ICSE Computer Applications
Reference Booklet

Principles of OOPS

Abstraction: It is the act of representing only the essential features/information, while hiding complex implementation details to the user, focusing on “what” a class does, than “how” it does it.

Encapsulation: It is the wrapping up of data and functions of an object as one unit, that can be used to together for a specific purpose, while restricting direct access to internal data and controlling its access.

Inheritance: It is the property by which a class [subclass] acquires (inherits) the features and behaviour of another class [superclass], promoting code reusability.

Polymorphism: It is the process of using a given method for multiple operations. A method with the same name is made to perform different function based on the given conditions.

ASCII Codes

A - Z	*65 - 90*
a - z	*97 - 122*
0 - 9	*48 - 57*
whitespace - ‘ ‘	*32*

Escape Sequences

\t	*horizontal tab*	**\v**	*vertical tab*
\b	*vertical tab*	\	*backslash*
\n	*new line*	**\f**	*form feed*
\’	*single quote*	**\0**	*null*
\”	*double quote*	**\r**	*carriage return*

Tokens: Each individual component of a Java program that carries some meaning and takes active part in the program execution. The tokens are:
Literals
Identifiers
Assignment operators
Operators: Arithmetic, Relational, Logical
Punctuators: comma, semicolon, period
Separators: comma, parentheses, braces, square brackets
Keywords

Data Types Spec-sheet

Type	Default	Size	Range
boolean	false	1 bit	true or false
char	\u0000	2 bytes	0 to 65,535 (unsigned, UTF-16)
byte	0	1 byte	-128 to 127
short	0	2 bytes	-2^{15} to $(2^{15} - 1)$
int	0	4 bytes	-2^{31} to $(2^{31} - 1)$
long	0L	8 bytes	-2^{63} to $(2^{63} - 1)$
float	0.0f	4 bytes	±1.4E-45 to ±3.4028235E38 **(7 digit precision)**
double	0.0	8 bytes	±4.9E-324 to ±1.7976931348623157E308 **(15-16 digit precision)**

Operator Precedence Table

Operator	Associativity
++ --	Right to Left
++ -- + - ~ *!* *(type)*	Right to Left
**** / %***	Left to Right
+ -	Left to Right
< <= >= >	Left to Right

== !=	Left to Right
&	Left to Right
^	Left to Right
|	Left to Right
&&	Left to Right
||	Left to Right
?:	Right to Left
= += -= *= /= %=	Right to Left

Java: Compiler, Interpreter, Platform Independece

Java is often called a **"compiler-interpreter language"** because it uses both a compiler and an interpreter: the Java compiler (javac) converts **source code** into **bytecode**. Bytecode is an **platform-independent, intermediate** code created after the source code is complied. The bytecode can be interpreted on any system with a **JVM** to run the program.

Java is platform-independent because it compiles to **bytecode**, a platform-neutral instruction set, which is then executed by the **JVM** on any device with a compatible JVM, enabling the "write once, run anywhere" - **WORA** capability.

Math Functions

Method	Description	Return Type

min(a, b)
max(a, b)
	return smallest & largest value respectively	int/long/float/double

sqrt(a)
cbrt(a)
	return square root and cube root respectively	double

pow(a, b)	returns value of $\mathbf{a^b}$	double
abs(a)	modulus function	int/long/float/double
round(a)	up to nearest integer - standard rounding	int/long
rint(a)	to nearest integer - precise rounding	double
floor(a)	nearest integer less than or equal to **'a'**	double
ceil(a)	nearest integer greater than or equal to **'a'**	double
random()	random real; 0 <= r < 1	double

log(a)
exp(a)
	return value of **ln a** and $\mathbf{e^a}$ respectively	double

sin(a), cos(a), tan(a)	give respective values **in radians**	double

Math.rint(a) v.s. Math.round(a)

```
System.out.println(Math.rint(2.5)); // 2.0 (to even integer)
System.out.println(Math.rint(3.5)); // 4.0


System.out.println(Math.round(2.5)); // 3 (standard rounding)
System.out.println(Math.round(3.5)); // 4
```

Random Number Generation

1) Math.random()
   0 <= r < 1
2) Math.random() + k
   k <= r < k + 1; **k is the shifting factor**
3) Math.random() * a

0 <= r < a; **a is the scaling factor**

4) (int)(Math.random() * max)
   0 <= i <= max - 1
5) (int)(Math.random() * (max - min + 1)) + min
   min <= i <= max

Scanners

System.out.println(): Prints on the current line and moves the cursor to a new line after printing. Empty statement simply moves the cursor to a new line.

System.out.print(): Prints on the current line and does not shift the cursor.

