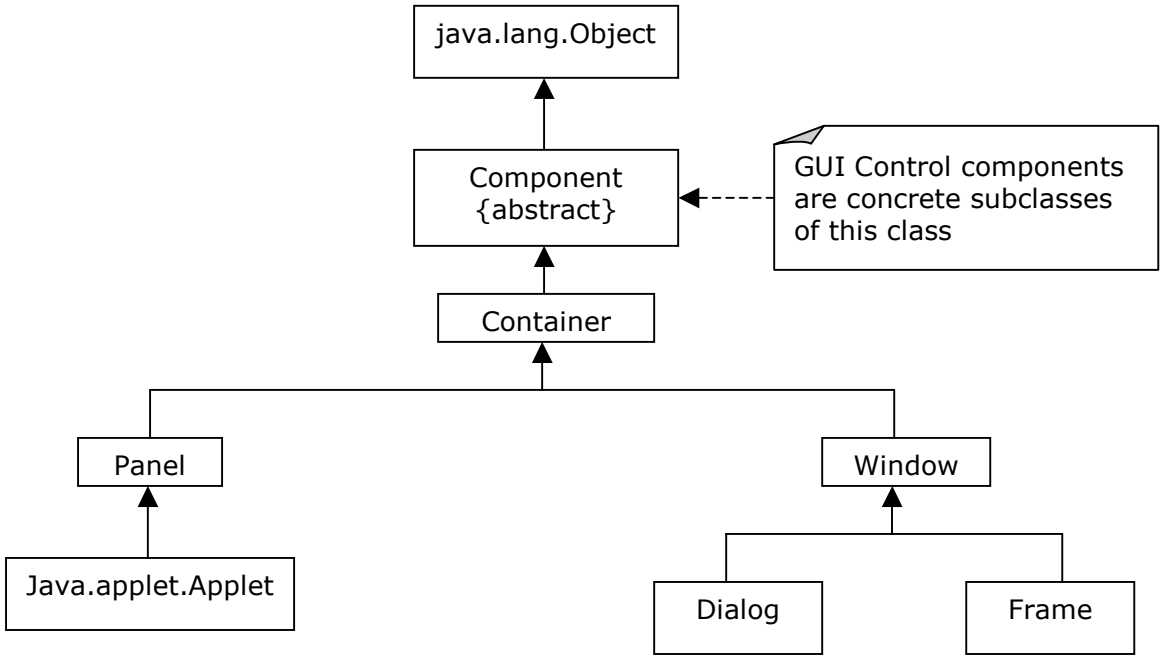
Chapter 11:

AWT components

11.1) The Java Foundation Classes (JFC) provide two frameworks for building GUI based application, **BUT** both relies on the same event handling model: [1]

AWT Abstract Windowing Toolkit	Swing
Relies on the underlying windowing system on a specific platform to represent its GUI components.	Implements a new set of light weight GUI components that are written in java and have a pluggable look and feel, they are not dependent on the underlying windowing system.

11.2) Partial inheritance hierarchy of components and containers in AWT: [1]



`Component`	The superclass of all non-menu related components that provides basic support for handling of events, changing of component size, controlling of fonts and colors, and drawing of components and their contents.
`Container`	A `container` is a component that can accommodate other components and also other containers. Containers provide the support for building complex hierarchical graphical user interface.
`Panel`	A `Panel` is a container ideal for packing other components and panels to build component hierarchies.
`Applet`	An `Applet` is a specialized panel that can be used to develop programs that run in a web browser.
`Window`	The `Window` class represents a top-level window that has no title, menus or borders.
`Frame`	A `Frame` is optionally user resizable and movable top-level window that can have a title-bar, an icon, and menus.
`Dialog`	The `Dialog` class defines an independent, optionally user resizable window that can only have a title-bar and a border. A Dialog window can be modal, meaning that all input is directed to this window until it is dismissed.

The objects of these classes can be populated with GUI control components like buttons, checkboxes, lists and text fields to provide the right interface and interaction with the user.

11.3) `Component` class: [1]

- All non-menu related elements that comprise a graphical user interface are divided from this abstract class.

```
Dimension getSize()          // get size in pixels
void setSize(int width, int height)
```

```
void setSize(Dimension d)

Point getLocation()         // return top-left corner of the component
void setLocation(int x, int y)
void setLocation(Point p)

Rectangle getBounds()
void setBounds(int x, int y, int width, int height)
void setBounds(Rectangle r)

void setForeground(Color c)
void setBackground(Color c)

Font getFont()
void setFont(Font f)

void setEnabled(boolean b)
```

If set to `true`, the components acts as normal, and can respond to user input and generate events, if the argument is `false`, then the component appears grayed out and does not respond to user interaction, initially all components are enabled.

```
void setVisible(boolean b)
```

It influence the visibility of the child components, default visibility is true for all components except `Window`, `Frame`, `Dialog`, whose instances must explicitly be made visible by this method.

11.4) `Container` class: [1]

- It defines components for nesting components in a container.
- It provides functionality for building complex hierarichal graphical user interfaces.
- It defines a component hierarchy in contrast to the inheritance hierarchy defined by classes.
- It provides the overloaded method `add()` to include components in the container.
- A container uses a layout manager to position its components in the container.

11.5) `Panel` class: [1]

- It provides intermediate level of GUI organization.
- It is a recursively nested container that is not a top-level window.
- It does not have a title, menus or borders.
- Ideal for packing other components and panels to build component hierarchies using inherited `add()` methnod.

11.6) `Applet` class: [1]

Used to develop programs that run in a web browser.

11.7) `Window` class:

- It does not have title, menus or borders.
- Represent a top level window, and cannot be incorporated into other components.
- It is seldom used directly, instead its subclasses `Frame` and `Dialog` are used to provide independent top-level windows.

`void pack()`

initialize the layout manager of the sub-components, leading to the window size being set to match the preferred size of its sub-components, usually called after the component hierarchy has been constructed to facilitate the layout of the subcomponents in the window.

`void show()`

used to make the window actually visible and bring it to front, unlike other components, windows are initially hidden, **NOTE:** `setVisible()` method can be used to make a window visible without bringing it to the front.

`void dispose()`

when a window is no longer needed, this method is called to free the windowing resources, it does not actually delete the window object, the window object should not be used often a call to this method.

11.8) `Frame` class: [1]

- A frame is an optionally user resizable and movable top-level window that can have a title-bar, an icon, and menus.
- Usually the starting point of a GUI application and serves as a root of the component hierarchy.
- It can contain several panels which inturn can hold other GUI control components and other nested panels

```
Frame()
Frame(String title)
```

All constructors create an initially invisible frame.

11.9) `Dialog` class: [1]

- A dialog is an optionally user resizable and movable top-level window with a title bar.
- It doesn't have an icon or a menu-bar.
- It can be a root of a component hierarchy.
- It can be modal, meaning that all input is directed to its window until it is dismissed.

```
Dialog(Frame parent)
Dialog(Frame parent, boolean modal)
Dialog(Frame parent, String title)
Dialog(Frame parent, String title, boolean modal)
```

All constructors create an initially invisible dialogbox.

- A dialog box is non-modal by default.

11.10) GUI control components: [1]

- Are the primary components of a graphical user interface that enable user interaction.
- They are concrete subclasses of the `Component` class

`Button`	A button with a textual label, designed to invoke an action when pushed, called a push button.
`Canvas`	A generic component for drawing and designing new GUI components.
`Checkbox`	A checkbox with a textual label that can be toggled on and off. Checkboxes can be grouped to represent radio button.
`Choice`	A component that provides a pop-up menu of choices. Only the current choice is visible in the Choice component.
`Label`	A label is a component that displays a single line of read-only, non-selectable text.
`List`	A component that defines a scrollable list of text items.
`Scrollbar`	A slider to denote a position or a value.
`TextField`	A component that implements a single line of optionally editable text.
`TextArea`	A component that implements multiple lines of optionally editable text.

- The following three steps are essential in making use of a GUI control manager:

(a) A GUI component is created by calling the appropriate constructor.

(b) The GUI component is added to a container using a layout manager; this usually invokes invoking the overloaded method `add()` on a container with the GUI control component as the argument.

(c) Listeners are registered with the GUI component, so that they can receive events when these occur.

- Since these controls are subclass of the `Component` class, they all generate keyboard and mouse events.

11.11) `Button` class: [1]

```
Button ();
Button (String label);

String getLabel ();
void setLabel (String label);
```

11.12) `Canvas` class: [1]

- It doesn't have any default graphical representation or any event handlers of its own.
- Is usually subclassed to customized GUI components consisting of drawings or images, and can handle user input events relating to mouse and keyboard actions.
- The `paint()` method is usually overridden to render graphics in the component.

11.13) `Checkbox` & `CheckboxGroup` classes: [1]

- A Checkbox object can be in one of two states:
  `true` : meaning it is checked.
  `false` : meaning it is unchecked.

```
Checkbox();
Checkbox(String label);
Checkbox(String label, boolean state);
Checkbox(String label, boolean state, CheckboxGroup group);
```

- If the state is not explicitly specified in the appropriate constructor, the initial state is unchecked.
- A `Checkbox` can be incorporated in a `CheckboxGroup` to implement radio buttons.
- Unless the `CheckboxGroup` is not specified in the appropriate constructor, the `Checkbox` is not part of any `CheckboxGroup`.

```
boolean getState();
void setState(boolean state);

String getLabel();
void setLabel(String label);

CheckboxGroup getCheckboxGroup();
void setCheckboxGroup(CheckboxGroup group);
```

- `CheckboxGroup` only allows a single selection.

```
Checkbox getSelectedCheckbox();
void setSelectedCheckbox(Checkbox box);
```

- `CheckboxGroup` object does **NOT** have a graphical representation and is **NOT** a subclass of component, it is just a class to implement mutual exclusion among a set of checkboxes.

11.14) `Choice` class: [1]

- Constructing a pop-up menu of choices involves the following steps:
  (a) Creating a `Choice` object using the single default constructor provided.
  (b) Adding the items using the `add()` method; **NOTE:** that the items in the pop-up menu are strings.

```
void add(String item);

int getItemCount();

String getItem(int index);

String getSelectedItem();
int getSelectedIndex();

void select(int pos);
void select(String str);
```

11.15) `Label` class: [1]

- It doesn't generate any special events.

```
Label();
Label(String text);
Label(String text, int alignment);
```

- Alignment can be specified by the following constants of the `Label` class; the default alignment is `LEFT`:

```
public static final int LEFT
public static final int CENTER
public static final int RIGHT

String getText();
void setText(String text);

int getAlignment();
void setAlignment(int alignment);
```

11.16) `List` class: [1]

- The number of items that can be visible in the list box is defined as the number of rows in the list.
- The list can be of course have any number of text items, and a scrollbar appears when necessary to scroll the list.
- A list can be constructed to allow either single or multiple selection(s).
- By default, multiple selection is not allowed.

```
List();
List(int rows);
List(int rows, boolean multipleMode);
```

- Constructing a list involves the following steps:
(a) Creating a list object.
(b) Adding the items using the `add()` method, the items are strings.

```
void add(String item);
void add(String item, int index);

int getRows();

boolean isMultipleMode();

int getItemCount();

String getItem(int index);
String[] getItems();

String getSelectedItem();
int getSelectedIndex();
String[] getSelectedItems();
int[] getSelectedIndexes();

void select(int index);
void select(String str);
void deselect(int index);
```

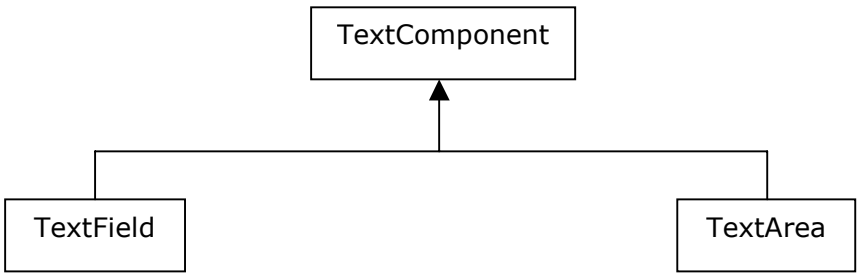
11.17) `Scrollbar` class: [1]

- can be used to:
  (a) indicate a relative position of the visible contents in relation to the whole document.
  (b) controller to specify a value from a given interval.

- `Scrollbar` has vertical orientation as default, and it can be either horizontal or vertical.
- Clicking on any gadgets(arrowheads) scrolls one unit, the default unit size is 1.
- Clicking on the area between an arrowhead and the slider scrolls by one block; the default size is 10 units.

```
Public static final int HORIZONTAL
Public static final int VERTICAL

Scrollbar();
Scrollbar(int orientation);
Scrollbar(int orientation, int value, int visible, int minimum, int
maximum);

Int getValue();

Int getMinimum();
Int getMaximum();
Int setMinimum(int newMinimum);
Int setMaximum(int newMaximum);

Int getVisibleAmount();
void setVisibleAmount(int newAmount);

int getUnitIncrement();
void setUnitIncrement(int v);
int getBlockIncrement();
void setBlockIncrement(int v);
```

11.18) `TextField` and `TextArea`: [1]



- `TextComponent` is a subclass of `java.awt.Component`
- Text in the `TextComponent` can be read-only or editable.
- The size of the text field is measured in columns.

```
TextField();
TextField(String text);
TextField(int columns);
TextField(String text, int columns);
```

- `TextArea` class implements multiple lines of optionally editable text, these lines are separated by the `'\n'` (new line character).
- The size of `TextArea` is measured in columns and rows

```
TextArea();
TextArea(String text);
TextArea(int rows, int columns);
TextArea(String text, int rows, int columns);
TextArea(String text, int rows, int columns, int scrollbars);

public static final int SCROLLBARS_BOTH
public static final int SCROLLBARS_VERTICAL_ONLY
public static final int SCROLLBARS_HORIZONTAL_ONLY
public static final int SCROLLBARS_NONE
```

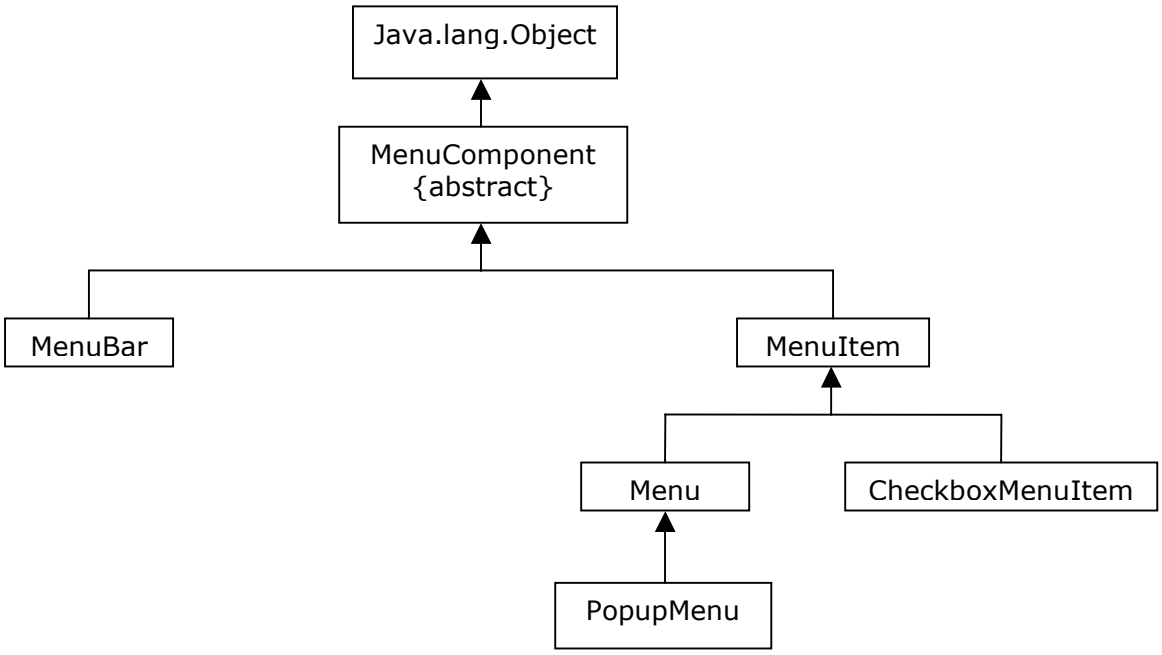
- By default, `SCROLLBARS_BOTH` is selected.

