SPECIAL EDITION FOR CSE, DU STUDENTS

TOUCH-'N'-PASS EXAM CRAM GUIDE SERIES

OOP IN JAVA

- ✓ Theory questions and answers from all the topics for exam.
- ✓ All needed concepts in just one place for each chapter.
- ✓ List of points which should be remembered so that one does not become confused at exam.
- ✓ All-in-one complete concepts programs for each chapter.
- ✓ Exercises with solutions for practice.

And much more…

Includes Solutions to DU Java Final Exam Questions of 5 Years (2002-2006)

Prepared By

Sharafat Ibn Mollah Mosharraf

CSE, DU
Batch: 2005-2006

OOP in Java

By: Sharafat Ibn Mollah Mosharraf

sharafat_8271@yahoo.co.uk

www.sharafat.info

First Edition: May, 2008.

How This Book is Organized

The book is organized as follows:

- The chapter numbers are assigned according to the chapter numbers in the text book (Java – The Complete Reference, By Herbert Schildt, 7th Edition).
- Each chapter is divided into five parts (in most cases) as follows:
    1. **Theories:** Includes all the theories I could find or think of from the chapter.
    2. **Concepts:** Includes all the concepts discussed in the chapter.
    3. **Points to be Remembered:** Includes a list of points to be remembered for successfully solving problems of types error-finding or output generating. Footnotes indicating the passages in the text book where the points are discussed in detail are added to each point.
    4. **Complete Concepts Programs:** Includes one or more all-in-one programs where all the concepts of the chapter are applied and demonstrated.
    5. **Exercises:** Includes some exercises to practise. Solutions as well as explanations are given at the end of each problem.
- Questions which appeared on the previous year final exams (from 2002-2005) as well as in the three in-course exams on 2007 are marked by appending the year and marks inside square brackets.
- Due to many reasons, the topics on Applets, GUI and Socket Programming have not been included.

On the Website

The following materials can be found on the website for this book (http://www.sharafat.info/java_guide):

- The electronic copy of this book (in PDF, DOCX and DOC formats).
- NetBeans source code for many exercises and complete concepts programs from this book.
- Some guidelines on which books or tutorials to study for learning Java and Swing and links to download those books or tutorials.
- A discussion on which IDE is the best for developing Java applications.
- Links to sites containing lots of MCQ type problems on Java.
- Some other useful links regarding Java programming.

Table of Contents

Chapter 1 (& 2)
The History, Evolution and Overview of Java

Theories

✮✮	1.1	**Describe the useful features of object-oriented programming over the procedure-oriented (or structured) programming language.** ***[2003. Marks: 3]*** OOP language has the following advantages over structured programming language: **1. Data abstraction** In structured languages, data abstraction or hiding is achieved through only local and global variables, whereas in object-oriented languages, a higher degree of data abstraction is achieved through the uses of objects and access modifiers. **2. Inheritance** In object-oriented languages, an object can get its general attributes from its parent through a mechanism called ‘inheritance’, without copying and editing the code of the parent object. But in case of structured languages, this cannot be done without copying and editing huge amount of code, and thus leaving a great scope for making mistakes. **3. Polymorphism** In object-oriented languages, a single named method can be used to operate on different types of data, which is known as ‘polymorphism’. However, in structured languages, differently-named methods are needed to operate on different types of data, thus bearing the stress of remembering more than one names for a single job.
✮	1.2	**Why is Java called *platform independent*? *[2003. Marks: 2]*** Java does not compile a program directly to machine code, rather it translates a program to an intermediate code named bytecode, which is later interpreted by JVM to respective machine codes. Thus, a single program can be run on any platform. This is why Java is called platform independent.
✮✮✮	1.3	**What is *bytecode*? Explain its usefulness while translating a Java program in a wide variety of environments. *[2003. Marks: 4]*** **OR, How does Java make platform independence possible?** Java makes platform independence possible by translating a program into bytecode instead of machine code. Bytecode is a highly optimized set of instructions designed to be executed by the Java run-time system or JVM (Java Virtual Machine). Translating a Java program into bytecode makes it much easier to run a program in a wide variety of environments, because only the JVM needs to be implemented from each platform. Once the run-time package exists for a given system, any Java program can run on it. Although the details of the JVM will differ from platform to platform, all understand the same Java bytecode. If a Java program were compiled to native code, then different versions of the same program would have to exist for each type of CPU. Thus, the execution of bytecode by the JVM is the easiest way to create truly portable programs.
	1.4	**Explain the following OOP terminologies: [2004. Marks: 4]** **i) Data Abstraction** **ii) Inheritance** **Data Abstraction:**

Abstraction refers to the act of representing essential features without including the background details or explanations. Hence, data abstraction means hiding detailed data from object behaviors. Objects can be treated as concrete entities that respond to messages telling them to *do something*, without knowing the details of *how* they would do it.

Inheritance:

Inheritance is the process by which one object acquires the properties of another object. By use of inheritance, an object would need only define those qualities that make it unique within its class. It can inherit its general attributes from its parent. Thus, it is the inheritance mechanism that makes it possible for one object to be a specific instance of a more general case.

1.5 **What is *polymorphism*? How does polymorphism promote extensibility? *[2004. Marks: 3]***

Polymorphism is a feature that allows one interface to be used for a general class of actions. The specific action is determined by the exact nature of the situation.

By dint of polymorphism, it is possible to design a generic interface to a group of related activities. This helps reduce complexity by allowing the same interface to be used to specify a general class of action. It is the compiler's job to select the specific action as it applies to each situation. The programmer does not need to make the selection manually. He needs only remember and utilize the general interface. In this way, polymorphism promotes extensibility.

1.6 **How Java changed the internet?**

OR, What is the usefulness of Java with regard to internet?

Java addressed some of the thorniest issues associated with the internet: portability and security. They are described below.

Portability:

Portability is a major aspect of the internet because there are many different types of computers and operating systems connected to it. As Java is a portable language, programs written in it runs just fine in any platform. Thus, Java solves the issue of portability.

Security:

Whenever a program is downloaded, there lies a risk, because the code downloaded may contain a malware or other harmful code. In order to ensure protection, Java confines an applet to the Java execution environment and does not allow it to access other parts of the computer.

1.7 **What are the core parts of OOP? Describe them in brief.**

The core parts of OOP are:

1. Encapsulation
2. Inheritance
3. Polymorphism

Encapsulation is the mechanism that binds together code and the data it manipulates, and keeps both safe from outside interference and misuse.

Inheritance is the process by which one object acquires the properties of another object. By use of inheritance, an object would need only define those qualities that make it unique within its class. It can inherit its general attributes from its parent. Thus, it is the inheritance mechanism that makes it possible for one object to be a specific instance of a more general case.

Polymorphism is a feature that allows one interface to be used for a general class of actions. The specific action is determined by the exact nature of the situation.

Chapter 3 (& 5)
Data Types, Variables, Arrays and Control Statements

Theories

3.1 **What do you mean by dynamic initialization of a variable in Java? Give an example.** ***[2003. Marks: 3]***

In Java, a variable can be initialized dynamically using any expression valid at the time the variable is declared. This is called dynamic initialization of a variable.

```java
class Area {
    public static void main(String[] args) {
        float height = 2.0f;
        float width = 3.0f;
        float areaOfTriangle = 1 / 2 * height * width;
    }
}
```

In the above example, variables `a` and `b` are initialized by constants, but variable `c` is initialized dynamically using the expression `1 / 2 * height * width`.

☆ 3.2 **What do you understand by scope and lifetime of a variable? Explain with examples.** ***[2002. Marks: 4]***

The scope of a variable determines to which extent that variable can be seen or used in a program.

The lifetime of a variable decides how long the storage for that variable exists in memory.

```java
class Scope {
    public static void main(String[] args) {
        int x; //known to all code within main
        x = 10;
        if (x == 10) {
            int y = 20; //known only to this 'if' block

            //x and y both are known here
            System.out.println("x and y: " + x + " " + y);
            x = y * 2;
        }
        // y = 100; //Error! y is not known here

        //x is still known here
        System.out.println("x is: " + x);
    }
}
```

In the above example, as the comments indicate, the variable `x` is declared at the start of `main()`'s scope and is accessible to all subsequent code within `main()`. Within the `if` block, `y` is declared. Since a block defines a scope, `y` is only visible to other code within its block.

Again, the variable `x` is created at the beginning of the `main()` method. So, its lifetime is until the end of the method. Variable `y` is created inside the `if` block. Hence, its lifetime is until the block ends.

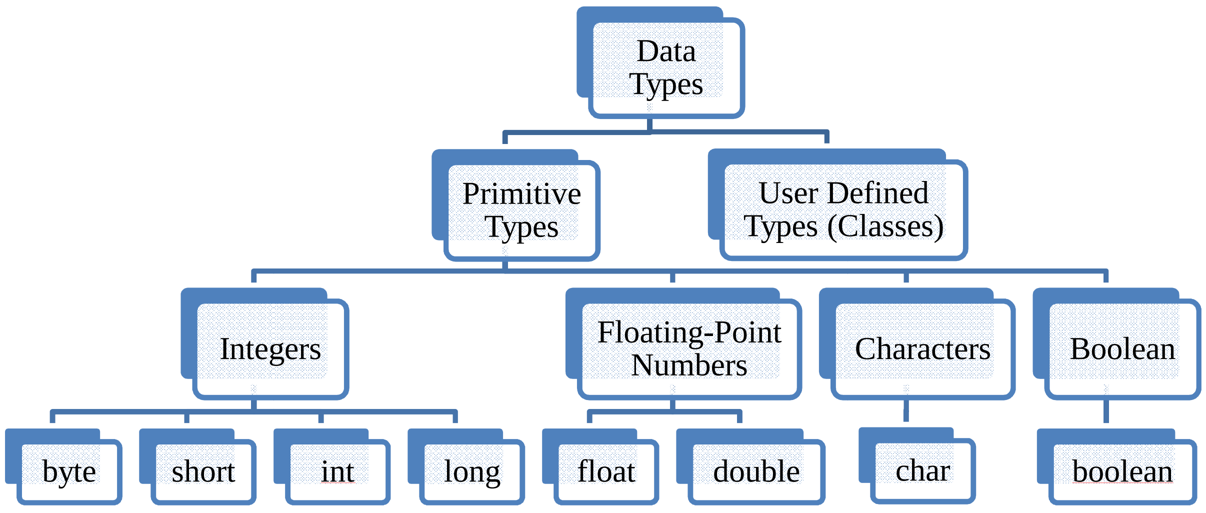
☆ 3.3 **What are the differences between *type conversion* and *casting*? What is automatic type promotion?** ***[2002. Marks: 2]***

Automatic conversion between *compatible* types is called type conversion, and manual

		conversion between *incompatible* types is called type casting. While evaluating expressions, Java automatically promotes all byte, short and char values to int; and in case of the presence of a float or double, they are promoted to float or double respectively. This is called automatic type promotion.
✮✮	3.4	**What are the differences between the constants 7, ‘7’ and “7”?** 7 is an integer literal, ‘7’ is a character literal, and “7” is a string literal.
	3.5	**Is the `switch` statement more efficient than the `if` statement? Why?** Yes, the switch statement is more efficient than the if statement. When Java compiles a switch statement, the compiler inspects each of the case constants and create a “jump table” that it will use for selecting the path of execution depending on the value of the expression. Therefore, when it is needed to select among a large group of values, a switch statement will run much faster than the equivalent logic coded using a sequence of **if-else**s.
	3.6	**How does a `switch` statement differ from an `if` statement?** A **switch** statement differs from the if in that **switch** can only test for equality, whereas **if** can evaluate any type of boolean expressions.

Concepts

3.1



3.2

Escape Sequence	Description
`\ddd`	Octal Character (ddd)
`\uxxxx`	Hexadecimal Unicode Character (xxxx)
`\'`	Single Quote
`\"`	Double Quote
`\\`	Backslash
`\r`	Carriage Return
`\n`	New Line (a.k.a. line feed)
`\f`	Form Feed
`\t`	Tab
`\b`	Backspace

3.3

Data Type	Width (Bits)	Range	Declaration
`byte`	8	$(-2^{8}/2)$ to $(2^{8}/2-1)$	`byte b = 6;`
`short`	16	$(-2^{16}/2)$ to $(2^{16}/2-1)$	`short s = 1024;`

`int`	32	$(-2^{32}/2)$ to $(2^{32}/2-1)$	`int i = 50000; //decimal`
			`int i = 05;    //octal`
			`int i = 0x5A;  //hexadecimal`
			`int i = 0X5a;  //hexadecimal`

`long`	64	$(-2^{64}/2)$ to $(2^{64}/2-1)$	`long l = 999999999l;`
			`long l = 999999999L;`
			`long l = 9999999999; //error`

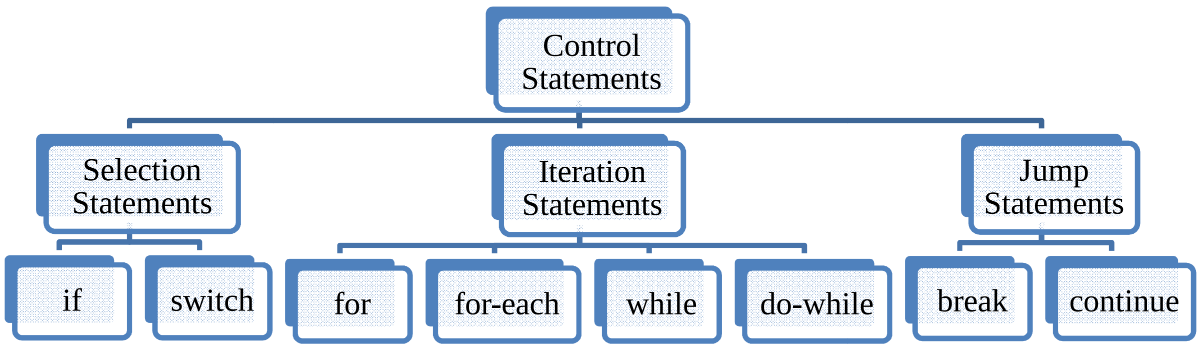
`float`	32	$1.4e^{-45}$ to $3.4e^{+38}$ (with 7 significant digits)	`float f = 1.6736421f;`
			`float f = 1.673e232f;`
			`float f = 1.673E-23F;`

`double`	64	$4.9e^{-324}$ to $1.8e^{+308}$ (with 15 significant digits)	`double d = 5.98e-105;`
			`double d = 5.98e+105d;`
			`double d = 5.98e105D;`

`boolean`	1	*true* or *false*	`boolean b = true;`
			`boolean b = false;`

`char`	16	'\u0000' to '\uFFFF'	`char c = '!';`
			`char c = '\u0995'; //ক`

3.4



3.5

Type Compatibility and automatic type conversion:[1]

To / **From**	byte	short	int	long	float	double	char	boolean
byte	Y	Y	Y	Y	Y	Y	N	N
short	N	Y	Y	Y	Y	Y	N	N
int	N	N	Y	Y	Y	Y	N	N
long	N	N	N	Y	Y	Y	N	N
float	N	N	N	N	Y	Y	N	N
double	N	N	N	N	N	Y	N	N
char	N	N	Y	Y	Y	Y	N	N
boolean	N	N	N	N	N	N	N	N

[1] Notice that we can assign char type variable to int, float, long or double. But we cannot assign byte or short type variables to char. This is because char type is unsigned, whereas byte and short types are signed.