Scanner issue with *'nextLine()'*
If you use nextLine() after nextInt(), it can seem like it is skipping input. `nextInt()` leaves a newline character in the buffer, so `nextLine()` immediately reads it.

```
sc.nextLine(); // Consume the leftover newline
```

```
sc.next(); //accepts next word
sc.next().chatAt(0); //next character
sc.useDelimiter(','); //changes the delimiter from ‘ ’ to ','
sc.next(); //accepts all characters until the next comma
```

Syntax Error: Occurs due to a *grammatical error* in the program, when the programmer breaks the structure of the program. ***For example, missing punctuators, incorrect instructions, undefined variables, etc.***

Logical Error: Occurs when the program gets compiled successfully, but does not produce the desired results. ***For example, incorrect formulae or conditions.***

Runtime Error: Occurs at runtime when the program gets successfully compiled, but the computer does not respond properly while executing a particular statement. ***For example, division by zero, null reference, array index out of bounds, etc.***

Type of Error	Detected by	When
Syntax	Compiler	Compilation
Logical	Programmer	Run-time
Runtime	JVM [Interpreter]	Run-time

Switch Statement - Fall Through

The condition where a break statement is not used after a case, causing the control to enter the next case for execution.

Wrapper Classes

Boxing: Conversion of a value of a primitive data type into an object of its equivalent Wrapper class. When boxing is done **automatically by the compiler** at compilation, it is called **Autoboxing.**

→ When we want to pass a primitive data type argument to a method that uses a wrapper class as the function argument. Ex. ArrayList.

Unboxing: It is the opposite of Boxing. It is the conversion of the object of a Wrapper class into its primitive data type. When unboxing is done **automatically by the compiler** at compilation, it is called **Auto-unboxing.**

→ When the value from the object of a wrapper class is to be passed to a function which accepts arguments of primitive data types
→ When the returned value of a method, whose return type is a Wrapper Class, is to be used as a primitive data type.

```java
int p = 10;
Integer P = Integer.valueOf(p); // Boxing
```

```java
Integer P = Integer.valueOf(10);
int p = P.intValue(); // Unboxing
```

```java
int p = 10;
Integer P = p; // Autoboxing
int p2 = P; // Autounboxing
```

Here ***p*** and **p2** are of `int` type and **P** is of `Integer` type.

Integer.parseInt(): Returns a primitive *'int'* value. Autoboxing may be required in some cases.

Integer.valueOf(): Returns an object of the *'Integer'* class. Auto-unboxing may be required in some cases.

OPTIONAL TOPIC

Integer Caching [Object Pooling]

```java
Integer a = Integer.valueOf(100);
Integer b = Integer.valueOf(100);
System.out.println(a == b); // true (same cached object)

Integer c = Integer.valueOf(200);
Integer d = Integer.valueOf(200);
System.out.println(c == d); // false (new objects)
```

Java caches Integer values from -128 to 127. For values outside this range, new objects are created.

The String Pool

→ "hello" is added to the string pool

```
String s1 = "hello";
String s2 = "hello";

System.out.println(s1 == s2); // true (both point to the same object)
```

→ **New object is added to the heap, different memery locations**

```
String s1 = new String("hello");
String s2 = new String("hello"); //new object, bypassing string pool

System.out.println(s1 == s2); // false (different objects)
```

→ **"intern()" method checks the pool for the string and returns the pooled address, if it exists, else it is created**

```
String s1 = new String("hello").intern(); //forces pooling
String s2 = "hello";

System.out.println(s1 == s2); // true (both now point to the pooled object)
```

OPTIONAL TOPIC CONLCUDED

Character Functions

Method	Return Type
Character.isLetter(c)	boolean
Character.isDigit(c)	
Character.isLetterOrDigit(c)	
Character.isWhiteSpace(c)	

Character.isUpperCase(c)	
Character.isLowerCase(c)	
Character.toUpperCase(c)	Character [auto-unboxed to char]
Character.toLowerCase(c)	

Array Algorithms

[Sorting]

Bubble Sort

```
int[] arr = {1, 3, 4, 8, 5, 6, 2, 9, 0, 7};
int temp = 0;
for(int i = 0; i < arr.length - 1; i++) {
      boolean swap = false;
      for(int j = 0; j < arr.length - 1 - i; j++) {
        if (arr[j] > arr[j+1]) {
          temp = arr[j]; arr[j] = arr[j + 1];
          arr[j + 1] = temp; swap = true;
        }
      }
      if(!swap) break;
}
for (int x : arr) System.out.print(x + " ");
```

Selection Sort

```
int[] arr = {1, 3, 4, 8, 5, 6, 2, 9, 0, 7};
int temp = 0;
for (int i = 0; i < arr.length; i++) {
      int minIndex = i;
      for (int j = i + 1; j < arr.length; j++)
        if (arr[j] < arr[minIndex]) minIndex = j;
      temp = arr[i];
      arr[i] = arr[minIndex];
```

```java
            arr[minIndex] = temp;
}
for (int x : arr) System.out.print(x + " ");
```