Methods for both `TextField` and `TextArea`:

```
int getColumns();
void setColumns(int columns);

String getText();
void setText(String text);

String getSelectedText();

boolean isEditable();
void setEditable(boolean b);
```

- For fixed pitch fonts, the number of columns is equal to the number of characters in the text line, for proportional (variable) pitch fonts, the column width is taken to be average character width of the font used for rendering the text.

11.19) Menu components: [1]



- `MenuBar` class implements a menu bar that can contain pull-down menus.
- `Menu` class are the pull-down menus; and the `add()` method can be used to add menus to a menu-bar, and the `remove()` method can be used to remove menus from a menu-bar.
- A menu-bar can be attached to a `Frame` object; **NOTE:** that an `Applet` is not a subclass of the `Frame` class, and therefore **CANNOT** have a menu bar.
- The `MenuItem` class defines a menu item that has a textual label, and a keyboard shortcut can also be defined for a menu item.
- Since a `Menu` is also a `MenuItem`, menus can be nested to create submenus.
- A `MenuItem` object can be used by `add()` method; a separator can be added using `addSeparator()` method in the `Menu` object.
- `PopupMenu` represent a pop-up menu that can be poppedup at a specified position within a component, and **CANNOT** be contained in a menu-bar.
- `CheckboxMenuItem` class implements a checkbox with a textual label that can appear in a menu.
- Follow these steps to create a menu-bar for a frame:
  (a) Create a menubar
  (b) Create a menu
  (c) Create menu items and add them to the menu (appear from top to bottom according to the order in which they are added to the menu).

(d) Add each menu to a menubar. The menus appear from left to right in the menu-bar.
(e) add the menu-bar to the frame.
<framename>.`setMenuBar`(*MenuBarObject*)

Chapter 12:

Layout Manager

12.1) A layout manager implements a layout policy that defines spatial relationships between components in a container; these relationships or constraints specify the placement and sizes of components and come into play when the container is resized. A layout manager works in conjunction with a container holding the components. [1]

12.2) Overview of layout manager: [1]

Manager	Description
`FlowLayout`	Lays out the components in row-major order: in rows growing from left to right, and rows placed top to bottom in the container(→↓). This is the default layout manager for the `Panel` and `Applet`.
`GridLayout`	Lays out the components in a specified rectangular grid, from left to right in each row, and filling rows from top to bottom in the container(→↓).
`BorderLayout`	Up to five components can be placed in a container in locations specified by the following directions: north, south, west, east and center. This is the default layout manager for `Window` and its subclasses (`Frame` and `Dialog`)
`CardLayout`	Components are handled as a stack of indexed cards with only the top component being visible in the container.
`GridBagLayout`	Customizable and flexible layout manager that lays out the components in a rectangular grid. A component can occupy multiple cells in the grid.

12.3) Common methods for designing a layout. [1]

```
LayoutManager getLayout();
void setLayout(LayoutManager mgr);
```

12.4) For adding components to a container: [1]

```
Component add(Component comp);
Component add(Component comp, int index);
void add(Component comp, Object constraints);
void add(Component comp, Object constraints, int index);
```

- The index argument can be used to specify a position where the component should be inserted; the value –1 inserts the component at the end, which is the default placement.
- The constraints argument specifies properties that are used by the layout manager to place the components in the container. These properties are specific to the layout manager used.

12.5) Components can be removed from a container: [1]

```
void remove(int index);
void remove(Component comp);
void removeAll();
```

12.6) Applications usually **NEVER** call the methods of the layout manager directly; since the layout manager is registered with a container, the container calls the appropriate methods in the layout manager. [1]

Methods of the Container class	Methods of the LayoutManager Interface
`add()`	`addLayoutComponent()`
`doLayout()`	`layoutContainer()`
`getMinimumSize()`	`minimumLayoutSize()`
`getPreferredSize()`	`preferredLayoutSize()`
`remove or removeAll()`	`removeLayoutComponent()`

A typical scenario for updating the layout, when the container size changes, is as follows:

(a) The container's `invalidate()` method is invoked. This makes the container and the parents above it in the component hierarchy as neading layout updating.

(b) The container's `validate()` method is called. This invocation leads to the following chain of events, resulting in the layout of the container and its parents being updated:
- The `validate()` method invokes the container's `doLayout()` method.
- The `doLayout()` method delegates the job to its layout manager, by calling the layout manager's `layoutContainer()` method and by calling the layout manager's `layoutContainer()` method and passing itself as the argument.

12.7) These are two types of containers: [1]
- Containers that must be attached to a parent container. They can't exist on their own. Objects of the `Panel` class and its subclass `Applet` are typical examples.
- Containers that exist independently and cannot be put in other containers. They are sometimes called top-level windows. They denote the root of a component hierarchy. `Window` class and its subclasses `Frame` and `Dialog` are typical examples.

12.8) A component can request a certain size, **NOTE:** it is not certain that the layout manager will honor it.

12.9) Layout manager in the AWT always gives precedence to placement if honoring the preferred size would violate the layout policy. [1]

12.10) `FlowLayout` manager: [1]
- Components added to the container are placed in rows that grow from left to right, the rows are constructed from top to bottom in the container (sometimes called row-major allocation), components towards the end of a row spill over to the next row if there is not enough space in the current row.
- It honors the preferred size of the components, i.e. the size of the components **NEVER** changes, regardless of the size of the container; if the container is too small, the rendering of the component appears cropped.
- It is the default layout manager of the `Panel`, and hence, `Applet` class.

```
FlowLayout();
FlowLayout(int alignment);
FlowLayout(int alignment, int horizontalgap, int verticalgap);
```

Alignmentand gap properties apply to all the components in the container, the default alignment is centered rows, and the default gap is five pixels both vertically and horizontally.

```
public static final int LEFT
public static final int RIGHT
public static final int CENTER
```

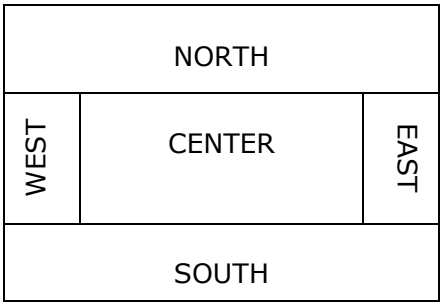
12.11) `GridLayout` manager: [1]
- It divides the region of the container into rectangular grid.
- Each component is placed in a cell in this grid, and this position uniquely identified by the row and column number that are one-based.
- Only one component can be placed in each cell.
- All the cells in the grid have the same size.
- The cell size is dependent on the number of components to be placed in the container and the container's size.
- A component is resized to fill the cell. (To avoid components being stretched is to first stick the component in a panel, and then add the panel to the container because components in a panel does not stretch when the `FlowLayout` manager is used).
- It ignores a component's preferred size.
- After a `GridLayout` has been constructed and registered with a container, components are added left to right and top to bottom (i.e. row-major)

```
GridLayout(); // ≡ GridLayout(1,0); // one row with any number of
  components added.
GridLayout(int rows, int columns);
```

```
GridLayout(int rows, int columns, int horizontalgap, int verticalgap);
```

- Either rows or columns can be zero, **BUT NOT** both; the geometry of the grid is then determined by the non-zero value and the number of components added. The default gap between components is zero pixels, both horizontally and vertically.

12.12) `BorderLayout` manager: [1]



- Allows one component to be placed in each of the four compass directions in a container, any space left over can be used for a fifth component in the center of the container.
- Not all regions need to be occupied with a component.
- Adding more than one component to a region is not recommended; **ONLY** the last component added to a region is shown.
- The order in which the components are added to the container is irrelevant.
- It is the default layout manager for the `Window` class and its subclasses (`Frame` and `Dialog`).

```
BorderLayout();
BorderLayout(int horizontalgap, int verticalgap);
```

The default gap in either direction is zero pixels.

```
Public static final String NORTH = "North"
Public static final String SOUTH = "South"
Public static final String EAST = "East"
Public static final String WEST = "West"
Public static final String CENTER = "Center"
```

- Adding components:

```
Component add(Component comp);
void add(Component comp, Object constraints);
```

- The default region is CENTER, and the region can be explicitly specified using the constraints argument.
- If a north or a south component exists, it will stretch horizontally across the width of the container, and the `BorderLayout` manager will attempt to honor the preferred height of the components in the north and south regions.
- A west or east component is sandwiched between any north or south component, otherwise it stretches vertically along the height of the container, and the `BorderLayout` manager will attempt to honor the preferred width of the components in the west and east regions.
- The center can be stretched both horizontally and vertically.

12.13) `CardLayout` manager: [1]

- Handles the component in a container like a stack of indexed cards, where **ONLY** the top is visible, and it fills the whole region of the container.
- The card layout does not give any visual clue that the container consists of a stack of components.

```
CardLayout();
CardLayout(int horizontalgap, int verticalgap);
```

- Both horizontal and vertical gaps between the edges of a component and borders of the container can be specified, the default gap in either direction is zero pixels.
- Individual components can be added to a container by using the `add()` methods from the `Container` class. The constraints argument in the `add()` methods is a `String` object which can be associated with the component, and later used to make this particular component visible using the `show()` method.

```
void show(Container parent, String name);
```

```
void first(Container parent);
void next(Container parent);
void previous(Container parent);
void last(Container parent);
```

The parent argument is the container associated with the card layout manager, these methods can be used to choose which card should be shown.

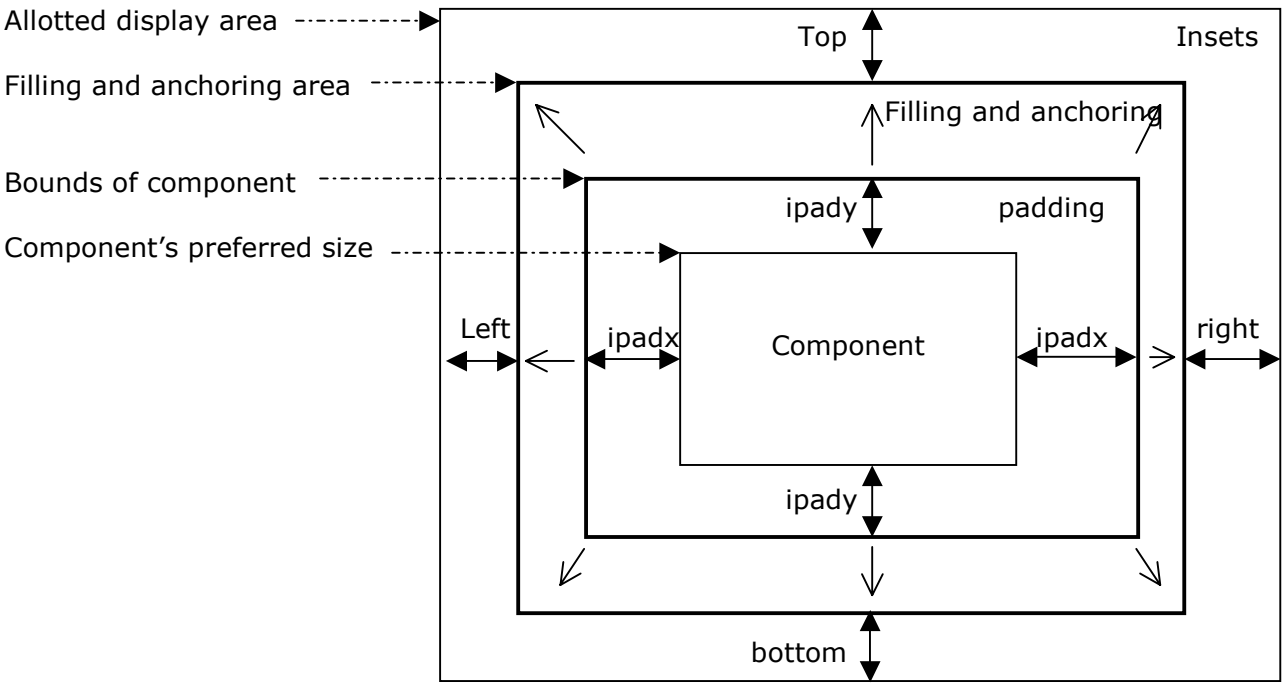
NOTE: methods are invoked on a `CardLayout` object not on the parent container.

12.14) `GridBagLayout` manager: [1]

- The `GridBagLayout` policy uses a rectangular grid but unlike the grid layout manager, a component can occupy multiple cells in the grid, and the width and the height of the cells need not to be uniform, i.e. the component can span several rows and columns, but the region it occupies is always rectangular.
- The components in the container can have different sizes, i.e the `GridBagLayout` allows different size components to be aligned in the container.
- Constructing the layout:
  (a) Create an object of the class `GridBagLayout` using the default constructor.
  (b) Set the layout manger for the container.
  (c) Create an object of the class `GridBagConstraints`.
  (d) For each component to be added:
    i- Fill in the layout information in the `GridBagConstraints` object.
    ii- Add the component supplying the `GridBagConstraints`.

