Table of Contents

Objective 1: Java Basics

1. Define the scope of variables

a. General information

- The scope of a variable is the section of the program in which the variable is visible.
- Variable identifiers are statically scoped, i.e. the scope of a variable is determined at compile time.

b. Class-level scope

- Class-level variables, also known as fields, are declared inside a class and outside any methods.
- A class-level variable is accessible from anywhere within the class, and maybe from the outside provided a suitable modifier is in place. The following table shows the access to members permitted by each modifier.

Modifier	*Class*	*Package*	*Subclass*	*World*
public	Y	Y	Y	Y
protected	Y	Y	Y	N
no modifier	Y	Y	N	N
private	Y	N	N	N

Code example:

```
public class MyClass {
   int number = 1234;
   void myMethod() {
      System.out.println(number);
      System.out.println(text);
   }
   private String text = "text";
}
```

In the above example, variables *number* and *text* are both members of the *MyClass* class and accessible to the body of *myMethod*. This method prints out *"1234"* and *"text"* when getting executed.

c. Method-level scope

- Variables declared inside a method, also known as local variables, are scoped to the method and cannot be accessed from outside the method.
- A method-scoped variable only exists since its declaration until the method returns.

- Method parameters are no different from method-scoped variables, except that instead of being declared and initialized within the method body, they are declared as part of the method signature and initialized when the method gets invoked.

d. Block-level scope

- A pair of curly brackets defines a scope.
- A variable can only be accessed since its declaration until the closing bracket. It is inaccessible to the code outside the block.

Code example:

```
void myMethod() {
   if (true) {
      // System.out.println(i);
      int i = 0;
      System.out.println(i);
   }
   // System.out.println(i);
}
```

The above method prints out number *0* when it is executed. If you uncomment the first comment, a compile error will be produced as the *i* variable is used before its declaration. Uncommenting the second leads to the same issue as this variable will be used out of scope.

- Subclass
  (other package)
  Y
  Y (only through sub class reference)
  N
  N

2. Define the structure of a Java class

A class may include the following members:

- Fields
- Methods
- Constructors

In addition to the above, a class may include other members, such as initialization blocks or nested classes. These members are not covered in the OCAJP exam, though.

a. Fields

Field declarations are composed of three components, in order:

- Zero or more modifiers such as public or private
- The type of the field
- The field name

b. Methods

Method declarations have the following components, in order:

- Modifiers, such as public, private
- The return type (the data type of the value returned by the method), or void if the method does not return a value
- The method name
- A comma-delimited list of input parameters, preceded by their data types, enclosed by parentheses (); if there are no parameters, we must use empty parentheses
- An exception list

- The method body, enclosed between curly braces; the method's code, including the declaration of local variables, goes here
- Variables, methods, constructors in a class can be written in any order.
  A Java source file also contains below elements
    - The package statement
    - The import statement
    - Comments
- The package statement :
  If a class includes a package statement, it must be the first non comment statement in the class definition.
- The import statement :
  If a class includes a package statement, import statement should come after package statement but before class declaration.
  If a class doesn't includes a package statement, import statement should be the first non comment statement in the class definition.
- Comments :
  You can write comments anywhere. A comment can appear before and after a package statement, before and after the class definition, before and within and after a method definition.

c. Constructors

A class contains constructors that are invoked to create objects from the class blueprint. Constructor declarations look like method declarations, except that they use the name of the class and have no return type.

3. Create executable Java applications with a main method; run a Java program from the command line; produce console output

a. Requirement

In order to be executable, a Java class must have one method with the following declaration:

```
public static void main(String[] args) {
   // code goes here
}
```

The parameter type may be changed to *String...*, and the parameter name can be whatever you want.