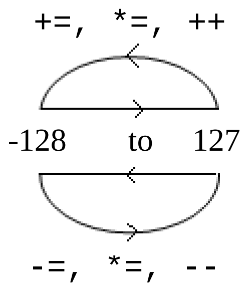
Points to be Remembered

✯✯	3.1	Integer literals[2] can be represented in decimal, octal or hexadecimal format. But floating-point literals can be represented *only* in decimal format.[3]
✯✯	3.2	Octal values are denoted in Java by a leading zero. Normal decimal numbers cannot have a leading zero. Hexadecimal values are represented using a leading zero-x (**0x** or **0X**).[4]
✯	3.3	All integer literals are by default of type `int`. So, to declare a literal as type `long`, an **`l`** or **`L`** should be appended to it.[5]
✯✯✯	3.4	All floating-point literals are by default of type `double`. So, to declare a literal as type `float`, an **`f`** or **`F`** must be appended to it.[6]
	3.5	A `double` literal can be declared by appending **`d`** or **`D`** to it, but it is not essential.[7]
✯✯✯	3.6	Boolean literals are only `true` or `false`. A `true` literal does not equal 1, nor does the `false` literal equal 0.[8]
✯✯✯	3.7	A block defines a scope and a variable is visible within that block.[9] The scope defined by a method begins with its opening curly brace. However, if that method has parameters, they too are incuded within the method's scope.[10]
✯	3.8	Variables declared inside a scope are not visible (i.e., accessible) to code that is defined outside that scope.[11]
✯✯✯	3.9	When one type of data is assigned to another type of variable, an automatic type conversion will take place if the following two conditions are met: 1. The two types are compatible. 2. The destination type is larger than the source type.[12]
✯✯✯	3.10	When a larger *integer* type value is cast into a smaller *integer* type value, it is reduced to the smaller type's modulo range.[13] But when a double value is cast into a float value and the value's range is out of the range of float, then the float value will contain *infinity*.
✯✯✯	3.11	**Type Promotion Rules:** First, all `byte`, `short` and `char` values are promoted to `int`. Then, if one operand is a `long`, the whole expression is promoted to `long`. If one operand is a `float`, the entire expression is promoted to `float`. If any of the operands is `double`, the result is `double`.[14]
	3.12	Type promotion rules are applicable when an expression *containing a **variable*** is evaluated, but when an increment/decrement (`++` , `--`) or arithmetic compound assignment operators (`+=` , `-=` , `*=` , `/=` , `%=`) are performed on a variable, no casting is needed even if

[2] *Literal* is a constant value which is assigned to a variable to is used to evaluate expressions. For example, consider the following:

`100`	`98.6`	`'a'`	`"Hello"`	`true`

Here, `100` is an *integer* literal, `98.6` is a *floating-point* literal, `'a'` is a *character* literal, `"Hello"` is a *string* literal, and `true` is a *boolean* literal.
[3] p.39, topic: "Integer Literals".
[4] p.39, topic: "Integer Literals".
[5] p.40, 1st para.
[6] p.40, 3rd para.
[7] p.40, 3rd para, 2nd line.
[8] p.40, 4th para, 3rd line.
[9] p.42, topic: "The Scope and Lifetime of Variables", 1st para, 3rd line.
[10] p.43, 2nd para, 1st line.
[11] p.43, 3rd para, 1st line.
[12] p.45, topic: "Java's Automatic Conversions".
[13] p.46, 1st para.
[14] p.47, topic: "The Type Promotion Rules", 1st para.

		the result is greater than the highest range of that variable. In that case, the value will become negative-to-positive or positive-to-negative. For example:
		`byte b = 127; //Highest range of byte type`
		`b++; //Now, b = -128`
		`b += 3; //Now, b = -125`
		`b = -128; //Lowest range of byte type`
		`b--; //Now, b = 127`
		`b -= 3; //Now, b = 124`
		
		Therefore, the rule is as follows:
		For range of type byte: 

☆☆	3.13	If an expression contains only literals, then the type promotion rule will not be applicable. But in that case, if the result is beyond the range of the type of variable in which we are assigning the value, then casting is needed. For example:
		`b = 2 * 3 + 5; //OK. b = 11`
		`//b = 100 * 3; //Error. Result 300 is beyond the range of byte`
		`b = (byte) (100 * 3); //OK. b = 44`

☆☆☆	3.14	In multidimensional arrays, only the first dimension needs to be specified. The other dimensions may or may not be specified and they can be different or the same.[15]
	3.15	The expression in a `switch` statement must be of type byte, short, int or char; each of the values specified in the `case` statements must be of a type compatible with the expression. Again, each `case` value must be a unique literal (i.e., it must be a constant, not a variable). Duplicate `case` values are not allowed.[16]

Complete Concepts Program

```
/* CompleteConcept_Chapter3.java */

public class CompleteConcept_Chapter3 {
    public static void main(String[] args) {
        //Literal assignments - Points 3.3, 3.4, 3.5
        byte bt = 45;
        short s = 666;
        int i1 = 45543;
        long l1 = 214234;
        long l2 = 214234L;
        //long l3 = 3000000000; //Error. Integer number too large.
        float f = 123.12345678f;
        System.out.println(f); //Prints 123.12346 (up to 7 significant digits)
        //f = 1.1; //Error. See point 3.4
        double d1 = 1234567.12345678910111883141 5;
        d1 = 12.25D;
        char c1 = 'a';
        boolean flag = true;
```

[15] p.51, last para.
[16] p.81, 1st para.

```java
//Octal & Hexadecimal representation - Points 3.1 & 3.2
bt = 0xA; //bt = 10
bt = 012; //bt = 10
f = 0xA; //f = 10.0
//f = 0x2.5; //Error. Malformed floating point literal


//Automatic type promotions
byte b1 = 10, b2 = 5;

//Using expression including variables - whether overflow or not
byte b = (byte) (b1 * b2); //b = 50. See points 3.11 & 3.10

//Using expression without variables - no overflow
b = (5 / 2) * 4; //Now, b = 8. See point 3.13

//Using expression without variables - overflow
b = (byte) (100 * 3); //Now, b = 44. See points 3.13 & 3.10

//Using arithmetic compound assignment operator - whether overflow or not
b *= 3; //Now, b = -124 [44 * 3 = 132 = (127 + 1) + 4 = -128 + 4 = -124].
        //See point 3.12

//Using expression without variables -
//with arithmetic compound assignment operator - whether overflow or not
b += 100 * 3 ; //Now, b = -80 [-124 + 300 = (-124 + 124 + 127 + 1) + 48
               //                         =  -128 + 48 = -80].
               //See point 3.12

//Using expression including variables -
//with arithmetic compound assignment operator - whether overflow or not
b += b1 * 21;  //Now, b = 10 [-80 + 210 = (-80 + 80 + 127 + 1) + 2
               //                       = -128 + 2 = -126]. See point 3.12


//char-to-others & others-to-char conversion – concept 3.4
//Note: the erroneous assignments can be fixed by casting.
int a = 65;
byte b3 = 65;
char c = 'A';
a = 'B'; //OK. a = 66 (acceptable literal)
c = 66; //OK. c = 'B' (acceptable literal)
//c = 70000; //Error. 70000 is beyond the range of char (0 - 65536).
a = c; //OK. a = 65.
//b3 = c; //Error. c is beyond the range of byte.
//c = b3; //Error. See footnote 1.


//Array declaration
//One-dimensional array
int[] a1 = new int[5];
int a2[] = {1, 2, 3, 4, 5};

//Two-dimensional array with second dimension defined
int[][] a3 = new int[2][3];
for (int i = 0; i < a3.length; i++) {
    for (int j = 0; j < a3[i].length; j++) {
        System.out.print(a3[i][j] + " ");
    }
    System.out.println();
}
/* Output:
 * 0 0 0
 * 0 0 0
 */

//Two-dimensional array with second dimension omitted
```

```java
        int[][] a4 = new int[5][];
        for (int i = 0; i < a4.length; i++) {
            a4[i] = new int[i];
            for (int j = 0; j < a4[i].length; j++) {
                System.out.print(a4[i][j] + " ");
            }
            System.out.println();
        }
        /* Output:
         * 0
         * 0 0
         * 0 0 0
         * 0 0 0 0
         */


        //Variable scope & Lifetime
        int x = 10; //known to the rest of the code within main()
        if (x == 10) { //start new scope
            int y = 20; //known only to this block

            //x & y both are known here
            System.out.println("x & y: " + x + " " + y);
            x = y * 2;
            //int x = 4; //Error. x is already defined.
        }
        //y = 100; //Error. y is not known here.

        //x is still known here
        System.out.println("x is: " + x);
    }
}
```

Exercises

☆☆ 3.1 **Identify errors in the following program, correct them and write the output.** ***[Incourse-1, 2007. Marks: 4]***

```java
 1 class test {
 2     public static void main(String[] args) {
 3         byte a = 100;
 4         short b = a * 3;
 5         long l = 2000;
 6         float k = 284.24;
 7         byte c = k;
 8         int m = a;
 9         double d = b;
10
11         System.out.println(b);
12         System.out.println(c);
13         System.out.println(d);
14     }
15 }
```

Solution:

Error 1: Line 4: Possible loss of precision. Found: int, required: short.
Correction: `short b = (short) (a * 3);`

Error 2: Line 6: Possible loss of precision. Found: double, required: float.
Correction: `float k = 284.24f;`

Error3: Line 7: Possible loss of precision. Found: float, required: byte.
Correction: `byte c = (byte) k;`

Output:

```
300
28
300.0
100
```

3.2 **Write a program in Java that will print the following output on the screen: *[2005. Marks: 3]***

0 0 0 0 0
0 0 1 2 3
0 1 3 5 7
0 2 5 8 11

Solution:

```
public class Main {
    public static void main(String[] args) {
        int[][] a = {{0, 0, 0, 0, 0},
                     {0, 0, 1, 2, 3},
                     {0, 1, 3, 5, 7},
                     {0, 2, 5, 8, 11}};

        for (int i = 0; i < a.length; i++) {
            for (int j = 0; j < a[i].length; j++) {
                System.out.print(a[i][j] + " ");
            }
            System.out.println();
        }

    }
}
```

3.3 **Write down the output of the following sequence of code:**

```
for (int I = 0; I < 8; I++) {
      for (int J = 4 - (I % 4); J > 0; J--)
           System.out.print("");
      for (int J = 0; J < (I % 4) + 1; J++)
           System.out.print("X");
      System.out.println();
}
```

Solution:

```
X
XX
XXX
XXXX
X
XX
XXX
XXXX
```

3.4 **Consider the following Java program:**

```
public class Main {
    public static void main(String[] args) {
        int i, j, k, a[];
        a = new int[5];
        for (k = 0; k < 5; k++) a[k] = 1;
        for (i = 1; i < 4; i++)
            for (j = i; j > 0; j--)
                a[j] += a[j-1];
    }
}
```

Generate the initial content of the array a (after the first loop) and then show the contents of it after each iteration (for each value of `i`) of the loop containing `i`.

Modify the program so that it displays the contents of a after each iteration.

Solution:

Initial contents of **a**: [1, 1, 1, 1, 1]

Contents of **a** after each iteration:

i	**a[0]**	**a[1]**	**a[2]**	**a[3]**	**a[4]**
1	1	2	1	1	1
2	1	3	3	1	1
3	1	4	6	4	1

Modified program:

```
public class Main {
    public static void main(String[] args) {
        int i, j, k, a[];
        a = new int[5];
        for (k = 0; k < 5; k++) a[k] = 1;
        for (i = 1; i < 4; i++) {
            for (j = i; j > 0; j--)
                a[j] += a[j-1];
            for (k = 0; k < 5; k++)
                System.out.print(a[k] + " ");
            System.out.println();
        }
    }
}
```

Chapter 4

Operators

Theories

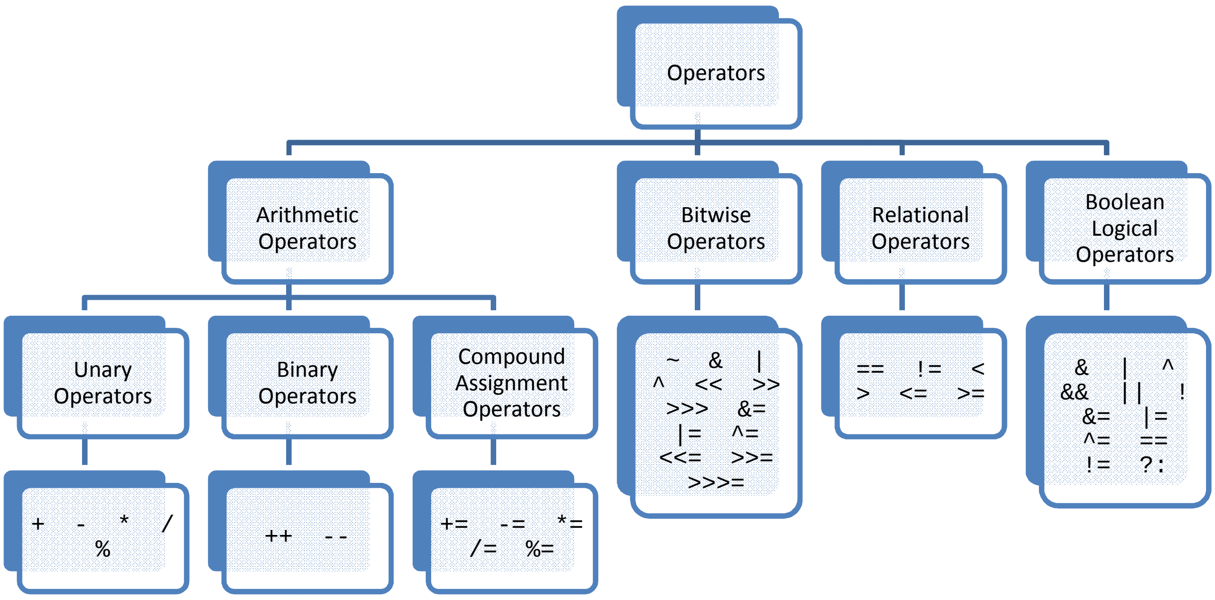
4.1 **What is *sign extension* in Java and why is it used?**

In Java, when a value is shifted right, the top (leftmost) bits exposed by the right shift are filled with the previous contents of the top bit. This is called *sign extension*.

It is used to preserve the sign of negative numbers when shifting them right.

Concepts

4.1



4.2 **Operator Precedence Chart:**

Highest			
( )	[ ]	.	
++	- -	~	!
*	/	%	
+	–		
>>	>>>	<<	
>	>=	<	<=
==	!=		
&			
^			
|			
&&			
||			
? :			
=	op=		
Lowest			

Points to be Remembered

	4.1	For each shift left, the *high*-order bit is shifted out (and lost), and a zero is brought in on the right.[17]
	4.2	Due to Java's automatic type promotion system, when a byte or short value is shifted left, the result must be cast back to byte or short type to get the correct result.[18]
	4.3	Each left shifting has the effect of doubling the original value. So, it's an efficient way to *multiply* by 2. But if a 1 is shifted to the MSB[19], then the value will become negative.[20]
	4.4	For each right shift, the *low*-order bit is shifted out (and lost), and on the left, a copy of the previous bit is brought (i.e., if the previous bit on that position was 0, then a 0 is brought; if there was 1, then a 1 is brought).[21]
	4.5	Each right shifting has the effect of dividing the original value by 2, and any remainder is discarded.[22]
	4.6	When a value is unsigned shifted right, a 0 is brought in on the left, no matter what its initial value was.[23]
	4.7	Java does not treat the boolean value `false` as equivalent to integer 0 or `true` as equivalent to integer 1.[24]
	4.8	In case of short-circuit logical operators (&&, `||`), if the left expression is false, the right side expression is not evaluated. But in case of single-character logical operators, all the expressions are evaluated.[25]

Exercises

✫✫✫	4.1	**Write a single Java statement to find the largest value of three integer variables a, b and c.** `int largest = (a > b) ? ((a > c) ? a : c) : ((b > c) ? b : c);`
	4.2	**Write a single Java statement to find the smallest value of three integer variables a, b and c.** `int smallest = (a < b) ? ((a < c) ? a : c) : ((b < c) ? b : c);`
✫✫	4.3	**Figure out the output of the following program code and explain your answer.** ***[2002. Marks: 4]***

```java
class E4_2 {
    public static void main(String[] args) {
        byte a = 112, b;
        int c = 8, d = 3;
        int i ;
        c %= d; d ^= c;
        b = (byte) (a << 2);
        System.out.println("a " + a);
        System.out.println("i " + i);
        System.out.println("b " + b);
```

[17] p.65, 3rd para, 3rd line.
[18] p.65, 4th para. [**Note:** when casting, the lower 8 or 16 bit (for byte and short type respectively) will be taken and the rest of the bits will be discarded.]
[19] *MSB* – Most Significant Bit, i.e., the left-most bit of a number.
[20] p.66, 2nd para.
[21] p.67, 5th para, 1st line.
[22] p.67, 4th para.
[23] p.68, topic: "The Unsigned Right Shift", 1st para.
[24] p.71, from top.
[25] p.73, 2nd para.

```
11          System.out.println("c " + c);
12          System.out.println("d " + d);
13      }
14 }
```

Solution:

The statement on line 9 will not compile as variable ‘i’ is not initialized. The output of the other `println()` statements will be as follows:

```
a 112
b -64
c 2
d 1
```

Explanation:

1. The value of variable ‘a’ is unchanged. So, 112 is printed as its value.
2. In the statement `c %= d;`, the result of the operation (2) is assigned to ‘c’. So, 2 is printed as its value in the 4th println() statement.
3. In the statement `d ^= c;`, the result of the operation [(3 ^ 2) = 1] is assigned to ‘d’. So, 1 is printed as its value in the 5th println() statement.
4. In the statement `b = (byte) (a << 2);`, the result of 2-bits left shifting is of type int. So, when it is casted into type byte, the lower 8 bits of the result (which is 1) is assigned to ‘b’. However, in this case, the MSB of the byte value is 1, which means that the value is negative. So, -64 is printed as its value in the 3rd println() statement.[26]

[26] See point 4.2 and the footnote on that point for details.