- **NOTE**: The same `GridBagConstraints` object can be reused for adding other component; and can be specified using the public data members.

```
GridBagConstraints();
GridBagConstraints( int gridx, int gridy,
                    int gridwidth, int gridheight,
                    double weightx, double weighty,
                    int anchor,
                    int fill,
                    Insets insets,
                    int ipadx, int ipady);
```



Location:

```
int gridx
int gridy
```

They define the column and row positions of the upper left corner of the component in the grid.
Both values can be set to `GridBagConstraints.RELATIVE`, the components are then added in relation to the previous component and its default values for these member variables.

Dimension:

```
int gridwidth
int gridheight
```

They specify the number of cells occupied by the component horizontally and vertically; can be either `GridBagConstraints.RELATIVE` or `GridBagConstraints.REMAINDER`, the value `GridBagConstraints.REMAINDER` indicates that the component extends to the end of the row or column (i.e. last component in the row or column) the default value is 1 for each cell.
The value `GridBagConstraints.RELATIVE` should be used to specify that the component is next-to-last in its row (for gridwidth) or column(for gridheight).

Growth Factor:

```
double weightx
double weighty
```

Define the portion of the "slack" that should be allocated to the area occupied by the component.
Default value is zero for both, i.e. the area allocated to the component does not grow beyond the preferred size.

Anchoring:

```
int anchor
public static final int CENTER
public static final int NORTH
public static final int NORTHEAST
public static final int EAST
public static final int SOUTHEAST
public static final int SOUTH
public static final int SOUTHWEST
public static final int WEST
```

`public static final int NORTHWEST`
Specify where a component should be placed within its display area. If the component does not fill its allocated area, it can be anchored specifying one of the constants. The default value is `GridBagConstraints.CENTER`.

Filling:
`int fill`
`public static final int NONE`
`public static final int BOTH`
`public static final int HORIZONTAL`
`public static final int VERTICAL`
How the component is to stretch and fill its display area. The default is `GridBagConstraints.NONE`

Padding:
`int ipadx`
`int ipady`
Specifies the padding that will be added internally to each side of the component. The dimension of the component will be padded in the horizontal ($2 \times$ ipadx) and the vertical ($2 \times$ ipady). The default value is zero pixels in either direction.

Insets:
`Insets insets`
The insets variable defines the external padding (border) around the component and its display area. The default value is (0, 0, 0, 0) specifying top, left, bottom, right.

Chapter 13:

Event Handling

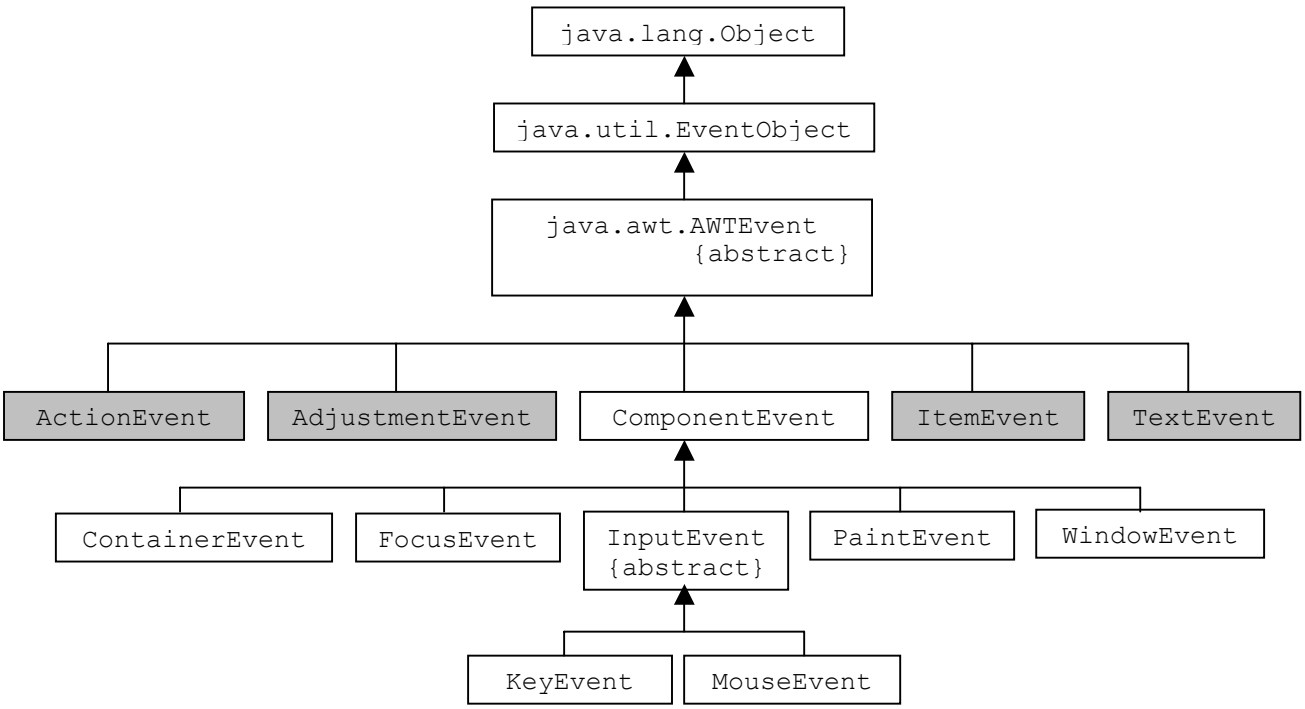
13.1) Event handling in java is based on the *event delegation model*. Its principal elements are: [1]

(1) Event classes that can encapsulate information about different types of user interaction.
(2) Event source objects that inform event listeners about events when these occur and supply the necessary information about these events.
(3) Event listener objects that are informed by an event source when designated events occur, so that they can take appropriate action.

13.2) Handling events in a GUI application, using the event delegation model, can be divided into the following two tasks when building the application: [1]

(1) Setting up the propagation of events from event sources to event listeners.
(2) Providing the appropriate actions in event listeners to deal with the events received.

13.3) Partial Inheritance hierarchy of Event classes. [1]



13.4) The `EventObject` class provides a method that returns the object that generated the event: [1]

```
Object getSource()
```

13.5) The `AWTEvent` class provides a method that returns an event's id: [1]

```
int getID()
```

13.6) The `AWTEvent` class is divided into two groups: [1]

Semantic Events	Low-level Events
ActionEvent	ComponentEvent
AdjustmentEvent	ContainerEvent
ItemEvent	FocusEvent
TextEvent	KeyEvent
	MouseEvent
	PaintEvent
	WindowEvent

High-level semantic events represent user interface action with a GUI component (clicking a button, selecting a menu item, selecting a checkbox, scrolling, and changing text); while Low-level events represent input and window operations. Several low-level events can constitute a single semantic event.

13.7) `ActionEvent` class (Semantic Event): [1]

- Generated when an action performed on a GUI component
- The GUI components that generate this event are:
  - `Button`: when a button is clicked.
  - `List`: when a list item is double-clicked.
  - `MenuItem`: when a menu item is selected.
  - `TextField`: when the Enter key is hit in the text field.
- This function returns the command name associated with this action(button label, list-item name, menu-item name or text)

```
String getActionCommand()
```

- This function returns the sum of the modifier constant corresponding to the keyboard modifiers held down during the action.

```
int getModifiers()
public static final int SHIFT_MASK
public static final int CTRL_MASK
public static final int META_MASK
public static final int ALT_MASK
```

13.8) `AdjustmentEvent` class (Semantic Event): [1]

- Generated when adjustments are made to an adjustable component like a scrollbar
- The GUI component that generates the adjustment event is
  - `Scrollbar`: when any adjustment is made to the scrollbar.
- This function returns the current value designated by the adjustable component.

```
int getValue()
```

13.9) `ItemEvent` class (Semantic Event): [1]

- Generated when an item is selected or deselected in an item selectable component.
- GUI components that generate this event are:
  - `Checkbox`: when the state of the checkbox changes.
  - `CheckboxMenuItem`: when the state of the checkbox associated with a menu item changes.
  - `Choice`: when an item is selected or deselected in a choice list.
  - `List`: when an item is selected or deselected from a list.
- This function returns the object that was selected or deselected in a choice-list (label of the `Checkbox` or `CheckboxMenuItem`, or label of the item in a `Choice` or a `List`, is returned as `String` object)

```
Object getItem()
```

- This function returns a value indicating whether it was selection or deselection that took place.

```
int getStateChange()
```

13.10) `TextEvent` class (Semantic Event): [1]

- Generated whenever the content of a text component is changed.
- GUI components that generate this event are subclasses of the TextComponent class:
  - `TextArea`
  - `TextField`

13.11) `ComponentEvent` class (Low-level Event): [1]

- Generated when a component is hidden, shown, moved, or resized, they are handled by the AWT, and normally not directly dealt by the application.
- This function returns the same object as `getSource()` method, but the returned reference is of type Component.

```
Component getComponent()
```

13.12) `FocusEvent` class (Low-level Event): [1]

- Generated when a component gains or loses focus, having the focus means that the component can receive keystrokes.

- The inherited method `getID()` from its superclass `AWTEvent` can be used to determine whether the focus was lost or gained (`FocusEvent.FOCUS_LOST`, `FocusEvent.FOCUS_GAINED`).
- Focus can be lost either permanently or temporarily, and this can be determined by the method.

```
Boolean isTemporary()
```

13.13) `KeyEvent` class (Low-level Event): [1]

- Generated when the user presses or releases a key, or does both (i.e. types a character)
- The inherited method `getID()` from its superclass `AWTEvent` can be used to denote the constant indicating the action:

```
public static final int KEY_PRESSED
public static final int KEY_RELEASED
public static final int KEY_TYPED
```

- The inherited method `long getWhen()` from the parent class `InputEvent` can be used to get the time when the event took place.
- To get the integer key-code (that are defined as constants in the KeyEvent) associated with the key if pressed or released, you can use the following function:

```
int getKeyCode()
```

- To get the integer Unicode that results from hitting a key, you can use the following function:

```
char getKeyChar()
```

13.14) `MouseEvent` class (Low-Event): [1]

- Generated when the user moves the mouse or presses a mouse button.
- The inherited method `getID()` from its superclass `AWTEvent` can be used to denote the exact action is identified by the following constants in the `MouseEvent` class:

```
public static final int MOUSE_PRESSED
public static final int MOUSE_RELEASED
public static final int MOUSE_CLICKED
public static final int MOUSE_DRAGGED
public static final int MOUSE_MOVED
public static final int MOUSE_ENTERED
public static final int MOUSE_EXITED
```

- The inherited method `long getWhen()` from the parent class `InputEvent` can be used to get the time when the event took place.[1]
- Use the following functions to get the x- and/or y- position of the event relative to the source component.[1]

```
int getX()
int getY()
Point getPoint()
```

- Use the following function to translates the event's coordinates to a new position by adding specified x (horizontal) and y (vertical) offsets

```
void translatePoint(int dx, int dy)
```

- Use the following function to return the number of clicks associated with the event, which is useful for detecting such events as double clicks.[1]

```
int getClickCount()
```

13.15) `PaintEvent` class (Low-Event): [1]

- Generated when a component should have its `paint()/update()` methods invoked. These events are handled internally by the AWT and should not directly be dealt with by the application.

13.16) `WindowEvent` class (Low-Event): [1]

- Generated when an important operation is performed on a window, these operations are identified by the following constants, the inherited `getID()` method returns the specific type of the event:

```
public static final int WINDOW_OPENED
```

Only once for a window when it is create , opened and made visible the first time.

`public static final int WINDOW_CLOSING`

When the user action dictates that the window should be closed, the application should explicitly call either `setVisible(false)` or `dispose()` on the window as a response to this event.

`public static final int WINDOW_CLOSED`

After the Window has been closed as result of a call to `setVisible(false)` or `dispose()`

`public static final int WINDOW_ICONIFIED`

When the window is iconified

`public static final int WINDOW_DEICONIFIED`

When the window is de-iconified

`public static final int WINDOW_ACTIVATED`

When the window is activated, i.e. keyboard events will be delivered to the window or its subcomponents.

`public static final int WINDOW_DEACTIVATED`

When the window is deactivated, i.e. keyboard events will no longer be delivered to the window or its subcomponents.

- It has a useful method that returns the Window object that caused the event.

`Window getWindow()`

13.17) Semantic Event handling: [1]

Event Type	Event Source	Listener Registration and Removal Methods provided by the source	Event Listener Interface implemented by a listener
ActionEvent	Button List MenuItem TextField	addActionListener removeActionListener	ActionListener
AdjustmentEvent	Scrollbar	addAdjustmentListener removeAdjustmentListener	AdjustmentListener
ItemEvent	Choice Checkbox CheckboxMenuItem List	addItemListener removeItemListener	ItemListener
TextEvent	TextArea TextField	addTextListener removeTextListener	TextListener

13.18) Low-Level Event Handling: [1]

Event Type	Event Source	Listener Registration and Removal Methods provided by the source	Event Listener Interface implemented by a listener
ComponentEvent	Component	addComponentListener removeComponentListener	ComponentListener
ContainerEvent	Container	addContainerListener removeContainerListener	ContainerListener
FocusEvent	Component	addFocusListener removeFocusListener	FocusListener
KeyEvent	Component	addKeyListener removeKeyListener	KeyListener
MouseEvent	Component	addMouseListener removeMouseListener addMouseMotionListener removeMouseMotionListener	MouseListener MouseMotionListener
WindowEvent	Window	addWindowListener removeWindowListener	WindowListener

13.19) Semantic Event listener interfaces and their methods:[1]

Event Listener Interface	Event Listener methods
`ActionListener`	`actionPerformed(ActionEvent evt)`
`AdjustmentListener`	`adjustmentValueChanged(AdjustmentEvent evt)`
`ItemListener`	`itemStateChanged(ItemEvent evt)`
`TextListener`	`textChanged(TextEvent evt)`

13.20) Low-level Event listener interfaces and their methods: [1]

Event Listener Interface	Event Listener methods

`ComponentListener`	`componentHidden(ComponentEvent evt)`
	`componentMoved(ComponentEvent evt)`
	`componentResized(ComponentEvent evt)`
	`componentShown(ComponentEvent evt)`

`ContainerListener`	`componentAdded(ComponentEvent evt)` `componentRemoved(ComponentEvent evt)`
`FocusListener`	`focusGained(FocusEvent evt)` `focusLost(FocusEvent evt)`

`KeyListener`	`keyPressed(KeyEvent evt)`
	`keyReleased(KeyEvent evt)`
	`keyTyped(KeyEvent evt)`

`MouseListener`	`mouseClicked(MouseEvent evt)`
	`mouseEntered(MouseEvent evt)`
	`mouseExited(MouseEvent evt)`
	`mousePressed(MouseEvent evt)`
	`mouseReleased(MouseEvent evt)`

`MouseMotionListener`	`mouseDragged(MouseEvent evt)` `mouseMoved(MouseEvent evt)`

`WindowListener`	`windowActivated(WindowEvent evt)`
	`windowClosed(WindowEvent evt)`
	`windowClosing(WindowEvent evt)`
	`windowDeactivated (WindowEvent evt)`
	`windowDeiconified(WindowEvent evt)`
	`windowIconified(WindowEvent evt)`
	`windowOpened(WindowEvent evt)`

13.21) How is the association between the source and listener established? [1]

(2) Each event source defines methods for registering(addXListener()) and removing(removeXListener()), which implement a particular listener interface.