[Searching]

Binary Search

```java
int low = 0, high = arr.length - 1, mid = 0;
int target = 5;
boolean found = false;

while(low <= high) {
            mid = low + (high - low)/2;
            if(mid == target) {
                found = true;
                break;
            } else if (mid < target) low = mid + 1;
            else high = mid - 1;
}

if(found) System.out.println("Target found at index: " + mid);
else System.out.println("Target not found");
```

Linear Search	Binary Search
Works on sorted and unsorted arrays	Words only on sorted arrays
Search begins at the start of the array, i.e., from the 0th index, and continues till the element is found	Array is divided into two halves and one of the halves is searched, which is further split until the target is found
Works on single and multi-dimensional arrays	Works only on single dimensional arrays
Has time complexity O(n)	Has time complexity O(logn)
It uses equality comparisions	It uses ordering comparisions

Inserting Elements

```
int[] set = new int[6]; //new array
for(int i = 0; i < set.length - 1; i++) set[i] = i;
//initialising all but one element
int index = 3, rep = 21;
//shifting elements
for (int i = set.length - 2; i >= 3; i--) set[i + 1] = set[i];
set[3] = rep;
for(int x : set) System.out.print(x + " ");
```

Deleting Elements

```
int[] set = new int[6]; //new array
for(int i = 0; i < set.length; i++) set[i] = i;
//initialising array
int index = 3;
//shifting elements
for (int i = index; i < set.length - 1; i++) set[i] = set[i + 1];
set[set.length - 1] = 0; //setting last element to zero
for(int x : set) System.out.print(x + " ");
//or you can choose to print only till the last element
```

Merging Arrays

```
int[] arr0 = new int[3]; //new arrays
int[] arr1 = new int[5];
int[] arr2 = new int[arr0.length + arr1.length];
for (int i = 0; i < arr0.length; i++) arr0[i] = i;
//initialisation
for (int i = 0; i < arr1.length; i++) arr1[i] = i;
for(int i = 0; i < arr0.length; i++) arr2[i] = arr0[i]; //merging
arrays
for(int i = 0; i < arr1.length; i++) arr2[i + arr0.length] =
arr1[i];
for(int x : arr2) System.out.print(x + " ");
```

Functions and Overloading

Formal parameters are the parameters described in the method header of the called function and receive their values from its calling function.

Actual parameters are the parameters/values passed to the calling function when it is called.

Calling Function: Method which calls another method.

```
obj.factorial(x);
```

Called Function: Method which is called by another method.

```
public int factorial(int a) {
      if (a == 0) return 1;
      else return a * factorial(a - 1);
}
```

Pass by value	Pass by reference
Process of passing a copy of the actual parameters to the formal parameters	Process of passing the reference [address/alibi] of the actual parameters to the formal parameters.
Any changes made in the formal parameters does not reflect in the actual parameters	Any changes made in the formal parameters are reflected in the actual parameters
Usually for primitive data types	Usually for objects and Arrays

Pure method	Impure method
Does not change the internal state of the object	Changes the internal state of the object

Generally returns a value	Generally does not return a value
Also known as getter/accessor	Also known as setter/mutator

Early/Static binding: During function overloading, when an overloaded method is called, the system finds the best match of the function arguments and the parameter list [i.e., types and number of parameters] *during the program compilation,* which is known as static or early binding.

Function overloading: It is the process of defining functions/methods with the same method name, but different number and types of parameters.

Recursive function: It is a function that calls itself in its body.

```
public static int add(int a, int b) { return a + b; }
```

Method Header: `public static int add(int a, int b)`

Method Signature: `add(int a, int b)`

Best Match - Method Overloading

Two method are said to be the best match for overloading if:
→ Types of actual and formal parameters are same
→ Number of actual and formal parameters are same
→ Order of data types of actual and formal parameters is same

---

String Functions

Function	Usage	Return Type
str.**toLowerCase()**	converts to lower case	String
str.**toUpperCase()**	converts to upper case	String
replace(ch,ch1**)**	replaces all occurrences of	String