Chapter 6 (& 7)
Classes, Objects, Methods and Fields

Theories

6.1 **What is a *class* and what is an *object*?**

A *class* is a collection of *fields* (data) and *methods* (that operate on those fields).

An *object* is an instance of a class.

6.2 **What are the characteristics of an object?**

Three properties characterize objects:

1. Identity: the property of an object that distinguishes it from other objects.
2. State: describes the data stored in the object.
3. Behavior: describes the methods in the object's interface by which the object can be used.

6.3 **"Class is a logical construct and an object has physical reality" – explain the statement with example. *[2005. Marks: 2]***

A class defines the general characteristics and behavior of an object. Therefore, an object is an instance of a class. Consider the following example:

```
class Box {
   int height;
   int width;
   Box(int h, int w) {
         height = h;
         width = w;
   }
}

class Main {
   public static void main(String[] args) {
         Box b1 = new Box(3, 2);
         Box b2 = new Box(5, 5);
   }
}
```

In the above example, the **Box** class defines the general characteristics of boxes, and the objects b1 and b2 are two specific instances of that class – which have physical realities.

6.4 **Describe the lifecycle of an object. *[2002. Marks: 2]***

An object is instantiated when the **new** statement is used. It is destroyed when the method in which it was instantiated returns.

6.5 **Differentiate among instance variable, class variable and local variable. *[2005. Marks: 3]***

Instance Variables (Non-Static Fields): An instance variable is any field declared *without* the **static** modifier. It is called such because its value is unique to each instance of a class (or object).

Class Variables (Static Fields): A class variable is any field declared with the **static** modifier; this tells the compiler that there is exactly one copy of this variable in existence, regardless of how many times the class has been instantiated.

Local Variables: A local variable is a variable declared inside a method.

6.6 **What type of variable should be used to store data that is important throughout an object's lifespan? *[2004. Marks: 1]***

Class variables (i.e., static fields) should be used to store data that is important throughout an object's lifespan.

6.7 **Is it possible to declare a class using only variables or using only methods? Justify your answer. *[2003. Marks: 3]***

Yes, it is possible to declare a class using only variables or using only methods.

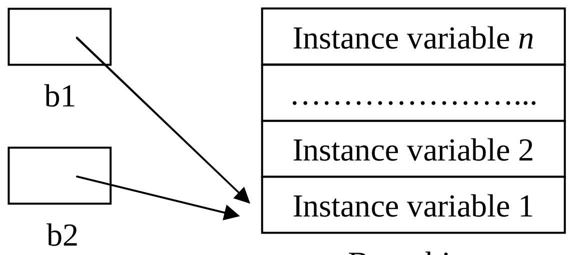
Following is an example of such cases:

```java
class Box {
    int height;
    int width;
}

class Print {
    void show(Box x) {
        System.out.println("Height: " + x.height
                                  + "\nWidth: " + x.width);
    }
}

public class test {
    public static void main(String[] args) {
        Box b = new Box();
        b.height = 5;
        b.width = 2;
        new Print().show(b);
    }
}
```

In the above example, the **Box** class has only variables, whereas the **Print** class has only methods.

6.8 **Suppose Box is a class. Now draw the memory allocation of the following three statements: *[2005. Marks: 3]***

```java
Box b1;
b1 = new Box();
Box b2 = b1;
```



Box object

☆ 6.9 **What does a constructor do? What are the syntactic differences between a constructor and a method? *[2006. Marks: 2]***

A constructor initializes an object immediately upon creation.

The syntactic difference between a constructor and a method is that a constructor has no return type (not even void), whereas a method must have a return type or void if it does not return anything.

6.10 **How is main() method declared in Java? Discuss briefly the meaning of each part of the main() method declaration. *[2005. Marks: 3]***

In Java, main() method is declared as follows:

```
public static void main(String[] args)
```

Meaning of each part of the method declaration is as follows:

1. **public:** Specifies that this method can be called from any class.
2. **static:** Specifies that this method can be called without instantiating an object of the main class.
3. **void:** Specifies that this method returns nothing.
4. **main:** This is the name of the method.
5. **String[] args:** Specifies that this method takes an array of String as its parameter.

☆ ☆ ☆ 6.11 **What problem will arise in the following constructor? How can you solve it?** ***[2005. Marks: 4]***

```
class Student {
    int roll;
    int marks;
    Student(int roll, int marks) {...............}
}
```

The following statements will have to be put inside the constructor:

```
roll = roll;
marks = marks;
```

In this case, the local variables **roll** and **marks** will be assigned the values they already contain. The fields **roll** and **marks** will not be assigned the expected value.

To solve this problem, we can use the **this** keyword as follows:

```
this.roll = roll;
this.marks = marks;
```

The **this** keyword refers to the fields of the current object and thus assigns the desired values to the fields instead of the local variables.

☆ 6.12 **Explain why you must be careful when passing objects to a method or returning objects from a method.** ***[2002. Marks: 3]***

Objects are passed to a method using *call-by-reference*. Thus, actually the reference of the object is passed to the method. Consequently, if any change in the object is made, the change reflects in the original object.

☆ 6.13 **What is *access specifier*? Distinguish among them with examples.** ***[2002. Marks: 3]***

An access specifier is a keyword which specifies how a class member can be accessed.

There are four types of access specifiers:

1. Anything declared `public` can be accessed from *anywhere*.
2. Anything declared `private` cannot be seen outside of its *class*.
3. Anything declared as default access (package-private) can be accessed from anywhere in the *same* package.
4. Anything declared as `protected` can be accessed from anywhere in the *same package*, plus from only its subclasses *outside* the package.

☆ ☆ ☆ 6.14 **Consider the following piece of code:**

```
Employee E1 = new Employee("Karim", 5001);
Employee E2 = new Employee(E1);
```

What are the values of the expressions `E1.equals(E2)` and `E1 == E2`? Why? ***[2004, 2005. Marks: 3]***

The value of `E1.equals(E2)` is **true**, whereas the value of `E1 == E2` is **false**.

This is because when two objects are compared using the == operator, only their references are compared; and as they are two different objects, their references are different. So, the latter expression produces **false**. On the other hand, the `equals()` method compares

		the fields within two objects; and as they are the same, the result of the first expression is **true**.
	6.15	**Why are Java's primitive data types not implemented as objects?** Java's primitive data types are not implemented as objects to increase efficiency. Objects have many features and attributes that require Java to treat them differently than it treats primitive types. By not applying the same overhead to the primitive types that applies to objectsa, Java can implement the primitive types more efficiently.
	6.16	**What is a *variable-arity* method?** **OR, What is a *varargs* method?** A method that takes a variable number of arguments is called a variable-arity method or a varargs method. For example: `void aMethod(int ... x) {}`
	6.17	**Why sometimes a variable is declared as *final*? *[2005. Marks: 2]*** Sometimes a variable is declared as final to make it unchangeable, i.e., to make it a constant.
✯	6.18	**What is "garbage" and what is the function of the "garbage collector"? *[2005. Marks: 2]*** **OR, How does Java manage free memory? *[2002. Marks: 2]*** When an object is no longer used, it is called "garbage". The function of the "garbage collector" is to determine objects which are no longer used (by finding out if no reference to an object exists) and reclaim the memory occupied by the object.
✯	6.19	**What is the use of the *finalize()* method? What is its disadvantage?** By using the finalize() method, some specific actions can be defined that will occur when an object is just about to be reclaimed by the garbage collector. finalize() is only called just prior to garbage collection. It is not called when an object goes out-of-scope, for example. This means that it cannot be known – or even if – finalize() will be executed. So, this method cannot be relied upon for normal program execution.

Points to be Remembered

✯	6.1	When one object reference variable is assigned to another object reference variable, a copy of the object is not being created, only a copy of the reference is being made.[27]
✯	6.2	Constructors have no return types, not even void. This is because the implicit return type of a class' constructor is the class type itself.[28]
✯	6.3	The default constructor automatically initializes all instance variables to their default values. The default initialization values are given below:

Instance Variable Type	Default Initialization Value
byte	0
short	0
int	0

[27] p.111, topic: "Assigning Object Reference Variables".
[28] p.117, topic: "Constructors", 2nd para, 4th line.

long	`0L`
float	`0.0f`
double	`0.0`
char	`'\u0000'` (null character)
boolean	`false`
String	`""` (Note that there is no space between the quotes)
Any Object	`null`[29]

☆ ☆ ☆	6.4	All *fields* (both static and non-static) are always initialized (either automatically by default values or by user-given values) upon an object initialization. But all *local variables* **must** be initialized by the user.
☆ ☆ ☆	6.5	Overloaded methods must differ in the type and/or number of their parameters.[30] Return types do not play a role in overload resolution.[31] In some cases, Java's automatic type conversions can play a role in overload resolution.[32]
	6.6	When a primitive type is passed to a method, it is done by use of call-by-value. Objects are implicitly passed by use of call-by-reference.[33]
☆ ☆ ☆	6.7	Methods declared as static have several restrictions: 1. They can only call other **static** methods. 2. They must only access **static** data. 3. They cannot refer to **this** or **super** in any way.[34]
☆ ☆ ☆	6.8	An inner class has access to all of the members of its enclosing class (whether they are public, private, protected or package-default). To access any member of inner class (whether they are public, private, protected or package-default), the outer class must instantiate an object of the inner class. [35]

	6.9	In a varargs method, the varargs parameter must be last.[36] For example:
		`int doIt(int a, boolean b, int ... vals) {}`
		`//int doIt(int a, int ... vals, boolean b) {} //Error!`

	6.10	There must be only one varargs parameter.[37] For example:
		`int doIt(int a, int ... vals) {}`
		`//int doIt(int a, int ... vals, boolean ... x) {} //Error!`

	6.11	***Zero*** or more arguments may be passed to a varargs parameter.[38] For example: `int doIt(int ... vals) {}` We can call the above method as `doIt()` or `doIt(2)` or `doIt(2, 3, ..., n)` etc.

Complete Concepts Program – Class, Objects and Methods

[29] Note that there is an important difference between *null* and *null character*. *null* is a Java keyword which means that a variable is not initialized, whereas *null character* is the first Unicode or ASCII character.
[30] p.125, topic: "Overloading Methods", 2nd para, 3rd line.
[31] p.126, 1st para, last line.
[32] p.126, 2nd para, 3rd line.
[33] p.134, the "Remember" para.
[34] p.141, topic: "Understanding Static", 3rd para.
[35] p.146, 3rd para, 1st line.
[36] p.153, 4th para.
[37] p.153, 6th para.
[38] p.155, topic: "Varargs and Ambiguity".

```java
/* CompleteConcept_Chapter6_1.java

   In this program, we'll learn the following:

   1. How to create objects with no parameterized constructor and two
      parameterized constructors.
   2. How to access methods of a class by using an object of that class.
   3. How to access private data of a class by using a public method of
      that class, hence learning how to protect data from unauthorized
      access.
   4. How to pass an object as a parameter and hence creating a duplicate
      copy of that object.
   5. How to return objects from a method.
   6. Whether we can access static and non-static data and methods from a
      static method [main()].
   7. Whether we can change the value of a final variable, which is
      essentially a constant.

 */


class Box {
    private float height;
    private float width;

    Box() {
        height = 0;
        width = 0;
    }

    Box(float h, float w) {
        height = h;
        width = w;
    }

    Box(Box obj) {
        height = obj.height; //Note why we're allowed to access private data...
        width = obj.width;
    }

    public float getArea() {
        return (float) height * width; //Note the type casting...
    }

    public Box inc(float val) {
        return new Box(height + val, width + val);
        /* The above statement works similar to the following lines:
         * Box temp = new Box(height + val, width + val);
         * return temp;
         */
    }

    public void setDimensions(float h, float w) {
        height = h;
        width = w;
    }

    public float getHeight() {
        return height;
    }

    public float getWidth() {
        return width;
    }
}
```

```
class CompleteConcept_Chapter6_1 {
    static boolean can_Be_Accessed_From_Main = true;
    boolean _can_Be_Accessed_From_Main = false;
    final static boolean CAN_BE_CHANGED = false; //Constant

    static void canBeAccessedFromMain() {
        System.out.println("Static method can be accessed from main.");
    }

    void cannotBeAccessedFromMain() {
        System.out.println("Non-static method cannot be accessed from main.");
    }

    public static void main(String args[]) {
        Box b1 = new Box();                 //Test Box() Constructor
        System.out.println(b1.getArea());

        b1.setDimensions(5.2f, 5.9f);       //Test setDimensions() Method
        System.out.println(b1.getHeight()); //Test getHeight() Method
        System.out.println(b1.getWidth());  //Test getWidth() Method
        System.out.println(b1.getArea());   //Test getArea() Method
        //System.out.println(b1.height);    //Cannot access private data

        Box b2 = new Box(5.2f, 5.9f);       //Test Box(float h, float w)
        System.out.println(b2.getArea());

        Box b3 = new Box(b2);               //Test Box(Box obj) Constructor
        System.out.println(b3.getArea());

        Box b4 = new Box(b3.inc(6.8f));     //Test inc(float val) Method
        System.out.println(b4.getArea());


        //Test static & non-static variable access from main():
        System.out.println(can_Be_Accessed_From_Main);
        //System.out.println(_can_Be_Accessed_From_Main);

        //Test static & non-static method access from main():
        canBeAccessedFromMain();
        //cannotBeAccessedFromMain();

        //Test whether final variable (i.e., a constant) can be changed:
        //CAN_BE_CHANGED = true;
    }
}


/*Program Output:

  0.0           //System.out.println(b1.getArea());
  5.2           //System.out.println(b1.getHeight());
  5.9           //System.out.println(b1.getWidth());
  30.68         //System.out.println(b1.getArea());
  30.68         //System.out.println(b2.getArea());
  30.68         //System.out.println(b3.getArea());
  152.40001     //System.out.println(b4.getArea());
  true          //System.out.println(can_Be_Accessed_From_Main);
  Static method can be accessed from main.     //canBeAccessedFromMain();

*/
```

Complete Concepts Program – Inner Classes

```
class Outer {
    public int outer_pub;
    private int outer_pri;
    protected int outer_pro;
    int outer_def; //Default access
    int general_var;

    void outerMethod() {
        //Can't access inner fields or methods without objects
        Inner in = new Inner();
        //Can access inner fields or methods whatever
        //access specifiers they may have
        in.inner_pub = 5;
        in.inner_pri = 5;
        in.inner_pro = 5;
        in.inner_def = 5;
        in.general_var = 5;
        in.innerMethod();
    }

    class Inner {
        public int inner_pub;
        private int inner_pri;
        protected int inner_pro;
        int inner_def; //Default access
        int general_var; //Hides the general_var in Outer class

        void innerMethod() {
            //Can access outer variables whatever
            //access specifiers they may have
            outer_pub = 5;
            outer_pri = 5;
            outer_pro = 5;
            outer_def = 5;
            general_var = 5; //This is the field in Inner class
            outerMethod(); //Can call outer method
        }
    }

}

public class CompleteConcept_Chapter6_2 {
    public static void main(String[] args) {
        //Instantiating an inner class object
        Outer a = new Outer();
        Outer.Inner in = a.new Inner();

        //Can access all the fields and methods of inner except private
        in.innerMethod();
        in.inner_pub = 5;
        in.inner_pro = 5;
        in.inner_def = 5;
        in.general_var = 5;
    }
}
```