(3) Each XListener interface defines methods which accept a specific event type as argument.

13.22) How does the event source inform the event listener that a particular event it is interested in has occurred? [1]

(1) It calls a particular method in the listener; to insure that the listener really does provide the relevant method that can be called by the event source, the listener must implement a listener interface XListener.

(2) Each registeration and removal method in an event source takes as argument the corresponding XListener interface.

13.23) The events generated by an event source are independent of any enclosing component, i.e. an event source transmits the same events regardless of its location in the component hierarchy. [1]

13.24) Events generated by an event source component are also genrated by subclasses of the source component, unless explicitly inhibited. [1]

13.25) An event listener interface can contain more than one method. This is true for all low-level event classes. [1]

13.26) `MouseEvent` has two listener interfaces: `MouseListener` and `MouseMotionListener`. A source that generates a `MouseEvent` provides two sets of registeration and removal methods corresponding to the two listener interfaces, and can dispatch the `MouseEvent` to the appropiate listeners, based on the interface these listeners implement. [1]

13.27) All listeners of particular event are notified, but the order in which they are notified is **NOT** necessarily the same as the order in which they were added as listeners. [1]

13.28) Notification of all listeners is not guaranteed to occur in the same thread. Access to any data shared between the listeners should be synchronized. [1]

13.29) Each listener interface extends the java.util.EventListener. [1]

13.30) The same listener can be added to several event sources, if required. [1]

13.31) If you register more than one Listener of the same type to a component ⇒ **ALL** registered listeners will be notified when an action happens. [1]

13.32) Event adapters facilitate implementing listener interfaces. When you implement an interface you have to provide the implementation for each method specified in this interface, also for the low-level listener interface the interface has several methods that must be implemented even if you don't want to handle these methods ⇒ the `java.awt.event` package defines an adapter class corresponding to each low-level listener interface; an event adapter implements stubs for all the methods of the corresponding interface, so a listener can subclass the adapter and override only stub-methods for handling events of interest. It makes sense to define such adapters for low-level event listener interfaces, as only these interfaces have more than one method in their specification. [1]
Example:

```
Import java.awt.*;
Import java.awt.event.*;

Public class SimpleWindowTwo extends Frame {
    Button quitButton;        // The source
    QuitHandler quitHandler;  // The listener

    Public SimpleWindowTwo() {
        // create the window
        super("SimpleWindow");

        // create button
        quitButton = new Button("Quit");

        // set a layout manager, and add the button to the window
        setLayout( new Layout(FlowLayout.CENTER) );
        add(quitButton);

        // Create and add the listener to the button
        quitHandler = new QuitHandler(this);
        quitButton.addActionListener(quitHandler);

        // Pack the window and pop it up.
        Pack();
        SetVisible(true);
    }
    /** Create an instance of the application */
    public static void main() { new SimpleWindowTwo(); }
}
```

```
// Definition of the listener
class QuitHandler implements ActionListener, WindowListener {
    private SimpleWindowTwo application;

    public QuitHandler(SimpleWindowTwo window) {
        application = window;
    }

    // Terminate the program
    private void terminate() {
        System.out.println("Quuiting the application");
        Application.dispose();
        System.exit(0);
    }

    // Invoked when the user clicks the quit button
    public void actionPerformed(ActionEvent evt) {
        if (evt.getSource() == application.quitButton) {
            terminate();
        }
    }

    // Invoked when the user clicks the close-box
    public void windowClosing(WindowEvent evt) {
        terminate();
    }

    // Unused methods of the WindowListener interface.
    Public void windowOpened(WindowEvent evt) {}
    Public void windowIconified(WindowEvent evt) {}
    Public void windowDeiconified(WindowEvent evt) {}
    Public void windowDeactivated(WindowEvent evt) {}
    Public void windowClosed(WindowEvent evt) {}
    Public void windowActivated(WindowEvent evt) {}
}
```

Example revisited after using event adapter:

```
Import java.awt.*;
Import java.awt.event.*;

Public class SimpleWindowTwo extends Frame {
    /* as before */
}

// Definition of the listener
class QuitHandler extends WindowAdapter implements ActionListener {

    private SimpleWindowTwo application;

    public QuitHandler(SimpleWindowTwo window) {
        application = window;
    }

    // Terminate the program
    private void terminate() {
        System.out.println("Quuiting the application");
        Application.dispose();
        System.exit(0);
    }

    // Invoked when the usre clicks the quit button
    public void actionPerformed(ActionEvent evt) {
        if (evt.getSource() == application.quitButton) {
            terminate();
```

```
        }
    }

    // Invoked when the user clicks the close-box
    public void windowClosing(WindowEvent evt) {
        terminate();
    }
}
```

13.33) Anonymous classes provide an elegant solution for creating listeners and adding them to event sources. [1]
Example:

```
Import java.awt.*;
Import java.awt.event.*;

Public class SimpleWindowThree extends Frame {
    Button quitButton;        // The source

    Public SimpleWindowThree() {
        // create the window
        super("SimpleWindowThree");

        // create button
        quitButton = new Button("Quit");

        // set a layout manager, and add the button to the window
        setLayout( new Layout(FlowLayout.CENTER) );
        add(quitButton);

        // Create and add the listener to the button
        quitButton.addActionListener(new ActionListener(){
            // Invoked when the usre clicks the quit button
            public void actionPerformed(ActionEvent evt) {
                if (evt.getSource() == application.quitButton) {
                    terminate();
                }
            }
        });

        // Create and add the listener to the button
        addWindowListener( new WindowAdapter() {
            // Invoked when the user clicks the close-box
            public void windowClosing(WindowEvent evt) {
                terminate();
            }
        });

        // Pack th window and pop it up.
        Pack();
        SetVisible(true);
    }

    // Terminate the program
    private void terminate() {
        System.out.println("Quuiting the application");
        Application.dispose();
        System.exit(0);
    }

    /** Create an instance of the application */
    public static void main() { new SimpleWindowTwo(); }
}
```

13.34) The AWT delivers AWTEvents to a component by calling the `processEvent(AWTEvent evt)` method of the component. This method is at the core of low-level event processing. Its default implementation calls an event-specific method in the component. If the component received an `ActionEvent`, the `processEvent()` method calls the `processActionEvent()` method of the component. In other words, when the XEvent is received by a component, it is dispatched by the `processEvent()` method to a corresponding processXEvent() method of the component. [1]

13.35) If a component is customized by subclassing another component, it has the opportunity to implement its own low-level event processing. This can be done in one of two ways:
(1) The subclass component can keep the default behavior of the `processEvent()` method, but provide its own implementations of the processXEvent() methods which override the default versions of these methods.
(2) The subclass component can override the `processEvent()` method and thereby bypass the default behavior of the processXEvent() methods.
In order for either scheme to work, one additional requirment must be met. The subclass component must explicitly enable all events of interest. This is done by calling the `enableEvents()` method of the component. This method is passed a bit mask formed from OR'ing `EVENT_MASK` constants defined in the `java.awt.AWTEvent` class shown in the following table: [1]

Enabling `EVENT_MASK`	Corresponding event processing method
`AWTEvent.COMPONENT_EVENT_MASK`	`ProcessComponentEvent()`
`AWTEvent.CONTAINER_EVENT_MASK`	`ProcessContainerEvent()`
`AWTEvent.FOCUS_EVENT_MASK`	`ProcessFocusEvent()`
`AWTEvent.KEY_EVENT_MASK`	`ProcessKeyEvent()`
`AWTEvent.MOUSE_EVENT_MASK`	`ProcessMouseEvent()`
`AWTEvent.MOUSE_MOTION_EVENT_MASK`	`ProcessMouseMotionEvent()`
`AWTEvent.WINDOW_EVENT_MASK`	`ProcessWindowEvent()`
`AWTEvent.ACTION_EVENT_MASK`	`ProcessActionEvent()`
`AWTEvent.ADJUSTMENT_EVENT_MASK`	`ProcessAdjustmentEvent()`
`AWTEvent.ITEM_EVENT_MASK`	`ProcessItemEvent()`
`AWTEvent.TEXT_EVENT_MASK`	`ProcessTextEvent()`

13.36) Steps for explicit event handling can be summarized as follows: [1]
(1) Define a subclass of the component.
(2) Enable the events by making the subclass constructor call the enableEvents() method with the appropriate bit mask formed from `AWTEvent.X_EVENT_MASK` constants.
(3) Choose one of the two strategies to intercept the events in the subclass:
- For each `AWTEvent.X_EVENT_MASK` constant, the subclass can provide an implementation of the corresponding `processXEvent()` method.
- The subclass can override the `processEvent()` method by providing an implementation to handle the events explicitly.

No matter which strategy is chosen, each event processing method must call the overridden version in the superclass before returning. This ensures that any registered listeners will also be notified.
Example: low level processing I:

```
Import java.awt.*;
Import java.awt.event.*;

Public class SimpleWindowFour extends Frame {
    QuitButton quitButton;
    Public SimpleWindowFour() {
        super("SimpleWindowFour");
        quitButton = new QuitButton("Quit", this);
        add(quitButton);
        enableEvents(AWTEvent.KEY_EVENT_MASK |
AWTEvent.WINDOW_EVENT_MASK);
```

```
        pack();
        setVisible(true);
    }

    // Invoked when the user clicks the close-box
    public void processWindowEvent( WindowEvent evt ) {
        if (evt.getID == WindowEvent.WINDOW_CLOSING)
            terminate();
        super.processWindowEvent(evt);
    }

    // Invoked when the user types 'q' or 'Q'
    public void processKeyEvent( KeyEvent evt ) {
        if (evt.getID() == KeyEvent.KEY_TYPED && (evt.getKeyChar() == 'q'
|| evt.getKeyChar() == 'Q'))
            terminate();
        super.processKeyEvent(evt);
    }

    public void terminate() {
        System.out.println("Quiting the application");
        dispose();
        System.exit(0);
    }

    public static void main(String args[]) {
        SimpleWindowFour window = new SimpleWindowFour();
    }
}
class QuitButton extends Button {
    private SimpleWindowFour application;

    public QuitButton(String name, SimpleWindowFour window) {
        super(name);
        application = window;
        enableEvents(AWTEvents.ACTION_EVENT_MASK |
AWTEvent.KEY_EVENT_MASK);
    }

    public void processActionEvent( ActionEvent evt ) {
        if ( evt.getSource() == this )
            application.terminate();
        super.processActionEvent(evt);
    }

    public void processKeyEvent( KeyEvent evt ) {
        if (evt.getID() == KeyEvent.KEY_TYPED && (evt.getKeyChar() == 'q'
|| evt.getKeyChar() == 'Q') )
            application.terminate();
        super.processKeyEvent(evt);
    }
}
```

Example: low level processing II:

The main difference between I & II is that II does all the event processing in the processEvent() method.

```
Import java.awt.*;
Import java.awt.event.*;

Public class SimpleWindowFive extends Frame {
    QuitButton quitButton;
    Public SimpleWindowFive() {
        super("SimpleWindowFive");
        quitButton = new QuitButton("Quit", this);
```

```
        add(quitButton);
        enableEvents(AWTEvent.KEY_EVENT_MASK |
AWTEvent.WINDOW_EVENT_MASK);
        pack();
        setVisible(true);
    }

    // Event processing
    public void processEvent() {
        // Invoked when the user clicks the close-box
        if (evt.getID() == WindowEvent.WINDOW_CLOSING)
            terminate();

        // Invoked when the user types 'q' or 'Q'
        if ( evt.getID() == KeyEvent.KEY_TYPED &&
             (((KeyEvent) evt).getKeyChar() == 'q' ||
             (((KeyEvent) evt).getKeyChar() == 'Q'))
            terminate();
        super.processEvent(evt);
    }

    public void terminate() {
        System.out.println("Quiting the application");
        dispose();
        System.exit(0);
    }

    public static void main(String args[]) {
        SimpleWindowFour window = new SimpleWindowFive();
    }
}
class QuitButton extends Button {
    private SimpleWindowFive application;

    public QuitButton(String name, SimpleWindowFive window) {
        super(name);
        application = window;
        enableEvents(AWTEvents.ACTION_EVENT_MASK |
AWTEvent.KEY_EVENT_MASK);
    }

    public void processEvent(AWTEvent evt) {
        // Invoked when the user clicks the quit button
        if ((evt instanceof ActionEvent) && ((ActionEvent)evt).getSource()
== this)
            application.terminate();

        // Invoked when the user types 'q' or 'Q'
        if (evt.getID() == KeyEvent.KEY_TYPED &&
            (((KeyEvent) evt).getKeyChar() == 'q' ||
            ((KeyEvent) evt).getKeyChar() == 'Q'))
            application.terminate();

        super.processEvent(evt);
    }
}
```

Chapter 14:

Painting

14.1) The abstract class `java.awt.Graphics` provides a device-independent interface for rendering graphics. An instance of the `Graphics` class or its subclass **CANNOT** be created directly using a constructor because it is abstract. [1]

14.2) The graphics context encapsulates the following state information: [1]
(a) The *target* of the graphics context.
(b) The *color* in which drawing is done.
(c) The *font* in which text is rendered.
(d) The *clip region* that defines the area in which drawing is done.
(e) The *translation origin* relative to which all drawing coordinates are interpreted.
(f) The *paint mode* for rendering graphics.
(g) The color of the XOR *paint mode* toggle.