	‘ch’ with ‘ch1’	
replace(str, str1**)**	replaces all occurrences of ‘str’ with ‘str1’	String
str.**trim()**	removes leading and trailing spaces	String

str.**equals(**str1**)**	if ‘str’ equals ‘str1’;
	case-sensitive
		boolean

str.**equalsIgnoreCase(**str1**)**	if ‘str’ equals ‘str1’;
	case-insensitive
		boolean

str.**length()**	returns length of String	int
str.**chatAt(**i**)**	returns the character at ‘i’	char
str.**substring(**m**)** ***[OR]*** str.**substring(**m, str.length()**)**	characters from ‘m’ (incl.)	String
str.**substring(**m, n**)**	characters from ‘m’ (incl.) to ‘(n - 1)’ (incl.)	String
str.**concat(**str1**)**	concatenates ‘str’ and ‘str1’	String
str.**indexOf(**ch**)**	index of first occurence	int
str.**indexOf(**ch, i**)**	index of first occurrence after ‘i’ (inclusive)	int
str.**lastIndexOf(**ch**)**	index of last occurence	int
str.**compareToIgnoreCase(**str1**)**	compareTo() case-insensitive	int
str.**startsWith(**str1**)**	if ‘str’ starts with ‘str1’	boolean
str.**endsWith(**str1**)**	if ‘str’ ends with ‘str1’	boolean
str.**compareTo(**str1**)** - returns an ‘int’ if str = str1, 0] if str > str1, +ve] - if str ≠ str1 **(or)** str = str1 and their lengths are equal if str < str1, -ve] str.length() - str1.length()] - if common characters are equal ‘ ’ **(32)** < ‘0’ to ‘9’ **(48 to 57)** < ‘A’ to ‘Z’ **(65 to 90)** < ‘a’ to ‘z’ **(97 to 122)**		

Encapsulation

Access Specifiers

[for variables]	default	public	private	protected
same class	✅	✅	✅	✅
same package subclass	✅	✅	❌	✅
same package non-sublcass	✅	✅	❌	✅
different package subclass	❌	✅	❌	✅
different package non-subclass	❌	✅	❌	❌

[final keyword]	**Behaviour**
variables	constant (cannot be changed)
methods	**cannot** be **overridden**, but **can** be **overloaded**
classes	cannot be inherited

Instance vs Class Variables

-	Instance Variables	Class Variables
Definition	Each object of the class has an individual copy	They are common fields for all objects of the class
Declaration	declared without ‘static’ keyword	declared using ‘static’ keyword

Access	Object name must be referred to handle them **objectName.var;**	Object name is not required; they can be access directly with the class name **className.var;**
Copies	One per object	One per class
Memory allocation	When the object is created	When the class is loaded
Use case	Object-specific data	Shared properties of all objects

---

Method Overriding

When a subclass provides its own implementation of a method that is already defined in the super class. For overriding, the method header (name, return type, parameters) of the original and overridden methods must be the same. **‘final**,’ ‘**private’** and ‘**static’** methods **cannot** be **overriden**.

Late/Binding binding: During method overriding, when an overridden method is called through a parent class reference, the system determines the appropriate method to invoke based on the runtime type of the object, not the reference type. This decision happens *during program execution,* which is known as dynamic or late binding.

-	Static Binding	Dynamic Binding
Timing	Compile-time (early binding)	Runtime (late binding)
Methods	static, final and private	overridden methods
Performance	Faster	Slightly slower (runtime lookup
Decided by	Compiler	JVM

Constructors

Special member methods with the same name as the class name. They help in intiailising the data members in the classes to some initial values. They are non-returnable.

Constructor Overloading

If a parameterised constructor is defined and no parameters are provided at the time of object creation, the default constructor will **not** be used. Another non-parameterised constructor must be explicitly defined in this case.

Copy Constructor

A constructor that is used to initialise the instance variables of an object by *copying the initial values of the instance variables from another object,* is known as a Copy Constructor. It can be either a **Direct Entry Copy Constructor,** or may require an **object to be passed.**

A Direct Entry Copy Constructor is **not recommended** as it leads to unintended affects due to reference sharing, where changes in one of the objects, are reflected in the other object as well, since they both point to the same memory location.

```
public class ClassExpt {
   private int age;

   ClassExpt(int age) { this.age = age; }

   public static void main(String[] argus) {
       ClassExpt c1 = new ClassExpt(23);
       ClassExpt c2 = c1;

       c1.age = 33;
       System.out.println(c1.age + " " + c2.age); //33 33

       c2.age = 43;
       System.out.println(c1.age + " " + c2.age); //43 43
   }
}
```

Instead a copy constructor, in which the required object is passed, is preferred. This way, the “new” keyword creates a new object, with the same values of the instance variables of the intended object, pointing to a different memory location. Any change in one of the objects, will not affect the other.

```
public class ClassExpt {
   private int age;

   ClassExpt(int age) { this.age = age; }

   ClassExpt(ClassExpt c) {
       age = c.age;
       //it is the same class, even though ‘age’ is private,
       //we can access it directly
   }

   public static void main(String[] argus) {
       ClassExpt c1 = new ClassExpt(23);
       ClassExpt c2 = new ClassExpt(c1);

       c1.age = 33;
       System.out.println(c1.age + " " + c2.age); //33 23

       c2.age = 43;
       System.out.println(c1.age + " " + c2.age); //33 43
   }
}
```