Exercises

✩✩✩ 6.1

The following complete program prints four lines when executed. Show the four lines that are printed in the order in which they are printed. ***[2006. Marks: 3]***

```
public class ArrayTest {
    public static void main(String[] args) {
        int[] test = new int[2];
        test[0] = test[1] = 5;
        System.out.println(test[0] + "," + test[1]);
        fiddle(test, test[1]);
        System.out.println(test[0] + "," + test[1]);
    }
    static void fiddle(int[] test, int element) {
        test[0] = 10;
        test[1] = 11;
        element = 12;
        System.out.println(test[0] + "," + test[1] + "," + element);
        test = new int[2];
        test[0] = 20;
        test[1] = 21;
        System.out.println(test[0] + "," + test[1]);
    }
}
```

Solution:

```
5,5
10,11,12
20,21
20,21
```

Explanation:

See point 6.6

☆☆☆☆☆ 6.2

Design a class named *Student* that has two private data – student id and score. The class should contain a parameterized constructor to initialize its data member and one method to display the information. Now write a Java program that will use an array of Student objects to represent information about 3 students. Your program should take input from the keyboard and display the information of the 3 students. *[Incourse-1, 2007. Marks: 3+3=6]*

Solution:

```
import java.util.Scanner;

class Student {
    private int student_ID;
    private int score;

    Student(int std_ID, int s) {
        student_ID = std_ID;
        score = s;
    }

    void display() {
        System.out.println("ID: " + student_ID + ", score: " + score);
    }
}

public class Main {
    public static void main(String[] args) {
        Student students[] = new Student[3];

        //Input Student information
        Scanner in = new Scanner(System.in);
        int stdID, stdScore;
        for (int i = 0; i < 3; i++) {
            System.out.print("Enter student ID: ");
            stdID = in.nextInt();
            System.out.print("Enter score: ");
            stdScore = in.nextInt();
            students[i] = new Student(stdID, stdScore);
        }
        //Display student information
        for (int i = 0; i < 3; i++) {
            students[i].display();
        }
    }
}
```

✮✮✮ 6.3 **Identify errors in the following program and state the reasons: *[Incourse-1, 2007. Marks: 5]***

```
 1 class QW1 {
 2     private int a;
 3     private int b;
 4     public QW1(int i, int j) {a = i; b = j;}
 5     public QW1(int i) {a = i; b = i;}
 6     public void show() {
 7         System.out.println(a);
 8         System.out.println(b);
 9     }
10
11     public static void main() {
12         final int ARRAY_SIZE;
13         int a[] = new int[ARRAY_SIZE];
14         int b[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
15
16         for (int i = 0; x <= b.length; i++) {
17             System.out.println(b[i]);
18         }
19         b += 10;
20         show();
21     }
22 }
```

Solution:

- **Line 11:**
  `main()` method has no parameters.
  OR,
  At runtime, the following exception will be generated:
  `Exception in thread "main" java.lang.NoSuchMethodError: main`
- **Line 13:**
  Variable `ARRAY_SIZE` might not have been initialized.
- **Line 16:**
  Cannot find symbol. Symbol: `x`, Class: `QW1`.
- **Line 17:**
  At runtime, the following exception will be generated:
  `Exception in thread "main"java.lang.ArrayIndexOutOfBoundsException:10`
- **Line 19:**
  Operator + cannot be applied to `int[]`, `int`
  OR,
  Operator + cannot be applied to array
- **Line 20:**
  Non-static method `show()` cannot be referenced from a static context.

✮ 6.4 **Identify errors and state the reasons: *[Incourse-3, 2007. Marks: 5]***

```
 1 class test {
 2     int x = 25;
 3     private int y = 45;
 4
 5     class in {
 6         private int p = 90;
 7         void set_p() {
 8             p = 25;
 9             set_val(100, 200);
10         }
11     }
12
13     private void set_val(int a, int b) {
14         x = a;
15         y = b;
16         p = 34;
17     }
18
19     void temp() {
20         in t = new in();
21         t.set_p();
22         t.p = 23;
23     }
24 }
25
26 class test_in {
27     public static void main(String args[]) {
28         test ob = new test();
29         temp();
30         in ob1 = new in();
31         in.set_p();
32     }
33 }
```

Solution:

Error 1: Line 16: Cannot find symbol. Symbol: variable `p`. Location: class `test`.

Error 2: Line 29: Cannot find symbol. Symbol: method `temp()`. Location: class `test_in`.

Error 3: Line 30: Class `in` is an inner class. Cannot be instantiated outside the outer class `test`.

OR, Cannot find symbol. Symbol: class `in`. Location: class `test_in`.

6.5 **Write a class definition for the following scenario. You have to write a class called *MyFraction* which works on fractions of the form *a/b* where *a* represents the numerator and *b* represents the denominator. Both *a* and *b* are integers (i.e. if a = 1 and b = 2, then the fraction will be 1/2). Your class should perform the following operations:**

- **i.** **Create two constructors – (1) with no parameters that set *a = 0* and *b = 1*; and (2) with 2 integer parameters that sets both *a* and *b*.**
- **ii.** **Write an instance method for *addition* which returns the resultant object. [int: a/b + c/d = (ad + bc)/bd]**
- **iii.** **Write a static method for *multiplication* which returns the resultant object. [Hint a/b * c/d = ac/bd]**
- **iv.** **Write a static *main()* that allocates two Fractions, 1/2 and 3/4 and stores their sum in a third variable. *[2005. Marks: 5]***

Solution:

```
class MyFraction {
    int a, b;

    MyFraction() {
        a = 0;
        b = 1;
    }

    MyFraction(int x, int y) {
        a = x;
        b = y;
    }

    MyFraction addition(MyFraction x) {
        return new MyFraction((a * x.b + b * x.a), (b * x.b));
    }

    static MyFraction multiplication(MyFraction x, MyFraction y) {
        return new MyFraction((x.a * y.a), (x.b * y.b));
```

```java
    }
}

public class E6_5 {
    public static void main(String[] args) {
        MyFraction f1 = new MyFraction(1, 2);
        MyFraction f2 = new MyFraction(3, 4);
        MyFraction f3 = f2.addition(f1);
    }
}
```

Chapter 8
Inheritance, Abstract Classes and Interface

Theories

☆ 8.1

What is *Inheritance*?

Inheritance is a mechanism which allows a class (called the *base* or *superclass*) to be extended to make a new class (called the *derived class* or *subclass*), while maintaining the integrity of the base class.

☆☆☆ 8.2

Why should you use the keyword `super` in your java program? Explain with example. *[Incourse-2, 2007. Marks: 3]*

There are two cases where we should use the keyword `super` in a java program:

1. To call the constructor of the superclass.
2. To access the non-private fields and methods of the superclass.

For example, consider the following code segments:

```
class A {
    int a;

    A(int x) {
        a = x;
    }

    void display() {
        System.out.println(a);
    }
}

class B extends A{
    int b;

    B(int x, int y) {
        super(x);
        b = y;
    }

    void display() {
        super.display();
        System.out.println(b);
    }
}
```

In the above example, insides `class B`, `super(x)` is used to call `class A`'s constructor, and `super.display()` is used to access `class A`'s `display()` method.

☆☆☆ 8.3

What is the difference between *method overriding* and *method overloading*? Explain with example. *[2003, 2005. Marks: 4/3]*

In a class hierarchy, when a method in a subclass has the same name and type signature as a method in its superclass, then the method in the subclass is said to *override* the method in the superclass, and this process is called *method overriding*.

On the other hand, when multiple methods have the same name, but different type signatures, then the methods are said to be overloaded, and this process is called *method overloading*.

Here is an example:

```
class A {
    int a;

    void display() {
        System.out.println(a);
    }
}

class B extends A{
    int b;

    void setVal(int x) {
        b = x;
    }

    void setVal(int x, int y) {
        super.a = x;
        b = y;
    }

    void display() {
        System.out.println(b);
    }
}
```

[39]In the above example, the `display()` method in `class B` has the same name and type signature as the `display()` method in `class A`. So, the `display()` method in `class B` *overrides* the `display()` method of `class B`. On the contrary, the `setVal()` methods in `class B` have the same name, but different signatures. So, they are *overloaded*.

Again, overloading defines a similar operation in different ways for different *data*, whereas overriding defines a similar operation in different ways for different *object types*.[40]

☆ 8.4 **What is the advantage of method overriding?**

Method overriding allows Java to support run-tme polymorphism. This run-time polymorphism allows a general class to specify methods that will be common to all of its derivatives, while allowing subclasses to define the specific implementation of some or all of those methods. Overridden methods are another way that Java implements the "one interface, multiple methods" aspect of polymorphism.

☆☆ 8.5 **"A superclass variable can refer a subclass object", do you agree? Justify your answer.** ***[2003. Marks: 3]***

Yes, a superclass variable can refer a subclass object. The following program demonstrates this:

```
class A {
    int x;

    void displayA() {
        System.out.println(x);
    }
}

class B extends A{
    int y;

    void displayB() {
        System.out.println(y);
```

[39] Note that I haven't declared any constructors in the code segment. In the exam, you need to save as much time as possible. Declaring constructors is irrelevant to the question, and the program is perfectly valid without constructors. So, you must omit irrelevant codes to save time at exam, provided that the irrelevant code is not vital for the program to be compiled successfully.

[40] Include this difference only if the question has 4 marks, *or*, if the question doesn't want any example.

```
            }
        }

        public class TestClass{
            public static void main(String[] args){
               B b = new B();
               A a = b;
               a.displayA();
            }
        }
```

In the above program, inside the main method, ‘a’ is a superclass reference which refers to its subclass object ‘b’. However, only those members of `class B` can be accessed through ‘a’ that are defined by `class A`. So, while we can access the field ‘x’ and the method `displayA()`, we cannot access the field ‘y’ and the method `displayB()` through the superclass reference variable ‘a’.

8.6 **Discuss why casting a superclass reference to a subclass reference is potentially dangerous. *[2005. Marks: 3]***

A superclass reference knows nothing about what members the subclass has added. But the subclass knows what it inherited as well as what it added. So, when a subclass will refer to a superclass reference (through casting), it will know all the added members of its own class as well as those of the superclass. Therefore, it is likely that the subclass reference may try to access its own added members. And when it tries so, it will not be able to find them in the object referred to by the superclass. Thus, an error will occur. Hence, casting a superclass reference to a subclass reference is potentially dangerous.

☆ 8.7 **Why should you use abstract class in your program? *[Incourse-1, 2007. Marks: 2]***

Sometimes there may be a need to create a superclass that only defines a generalized form that will be shared by all of its subclasses, leaving it to each subclass to fill in the details. Such a class determines the nature of the methods that the subclasses must implement. Abstract class provides a way to solve this type of situation.

8.8 **What is interface? What are the possible contents of an interface? Explain. *[2003. Marks: 1+2]***

An interface defines a protocol of behavior that can be implemented by any class.

The possible contents of an interface are:

1. Method declarations without bodies.
2. Initialized variables which are implicitly static and final.

8.9 **State the advantages of using interface. *[Incourse-2, 2007. Marks: 2]***

1. Capturing similarities among unrelated classes without artificially forcing a class relationship.
2. Revealing an object's programming interface without revealing its class.
3. Helps achieving multiple inheritance.

☆ 8.10 **What are the differences between *abstract class* and *interface*?**

Abstract Class	Interface
A class may extend only one abstract class.	A class may implement several interfaces.
Can extend another abstract or non-abstract class.	Can extend only another interface.
Can have methods with bodies defined as well as methods without bodies.	Cannot have any method with a defined body.
Can contain both instance variables and constants.	Can contain only static final variables (i.e., constants).

✯ 8.11 **What are the differences between a *class* and an *interface*? *[2002. Marks: 3; 2005. Marks: 2]***

Class	Interface
Defines what states[41] and behaviors an *object* can have.	Defines what methods a *class* can have.
Can extend another class or implement interfaces.	Can extend only another interface.
Can have methods with bodies defined as well as methods without bodies (i.e., abstract methods).	Cannot have any method with a defined body.
Can contain both instance variables and constants.	Can have only static final variables(i.e., constants).
Objects of a class can be instantiated.	Objects of an interface cannot be instantiated.

✯ 8.12 **Is it possible to partially implement an interface? Justify your answer with example. *[2003. Marks: 3]***

Yes, it is possible to partially implement an interface by declaring the implementing class as abstract. For example:

```
interface Characteristics {
    boolean hasArea();
    boolean hasVolume();
    boolean is2D();
    boolean is3D();
}

abstract class Figure implements Characteristics{
    int dim1, dim2;
    public[42] boolean is2D() {
        return true;
    }
}
```

In the above example, class Figure partially implemented the interface Characteristics by implementing only one method from it. However, class Figure must be declared as abstract and any class that will inherit Figure must implement the remaining methods declared in the interface 'Characteristics'.

✯✯✯ 8.13 **How multiple inheritance is achieved in Java? *[2005. Marks: 1]***

OR, Write a Java program to implement multiple inheritance. *[Incourse-2, 2007. Marks: 4]*

Multiple inheritance can be achiedved in Java by allowing a class to inherit from one other class and an unlimited number of interfaces. Below is a program demonstrating multiple inheritance, which inherits a class and an interface:

```
interface Engine{
    void setHorsePower(int hrsPwr);
    int getHorsePower();
    void setMaker(String mk);
    String getMaker();
}
```

[41] i.e., attributes or properties.
[42] Don't forget to include the modifier 'public', ignoring it would result in a compile-error. See point 8.15.

```java
class Body{
     private String color;

    void setColor(String clr) {
        color = clr;
     }

    String getColor() {
        return color;
     }
}

class Car extends Body implements Engine {
    int hp;
    String maker;

    Car(int hrsPwr, String clr, String mkr) {
        hp = hrsPwr;
        setColor(clr);
        maker = mkr;
    }

    public void setHorsePower(int hrsPwr){
        hp = hrsPwr;
    }
    public int getHorsePower(){
        return hp;
    }
    public void setMaker(String mk){
        maker = mk;
    }
    public String getMaker(){
        return maker;
    }
}

public class Multi_Inherit {
    public static void main(String[] args) {
        Car toyota_Corolla = new Car(100, "White", "Toyota");
        Car pajero = new Car(800, "Black", "Mitsubishi");
    }
}
```

☆ 8.14 **X is a subclass of Y. Does the last two assignments below produce a compile-time error? *[2006. Marks: 2]***

```java
X x = new X();
Y y = new Y();
y = x;
x = y;
```

The assignment y = x does not produce any compile-time error as subclass object can be assigned to superclass reference. But the assignment x = y produces a compile-time error.

8.15 **What is *final variable* and *final method*? Write down the reasons to use these. *[2006. Marks: 3]***

A final variable is a variable whose value cannot be changed. It is used as a constant.

A final method is a method which cannot be overridden. It is used to prevent overriding of a method.

8.16 **What is *early binding, late binding* and *dynamic binding* (or *dynamic method dispatch*)?**

Normally, Java resolves calls to methods dynamically, at run time. This is called late binding. However, since final methods cannot be overridden, a call to one can be resolved at

		compile time. This is called early binding. Dynamic binding or dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time.