14.3) A component may need to be redrawn for a variety of reasons: its size might have changed, it might have been covered but now has become uncovered, user ineraction may initiate a redraw of the component. [1]

14.4) The following methods defined in the `Component` class are involved in drawing components:

```
void repaint();
void update(Graphics g);
void paint(Graphics g);
```

The code for drawing the component is usually implemented in the overriding method `paint()` defined by a concrete component. This method is passed an object of the `Graphics` class that provides the graphics context for drawing the component. The AWT will automatically call `paint()` when the size of the component has changed or the component has been uncovered. The `paint()` method is seldom called directly by the component, instead the following procedure is relied upon to change its appearance:
- The `repaint()` method is usually called by the application for screen updating.
- The call to the `repaint()` method eventually leads to invocation of the `update()` method. By default, this method does the following:
    (a) It clears the component's screen-area by filling it with the current component background color.
    (b) It sets the current drawing color to the foreground color of the component.
    (c) It invokes the `paint()` method, passing it the same `Graphics` object that it received.

14.5) In the AWT, most GUI control components(button, text fields, checkboxes) are drawn using the underlying windowing system, and therefore their graphics contexts should not be used by an application to draw on them. In other words, an instead on the default implementation of this method to display them on the screen. Components that lend themselves to graphics rendering by the application are those that do not have any default external graphical representation:
    (a) `Canvas` class.
    (b) Subclasses of the `Component` class that are not part of the AWT.
    (c) Container class and its subclasses: `Window`, `Frame`, `Dialog`, `Panel`, `Applet`.
A subclass of these components can override the `paint()` method, and use the graphics context passed to this method to render graphics onto the component.

14.6) A user thread (i.e. the application) usually relies on the indirect calls to the `update()` method through the `repaint()` method to update components. However, the AWT thread (i.e. the AWT event handler) calls the `paint()` method directly on a component if the component needs refreshing (for example, when the component is resized).

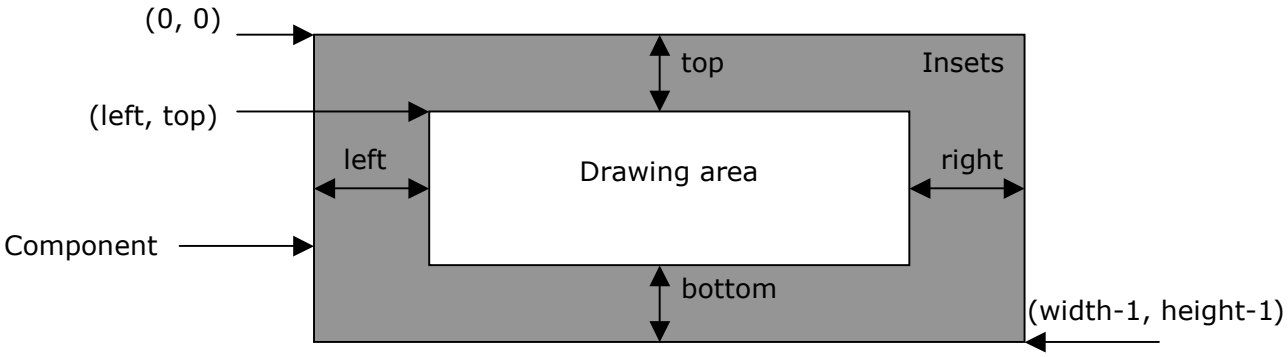
14.7) **NOTE**: The graphics context passed to the `paint()` method by the AWT **NEED NOT** be the same every time this method is called.

14.8) A graphics context cannot be created directly by calling a constructor, because te `Graphics` class is `abstract`. Inside the `paint()` method this is not a problem, as the method is passed such a context via a `Graphics` reference. In other situations, a graphics context can be obtained in one of the following two ways: [1]

- An existing `Graphics` object can be used to create a new one, by invoking the `create()` method in the `Graphics` class.
- Since every component has an associated graphics context, this can be explicitly obtained by calling the `getGraphics()` method of the `Component` class.

NOTE:The creator of a graphics context should ensure that the `dispose()` method is called to free the resources it uses when the graphics context is no longer needed.

14.9) Coordinates of a component: [1]



14.10) The coordinates are measured in pixels and supplied as integer values to the many drawing methods of the `Graphics` class. [1]

14.11) The drawing area in a component is not necessarily the same size as the component. The size of the component is returned by the following method of the `Component` class: [1]

```
Dimension getSize() // (width, height)
```

However, the size returned includes the borders (and any title-bar in the case of a frame). The insets(i.e. size of the borders) of the component are given by the following method of the `Container` class:

```
Insets getInsets() // (top, bottom, left, right)
```

14.12) The size of the drawing region in a component can be calculated as follows: [1]

```
Dimension size = getSize();
Insets insets = getInsets();
int drawHeight = size.height – insets.top – insets.bottom;
int drawWidth = size.width – insets.left – insets.right;
```

14.13) The origin of the drawing region in a component is given by `(getInsets.left, getInset.top)`. The method `translate(int x, int y)` of the `Graphics` class can be used to set the translation origin of the graphics context. All coordinate arguments to the methods of the `Graphics` object are then considered relative to this origin in subsequent operations. [1]

14.14) The following methods of the `Graphics` class can be used to get the current color or to set a color in the graphics context. Any change of color applies to all subsequent operations. [1]

```
Color getColor();
void setColor(Color c);
```

14.15) There are 13 predefined colors designated by constants in the `Color` class: [1]

```
Color.black
Color.blue
Color.cyan
```

```
Color.darkGray
Color.gray
Color.green
Color.lightGray
Color.magenta
Color.orange
Color.pink
Color.red
Color.white
Color.yellow
```

14.16) The Color class constructors: [1]

`Color(int r, int g, int b)`

Creates a color using the separate red, green, and blue(RGB) values for the color in the range (0-255)

`Color(int rgb)`

Creates a color with the specified combined RGB value consisting of the red component in bits 16-23, the green component in bits 8-15, and the blue component in bits 0-7

`Color(float r, float g, float b)`

Creates a color with the specified red, green, and blue(RGB) values in the range (0.0 – 1.0)

14.17) The class `SystemColor` provides the desktop color scheme for the current platform. Constants are provided for properties such as the background color of the desktop (`SystemColor.desktop`), background color for the controls(`SystemColor.control`) and text color for menus (`SystemColor.menuText`). These color properties can be used to provide a look which is consistant with that of the host platform. [1]

14.18) Text rendering in a component is done using the following methods of the Graphics class: [1]

`void drawString(String str, int x, int y);`

String is drawn with the baseline of the first character at the specified coordinates, using the current font and color.

`void drawChars(Char[] data, int offset, int length, int x, int y);`

Starting at the offset argument, length characters from the character array are drawn with baseline of the first character at the specified coordinates, using the current font and color.

`void drawBytes(byte[] data, int offset, int length, int x, int y);`

Starting at the offset argument, length bytes from the byte array are drawn with baseline of the first character at the specified coordinates, using the current font and color.

14.19) The following methods of the Graphics class can be used to get the font property or set the font in the graphics context. Methods with the same signatures are also defined in the `Component` class to obtain the current font or set a font for a component. [1]

```
Font getFont()
void setFont(Font f)
```

14.20) The `Font` class defines a constructor that can be used to obtain available fonts. [1]

`Font(String name, int style, int size)`

The following font names (called logical font names) are standard on all platforms and are mapped to actual fonts on a particular platform:
"`Serif`" which is a variable pitch font with serifs.
"`SansSerif`" which is a variable pitch font without serifs.
"`Monospaced`" which is a fixed pitch font.
"`Dialog`" which is a font for dialogs.
"`DialogInput`" which is a font for dialog input.
"`Symbol`" which is mapped to a symbol font.

Font style can be specified using constants from the `Font` class:

```
Font.BOLD
Font.ITALIC
Font.PLAIN
(Font.BOLD | Font.ITALIC) // Both bold and italic
```

14.21) Font availability is platform-depenedent. The glyphs which make up a font are also platform-dependent. [1]

```
boolean canDisplay(char c)
```

In the `Font` class can be used to find out if a font has a glyph for a specific character.

The `GraphicsEnvironment` class provides access to platform-specific information about fonts. The local `GraphicsEnvironment` can be used tgo get a list of names for the available fonts:

```
GraphicsEnvironment ge=GraphicsEnvironment.getLocalGraphicsEnvironment();
String[] fontNames = ge.getAvailableFontFamilyNames();
```

14.22) Properties of a font are accessed by using a font metrices (represented by objects of the `FontMetrices class`) associated with the `Font`. All font measurments are in pixels. [1]

A font metrices can be obtained in one of the following ways:
(a) A component can be used to get a font metrices for a font:

```
    Font font12 = new Font("Dialog", Font.ITALIC, 12);
    FontMetrices metrices2 = component.getFontMetrices(font12);
```

(b) A graphics context can be used to get a font metrices for a font:

```
    FontMetrices metrices3 = graphicsContext.getFontMetrices();
    FontMetrices metrices4 = graphicsContext.getFontMetrices(font18);
```

```
int getAscent()
int getDescent()
int getMaxAscent()
int getMaxDescent()
```

```
int getLeading()
```

Returns the standard leading value, a.k.a. *interline spacing*, which is the amount of space between the descent of one line of text and the ascent of the next line.

```
int getHeight()
```

Returns the standard height of a line of text in the font associated with the metrices. Font height is the distance between the baseline of adjacent lines of text. It is sum of the leading + ascent + descent of the font.

```
int getMaxAdvance()
```

Returns the maximum advance of any character in this font. This is defined as the maximum distance between one character to the next, in a line of text.

```
int charWidth(int ch)
int charWidth(char ch)
```

Returns the advance width of the specified character.

`int stringWidth(String str)`
Returns the advance width of the characters in the specified string.

14.23) `Graphics` class provides the following method for drawing lines: [1]

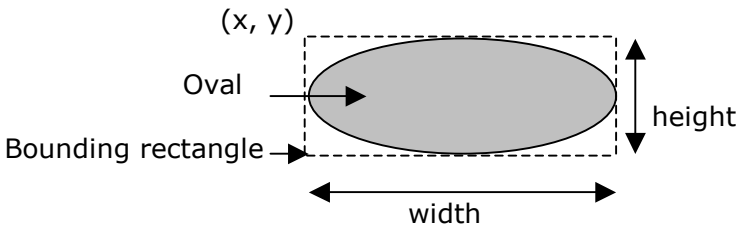
```
void drawLine( int x1, int y1,       // from point
               int x2, int y2)       // to point
```

14.24) `Graphics` class provides the following method for drawing outlines of rectangles, and to fill a rectangle with the current color. For methods that draw the outline of a rectangle, the resulting rectangle will cover an area of `(width+1) x (length+1)` pixels. For methods that fill a rectangle will cover an area of `width x height` pixels. [1]

```
void drawRect(int x, int y,          // top left corner
              int width, int length) // of rectangle
void fillRect(int x, int y,          // top left corner
              int width, int length) // of rectangle
void drawRoundRect(int x, int y,                 // top left corner
                   int width, int length,        // of rectangle
                   int arcwidth, int arclength) // horizontal & vertical
                                                 // diameters
void fillRoundRect(int x, int y,                 // top left corner
                   int width, int length,        // of rectangle
                   int arcwidth, int arclength)
void draw3DRect(int x, int y,                    // top left corner
                int width, int length,           // of rectangle
                int arcwidth, int arclength,     // horizontal & vertical
                                                 // diameters
                boolean raised)                  // raised or sunk
void fill3DRect(int x, int y,                    // top left corner
                int width, int length,           // of rectangle
                int arcwidth, int arclength,     // horizontal & vertical
                                                 // diameters
                boolean raised)                  // raised or sunk
void clearRect(int x, int y,                     // top left corner
               int width, int length)            // of rectangle
```

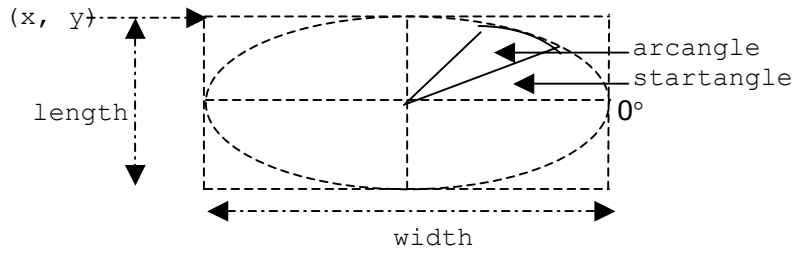
14.25) `Graphics` class provides the following method for drawing ovals, and to fill an oval with the current color. [1]

```
void drawOval(int x, int y,                  // top left corner
              int width, int length)         // of bounding rectangle
void fillOval(int x, int y,                  // top left corner
              int width, int length)         // of bounding rectangle
```



14.26) `Graphics` class provides the following method for drawing arcs, and to fill an oval with the current color. The starting point of the arc is given by a starting angle, and the ending point is given by the angle swept by the arc. Angles are measured are measured in degrees. All positive angles are measured in a counterclockwise direction with the 0 degrees given by the three-o'clock position. Negative angles are measured in a

clockwise direction from the 0 degrees position. The arc is bounded by a rectangle. The center of the arc coincides with the center of the bounding rectangle. [1]




```
void drawArc(int x, int y,                  // top left corner
             int width, int length,         // of bounding rectangle
             int startAngle,
             int arcAngle);
void fillArc(int x, int y,                  // top left corner
             int width, int length,         // of bounding rectangle
             int startAngle,
             int arcAngle);
```

14.27) A polygon is a closed sequence of line segments. Given a sequence of points (called vertices), line segments connect one vertex to the next in the sequence, finishing with the last vertex being connected with the first one. The Polygon class has the following constructors: [1]

```
Polygon(int[] xpoints, int[] ypoints, int npoints);
```

14.28) `Graphics` class provides the following method for drawing polygons, and to fill a polygons with the current color. [1]

```
void drawPolygon(int[] xPoints, int[] yPoints, int nPoints);
void drawPolygon(Polygon p);
void fillPolygon(int[] xPoints, int[] yPoints, int nPoints);
void fillPolygon(Polygon p);
```

14.29) The clip region of a graphics context defines the area in which all drawing will be done. In other words, the clip region defines the actual drawing area used for rendering operations. This region can be all or part of the associated component. Rendering operations have no effect outside the clip region. Only pixels that lie within the clip region can be modified. The `Graphics` class defines the following methods for the clip region: [1]

```
Rectangle getClipBounds()
Shape getClip()
void setClip(int x, int y, int width, int height)
void setClip(Shape clip)
```

14.30) The default paint mode is to overwrite pixels in the drawing region. The AWT offers another rendering mode called the XOR paint mode. The following methods of the `Graphics` class can be used to switch between overwrite and XOR paint modes: [1]

`void setPaintMode()`
Sets the mode to overwrite paint mode. All subsequent rendering operations will overwrite the destination with the current color.