Code example:

```
public class MyClass {
   public static void main(String[] args) {
      for (String arg : args) {
         System.out.print(arg.toUpperCase() + " ");
      }
   }
}
```

The example application takes a list of strings, converts them to uppercase, then prints out on the console output.

public, static keywords positions can be interchanged as follows

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

We can also add final , strictfp keywords as follows

```
final strictfp static public void main(String[] args) {  }
```

b. Program execution

- Name the file containing the example code *MyClass.java*.
- Compile the source file into a class file using Java compiler:
  javac MyClass.java
- Execute the application using Java application launcher tool with the command:
  java MyClass i will pass the ocajp exam

This application prints out the text *"I WILL PASS THE OCAJP EXAM"* on the console. Remember that the first argument passed to the *java* command is the name of the class containing the main method without any suffixes. The first argument of the application is the second of the command.

4. Import other Java packages to make them accessible in your code

To import a specific type into the current file, put an *import* statement at the beginning of the file before any type definitions but after the *package* statement. The type is referenced by its fully qualified name. For instance:

```
import java.util.List;
```

To import all the types contained in a particular package, use the *import* statement with the asterisk (*) wildcard character, such as:

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

After a type is imported, it may be referred to by simple name.

Code example:

```
import java.util.*;
public class MyClass {
   public static void main(String[] args) {
      List<String> list = Arrays.asList(args);
      System.out.println(list);
   }
}
```

Without the *import* statement, the type of the *list* variable must be referred to as *java.util.List*.

5. Compare and contrast the features and components of Java such as: platform independence, object orientation, encapsulation, etc.

The Java programming language is a high-level language that can be characterized by the following features:

a. Simple: Java was designed to be easy for professional programmers to learn and use effectively.

b. Object oriented: Java is a true object oriented language. Object oriented means we organize our software as a combination of different types of objects that incorporates both data and behavior.

Basic concepts of object oriented programming are:

- *Inheritance:* Allows a class to use properties and methods of another class.
- *Polymorphism:* Subclasses of a class can define their own unique behavior and yet share some of the same functionality of the parent class.
- *Abstraction:* The process used to hide certain details and only show the essential features of the object.
- *Encapsulation:* The mechanism of binding or wrapping the data and the codes that operates on the data into a single entity.

c. Platform independent: Applications written using Java can be executed on a computer with any CPU or operating system.
d. Other features: Simple, distributed, robust, secure, high performance, dynamic, threaded, interpreted, architecture neutral.

Objective 2: Working with Java Data Types

1. Declare and initialize variables (including casting of primitive data types)

a. Class-level variables

Class-level variables, including both static and non-static fields, are declared and initialized at the same time. If you declare a variable without initializing it, the complier will set the variable to a reasonable default value based on its type. The default values for variables of various data types are listed below:

Data Type	*Default Value*
byte	0
short	0
int	0
long	0L
float	0.0F
double	0.0D
char	'\u0000'
boolean	false
Object	null

It is not required to remember all primitive data types ranges. But it is good to remember byte , char data types ranges. byte ranges from -128 to 127 . char ranges from 0 to 65536 .

b. Local variables

Local variables are used to store temporary states within methods. They must be declared and explicitly initialized before being used as the compiler does not assign them default values.
A local variable may be initialized anywhere after it is declared (but must be within the enclosing method).

c. Primitive data type casting

Widening casting

This type of casting is implicitly done in the following order:
byte --> short --> int --> long --> float --> double
A widening casting does not always preserve the magnitude and precision of a value:

- Casting from *float* to *double* that is not *strictfp* may lose information about the overall magnitude of the converted value
- Casting from *int* to *float*, or from *long* to *float*, or from *long* to *double*, may result in loss of precision - that is, the result may lose some of the least significant bits of the value

Narrowing casting

This type of casting can only be done explicitly. The narrowing order is the opposite of the widening:
double --> float --> long --> int --> short --> byte
Narrowing casting may lose information about the magnitude of a numeric value, as well as precision and range.