Points to be Remembered – Inheritance

	8.1	কোন class কেবল **একটি** মাত্র class-কেই inherit করতে পারবে।[43]
	8.2	Constructor call হবে সর্বোচ্চ superclass থেকে সর্বনিম্ন subclass পর্যন্ত।[44]
	8.3	যদি Superclass-এর constructor না থাকে অথবা constructor parameter**less** হয়, তাহলে Subclass-এর constructor-এ `super()` call না করলেও চলবে। সেক্ষেত্রে Java নিজেই সেটা call করবে। কিন্তু যদি superclass-এর constructor parameterized হয়, তাহলে Java নিজে নিজে সেটা call করবে না এবং **অবশ্যই** তা ম্যানুয়ালি `super(`*`parameter-list`*`)` দিয়ে call করতে হবে।[45]
	8.4	Subclass-এর constructor-এ `super()` call করলে তা অবশ্যই constructor-এর **প্রথম** statement হতে হবে।[46]
	8.5	Subclass থেকে `super` কী-ওয়ার্ড দ্বারা **ঠিক উপরের** superclass-এর method বা variable access করা যাবে।[47]
	8.6	Superclass reference variable দ্বারা subclass object-কে refer করা যায়। তবে সেক্ষেত্রে ঐ variable দ্বারা subclass-এর সে সকল variable বা method-ই access করা যাবে যেগুলো superclass-এ define করা আছে। এর বিপরীতটি সম্ভব নয়, অর্থাৎ subclass reference variable দ্বারা superclass-এর object-কে refer করা যায় না।[48]
	8.7	Overridden method call-এর ক্ষেত্রে, কোন superclass variable যখন subclass-এর কোন object-কে point করবে, তখন **subclass**-এর overridden method-টাই call হবে।[49]
	8.8	Overridden method-এর access modifier অবশ্যই superclass-এর method-এর access privilege-এর তুলনায় সমান বা বেশি privilege সম্পন্ন হতে হবে। অর্থাৎ, superclass-এর method package default হলে subclass-এর method package default বা public হতে হবে, private হলে হবে না। একইভাবে, superclass-এর method public হলে subclass-এর method public হতে হবে, package default বা private হলে হবে না।

Points to be Remembered – Abstract Classes

	8.9	কোন class-এ ন্যূনতম একটি abstract method থাকলেই ঐ class-কে abstract হিসেবে declare করতে হবে।[50]
	8.10	Abstract class-এর কোন object declare করা যায় না, তবে সেটার reference variable declare করা যায়।[51]
	8.11	Abstract class-এ abstract constructor বা abstract static method declare করা যায় না।[52]
	8.12	Abstract class-এর subclass-এ ঐ Abstract class-এর **সকল abstract** method override করতে

[43] p.159, 1st para.
[44] p.170, topic: When Constructors are Called, 1st para, 3rd line.
[45] P.170, 2nd para, 3rd line.
[46] p.163, topic: Using super to Call Superclass Constructors, 3rd line.
[47] p.163, 2nd para, 1st line.
[48] p.162.
[49] p.174, the program.
[50] p.178, 5th line.
[51] p.178, 7th line; last para, 1st line.
[52] p.178, 9th line.

		হবে।[53] অবশ্য subclass-টি যদি নিজেই আবার abstract হয়ে থাকে, তাহলে superclass-এর সকল abstract method override না করলেও চলবে।[54]
	8.13	কোন class-কে abstract এবং final উভয় হিসেবে কখনোই declare করা যায় না।[55]

Points to be Remembered – Interfaces

	8.14	Variables can be declared inside of interface declarations. They are *implicitly* **static** and **final**, i.e., they cannot be changed by the implementing class. They **must** also be initialized.[56]
	8.15	All methods and variables in an interface are *implicitly* **public**. So, when they are implemented, they **must** have 'public' as their access modifier.[57]
	8.16	An interface can extend only interfaces; it cannot extend any normal or abstract class.
	8.17	Reference variables of an interface can be declared, and that variable works just like the reference variable of an abstract class *(See point 8.6).* And no objects of an interface can be instantiated, since interfaces are not classes.[58]
	8.18	If a class includes an interface but does not fully implement the methods defined by that interface, then that class **must** be declared as *abstract.*[59]

Exercises

☆☆☆	8.1	**Generate the output of the following program: *[Incourse-1, 2007. Marks: 6]***

```
class Add {
    protected int i;
    Add(int a) {i = a;}
    protected void addIt(int amount) {i += amount;}
    protected int getIt() {return i;}
}

class DAdd extends Add {
    private int i;
    DAdd(int a, int b) {
        super(a);
        i = b;
    }
    protected void addIt(int amount) {i = i + super.i + amount;}
    protected int getIt() {return i;}
    protected void doubleIt(int amount) {addIt(2 * amount);}
}

public class TestAdder {
    public static void main(String args[]) {
        Add A = new Add(3);
        DAdd DA = new DAdd(1, 5);
        A.addIt(20);
        System.out.println(A.getIt());
        A = DA;
        A.addIt(20);
        System.out.println(A.getIt());
```

[53] Note that all ***abstract*** methods must be overridden, not ***all*** of the methods (i.e., both abstract and non-abstract).
[54] p.178, 10th line.
[55] p.181, 1st para, 3rd line.
[56] p.193, the para before last para.
[57] p.193, last line of the para before last para; p.194, 2nd para, 3rd line.
[58] p.195, 1st para, lines 1-4.
[59] p. 196, topic: Partial Implementations.

```
28          DA.doubleIt(20);
29          System.out.println(A.getIt());
30      }
31  }
```

Solution:

23
26
67

Explanation:

1. Line 23 invokes line 4, and line 24 invokes line 5.
2. Line 26 invokes line 14, and line 27 invokes line 15. *See point 8.7*
3. Line 28 invokes line 16, which in turn invokes line 14; and line 29 invokes line 15.[60]

☆☆☆ 8.2

Identify errors in the following program and state the reasons. *[Incourse-2, 2007. Marks: 7]*

```
 1  import java.io.*;
 2
 3  class A {
 4      int p = 120;
 5      public void print() { System.out.println("p:" + p); }
 6  }
 7
 8  abstract class B extends A {
 9      int d = 525;
10      public void print() { System.out.println("d:" + d); }
11      public void print(int k) { System.out.println("In B"); }
12      abstract int add();
13  }
14
15  class C extends B {
16      int m = 424;
17      public void print() { System.out.println("m:" + m); }
18  }
19
20  class test {
21      public static void main(String[] args) {
22          A a = new A();
23          B b = new B();
24          C c = new C();
25
26          c = a;
27          c.print();
28
29          a = b;
30          a.d = 230;
31          a.print();
32          a.print(120);
33
34          b = c;
35          b.add();
36      }
37  }
```

Solution:

Error 1: Class C is not abstract and does not override the abstract method `add()` in class B. *See point 8.12*

Error 2: Line 23: Class C is abstract, cannot be instantiated. *See point 8.10*

[60] Note that in line 29, **A**`.getIt()` is called, not **DA**`.getIt()`. But the effect of both is the same according to point 8.7.

Error 3: Line 26: Superclass object cannot be assigned to subclass reference. *See point 8.6*

Error 4: Line 30: Cannot find symbol (variable d) in Class A, **OR**, Superclass reference ‘`a`’ does not know anything about subclass field ‘`d`’ which is not in class A. *See point 8.6*

Error 5: Line 32: Cannot find method `print(int)` in Class A, **OR**, Superclass reference ‘`a`’ does not know anything about subclass method `print(int)`which is not in class A. *See point 8.6*

Error 6: Line 35: Method `add()` is abstract in `class B` and is not overridden in `class C`. *See point 8.6 & 8.7*

☆ 8.3

What would be the output of the following statements? ***[2003. Marks: 3]***

```
 1 class A {
 2     A() { System.out.println("Inside A"); }
 3 }
 4 class B extends A {
 5     B() { System.out.println("Inside B"); }
 6 }
 7 class C extends A {
 8     C() { System.out.println("Inside C"); }
 9 }
10 class Example {
11     public static void main(String[] args) {
12         C obj = new C();
13     }
14 }
```

Solution:

```
Inside A
Inside C
```

Explanation:

See point 8.2. Note that on line 7, class C extends class **A**, not class **B**. Don’t forget to thoroughly check the program at exam, lest you may be fooled!

☆☆ 8.4

Consider the following code segment:

```
 1 interface Face1 {
 2     public void bar();
 3 }
 4
 5 class Class1 {
 6     private int size = 0;
 7     private String name;
 8
 9     public Class1(String name) { this.name = name; }
10     public String getName() { return (name); }
11     public void foo() { System.out.println("Class1.foo()"); }
12 }
13
14 class Class2 extends Class1 implements Face1 {
15     int size = 0;
16     int x;
17
18     public Class2(String name) { super(name); }
19     public void foo() { System.out.println("Class2.foo()"); }
20     public void bar() { System.out.println("Class1.bar()"); }
21 }
```

What will be the output after the following code is executed? ***[2005. Marks: 4]***

```
Face1 c = new Class2("ME");
```

```
c.bar();
```

Solution:

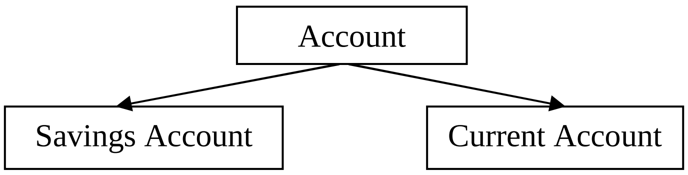
`Class1.bar()`

Explanation:

See point 8.17

✮✮✮ 8.5

You have to design a class hierarchy as shown below:



The parent class contains the general information about an account and an abstract method to calculate the yearly interest. For savings account, the interest rate is 10% and for current account the interest rate is 6%. All the data members of the Account class are initialized through a parameterized constructor. Your program should be able to deposit and withdraw money from a saving account. Perform the same operation on a current account. [Incourse-1, 2007. Marks: 4 + 3 = 7]

Solution:

```
abstract class Account {
    int acc_No;
    float balance;

    Account(int acc, float initial_Balance) {
        acc_No = acc;
        balance = initial_Balance;
    }

    void deposit(float amount) {
        balance += amount;
    }

    void withdraw(float amount) {
        balance -= amount;
    }

    abstract void calc_Interest();
}

class Sav_Acc extends Account {
    Sav_Acc(int acc_No, float initial_Balance) {
        super(acc_No, initial_Balance);
    }

    void calc_Interest() {
        balance += (balance * 10) / 100;
    }
}

class Curr_Acc extends Account {
    Curr_Acc(int acc_No, float initial_Balance) {
        super(acc_No, initial_Balance);
    }

    void calc_Interest() {
        balance += (balance * 6) / 100;
    }
}
```

```java
public class Acc_Test {
    public static void main(String[] args) {
        Sav_Acc acc1 = new Sav_Acc(1001, 5000.0f);
        acc1.deposit(1000);
        acc1.calc_Interest();
        acc1.withdraw(500.0f);

        Curr_Acc acc2 = new Curr_Acc(1001, 5000.0f);
        acc2.deposit(1000);
        acc2.calc_Interest();
        acc2.withdraw(500.0f);
    }
}
```

Chapter 9
Packages

Theories

✯ 9.1 **What is a package? How can you define your own package? *[2003. Marks: 1+2]***

A package is a namespace[61] that organizes a set of related classes and interfaces.

To define a custom package, a `package` statement has to be included as the *first* statement in a Java source file. More than one file can include the same `package` statement. For example, to define a package named MyPackage, the following line has to be added as the first statement in a Java source file:

```java
package MyPackage;
```

9.2 **How a class or an interface is added to a package? *[2005. Marks: 2]***

To add a class or an interface to a package, the class or interface should be declared in a Java source file and a `package` statement has to be included as the first statement in that file. Also, the file should be kept under a directory named after the package name. For example, to add a class named “MyClass” to a package named “MyPackage”, the following code should be included in a file named MyClass.java, and the file should be kept under a directory named “MyPackage”.[62]

```java
package MyPackage;

class MyClass {
    //Class body goes here.
}
```

9.3 **Show with an example, how you can organize two classes in two different files into a single package. *[2002. Marks: 3]***

```java
package MyPackage;

class Class1 {
    //Class body goes here.
}
```

Class1.java

```java
package MyPackage;

class Class2 {
    //Class body goes here.
}
```

Class2.java

In the above example, `Class1` and `Class2` are two classes in two different files, but they are under the same package named ‘MyPackage’.

9.4 **Discuss the relative merits of using protected access vs. private access in superclass. *[2005. Marks: 3]***

A private member in a superclass cannot be accessed from any of its subclasses or any other classes. On the contrary, a protected member in a superclass can be accessed from only its subclasses, not from any other classes.

Concepts - Rules for Package Access Specifiers

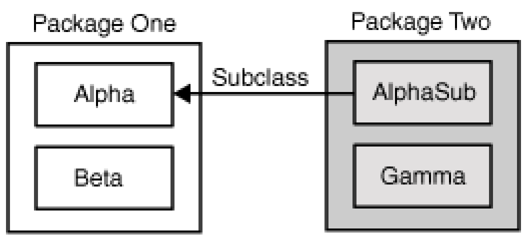
✯✯✯ 9.1

5. Anything declared `public` can be accessed from *anywhere*.
6. Anything declared `private` cannot be seen outside of its *class*.
7. Anything declared as default access (package-private) can be accessed from anywhere

[61]A *namespace* is an abstract container for various items. Each item within a namespace has a unique name, but the namespace allows disambiguation of items with the same name that are in *different* namespaces.
[62] If you don’t have much time at exam, don’t bother including the example.

in the *same* package.

8. Anything declared as `protected` can be accessed from anywhere in the *same package*, plus from only its subclasses *outside* the package.[63]



Visibility

Modifier	Alpha	Beta	Alphasub	Gamma
`public`	Y	Y	Y	Y
`protected`	Y	Y	Y	N
no modifier	Y	Y	N	N
`private`	Y	N	N	N

Notes on the above figures:

1. Alphasub is a subclass of Alpha.
2. The rulings for a subclass in the same package is identical to the rulings for another class in the same package. i.e., If another subclass of Alpha – for example – AlphaSub2 is declared inside Package 1, then its rulings would be the same as those for Beta. So, no subclasses of Alpha was declared in Package One.

☆☆☆ 9.2 A class can have only two access modifiers – *public* and no modifier (i.e., package default). However, only *one* class can be public in a file, and the file name must be after that public class name.

Complete Concepts Program

P1.java

```
package p1;

public class P1 {
    public int p1_pub;
    private int p1_pri;
    protected int p1_pro;
    int p1_def; //Default access
}

class P1_Sub extends P1 {
    P1_Sub() {
        super.p1_pub = 5;
        super.p1_pro = 5;
        super.p1_def = 5;
        //super.p1_pri = 5;
    }
}

class P1_Test {
    P1_Test() {
        P1 p1 = new P1();
        p1.p1_pub = 5;
        p1.p1_pro = 5;
        p1.p1_def = 5;
        //p1.p1_pri = 5;
    }
}
```

P2.java

```
package p2;
import p1.*;

public class P2 {
    P2() {
        P1 p1 = new P1();
        p1.p1_pub = 5;
        //p1.p1_pro = 5;
        //p1.p1_def = 5;
        //p1.p1_pri = 5;
    }
}

class P2_SubP1 extends P1 {
    P2_SubP1() {
        super.p1_pub = 5;
        super.p1_pro = 5;
        //super.p1_def = 5;
        //super.p1_pri = 5;
    }
}
```

[63] p.186, last para, from 2nd line to the last of that para.

Exercises

✮ ✮ ✮ 9.1

Consider the following Java program fragments: ***[Incourse-2, 2007. Marks: 4]***

A.java

```java
package pA;

public class A {
    private int fAOne;
    protected int fATwo;
    int fAThree;
    public void mAOne(){}
}
```

B.java

```java
package pA;

public class B extends A{
    public int fBOne;
    public void mBOne(){}
}
```

C.java

```java
package pA;

public class C {
    public int fCOne;
    public void mCOne(){
        B k = new B();
    }
}
```

D.java

```java
import pA.*;

class D {
    public static void main(String[] args) {
        B b = new B();
        C c = new C();
    }
}
```

i. Which instance variables are accessed using the object **b** inside **main()** method?
ii. Which instance variables are accessed using the object **c** inside **main()** method?
iii. Which instance variables are accessed using the object **k** inside **mCOne()** method?

Solution:

i. fBOne.
ii. fCOne.
iii. fBOne, fATwo, fAThree.

Chapter 10
Exception Handling

Theories

10.1 **What is an *exception*? Why do we need to handle exception? *[2002. Marks: 2]***

An exception is an abnormal condition that arises in a code sequence at run time.

We need to handle exception so that the program does not terminate abruptly[64] and we can display the user a meaningful error message of the situation.

10.2 **What is a *Java exception*?**

A Java exception is an **object** that describes an exceptional condition that has occurred in a piece of code.

☆☆☆ 10.3 **What is `finally` block? Discuss the utility of using the `finally` block. *[2003. Marks: 1+2]***

OR, What is a `finally` block? When and how is it used? Give a suitable example. *[2004. Marks: 3]*

OR, Why should you use `finally` in your program? *[Incourse-2, 2007. Marks: 1]*

A finally block is a block of code that will be executed after a try/catch block has completed and before the code following the try/catch block.

The finally block is useful for closing file handles and freeing up any other resources that might have been allocated at the beginning of a method with the intent of disposing of them before returning.