`void setXORMode(Color c1)`
Sets the mode to XOR paint mode, which alternates pixels between the current color and a new specified XOR alternation color.

14.31) The current toolkit can be used to read the graphics file (GIF or JPEG formates) into an `Image` object: [1]

```
Toolkit currentTK = Toolkit.getDefaultToolkit();
```

```
Image image1 = currentTK.getImage("Cover.gif");
```

For a graphics file on the net, a URL must be supplied:

```
URL url = new URL("http://www.example.com/Cover.gif");
Image image2 = currentTK.getImage( url );
```

14.32) The `Applet` class also provides `getImage()` methods for reading graphics files:

```
Image getImage(URL url);
Image getImage(URL url, String name);
```

The *url* argument must specify an absolute URL. The *name* argument is the name of the file, interpreted relative to the URL. These methods returns immediately.

The `Graphics` class defines a variety of `drawImage()` methods which can scale and fit the image. The simplest form of the `drawImage()` method is shown here:

```
boolean drawImage(Image img,
                  int x, int y,
                  ImageObserver observer)
```

Chapter 15:

Files & Streams

15.1) It is worth a while to review how Java represents text before looking to file I/O: [1][3]
Java uses two kinds of text representation:
- Unicode for internal representation of characters and strings.
- UTF for input and output.

Unicode uses 16 bits to represent each character. If the high-order 9 bits all zeros, then the encoding is simply standard ASCII, with the low-order byte containing the character representation. Otherwise, the bits represent a character that is not represeneted in 7-bit ASCII. Java's char type uses Unicode encoding, and the String class contains a collection of Java chars. It is sufficient to encode most alphapets, but pictographic Asian languages present a problem. Standards committees have developed compromises to allow limited but useful subsets of Chinese, Japanese, and Korean to be represented in Unicode, but it has become clear that an ideal global text representation scheme must use more than 16 bits per character.
The answer is **UTF**. The abbreviation stands for "UCS Transformation Format", and the UCS stands for "Universal Character Set". It uses as many bits for the larger Asian alphapets. Since every character can be represented, UTF is a truly global encoding scheme.
A character encoding is a mapping between a character set and a range of binary numbers. Every Java platform has a default character encoding, which is used to interpret between internal Unicode and external bytes. When an I/O operation is performed in Java, the system needs to know which character encoding to use.
NOT all Unicode characters can be represented in other encoding schemes, in that case the '?' character is usually used to denote any such character in the resulting ouput, during translation from Unicode.
The raw 16-bit Unicode is not particularly space effiecient for storing characters derived from the Latin alphabet, because the majority of the characters can be represented by one byte (same as ASCII), making the higher byte in the 16-bit Unicode superfluous. For this reason, Unicode characters are usually encoded externally, using the UTF8 encoding which has a multi-byte encoding format. It represents ASCII characters as one-byte characters but uses multiple bytes for others. The readers and writers can correctly and efficiently translate between UTF8 and Unicode.

15.2) `File` class is not meant for handling the contents of files, it represent the name of a file or directory that might exist on the host machine's file system. [1] [3]

15.3) The pathname for a file or a directory is specified using the naming conventions of the host system. However, the `File` class defines platform-dependent constants that can be used to handle file and directory names in a platform-independent way: [1]

`public static final char separatorChar`
The system-dependent default name-separator character. This field is initialized to contain the first character of the value of the system property `file.separator`. On UNIX systems the value of this field is `'/'`; on Win32 systems it is `'\'`.

`public static final String separator`
The system-dependent default name-separator character, represented as a `String` for convenience. This string contains a single character, namely `separatorChar`.

`public static final char pathSeparatorChar`
The system-dependent path-separator character. This field is initialized to contain the first character of the value of the system property `path.separator`. This character is used to separate filenames in a sequence of files given as a *path list*. On UNIX systems, this character is `':'`; on Win32 systems it is `';'`.

`public static final String pathSeparator`
The system-dependent path-separator character, represented as a `String` for convenience. This string contains a single character, namely `pathSeparatorChar`.

15.4) The `File` class: [2] [3]

Category	Methods	Example & declaration
Constructors	`File(String pathname)`	The pathname (of a file or a directory) can be an absolute pathname or pathname relative to the current directory. An empty string as argument results in an abstract pathname for the current directory.
	`File(String directoryPathname, String filename)`	Creates a `File` object whose pathname is as follows: directoryPathname + separator + filename
	`File(File directory, String filename)`	If the directory argument is `null`, the resulting `File` object represents a file in the current directory. If not `null`, it creates a `File` object whose pathname is as follows: pathname of the directory `File` Object + separator + filename
Navigation	`public boolean exists()`	Tests whether the file or directory denoted by this abstract pathname exists.
	`public String getAbsolutePath()`	Returns the absolute pathname string of this abstract pathname.
	`public String getPath()`	Returns the absolute or relative pathname of the file represented by the `File` object.
	`public String getCanonicalPath()`	Returns the name of the canonical path of the file or directory. This is similar to `getAbsolutePath()`, but the symbols `.` and `..` are resolved.
	`public String getParent()`	Returns the name of the directory that contains the `File` Object.
	`public boolean isAbsolute()`	Tests whether this abstract pathname is absolute. The definition of absolute pathname is system dependent. On UNIX systems, a pathname is absolute if its prefix is `"/"`. On Win32 systems, a pathname is absolute if its prefix is a drive specifier followed by `"\\"`, or if its prefix is `"\\"`.
Properties	`public boolean canRead()`	Tests whether the application can read the file denoted by this abstract pathname.
	`public boolean canWrite()`	Tests whether the application can modify to the file denoted by this abstract pathname.
	`public long lastModified()`	Returns the time that the file denoted by this abstract pathname was last modified.
	`public long length()`	Returns the length of the file denoted by this abstract pathname.
	`public boolean equals(Object obj)`	Returns true if the comparing the pathnames of the `File` objects is identical.

	`public boolean isDirectory()`	Tests whether the file denoted by this abstract pathname is a directory.
	`public boolean isFile()`	Tests whether the file denoted by this abstract pathname is a normal file.
Listing	`String[] list()`	Returns an array of strings naming the files and directories in the directory denoted by this abstract pathname.
	`String[] list(FilenameFilter filter)`	Returns an array of strings naming the files and directories in the directory denoted by this abstract pathname that satisfy the specified filter.
	`File[] listFiles()`	Returns an array of abstract pathnames denoting the files in the directory denoted by this abstract pathname.
	`File[] listFiles(FilenameFilter filter)`	Returns an array of abstract pathnames denoting the files and directories in the directory denoted by this abstract pathname that satisfy the specified filter.
	`File[] listFiles(FileFilter filter)`	Returns an array of abstract pathnames denoting the files and directories in the directory denoted by this abstract pathname that satisfy the specified filter.

	A filter is an object of a class that implements either of these two interfaces:
	`interface FilenameFilter {`
	`  boolean accept(File currentDirectory, String entryName);`
	`}`
	`interface FileFilter {`
	`  boolean accept(File pathname);`
	`}`	

Manipulating files & directories	`public boolean createNewFile()`	Atomically creates a new, empty file named by this abstract pathname if and only if a file with this name does not yet exist. The check for the existence of the file and the creation of the file if it does not exist are a single operation that is atomic with respect to all other filesystem activities that might affect the file.
	`public boolean renameTo(File dest)`	Renames the file denoted by this abstract pathname.
	`public boolean delete()`	Deletes the file or directory denoted by this abstract pathname. If this pathname denotes a directory, then the directory must be empty in order to be deleted.
	`public boolean mkdir()`	Creates the directory named by this abstract pathname.

	`public boolean mkdirs()`	Creates the directory named by this abstract pathname, including any necessary but nonexistent parent directories. Note that if this operation fails it may have succeeded in creating some of the necessary parent directories.

15.5) Difference between: [1]

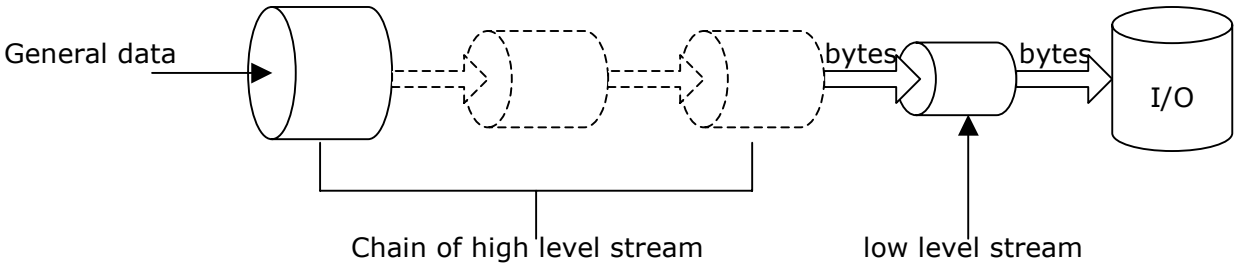
```
String getPath()
String getAbsolutePath()
String getCanonicalPath()
```

If the `File` object represented the relative pathname'`..\book\chapter1`' and the current directory had the absolute pathname '`c:\documents`' ⇒ they will return respictively

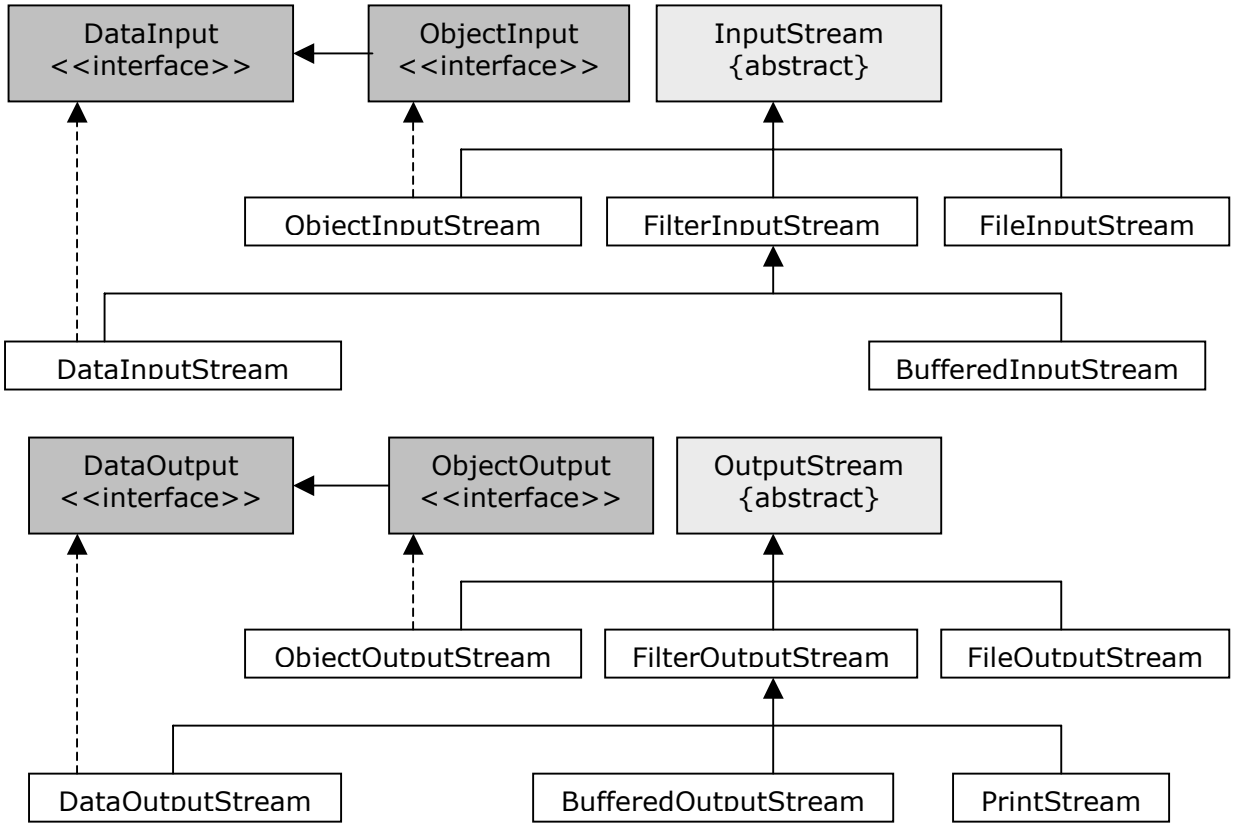
- '`\..\book\chapter1`'
- '`c:\documents\..\book\chapter1`'
- '`c:\book\chapter1`'

15.6) Java's general I/O classes provide this approach for I/O approach: [3] [**I made the graph, any mistake please report**]

- A low level output stream recieves bytes and writes bytes to an output device.
- A high level filter output stream receives general-format data, such as primitive, and write bytes to a low-level stream or to another filter output stream.



15.7) Byte Stream Inheritance hierarchies: [1]



15.8) The abstract classes `InputStream` and `OutputStream` are the root of the inheritance hierarchies for handling the reading and writing of bytes. [1]

15.9) Methods for `InputStream` and `OutputStream` classes:

Methods in `InputStream`	Methods in `OutputStream`
`int read() throws IOException`	`void write(int b) throws IOException`
`int read(byte[] b) throws IOException`	`void write(byte[] b) throws IOException`
`int read(byte[] b, int off, int len) throws IOException`	`void write(byte[] b, int off, int len) throws IOException`

Note: that `int read()` method read a `byte`, but returns an `int` value, the `byte` resides in the eight least significant bits of the unit, while the remaining bits in the int are zerod out. It returns the value `-1` when the end of stream is reached. The `void write(int b)` method takes an `int` as argument, but truncates it down to the eight least significant bits before writing it out as a `byte`. [1]

15.10) Closing a stream automatically flushes the stream, meaning that any data in its internal buffer is written out, and it can be manually flushed using `void flush()` method. [1]

15.11) Read & Write operations on streams are synchronous (blocking) operation, i.e. a call to `read` or `write` method does not return before a `byte` has been read or written. [1]

15.12) Input Streams: [1]

`ByteArrayInputStream`	Data is read from a byte array that must be specified.
`FileInputStream`	Data is read as bytes from a file. The file acting as the input stream can be specified by a `File` object, a `FileDescriptor` or a `String` file name.
`FilterInputStream`	Superclass of all input stream filters. An input filter **MUST** be chained to an underlying input stream.
`BufferedInputStream`	A filter that buffers the bytes read from an underlying input stream. The underlying input stream must be specified, and an optional buffer size can be included.
`DataInputStream`	A filter that allows the binary representation of Java primitive values to be read from an underlying input stream. The underlying input stream must be specified.
`PushbackInputStream`	A filter that allows bytes to be "unread" from an underlying input stream. The number of bytes to be unread can optionally be specified.
`ObjectInputStream`	Allows binary representation of Java objects and Java primitive values to be read from a specified input stream.
`PipedInputStream`	Reads bytes from a `PipedOutputStream` to which it must be connected. The `PipedOutputStream` can optionally be specified when creating the `PipedInputStream`.
`SequenceInputStream`	Allows bytes to be read sequentially from two or more input streams consecutively. This should be regarded as concatenating the contents of several input streams into a single continuous input stream.