Casting to and from char

- Widening casting: from *char* to *int*, *long*, *float*, or *double* - this is done implicitly
- Narrowing casting: *char* to *byte* or *short*; *short*, *int*, *long*, *float* or *double* to *char* - this must be done explicitly
- Widening and narrowing casting: *byte* to *char* - the *byte* value is first converted to an *int* via widening casting, then the resulting *int* is converted to a *char* by narrowing - this casting must be done explicitly

Notes:

- A boolean value cannot be cast to and from any other primitive data type
- Even though loss of precision may occur, a primitive conversion never results in a run-time exception

2. Differentiate between object reference variables and primitive variables

The basic difference between primitive and object reference variables is that the former store actual values, whereas the latter store the addresses of objects they refer to. This difference leads to distinctions in their behaviors:

- *Comparison:* Primitive variables are compared based on their literal values; while object reference variables are based on their addresses (except when the equals methods of the classes of their associated objects are overridden).
- *Assignment:* In the assignment of a primitive variable, the value of the right-hand side (RHS) is copied to the left-hand side (LHS), meaning that the LHS variable holds its own value; while the address is copied when assigning a reference variable, resulting in the LHS variable pointing to the same object returned by the expression on the RHS.
- *Passing parameter:* A primitive variable is passed to a method by value, implying that the changes made to parameters within the method is independent of the original variable. Reference data type parameters, such as objects, are also passed into methods by value, meaning that an object reference is manipulated inside the method and any changes made to the object is reflected in the original variable.
- *Returning value:* In a similar way to passing parameter, a primitive variable is returned from a method by value and this variable dies after the method finishes execution. In contrast, an object is returned by reference and the original object may survive the method completion if it is assigned to another variable.
- There are a few important differences you should know between primitives and reference types. First, reference types can be assigned null, which means they do not currently refer to an object. Primitive types will give you a compiler error if you attempt to assign them null.

  Ex : String name = null;
  int value = null; // compile time error

  Reference types can be used to call methods when they do not point to null.
  Primitives do not have methods declared on them.
  Ex: String name = "Whiz";
  int value = name.length();
  int count = value.length(); // compile time error

3. Know how to read or write to object fields

a. Inside the object

Statements within an object may access and mutate a field of this object using the *this* keyword:

```
class MyClass {
   String field = "I am an object field";
   String getField() {
      // read a field
      return this.field;
   }
   void setField() {
      // write a field
      this.field = "I have been changed";
   }
}
```

If the field is not shadowed by any local variable, the *this.* prefix may be left out:

```
class MyClass {
   String field = "I am an object field";
   String getField() {
      return field;
   }
   void setField() {
      field = "I have been changed";
   }
}
```

b. Outside the object

An object field can be accessed and changed from outside through a variable referencing the object like the following:

```
MyClass myObject = new MyClass();
// read a field
String myField = myObject.getField();
// write a field
myObject.setField();
```

4. Explain an Object's Lifecycle (creation, "dereference by reassignment" and garbage collection)

In Java, all objects must go through two phases: creation and destruction.

a. Creation

The creation of an object includes two steps: instantiation and initialization.

Code example:

Given a class declaration:

```
class MyClass {
   int number = 0;
   MyClass(int number) {
      this.number = number;
   }
}
```

This class can be instantiated and initialized with the following statement:

```
MyClass myObject = new MyClass(1);
```

The *new* keyword is a Java operator that instantiates the class, creating a new object; while the constructor initializes states of the created object, or you can say the constructor

initializes the object.
An object is eligible for garbage collection when one of two situations occurs:
1. The object no longer has any references pointing to it.
2. All references to the object have gone out of scope.

Java allows objects to implement a method called finalize() that might get called.
Syntax :
protected void finalize() throws Throwable
finalize() is only run when the object is eligible for garbage collection. If the garbage collector fails to collect the object and tries to run it again later, the method doesn't get called a second time.

b. Destruction

After being used, an object must be destroyed to release the memory resource. However, you cannot explicitly destroy an object immediately. Instead, this is done automatically by JVM using garbage collection.
An object is eligible for garbage collection when it is unreferenced. This means it is no longer referenced by any part of the program. You can make an object unreferenced by dereferencing it using a technique called reassignment.

Code example:

Assume you have an object referenced by variable *myObject*:

```
MyClass myObject = new MyClass();
```

When you reattach the *myObject* variable to another expression:

```
myObject = null;
```

The object in the heap that was originally referenced by the *myObject* variable becomes unreferenced, and the memory space it occupies will then be freed by the garbage collector.