☆☆☆ 10.4 **Explain `finally` with an example. *[2005. Marks: 4]***

A *finally* block is a block of code that will be executed after a try/catch block has completed and before the code following the try/catch block. The finally block will execute whether or not an exception is thrown. If an exception is thrown, the finally block will execute even if no catch statement matches the exception.

```
import java.io.*;

class Test {
    public static void main(String[] args) {
        BufferedWriter bout = null;
        try {
             bout = new BufferedWriter(new FileWriter("c:/a.txt"));
             bout.write("Hello!");
        } catch (IOException e) {
            System.out.println("Error! " + e);
        } finally {
            try {
                bout.close();
            } catch (IOException ex) {}
        }
    }
}
```

In the above example, if any IOException occurs while executing the code in the try block, then after executing the code in the catch block, the file will be closed (this instruction is given in the finally block). If any other type of exceptions occur which is not handled in the catch block (such as SecurityException), then still the code in the finally block will execute before the program terminates. If all the statements in the try block

[64] *Abruptly* means “*quickly and without warning*”.

executes successfully, then again, the code in the finally block will execute.

10.5 **Can there be a try block without a catch block?**

Yes, there can be a try block without a catch block, provided that there is a finally block immediately following the try block. Below is an example of such case:

```
try {
     //Some code
} finally {
     //Some code
}
```

✯ ✯ ✯ 10.6 **State the tasks of `throw` and `throws`. *[Incourse-3, 2007. Marks: 2]***

‘throw’ is used to throw an exception explicitly.

A ‘throws’ clause lists the types of exceptions that a method might throw.

10.7 **Why should exception handling techniques not be used for conventional program control? *[2004. Marks: 1]***

Exception handling techniques should not be used for conventional program control, because it will only confuse the code and make it hard to maintain.

10.8 **What happens if an exception occurs and an appropriate exception handler cannot be found? *[2004. Marks: 2]***

If an exception occurs and an appropriate exception handler cannot be found, then the exception is handled by the default handler. The default handler displays a string describing the exception, prints a stack trace from the point at which the exceptoin occurred, and terminates the program.

10.9 **What happens when a catch handler throws an exception? *[2004. Marks: 2]***

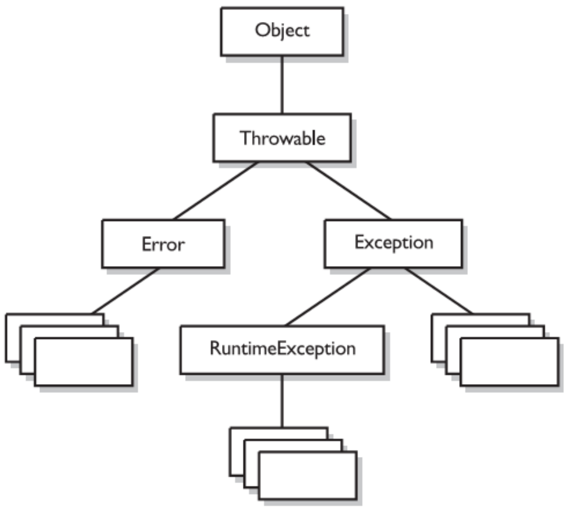
When a catch handler throws an exception, a matching catch handler is searched in the outer try-catch block. If no match is found, then the default handler handles it.

Concepts

10.1 **How an exception is handled[65]:**

1. When an exception is occurred, JVM throws an object of that type of exception on the line where the exception occurred. If that line is inside a try-catch block, then a match for that exception is searched among the catch blocks.
2. If no match is found, it is checked whether this try-catch is a nested try-catch. If it is, then a match for the exception is searched among the parent try-catch block. But before going there, the finally block of the nested try-catch is executed.
3. If no match is found in the parent try-catch block, then the exception is thrown back on the line of the calling method. If that calling line is inside a try-catch block, the catch statements are checked for a match according to rules 1 and 2.
4. If no match is found, again the exception is thrown back on the line of the caller of this method. But before going there, the finally block of the current method is executed. This continues until the main method is reached.
5. When no match for the exception is found in the main method, the exception is thrown to JVM. But before going there, the finally block of the main method is executed. Now, JVM prints the exception and terminates the program.
6. If, in the middle of travelling, a valid return statement is found, then the exception is not thrown to the caller method, rather it is destroyed. Therefore, the current method returns and program execution continues from the next line of the calling method.

[65] p. 211, topic: “Nested try Statements”.

	10.2	**How to create a user-defined exception:** 1. Create a class named after your exception, *extending* the **Exception** class. a. In the class, declare a String field and a constructor taking a String variable as a parameter. b. Override the `toString()` method in that class. 2. Declare a parameterized method where you will check for the occurrence of your exception, and if it occurs, you will throw an object of your exception type. The method must include the `throws` keyword. 3. Now, in the main method, in a `try` block, call the method declared on step 2, passing it the argument which is to be checked for exception. In the `catch` block, catch that exception.
	10.3	**Exception class hierarchy:**  Object Throwable Error Exception RuntimeException

	10.4	**Some important runtime exceptions to be remembered:**
		`ArithmeticException`
		`IndexOutOfBoundsException`
		   `ArrayIndexOutOfBoundsException`
		   `StringIndexOutOfBoundsException`
		`NullPointerException`
		`SecurityException`

Points to be Remembered

	10.1	Any time a method is about to return to the caller from inside a `try-catch` block, whether via an ***uncaught exception*** or an ***explicit return statement***, the `finally` clause is also executed just before the method returns.[66]
	10.2	A method *must* catch or throw all the exceptions that can occur in the containing code – *except* for those of type **Error** or **RuntimeException** or any of their subclasses.[67]
	10.3	Each `try` block requires at least one `catch` block *or* a `finally` block. In other words, either the `catch` clause or the `finally` clause can be omitted, but not both.[68]
	10.4	In multiple catch statements, exception *subclasses* must come **before** any of their *superclasses*.[69]
	10.5	If any *checked* exception[70] is caught in a catch clause which will never occur in any statement inside the try block, then a compile-time error will occur.[71]

[66] p.216, 2nd para, 4th line.
[67] p.214, topic: throws.
[68] p.216, 2nd para, last line.
[69] p.210, 3rd para (which starts with “When you use…”).

Complete Concepts Program – How an Exception is Handled

See the program of Exercise 10.1

Complete Concepts Program – How to Create a User-Defined Exception

```java
/* UserDefinedExceptin.java
 *
 * In this program, some integer numbers are taken from keyboard,
 * and if any negative integer number is found, then a user-defined
 * exception named "NegativeNumberException" is thrown. At the end,
 * the summation of the integers are printed.
 */

import java.util.Scanner;

class NegativeNumberException extends Exception {
    String a;

    NegativeNumberException(String x) {
        a = x;
    }

    public String toString() {
        return "Error! Negative number found: " + a;
    }
}


public class UserDefinedException {

    static int check(int x) throws NegativeNumberException {
        if (x <0) {
            throw new NegativeNumberException(Integer.toString(x));
        } else {
            return x;
        }
    }

    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int sum = 0;
        try {
            while (in.hasNextInt()) {
                sum += check(in.nextInt());
            }
        } catch (NegativeNumberException e) {
            System.out.println(e);
        }
        System.out.println(sum);
    }
}
```

Exercises

☆☆☆	10.1	**Generate the output of the following program:** ***[Incourse-2, 2007 (Modified). Marks: 4]***
		`import java.io.*;`
		
		`class TestException {`

---

[70] *Checked exception* means an exception which is checked at compile-time. Exceptions of type **Error** or **RuntimeException** or any of their subclasses are called *unchecked* exceptions. All other exceptions are *checked* exceptions.
[71] *See exercise 10.4 for an application of this point.*

```
 4
 5  public static void main(String[] args) throws IllegalAccessException{
 6          TestException ob = new TestException();
 7          try {
 8              System.out.println("return value: " + ob.m());
 9          } catch (ArithmeticException e) {
10              System.out.println("Exception caught in main");
11          } finally {
12              System.out.println("finally--main");
13          }
14          System.out.println("End of main");
15      }
16
17      int m() throws IllegalAccessException {
18          try {
19              return method();
20          } catch (ArithmeticException e) {
21              return 2;
22          } finally {
23              System.out.println("finally--m.");
24          }
25      }
26
27      int method() throws IllegalAccessException {
28          try {
29              int x = 5;
30              if (x == 5) {
31                  throw new IllegalAccessException("test");
32              }
33              return x;
34          } catch (IllegalAccessException e) {
35              try {
36                  throw new IllegalAccessException("test");
37              } catch (ArithmeticException e1) {
38                  return 2;
39              } finally {
40                  System.out.println("nested finally: " + e);
41              }
42          }catch (SecurityException e) {
43              return 9;
44          } finally {
45              System.out.println("finally---method");
46          }
47      }
48  }
```

Solution:

```
nested finally: java.lang.IllegalAccessException: test
finally---method
finally--m.
finally--main
Exception in thread "main" java.lang.IllegalAccessException: test
        at TestException.method(TestException.java:36)
        at TestException.m(TestException.java:19)
        at TestException.main(TestException.java:8)
```
[72]

Explanation:

See point 10.1& concept 10.1

✩ ✩ ✩	10.2	**Write a Java code segment that will take a sequence of positive integer numbers as input from the keyboard and find the summation of the odd numbers only. If the input**

[72] Note: In exam, the last output will never be asked to write. I just put it here as an example of concept 10.1.

is a negative number, your code segment should throw a user-defined exception. The main() method should handle this exception and print the error message.[73] [Incourse-2, 2007. Marks: 5]

Solution:

```java
public class UserDefinedException {

    static int check(int x) throws NegativeNumberException {
        if (x <0) {
            throw new NegativeNumberException(x);
        } else {
            return x;
        }
    }

    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int sum = 0, num;
        try {
            while (in.hasNextInt()) {
                num = check(in.nextInt());
                if ((num % 2) != 0) {
                    sum += num;
                }
            }
        } catch (NegativeNumberException e) {
            System.out.println(e);
        }
        System.out.println(sum);
    }
}
```

☆☆☆ 10.3

Determine errors in the following program. Correct them and generate the output.

```java
class TestException {
    public static void main(String args[]) {
        try {
            method();
            System.out.println("After method()");
        }
        catch(RuntimeException ex) {
            System.out.println("Exception in main");
        }
        System.out.println("End of main");
    }

    static void method() throws Exception {
        try {
            final int zero=0;
            int y=2/zero;
            System.out.println("Recovered from error");}
        catch(RuntimeException ex) {
            System.out.println("Runtime Exception in method");
            throw ex;
        } finally {
            System.out.println("Finally in method");
        }
        System.out.println("End of method");
        }
}
```

Solution:

[73] Note that the question asked to write a *segment* of code, not the full program. So, we don't need to define the user-defined exception. Still, it's better to ask the teacher in charge at the exam hall whether we should design the exception or not.

Error: Line 4: Unreported exception java.lang.Exception, must be caught or declared to be thrown. *See point 10.2*

Correction: Line 2 should be:

```
        public static void main(String args[]) throws Exception {
```

Output:

```
Runtime Exception in method
Finally in method
Exception in main
End of main
```

Explanation:

See point 10.1& concept 10.1

10.4 **Consider the following two programs:**

1.

```
 1 class throwdemo {
 2     static void procdemo() {
 3         try {
 4             throw new IllegalAccessException();
 5         } catch(ArithmeticException e) {
 6             System.out.println("Arithmetic Exception");
 7         }
 8     }
 9
10     public static void main(String args[]) {
11         try {
12             procdemo();
13         } catch(ArithmeticException e) {
14             System.out.println("Caught in main:"
15                     + "Arithmetic Exception");
16         }
17     }
18 }
```

2.

```
 1 class throwdemo {
 2     static void procdemo() {
 3         try {
 4             throw new ArithmeticException();
 5         } catch(IllegalAccessException e) {
 6             System.out.println("IllegalAccessException");
 7         }
 8     }
 9
10     public static void main(String args[]) {
11         try {
12             procdemo();
13         } catch(IllegalAccessException e) {
14             System.out.println("Caught in main:"
15                     + "IllegalAccessException");
16         }
17     }
18 }
```

Correct any errors the programs may contain and generate the output of both programs. Explain why the output of the two programs are different.[74]

Solution:

Errors in program 1:

[74] There is almost 0% possibility of this problem to appear at exam. This problem is mainly for making the concept clear.

Error 1 (Line 4): Unreported exception java.lang.IllegalAccessException; must be caught or declared to be thrown.

Correction: Line 2 should be:

```
        static void procdemo() throws IllegalAccessException {
```

Error 2 (Line 12): Unreported exception java.lang.IllegalAccessException; must be caught or declared to be thrown.

Correction: Line 10 should be:

```
public static void main(String args[]) throws IllegalAccessException {
```

Output of program 1:

```
Exception in thread "main" java.lang.IllegalAccessException
        at throwdemo.procdemo(E10_4_1.java:4)
        at throwdemo.main(E10_4_1.java:12)
```

Errors in program 2:

Error 1 (Line 5): Exception java.lang.IllegalAccessException is never thrown in body of corresponding try statement.

Correction: Line 4 should be:

```
                throw new IllegalAccessException();
```

Error 2 (Line 13): Exception java.lang.IllegalAccessException is never thrown in body of corresponding try statement.

Correction: Line 2 should be:

```
        static void procdemo() throws IllegalAccessException {
```

Output of program 2:

```
IllegalAccessException
```

Explanation of the reason for the difference in the output of the two programs:

In the first program, the IllegalAccessException was not caught in any of the catch statements, hence an error message was printed and the program terminated abruptly.

In the second program, the IllegalAccessException was caught inside the `procdemo()` method and the statement inside the catch clause was executed.

10.5 **Write a program that will read an integer number and a file name as input from the keyboard. Your program should add the number at the end of the file. The input should first take an integer number and then a file name. For example:**

11 c:\tem12.txt

Your program should generate a user-defined exception invalidInput if the input is given in wrong order. For example:

c:\temp12.txt 23

Your program should check whether the filename is valid. If invalid then generate a user-defined exception invalidFileName (Use exists() method of File class). *[Incourse-3, 2007. Marks: 8]*

Solution:

See exercise 19.3

10.6

Consider the BankAccount class has three methods with preconditions:

1. **The constructor (initial balance must not be negative)**
2. **The withdraw method (withdrawal amount must not be less than the balance)**
3. **The deposit method (deposit amount must not be negative)**

Write code for the BankAccount class so that each of the three methods throws an IllegalArgumentException if the precondition is violated. ***[2005. Marks: 6]***

Solution:

```
class BankAccount {
    int balance;

    BankAccount(int initialBalance) {
        if (initialBalance < 0) {
            throw new IllegalArgumentException(
                    "Initial balance must not be negative.");
        } else {
            balance = initialBalance;
        }
    }

    void withdraw(int amount) {
        if (amount < balance) {
            throw new IllegalArgumentException(
                 "Withdrawal amount must not be less than the balance.");
        } else {
            balance -= amount;
        }
    }

    void deposit(int amount) {
        if (amount < 0) {
            throw new IllegalArgumentException(
                    "Deposit amount must not be negative.");
        } else {
            balance += amount;
        }
    }
}
```

Chapter 11
Multithreaded Programming

Theories & Concepts

	11.1	**What is *multithreading*? What is *multithreaded programming*? *[2005 (Only the second question. Marks: 1]*** Simultaneously running two or more parts of the same program is called multithreading. Developing a program which can execute multiple tasks simultaneously is called multithreaded programming.
	11.2	**What are the advantages of multithread? *[2002. Marks: 2]*** **OR, What are the reasons for using multithreading? *[2004. Marks: 2]*** Multithreading enables one to write very efficient programs that make maximum use of the CPU, because idle time can be kept to a minimum.