15.13) Output Streams: [1]

`ByteArrayOutputStream`	Data is written from a byte array. The size of the byte array created can be specified.
`FileOutputStream`	Data is written as bytes from a file. The file acting as the output stream can be specified by a `File` object, a `FileDescriptor` or a `String` file name.
`FilterOutputStream`	Superclass of all output stream filters. An output filter **MUST** be chained to an underlying output stream.
`BufferedOutputStream`	A filter that buffers the bytes written to an underlying output stream. The underlying output stream must be specified, and an optional buffer size can be included.
`DataOutputStream`	A filter that allows the binary representation of Java primitive values to be written to an underlying output stream. The underlying output stream must be specified.
`ObjectOutputStream`	Allows binary representation of Java objects and Java primitive values to be written to a specified output stream.
`PipedOutputStream`	Writes bytes from a `PipedInputStream` to which it must be connected. The `PipedInputStream` can optionally be specified when creating the `PipedOutputStream`.

15.14) `FileInputStream`, `FileOutputStream` classes define byte I/O streams that are connected to files, data can only be read or written as a sequence of bytes. [1]

```
FileInputStream(String name) throws FileNotFoundException
FileInputStream(File file) throws FileNotFoundException
FileInputStream(FileDescriptor fdObj) throws FileNotFoundException
```

If the file does not exist, a `FileNotFoundException` is thrown. If it exists, it is set to be read from the beginning. A `SecurityException` is thrown if the file does not have read access.

```
FileOutputStream(String name) throws FileNotFoundException
FileOutputStream(String name, boolean append) throws FileNotFoundException
FileOutputStream(File file) throws FileNotFoundException
FileOutputStream(FileDescriptor fdObj) throws FileNotFoundException
```

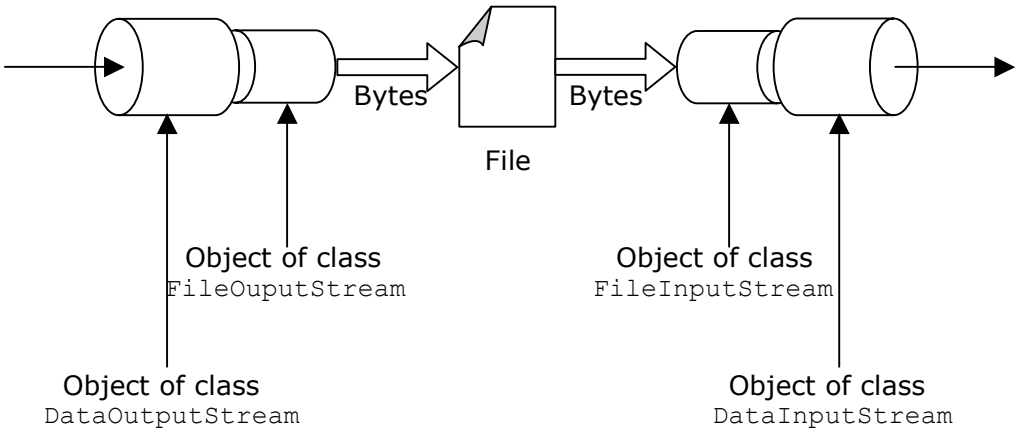
If the file does not exist, it is created. If it exists, its contens are reset, unless the appropriate constructor is used to indicate that output should be appended to the file. A `SecurityException` is thrown if the file does not have write access or it cannot be created.

15.15) DataInput and DataOutput interfaces: [1]

Type	Methods in DataInput	Methods in DataOutput
Boolean	`readBoolean()`	`writeBoolean( boolean v)`
Char	`readChar()`	`writeChar(int v)`
Byte	`readByte()`	`writeByte(int v)`
Short	`readShort()`	`writeShort(int v)`
Int	`readInt()`	`writeInt(int v)`
Long	`readLong()`	`writeLong(long v)`
Float	`readFloat()`	`writeFloat(float v)`
Double	`readDouble()`	`writeDouble(double v)`
String	`readLine()`	`writeLine(String s)`
String	`readUTF()`	`writeUTF(String s)`

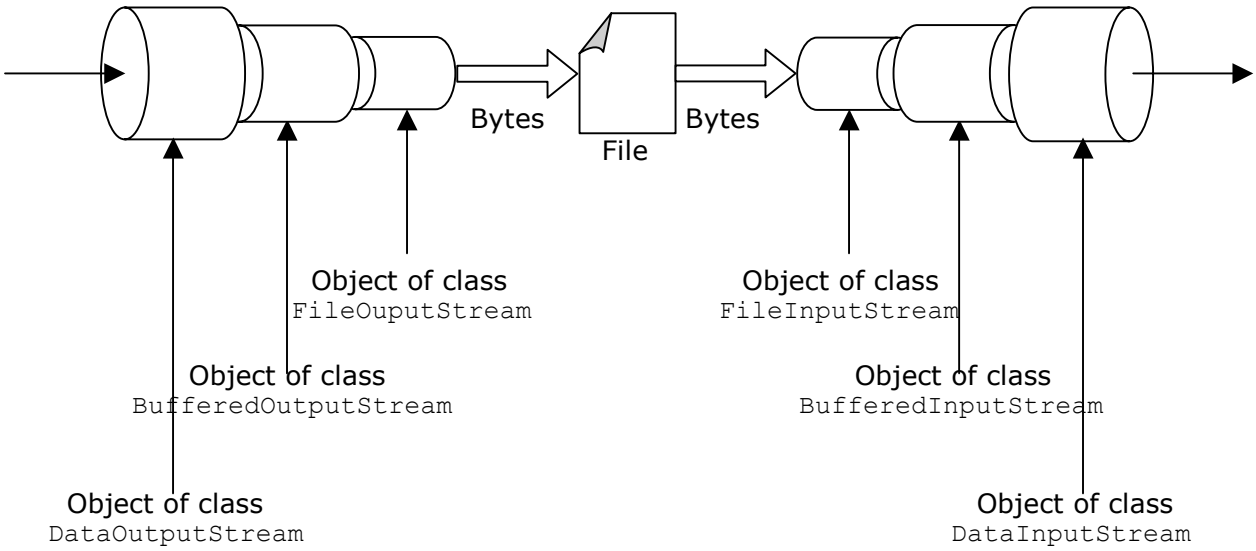
15.16) It is all very well to read bytes from input devices and write bytes to output devices, if bytes are the unit of information you are interested in. However, more often than not the bytes to be read or written constitute higher-level information such as ints or strings. The most common of high-level streams extending the super-classes are `FilterInputStream` and `FilterOutputStream`. [3]

15.17) Stream Chaining: [1]



15.18) Buffering Byte Streams: [1]

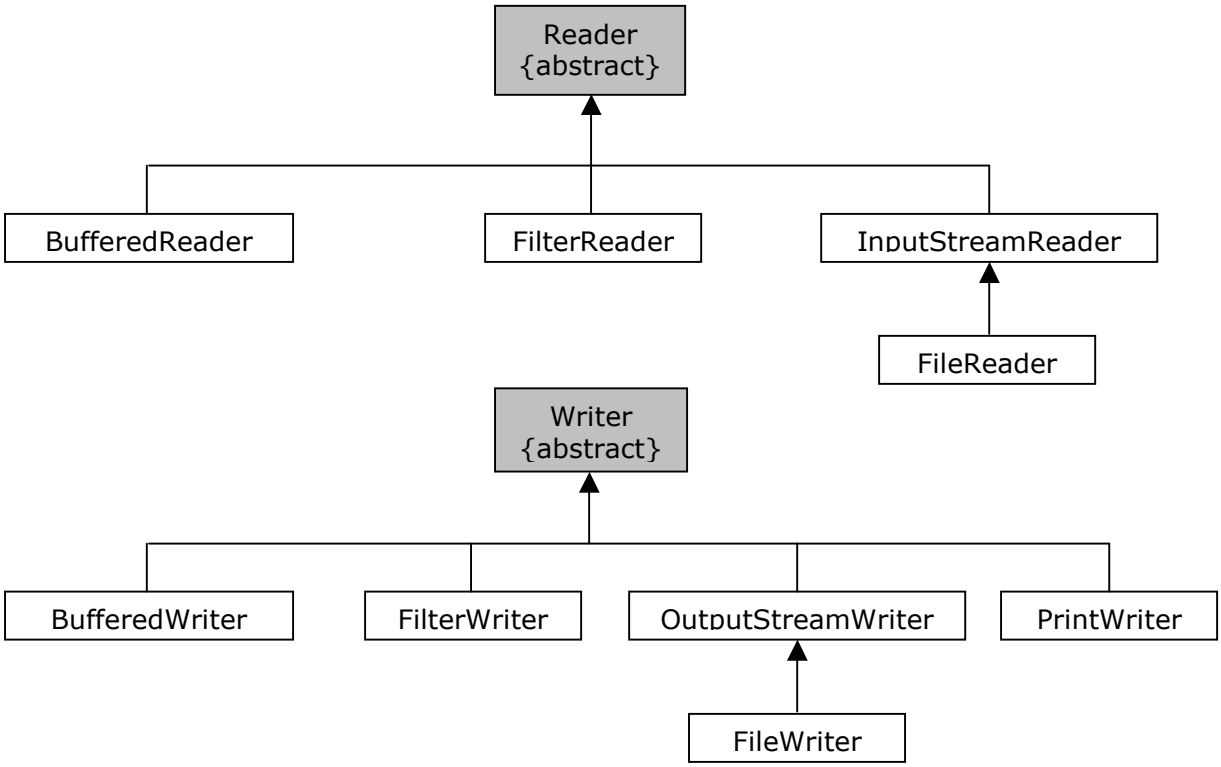
The filter classes `BufferedInputStream` and `BufferedOutputStream` implement buffering of bytes for input and output streams, respectively. Data is read and written in blocks of bytes, rather than a single byte at a time. Buffering can enhance performance significantly. These filter classes only provide methods for reading and writing bytes. A buffering filter must be chained to an underlying stream.



15.19) The table below shows the correspondace between byte ouput and input streams, note that not all classes have a correspondance counterpart.[1]

OutputStreams	InputStreams
`ByteArrayOutputStream`	`ByteArrayInputStream`
`FileOutputStream`	`FileInputStream`
`FilterOutputStream`	`FilterInputStream`
`BufferedOutputStream`	`BufferedInputStream`
`DataOutputStream`	`DataInputStream`
No counterpart	`PushbackInputStream`
`ObjectOutputStream`	`ObjectInputStream`
`PipedOutputStream`	`PipedInputStream`
No counterpart	`SequenceInputStream`

15.20) Character Stream inheritance hierarchies: [1]



15.21) Readers: [1]

`BufferedReader`
 	
`LineNumberReader`
	A reader that buffers the characters read from an underlying reader. The underlying reader must be specified, and an optional buffer size can be given.
	A buffered reader that reads characters from an underlying reader while keeping track of the number of lines read. The underlying reader must be specified, and an optional buffer size can be given.

`CharArrayReader`	Characters are read from a character array that must be specified.

`FilterReader`
 	
`PushbackReader`
	Abstract superclass of all character input stream filters. A `FilterReader` must be chained to an underlying reader, which must be specified.
	A filter that allows characters to be "unread" from a character input stream. A `PushbackReader` must be chained to an underlying reader, which must be specified. The number of characters to be unread can optionally be specified.

`InputStreamReader`
 	
`FileReader`
	Characters are read from a byte input stream, which must be specified. The default character encoding is used if no character encoding is explicitly specified.
	Reads characters from a file using the default character encoding. The file can be specified by a `File` object, a `FileDescriptor`, or a `String` file name. It automatically creates a `FileInputStream` for the file.

`PipedReader`	Reads characters from a `PipedWriter` to which it must be connected. The `PipedWriter` can optionally be specified when creating the `PipedReader`.
`StringReader`	Characters are read from a String, which must be specified.

15.22) Writers: [1]

`BufferedWriter`	A writer that buffers the characters before writing them to an underlying writing. The underlying writer must be specified, and an optional buffer size can be given.
`CharArrayWriter`	Characters are written to a character array that grows dynamically. The size of the character array initially created can be specified.
`FilterWriter`	Abstract superclass of all character input stream filters. A `FilterWriter` must be chained to an underlying writer, which must be specified.
`OutputStreamWriter`	Characters are written to a byte output stream, which must be specified. The default character encoding is used if no explicit character encoding is specified.
`FileWriter`	Writes characters to a file, using the default character encoding. The file can be specified by a `File` object, a `FileDescriptor`, or a `String` file name. It automatically creates a `FileOutputStream` for the file.
`PipedWriter`	Writes characters to a `PipedReader`, to which it must be connected. The `PipedReader` can optionally be specified when creating the `PipedWriter`.
`PrintWriter`	A filter that allows textual representations of Java objects and Java primitive values to be written to an underlying output stream or writer. The underlying output stream or writer must be specified.
`StringWriter`	Characters are written to a `StringBuffer`. The initial size of the `StringBuffer` created can be specified.