☆☆☆	11.3	**How can you create a thread in your Java program? *[2003. Marks: 3]***
		
		 OR, Write down the ways by which Java can create multiple threads. *[2005. Marks: 3]*
		
		 In a Java program, threads can be created in two ways:
		
		 1. By implementing the **Runnable** interface. For example:
		
		 `class MultiThread implements Runnable {`
		 `    Thread t;`
		
		 `    MultiThread() {`
		 `        t = new Thread(this);`
		 `        t.start();`
		 `    }`
		
		 `    public void run() {`
		 `        //Some code here`
		 `    }`
		 `}`
		
		 2. By extending the **Thread** class. For example:
		
		 `class MultiThread extends Thread {`
		 `    Thread t;`
		
		 `    MultiThread() {`
		 `        start();`
		 `    }`
		
		 `    public void run() {`
		 `        //Some code here`
		 `    }`
		 `}`

☆☆☆	11.4	**Explain the usefulness of `isAlive()` and `join()` functions. *[2003. Marks: 2]*** **OR, Why should you use the `isAlive()` method in your program? *[Incourse-3, 2007. Marks: 2]*** When we need to know whether a thread is still alive or not, we use the `isAlive()` method. This method returns `true` if the thread is alive, and otherwise `false`. The `join()` method waits until the thread on which it is called terminates. Also, an overloaded form of this method allows to specify a maximum amount of time that the

programmer wants to wait for the specified thread to terminate.

11.5 **Describe the complete lifecycle of a thread.** ***[2005. Marks: 3]***

A thread starts when its start() method is called. In return, the start() method calls the run() method, in which the code to be executed in the new thread is present. When the run() method returns, the thread is destroyed.

✯ 11.6 **How do the priorities for threads can be set?** ***[2005. Marks: 2]***

The priorities for threads can be set by calling the setPriority() method and passing it an integer value between 1 and 10. For example:

```
Thread t = new Thread(this);
t.setPriority(5);
```

The above example creates a thread and assigns it a normal priority (which is 5).

11.7 **What are the two methods by which threads can be stopped? Describe these methods.** ***[2005. Marks: 3]***

The two methods by which threads can be stopped are:

1. The `stop()` method:

   This method stops the thread on which it is invoked. This method is deprecated[75] as it might cause serious system failures.

2. The `interrupt()` method:

   This method interrupts the execution flow of the thread on which it is invoked. When the thread interrupts, it may do some other tasks or stop right away. The following example demonstrates this method:

```
class newThread extends Thread {
    newThread() {
        start();
    }

    public void run(){
        try {
            do {
                System.out.println(this.isAlive());
                Thread.sleep(1000);
            } while (true);
        } catch (InterruptedException e) {
            return;
        }
    }
}

class test {
    public static void main(String[] args) {
        newThread a = new newThread();
        a.interrupt();
    }
}
```

✯ 11.8 **What is *synchronization*? When do we use it?** ***[2004. Marks: 3]***

OR, What do you understand by *thread synchronization*? ***[2002. Marks: 1]***

When two or more threads need access to a shared resource, they need some way to ensure that the resource will be used by only one thread at a time. The process by which this is achieved is called synchronization.

An example of a case when we need synchronization is when two threads try to push in

[75] *Deprecated* means *disapproved, rejected*.

and pop out data from a stack simultaneously.

11.9 **Explain with an example what happens when threads are not synchronized.** ***[2002. Marks: 3]***

The following program demonstrates the situation when threads are not synchronized:

```
class Print {
    void print(String msg) {
        System.out.print("[" + msg);
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {}
        System.out.println("]");
    }
}

class CallPrint implements Runnable {
    Print target;
    String msg;

    CallPrint(Print trg, String str) {
        target = trg;
        msg = str;
        Thread t = new Thread(this);
        t.start();
    }

    public void run() {
        target.print(msg);
    }
}

class test {
    public static void main(String[] args) {
        Print cp = new Print();
        new CallPrint(cp, "A");
        new CallPrint(cp, "B");
        new CallPrint(cp, "B");
    }
}
```

The output of the program is as follows:

```
[A[B[C
]
]
]
```

But the output should have been as follows:

```
[A]
[B]
[C]
```

✮ 11.10 **How can threads be synchronized?**

Threads can be synchronized in two ways:

1. By preceding the definition of the method to be synchronized with the keyword `synchronized`[76]. For example:

```
class Sync {
   synchronized void synchedMethod() {
         //Some code here
   }
}
```

[76] This is known as ***method serializing***.

2. By using the **synchronized** statement[77]. For example:

```
Sync a = new Sync();
synchronized (a) {
   a.synchedMethod();
}
```

11.11 **What is serialization? *[2006. Marks: 2]***

Synchronizing a method so that only one thread can access it at a time is called method serialization.

Serialization is achieved by preceding the definition of the method to be synchronized with the keyword `synchronized`. For example:

```
class Sync {
   synchronized void synchedMethod() {
         //Some code here
   }
}
```

Exercises

✮✮ 11.1 **Write a program that will create two threads named *one* and *two* from the main thread. Each of the thread will display the message “Thread *name* Starting”, where *name* is the name of the thread. Each thread will then print a message “Hello from thread *name*” 3 times on the screen. Here, *name* is the name of the child thread. After each write on the screen it will sleep for 500 milliseconds. Main thread should wait for the termination of the child threads. [Incourse-3, 2007. Marks: 8]**

OR, Develop a simple application program to illustrate the use of multithreads. *[2004. Marks: 5]*

Solution:

```
class NewThread implements Runnable {
    Thread t;
    String threadName;

    NewThread(String name) {
        threadName = name;
        System.out.println("Thread " + threadName + " Starting");
        t = new Thread(this, threadName);
        t.start();
    }

    public void run() {
        try {
            for (int i = 0; i < 3; i++) {
                System.out.println("Hello from therad " + threadName);
                Thread.sleep(500);
            }
        } catch (InterruptedException e) {}
    }
}

public class E11_1 {
   public static void main(String[] args) throws InterruptedException{
        NewThread t1 = new NewThread("one");
        NewThread t2 = new NewThread("two");
        t1.t.join();
        t2.t.join();
    }
}
```

[77] Also known as ***synchronized block***.

Explanation & Warnings:

1. The first line of the question was: “Write a program that will create two threads named *one* and *two* from the main thread.” This is applied on line 24 & 25.
2. The second line was: “Each of the thread will display the message “Thread *name* Starting”, where *name* is the name of the thread”. This is applied on line 7.
3. The third line was: “Each thread will then print a message “Hello from thread *name*” 3 times on the screen. Here, *name* is the name of the child thread”. This is applied on lines 14 & 15 by enclosing the print statement by a for-loop.
4. The fourth line was: “After each write on the screen it will sleep for 500 milliseconds”. This is applied on line 16. As the `sleep()` method throws InterruptedException, we enclosed it within a try-block. **Note that we cannot use the *throws* clause on the `run()` method as the `run()` method is overridden and the original method did not include a *throws* clause.**
5. The fifth line was: “Main thread should wait for the termination of the child threads”. This is applied on lines 26 & 27. Note that we used `t1.t.join();`, not `t1.join();`. This is because the method `join()` is defined in the **Thread** class, not in the **NewThread** class. Here, `t1` is an object of **NewThread** class, and `t` is an object of **Thread** class. So, to access the `join()` method, we must access the `t` object first. Also note that the `join()` method throws InterruptedException. We are using the *throws* clause on the `main()` method to handle it.

11.2

Write a program to create two child threads. Now distribute the task of printing numbers from 1 to 50 between these two threads. Child thread 1 will print the odd numbers and child thread 2 will print the even numbers. *[2003. Marks: 5]*

Solution[78]:

```
class Print implements Runnable {
    Thread t;
    int startingNumber;

    Print(String name, int x) {
        startingNumber = x;
        t = new Thread(this, name);
        t.start();
    }

    public void run() {
        for (int i = startingNumber; i < 50; i+=2) {
            System.out.println(i);
        }
    }
}

public class E11_2 {
    public static void main(String[] args) {
        Print t1 = new Print("Child Thread 1", 1);
        Print t2 = new Print("Child Thread 2", 2);
    }
}
```

[78] Note that this solution is not perfect. The program will display all the odd numbers first and then the even numbers. To synchronize printing between the two threads, not only thread synchronization, but also interthread communication is needed. Questions on interthread communication are *most unlikely* to appear at exam (unless the course teacher teaches them well). So, interthread communication as well as thread synchronization has been omitted from this program.

Chapter 19
File I/O and Taking Input From Keyboard

Theories

	19.1	**What is byte stream and charater stream? Suppose in a program byte stream is required for IO operation. Name some classes that can be used for these purposes. *[2006. Marks: 2]*** Byte stream are classes that provide a rich environment for handling byte-oriented I/O. Character stream are classes that handles I/O of Unicode characters. Some classes that can be used for byte stream I/O: BufferedInputStream, BufferedOutputStream, DatainputStream, DataOutputStream etc.

Concepts

	19.1	Files and folders are considered as objects. To make a file or folder object, an instance of the File class is declared and the path of the file/folder is passed as the constructor argument. For example: `File file = new File("C:/autoexec.bat");`[79] `File folder = new File("C:/windows");` In the above example, the `file` object represents a file and the `folder` object represents a folder[80]. Remember that it is *not* necessary for File objects to exist. We can create a File object with such a path that does not exist currently. However, if we try to perform operations on a non-existent file, then errors will occur. To perform operations on a file or folder, first an object of that file or folder is created and then methods of the File class are called through that object to perform operations.
	19.2	**File and Folder manipulation:** The following methods need to be remembered to manipulate files and folders:

Method	Description	Example
`boolean exists()`	Checks whether the File object exists	`if (file.exists()) {}`
`boolean isFile()`	Checks whether the File object is a file.	`if (file.isFile()) {}`
`boolean isDirectory()`	Checks whether the File object is a directory.	`if (folder.isDirectory()) {}`
`String getName()`	Obtains the name of a file or folder	`System.out.println(file.getName());`
`String getPath()`	Obtains the path of a file or folder	`System.out.println(file.getPath());`

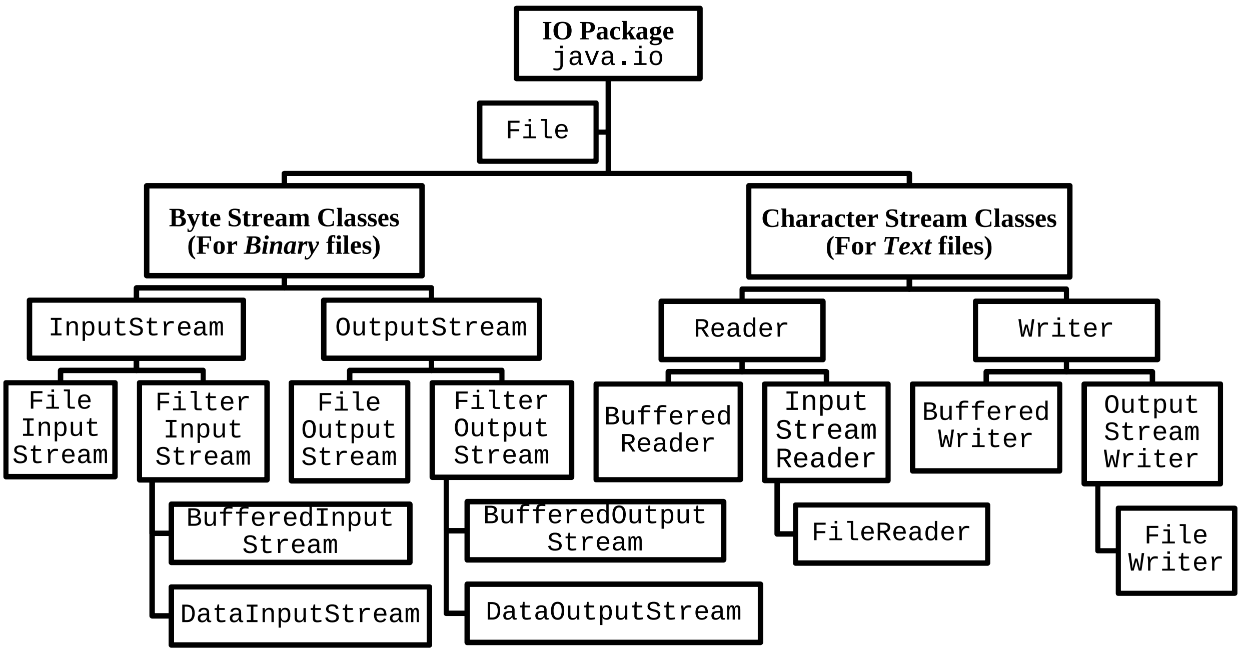
`String[] list()`
 `File[] listFiles()`
	Lists the files inside a folder	`File[] contents = folder.listFiles();`

`boolean mkdir()`	Makes a folder according to the path and name of the File object	`folder.mkdir();`

[79] Note that pathnames are not case sensitive. i.e., "C:/Windows" and "c:/windows" are the same.
[80] *Folders* are also called *directories*.

`boolean delete()`	Deletes the file or folder represented by the File object[81]	`file.delete();`
`long length()`	Gets the file size in bytes.[82]	`long size = file.length();`

19.3 **Tree representation of the classes in the IO package (Only the classes that we will use):**



19.4 **How to read/write binary and character files:**

1. To read/write binary files, declare objects of **BufferedInputStream/ BufferedOutputStream**. To read/write text files, declare objects of **BufferedReader/ BufferedWriter**.
2. To specify that we want to get input from a file, pass an anonymous object of **FileInputStream/ FileOutputStream/ FileReader/ FileWriter** as an argument to the constructor of the previous object. Again, specify the file name along with path as the argument of the constructor of the FileStream/ FileReader/Writer object. Example:

```
BufferedReader br = new BufferedReader(new FileReader("c:/abc.txt"));
```

3. To make a file appendable, simply pass `true` as the second parameter of the **FileOutputStream/ FileWriter** object. For example:

```
BufferedReader br = new BufferedReader(new FileReader("c:/abc.txt", true));
```

4. To read a file, use the `read()` method. This method reads a byte at a time, returns it as an ***int*** type and returns -1 when EOF[83] is found. On the other hand, to write a file, use the `write()` method. This method takes an **int**, a **char**, a **char[]** or a **String** as its parameter.
5. To read one line at a time from a ***text*** file, use the `readLine()` method. This method returns `null` when EOF is found.
6. To write a new line character, use the `newLine()` method.
7. Don't forget to close the files using the `close()` method.
8. To read/write primitive data types from/into a file, use the **DataInputStream/ DataOutputStream** classes. Use the `readInt()`, `readFloat()`, `writeInt()`,

[81] Note: To delete a folder, all the contents of that folder must be deleted before deleting the folder. i.e., only an empty folder can be deleted.
[82] Note that the returned file size is of type **long**. So, don't try to assign it to an **int**. Also note that you can get only *file* size with this method. *Folder* size cannot be get using this method.
[83] EOF: End Of File.

		`writeFloat()` etc. methods to read/write primitive data types. 9. Don't forget to use either try-catch blocks or the ***throws*** clause to handle exceptions.[84]
	19.5	**How to take inputs from keyboard:** 1. Create an object of the **Scanner** class and pass **`System.in`** as the argument of the constructor. 2. To check for the presence of a particular primitive type, use the `hasNextInt()`, `hasNextFloat()` etc. methods. 3. To actually read those primitive types, use the `nextInt()`, `nextFloat()` etc. methods.
	19.6	**How to read a string from console (Keyboard):** 1. Instantiate an object of the class **BufferedReader** and pass **`System.in`** as its argument. 2. Use the `readLine()` method to read a line of text.

Complete Concepts Program

```
/* IO.java

   In this program, we'll learn the following:

   1. How to create file objects.
   2. How to manipulate files.
   3. How to read/write both Binary and Text files one byte at a time.
   4. How to read/write Text files one line at a time.
   5. How to read/write primitive data types from/to files.
   6. How to append data to both Binary and Text files.
   7. How to read primitive data types as well as Strings from console.

 */

import java.io.*;
import java.util.*;

public class IO {

    public static void main(String[] args) throws IOException{
        //Manipulating files
        //Displaying the name, path and size of only the files contained in a directory
        File f = new File("c:");
        File[] list = f.listFiles();
        for (int i = 0; i < list.length; i++) {
            if (list[i].isFile()) {
                System.out.println(list[i].getName() + "\t" + list[i].getPath() + "\t" +
list[i].length());
            }
        }

        //Checking whether a file or folder exists and delete it.
        File f2 = new File("c:/abcd.txt");
        if (f2.exists()) {
            f2.delete();
        }

        //Copying files using Byte Stream
        BufferedInputStream bin = null;
        BufferedOutputStream bout = null;

        int b;
        try {
```

[84] Use the *throws* clause at exam (if the question doesn't require you to handle exceptions using try-catch) so that you don't lose time unnecessarily.

```
            bin = new BufferedInputStream(new FileInputStream("c:/abc.pdf"));
            bout = new BufferedOutputStream(new FileOutputStream("c:/a/abc.pdf"));
            while ((b = bin.read()) != -1) {
               bout.write(b);
            }
        } catch (IOException e) {
            System.out.println(e);
        } finally {
            try {
                bin.close();
                bout.close();
            } catch (IOException e) {
                System.out.println(e);
            }
        }

        //Copying Text files using Character Stream
        BufferedReader br = null;
        BufferedWriter bw = null;

        int b2;
        try {
            br = new BufferedReader(new FileReader("C:/abc.txt"));
            bw = new BufferedWriter(new FileWriter("C:/a/abc.txt"));
            while ((b2 = br.read()) != -1) {
               bw.write(b2);
            }
        } catch (Exception e) {
            System.out.println(e);
        } finally {
            try {
                br.close();
                bw.close();
            } catch (IOException e) {
                System.out.println(e);
            }
        }

        //Copying Text files by reading and writing one line at a time
        BufferedReader br2 = new BufferedReader(new FileReader("C:/abc.txt"));
        BufferedWriter bw2 = new BufferedWriter(new FileWriter("C:/a/abc.txt"));

        String s;
        while ((s = br2.readLine()) != null) {
            bw2.write(s);
            bw2.newLine();
        }

        br2.close();
        bw2.close();

        //Reading and writing primitive data types using Data Stream
        DataOutputStream dout = new DataOutputStream(new
FileOutputStream("C:/a.data"));
        DataInputStream din = new DataInputStream(new FileInputStream("C:/a.data"));

        for (int i = 0; i < 5; i++) {
            dout.writeInt(i);
        }

        int sum = 0;
        for (int i = 0; i < 5; i++) {
            sum += din.readInt();
        }
        System.out.println(sum);

        dout.close();
```

```java
        din.close();

        //Appending files (just include an argument "true" in the file constructor)
        BufferedOutputStream bout2 = new BufferedOutputStream(new
FileOutputStream("c:/a/abc.pdf", true));
        BufferedWriter bw3 = new BufferedWriter(new FileWriter("C:/a/abc.txt", true));
        DataOutputStream dout2 = new DataOutputStream(new FileOutputStream("C:/a.data",
true));
        /*  Remember, never try to include "true" as the argument for Stream
constructor. For example, the following will be an error:
       BufferedWriter bw3 = new BufferedWriter(new FileWriter("C:/a/abc.txt"), true);
        */

        //Taking input from keyboard
        //This program takes some integer numbers from keyboard and displays the
       //summation of them.
        Scanner in = new Scanner(System.in);
        int summation = 0;
        while (in.hasNextInt()) {
            summation += in.nextInt();
        }
        System.out.println(summation);

        //Reading Strings from console
        BufferedReader br3 = new BufferedReader(new InputStreamReader(System.in));
        String str;
        do {
            str = br3.readLine();
        } while (!str.equals(""));

    }
}
```

Exercises

☆☆☆	19.1	**Write a Java program that will write a list of integer numbers into a file. Your program will then read the content of the file and find the summation of the numbers.**[85] ***[Incourse-3, 2007. Marks: 7]***

Solution:

```java
import java.io.*;

public class E19_1 {
    public static void main(String[] args) {
        DataOutputStream dout = null;
        DataInputStream din = null;
        try {
            dout=new DataOutputStream(new FileOutputStream("C:/a.data"));
            din = new DataInputStream(new FileInputStream("C:/a.data"));

            for (int i = 0; i < 5; i++) {
                dout.writeInt(i);
            }

            int sum = 0;
            for (int i = 0; i < 5; i++) {
                sum += din.readInt();
            }
            System.out.println(sum);
        } catch (IOException e) {
            System.out.println("Error: " + e);
        } finally {
```

[85] Note that the marks are 7. So, the examiner expects exception handling. If the marks were 4 or 5, we could have omitted the try-catch block and simply add the ***throws*** clause.

```
                try {
                    dout.close();
                    din.close();
                } catch (IOException ex) {}
            }

        }
}
```

☆☆☆ 19.2

Write a Java code segment that will display the contents of a directory. *[Incourse-3, 2007. Marks: 3]*

Solution:

```
File dir = new File("c:/windows");
String[] contents = dir.list();
for (int i = 0; i < contents.length; i++) {
    System.out.println(contents[i]);
}
```

☆☆☆ 19.3

Write a program that will read an integer number and a file name as input from the keyboard. Your program should add the number at the end of the file. The input should first take an integer number and then a file name. For example:

11 c:\tem12.txt

Your program should generate a user-defined exception InvalidInput if the input is given in wrong order. For example:

c:\temp12.txt 23

Your program should check whether the filename is valid. If invalid then generate a user-defined exception InvalidFileName (Use exists() method of File class). *[Incourse-3, 2007. Marks: 8]*

Solution:

```
import java.io.*;
import java.util.*;

class InvalidInput extends Exception {
    public String toString() {
        return "Invalid input. Usage: <number> <filename>";
    }
}

class InvalidFileName extends Exception {
    public String toString() {
        return "The specified file does not exist.";
    }
}

public class E19_3 {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int num;
        String fileName;
        BufferedWriter bw = null;

        try {
            System.out.print("Enter the number and the file name: ");
            if (in.hasNextInt()) {
                num = in.nextInt();
            } else {
                throw new InvalidInput();
            }

            fileName = in.next();
            File file = new File(fileName);
            if (!file.exists()) {
                throw new InvalidFileName();
            }

            bw = new BufferedWriter(new FileWriter(fileName, true));
            bw.write(Integer.toString(num));
        } catch (InvalidInput e) {
            System.out.println("Error: " + e);
        } catch (InvalidFileName e) {
            System.out.println("Error: " + e);
        } catch (IOException e) {
            System.out.println("Error: " + e);
        } finally {
            try {
                bw.close();
            } catch (Exception e) {}
        }

    }
}
```

Chapter 35
Wrapper Classes, String, Generics and The Collections Framework

Theories

✯✯	35.1	**What are *Wrapper Classes*?** **OR, What are *Type Wrappers*?** Type Wrappers or Wrapper Classes are classes that encapsulate a primitive type within an object.
✯✯	35.2	**Why should you need a wrapper class? *[Incourse-2, 2007. Marks: 2]*** We need a wrapper class for the following cases: 1. To pass a primitive type as a reference to a method. 2. To use the standard data structures implemented by Java which can operate only on objects. 3. To convert primitive types to and from strings.
✯	35.3	**What is *autoboxing* and *auto-unboxing*?** Autoboxing is the process by which a primitive type is automatically encapsulated (boxed) into its equivalent type wrapper whenever an object of that type is needed. Auto-unboxing is the process by which the value of a boxed object is automatically extracted (unboxed) from a type wrapper when its value is needed.
✯✯✯	35.4	**State the advantages of autoboxing & auto-unboxing. *[Incourse-3, 2007. Marks: 2]*** The advantages of autoboxing and auto-unboxing are: 1. Removes the tedium of manually boxing and unboxing values. 2. Helps prevent errors. 3. Makes working with the Collections Framework much easier.
✯	35.5	**What is the disadvantage of the *String* class? How can it be solved?** The disadvantage of the String class is that objects of String class are immutable[86]. Using the **StringBuffer** or the **StringBuilder** class is the solution for this problem.
	35.6	**What is the difference between *String* class & *StringBuffer* class? *[2002. Marks: 2]*** Objects of String class are immutable, whereas objects of StringBuffer class are changeable.
	35.7	**What is the difference between *StringBuffer* class and *StringBuilder* class?** **Or, What is the advantage of *StringBuilder* over *StringBuffer* class?** The difference between StringBuffer and StringBuilder is that the latter is not synchronized, which means that it is not thread-safe. The advantage of the StringBuilder class is faster performance.
	35.8	**What is the specialty (or utility) of the `toString()` method?** When an object is used in a concatenating expression or in a call to `println()` method, the `toString()` method of that object is automatically involved. Usually this method returns a string that appropriately describes an object of a class.
	35.9	**What is the Collections Framework?** The Collections Framework[87] is a sophisticated hierarchy of interfaces and classes that

[86] *Immutable* means *unchangeable.*

		provide state-of-the-art[88] technology for managing groups of objects.
	35.10	**State the advantages of using collection classes. *[Incourse-3, 2007. Marks: 3]*** The advantages of using collection classes are: 1. They are high-performance. 2. They allow different types of collections to work in a similar manner and with a high degree of interoperability. 3. Extending and/or adapting a collection is easy.

Concepts – Primitive Types and Their Respective Wrapper Types

Primitive Types	Wrapper Types
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
char	Character
boolean	Boolean

Concepts – String Constructors and Methods

Constructors:

Constructors	Examples	Content of the String a
`String()`	`String a = new String();`	`""`
`String(String str)`	`String a = new String("Hello");`	`"Hello"`

`String(char[] chars)`	`char[] c = {'a', 'b', 'c'};`
	`String a = new String(c);`
		`"abc"`

`String(byte[] bytes)`	`byte[] b = {65, 66, 67};`
	`String a = new String(b);`
		`"ABC"`

Methods[89]:

```
String a = new String("Hello Universe!");
String b;
String[] str;
int i;
char c;
char[] charArr;
```

Methods	Examples
`String toUpperCase()`	`b = a.toUpperCase();`

[87] *Collections* in this context means different data structures (for example: array, linked lists, trees, graphs etc.).
Framework: A structure supporting or containing something (in this case – collections).
[88] **State-of-the-art**: The highest level of development of an art or technique at a particular time (specially the present time).
[89] Note: only the shaded methods are important for the exam.

String toLowerCase()	b = a.toLowerCase();
int length()	i = a.length();
char charAt(int index)	c = a.charAt(0);

int indexOf(Char ch)
int indexOf(String str)
	c = a.indexOf('!');
	c = a.idexOf("Uni");

String substring(int beginIndex)
String substring(int beginIndex, int endIndex)
	b = a.subString(6);
	b = a.subString(6,9);

boolean equals(String anotherString)
boolean equalsIgnoreCase(String anotherString)
	if (a.equals("Hi")) {}
	if (a.equalsIgnoreCase("Hi")) {}

char[] toCharArray()	charArr = a.toCharArray();

boolean startsWith(String prefix)
boolean startsWith(String prefix, int offset)
	if (a.startsWith("He")) {}
	if (a.startsWith("Un", 6)) {}

boolean endsWith(String suffix)	if (a.endsWith("e!")) {}

String replace(char oldChar, char newChar)
String replace(String target, String replacement)
	b = a.replace('o', 'u');
	b = a.replace("o", "oo");
	//b = a.replace('o', "oo"); Error!

boolean contains(String str)	if (a.contains("Uni")) {}
String[] split(String regex)	str = a.split(" ");
String trim()	b = a.trim();

Concepts – Comparative Analysis of the Methods of `ArrayList` and `Vector` Classes

Constructors:

ArrayList	Vector
ArrayList(int initialCapacity)	**Vector(int initialCapacity, int capacityIncrement)**
ArrayList() [Capacity = 10]	**Vector(int initialCapacity)** [Increment = 0]
	Vector() [Similar to calling Vector(10, 0)]

Methods:

boolean add(E element)	boolean add(E element) void addElement(E obj)
void add(E element)	void add(E element)
boolean remove(Object obj)	boolean remove(Object obj) void removeElement(Object obj)
E remove(int index)	E remove(int index) void removeElementAt(int index)
E set(int index, E element)	E set(int index, E element) void setElementAt(E obj, int index)
E get(int index)	E get(int index) E elementAt(int index)
int indexOf(Object o)	int indexOf(Object o)
boolean contains(Object obj)	boolean contains(Object obj)

`int size()`	`int size()`
`void trimToSize()`	`void trimToSize()`
`<T> T[] toArray(T[] a)`	`<T> T[] toArray(T[] a)`
	`int capacity()`
	`void setSize(int newSize)`

Points to be Remembered

	35.1	Autoboxing and auto-unboxing occurs whenever primitive types or type wrapper objects are passed to a method or returned by a method. So, type wrapper objects will not work as call-by-reference as expected.

Complete Concepts Program – `ArrayList` and `Vector`

```
import java.util.*;

public class CompleteConcept_Chapter35_1 {
    public static void main(String[] args) {
        //Creating an ArrayList object with initial capacity of 5 elements
        ArrayList<Float> a = new ArrayList<Float>(5);

        a.add(1.1f);
        a.add(2.2f);
        a.add(3.3f);
        System.out.println(a); //Prints: [1.1, 2.2, 3.3]

        a.set(1, 5.5f);
        System.out.println(a); //Prints: [1.1, 5.5, 3.3]

        System.out.println(a.get(0)); //Prints: 1.1
        System.out.println(a.indexOf(5.5f)); //Prints: 1
        System.out.println(a.contains(5.5f)); //Prints: true
        System.out.println(a.size()); //Prints: 3

        a.remove(5.5f);
        System.out.println(a); //Prints: [1.1, 3.3]
        a.remove(0);
        System.out.println(a); //Prints: [3.3]

        //Creating a Vector object with initial capacity of 5 elements
        //and an increment value of 2
        Vector<Integer> v = new Vector<Integer>(5, 2);

        //All the above methods apply to Vector, too.
        //So, they are not repeated here.
    }
}
```

Complete Concepts Program – `String`

```
public class CompleteConcept_Chapter35_2 {
    public static void main(String[] args) {
        String a = new String("Hello Universe!");
        String str[];
        char[] charArr;

        System.out.println(a.toUpperCase());                //HELLO UNIVERSE!
        System.out.println(a.toLowerCase());                //hello universe!
        System.out.println(a.length());                   //15
        System.out.println(a.charAt(0));                  //H
        System.out.println(a.indexOf('e'));              //1
        System.out.println(a.indexOf("Uni"));            //6
        System.out.println(a.substring(6));              //Universe!
        System.out.println(a.substring(6, 9));           //Uni
        System.out.println(a.equals("hello universe!")); //false
        System.out.println(a.equalsIgnoreCase("hello universe!")); //true
        System.out.println(a.startsWith("Hello"));       //true
        System.out.println(a.startsWith("Uni", 6));      //true
        System.out.println(a.endsWith("e!"));            //true
        System.out.println(a.contains("Uni"));           //true
        System.out.println(a.replace('e', 'u'));         //Hullo Univursu!
        System.out.println(a.replace("ll", "lll"));      //Helllo Universe!

        charArr = a.toCharArray();
        for (int i = 0; i < charArr.length; i++) {
            System.out.print(charArr[i] + " ");
        }
        System.out.println();                         //H e l l o   U n i v e r s e !

        str = a.split(" ");
        for (int i = 0; i < str.length; i++) {
            System.out.println(str[i] + " ");
        }                                            //Hello
                                                     //Universe!

        a = "   A B C  \n  ";
        System.out.println(a);                       //   A B C
                                                     //
        System.out.println(a.trim());                //A B C
    }
}
```

Exercises

☆☆☆	35.1	**Write a Java program that will perform the following operations: *[Incourse-3, 2007. Marks: 5]***
		
		1. **Create an object of type *ArrayList* that will contain a list of floating-point numbers.**
		2. **Now insert the following data: 12.34, 34.5, 5.6, 7.89, 10.12, 3.45**
		3. **Show the number of elements in the object.**
		4. **Remove 5.6 and 10.12**
		5. **Display the content of the object.**
		
		Solution[90]:
		
		`import java.util.ArrayList;`
		
		`public class E35_1 {`
		`    public static void main(String[] args) {`
		`        ArrayList<Float> a = new ArrayList<Float>();`
		
		`        a.add(12.34f);`
		`        a.add(34.5f);`
		`        a.add(5.6f);`
		`        a.add(7.89f);`
		`        a.add(10.12f);`
		`        a.add(3.45f);`
		
		`        System.out.println(a.size());`

[90] Note: Don't forget to append the '`f`' suffix when adding or removing float type elements in the program.

```java
        a.remove(5.6f);
        a.remove(10.12f);

        System.out.println(a);
    }
}
```

35.2 **Write a generic class and implement it.**

Solution:

```java
class Gen<T> {
    T obj;

    Gen(T o) {
        obj = o;
    }

    T getObj() {
        return obj;
    }
}

public class E35_2 {
    public static void main(String[] args) {
        Gen<Integer> a = new Gen<Integer>(100);
        int i = a.getObj();
        System.out.println(i);
    }
}
```