15.23) Methods for `Reader` and `Writer` classes:

Methods in `Reader`	Methods in `Writer`
`int read() throws IOException`	`void write(int c) throws IOException`
`int read(char cbuf[]) throws IOException`	`void write(char[] cbuf) throws IOException`
`int read(char cbuf[], int off, int len) throws IOException`	`void write(String str) throws IOException`
	`void write(char[] cbuf, int off, int len) throws IOException`
	`void write(String str, int off, int len) throws IOException`
`long skip(long n) throws IOException`	`void close() throws IOException`
	`void flush() throws IOException`

15.24) `PrintWriter` class:

Constructors:

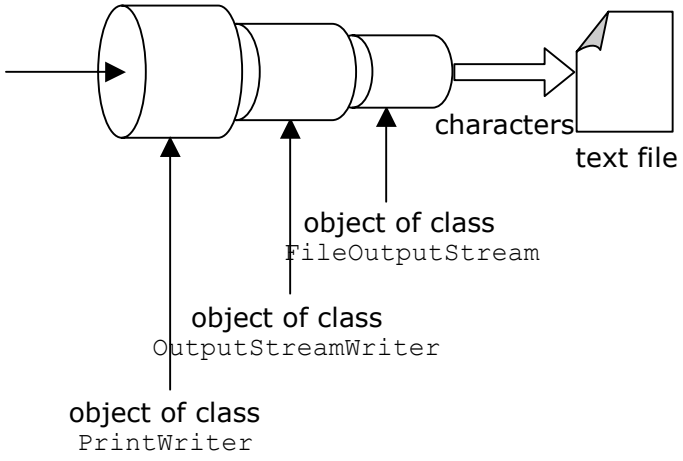
```
PrintWriter(Writer out)
PrintWriter(Writer out, boolean autoFlush)
PrintWriter(OutputStream out)
PrintWriter(OutputStream out, boolean autoFlush)
```

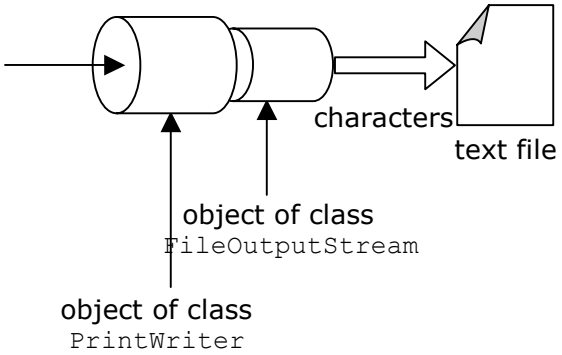
`print()` methods	`println()` methods
`print(boolean b)`	`println(boolean b)`
`print(char c)`	`println(char c)`
`print(int i)`	`println(int i)`
`print(long l)`	`println(long l)`
`print(float f)`	`println(float f)`
`print(double d)`	`println(double d)`
`print(char[] s)`	`println(char[] s)`
`print(String s)`	`println(String s)`
`print(Object obj)`	`println(Object obj)`

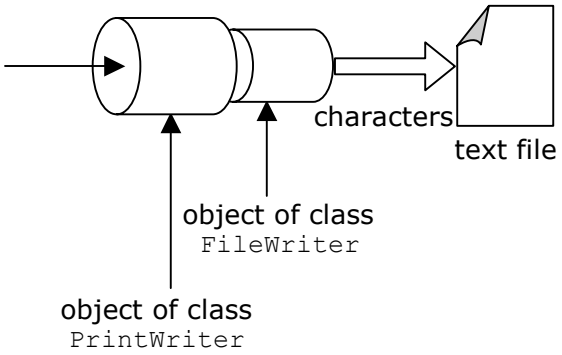
The `println()` methods write the text representation of their argument to the underlying stream, and then append a line-separator. They use the correct platform-dependentline-separator.
Example:
On Unix platforms the line separator is '`\n`'(linefeed), while on windows it is '`\r\n`' (carriage return linefeed) and on Macintosh it is '`\r`' (Carriage return).

15.25) The `print()` methods does not throw any `IOException`, instead the `checkError()` method of the `PrintWriter` class **MUST** be called to check for errors. [1]

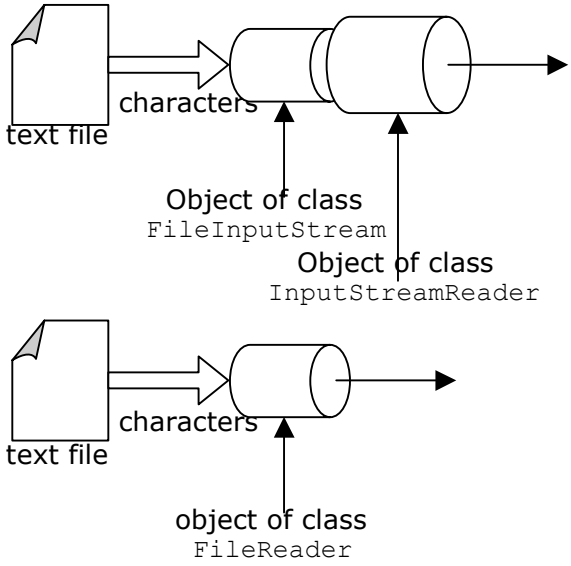
15.26) When writing text to a file using the default character encoding, the following three procedures for setting up a `PrintWriter` are equivalent. [1]





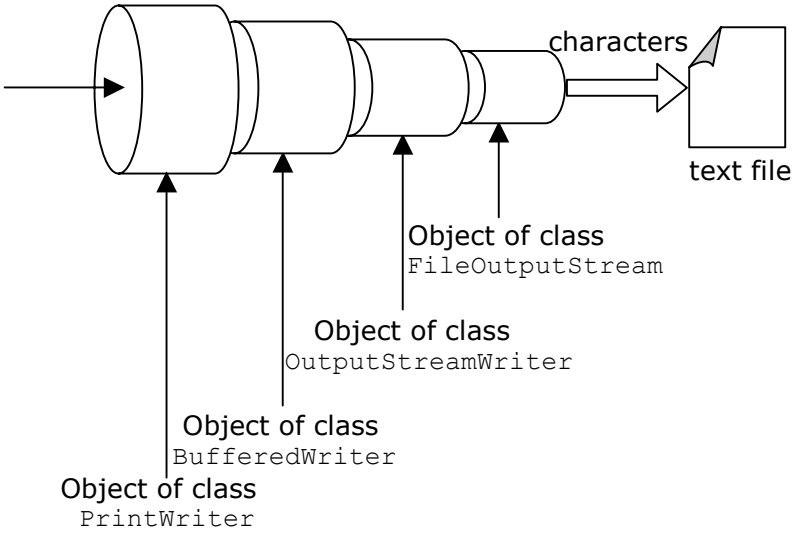


15.27) When reading characters from a file using default character encoding, the following two procedures for setting up an `InputStreamReader` are equivalent. [1]

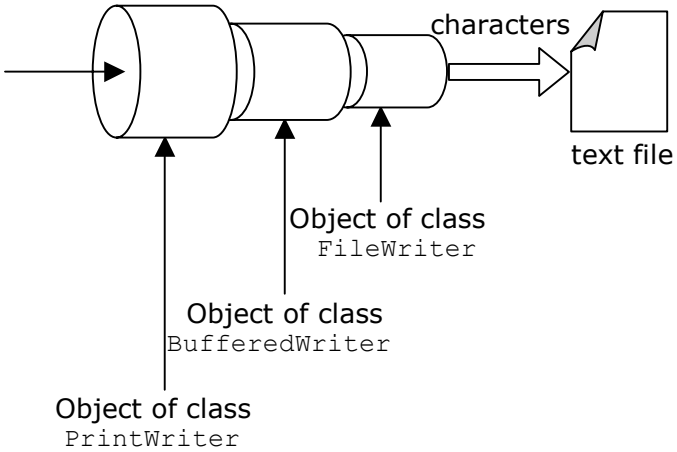


15.28) Using a buffered writers: [1]

For using a specified encoding:



For using the default character encoding:



15.29) `BufferedWriter` class provides the method `newLine()` for writing the platform dependent line separator. [1]

15.30) Standard output stream (usually the screen) is presented by `PrintStream` object `System.out`
Standard input stream (usually the keyboard) is represented by the `InputStream` object `System.in`
i.e. it is the I/P stream.
The standard error stream (also usually the screen) is repressented by the `System.err` which is another object of the `PrintStream` class

15.31) Comparison of character writers and readers: [1]

Writers	Readers
`BufferedWriter`	`BufferedReader`
No counterpart	`LineNumberReader`
`CharArrayWriter`	`CharArrayReader`
`FileWriter`	`FilterReader`
No counterpart	`PushbackReader`
`OutputStreamWriter`	`InputStreamReader`
`FileWriter`	`FileReader`
`PipedWriter`	`PipedReader`
`PrintWriter`	*No counterpart*
`StringWriter`	`StringReader`

15.32) Comparison between byte streams and character streams: [1]

Byte Streams	Character Streams
`OutputStream`	`Writer`
`InputStream`	`Reader`
`ByteArrayOutputStream`	`CharArrayWriter`
`ByteArrayInputStream`	`CharArrayReader`
No counterpart	`OutputStreamWriter`
No counterpart	`InputStreamWriter`
`FileOutputStream`	`FileWriter`
`FileInputStream`	`FileReader`
`FilterOutputStream`	`FilterWriter`
`FilterInputStream`	`FilterReader`
`BufferedOutputStream`	`BufferedWriter`
`BufferedInputStream`	`BufferedReader`
`PrintStream`	`PrintWriter`
`DataOutputStream`	*No counterpart*
`DataInputStream`	*No counterpart*
`ObjectOutputStream`	*No counterpart*
`ObjectInputStream`	*No counterpart*
`PipedOutputStream`	`PipedWriter`
`PipedInputStream`	`PipedReader`
No counterpart	`StringWriter`
No counterpart	`StringReader`
No counterpart	`LineNumberReader`
`PushbackInput`	`PushbackReader`
`SequenceInputStream`	*No counterpart*

15.33) `RandomAccessFile` class implements direct access for files, it implements both the `DataInput` and `DataOutput` Interfaces and inherits directly from the `Object` class. It presents a model of files that is incompatible with the stream/reader/writer model described previously, you can seek to a desired position within a file, and then read or write a desired amount of data. [1] [3]

15.34) The mode argument in the constructor must be either '`r`' or '`rw`' otherwise `IllegalArgumentException` is thrown. [1]

```
RandomAccessFile(String name, String mode) throws IOException
RandomAccessFile(File file, String mode) throws IOException
```

15.35) Opening a file for writing does not reset the contents of the file. An `IOException` is thrown if an I/O error occurs, must notably when the mode is "`r`" and the file does not exist, however, if the mode is "`rw`" and the file does not exist, a new emptyfile is created regardless ofthe mode, if the file does exist, its file pointer is set to the begining of the file. A `SecurityException` is thrown if the application does not have the neccessary access rights. [1]

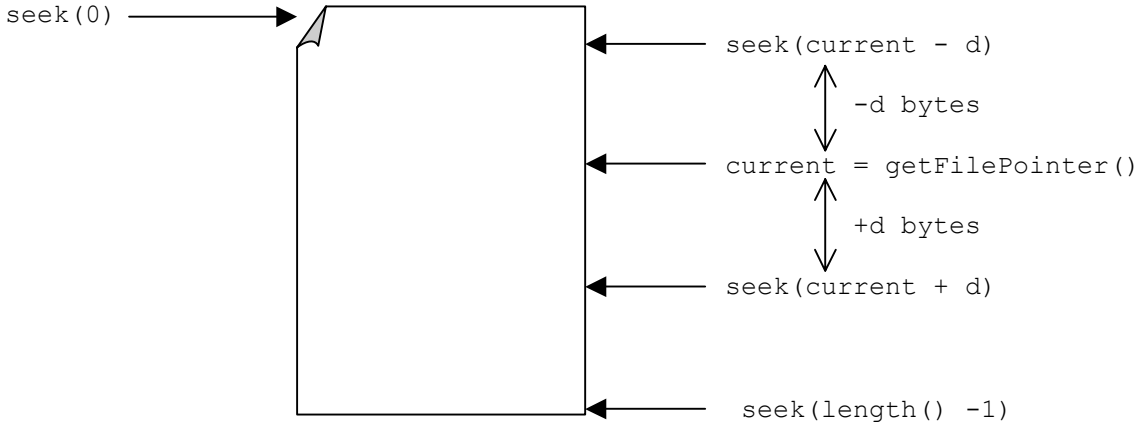
15.36) The `RandomAccessFile` class: [1] [3]

Category	Methods	Example & declaration
Navigating	`public long getFilePointer()`	Returns the current position within the file, in bytes. Subsequent reading and writing will take place starting at this position.
	`public void seek(long pos)`	Sets the file-pointer offset, measured from the **BEGINNING** of this file, at which the next read or write occurs.
	`public long length()`	Returns the length of this file.
Handling	`public void close()`	Closes this random access file stream and releases any system resources associated with the stream.

15.37) It implements all methods in table mentioned in tip (15.15), because it implements the two interface `DataInput`, `DataOutput`. Also support more common methods that support byte reading and writing as: [2] [3]

Category	Methods	Example & declaration
Reading	`public int read()throws IOException`	Reads a byte of data from this file. The byte is returned as an integer in the range 0 to 255 (`0x00-0x0ff`), or –1 if the end of the file is reached. This method blocks if no input is yet available.
	`public int read(byte[] b) throws IOException`	Reads up to `b.length` bytes of data from this file into an array of bytes. or –1 if the end of the file I reached. This method blocks until at least one byte of input is available.
	`public int read(byte[] b, int off, int len) throws IOException`	Reads up to `len` bytes of data from this file into an array of bytes. This method blocks until at least one byte of input is available.
Writing	`public void write(int b) throws IOException`	Writes the specified byte to this file. The write starts at the current file pointer.
	`public void write(byte[] b) throws IOException`	Writes `b.length` bytes from the specified byte array to this file, starting at the current file pointer.
	`public void write(byte[] b, int off, int len) throws IOException`	Writes `len` bytes from the specified byte array starting at offset `off` to this file.

15.38) Positioning the file pointer for direct file access: [1]



15.39) Object serialization allows an object to be transformed into a sequence of bytes that can be re-created into the original object. [1]

References:

[1] "A Programmer's Guide to Java Certification" A Comprehensive Primer
Khalid A. Mughal
Rolf W.Rasmussen
ISBN: 0-201-59614-8
Publisher: Addison-Wesley

[2] "Java Platform 1.2 documentation"
http://java.sun.com/docs/index.html

[3] "The complete Java 2 Certification study guide"
Simon Roberts
Philip Heller
Michael Ernest
ISBN: 0-7821-2700-2
Publisher: Sybex

[4] http://groups.yahoo.com/group/jcertification/

[5] http://www.absolutejava.com

[6] http://www.javaranch.com/campfire/StoryBits.jsp

[7] "The Java Handbook"
Patrick Naughton

[8] "Exam cram, Java 2 Exam 310-025"
Bill Brogden
ISBN:1-57610-291-2
Publisher:Coriolis

[9] http://java.sun.com/docs/books/tutorial/java/nutsandbolts/_keywords.html

[10] http://members.spree.com/education/javachina/Cert/FAQ_SCJP2.htm

[11] Veena Iyer Notes
Located in: http://groups.yahoo.com/group/jcertification/files/VeenaNotes.pdf

[12] Velmurugan Notes
mail to or inquiries: velmurugan_p@yahoo.com

[13] http://www.go4java.20m.com/mock1.htm