5. Develop code that uses wrapper classes such as Boolean, Double, and Integer

a. Boolean

```
Boolean bool = Boolean.logicalAnd(false, true);
int compare = bool.compareTo(false);
System.out.println(compare);
```

The above code fragment prints out number *0* since the first statement applies the logical *AND* operator to the *false* and *true* operands. The result is *false*, which is then assigned to variable *bool*. The *bool* variable is thus equal to the *false* argument passed to the *compareTo* method, resulting in number *0* getting printed.

b. Double

```
Double d = Double.sum(1.2, 2.1);
boolean check = d.isFinite(d);
System.out.println(check);
```

The above code fragment prints out *true* since the *d* variable, which is the sum of two finite *double* values, is finite.

c. Integer

```
Integer max = Integer.MAX_VALUE;
String base16 = max.toHexString(max);
System.out.println(base16);
```

The above code fragment prints out the hexadecimal number *7FFFFFFF* since the maximum value an integer can have is *2^31 - 1*, which is equivalent to *7FFFFFFF* in base 16.

d. Other wrapper classes

They are *Byte*, *Short*, *Long*, *Float* and *Character*.

Objective 3: Using Operators and Decision Constructs

1. Use Java operators; use parentheses to override operator precedence

a. Operators

The Java programming language introduces various operators that are listed below per precedence order:

- Postfix: *expr++ expr--*
- Unary: *expr++ expr-- ++expr --expr +expr -expr ~ !*
- Multiplicative: ** / %*
- Additive: *+ -*
- Shift: *<< >> >>>*
- Relational: *< > <= >= instanceof*
- Equality: *== !=*
- Bitwise AND: *&*
- Bitwise exclusive OR: *^*
- Bitwise inclusive OR: *|*
- Logical AND: *&&*
- Logical OR: *||*
- Ternary: *? :*
- Assignment: *= += -= *= /= %= &= ^= |= <<= >>= >>>=*

Numeric Promotion Rules :

1. If two values have different data types, Java will automatically promote one of the values to the larger of the two data types.
2. Smaller data types, namely byte, short, and char, are first promoted to int any time they're used with a Java binary arithmetic operator, even if neither of the operands is int.
3. After all promotion has occurred and the operands have the same data type, the resulting value will have the same data type as its promoted operands.

b. Overriding operator precedence using parenthesis

The operator precedence can be overridden using parentheses as illustrated in the examples below.

Code example 1:

```
int integer1 = 1 + 2 * 3;
int integer2 = (1 + 2) * 3;
System.out.println(integer1);
System.out.println(integer2);
```

The above code fragment prints out number *7* and then *9*.

Code example 2:

```
boolean boolean1 = true || true && false;
boolean boolean2 = (true || true) && false;
System.out.println(boolean1);
System.out.println(boolean2);
```

The above code fragment prints out *true* and then *false*.

2. Test equality between Strings and other objects using == and equals ()

One important thing to remember is that the == operator compares objects by reference, whereas the *equals* method compares based on how it is overridden in the classes of objects in comparison.

a. String literal comparison

Given a code fragment:

```
String string1 = "I am a string";
String string2 = "I am a string";
System.out.println(string1 == string2);
System.out.println(string1.equals(string2));
```

The above fragment prints out *true* and *true*. The reason is that with *String* instances created using *String* literals, the compiler ensures only a single instance exists in the *String* constant pool for each *String* value, resulting in both variables *string1* and *string2* referencing the same instance. As such, the comparison using the == operator evaluates to true, so is the comparison using the *equals* method.

b. String object comparison

Given a code fragment:

```
String string1 = new String("I am a string");
String string2 = new String("I am a string");
System.out.println(string1 == string2);
System.out.println(string1.equals(string2));
```

The above fragment prints out *false* and *true*. The reason is that each time the *String* constructor is invoked, a new *String* instance is created in the heap. The *string1* and *string2* variables therefore point to two different instances, causing the comparison using the == operator to evaluate to *false*. Meanwhile, the *equals* method is overridden in the *String* class, making it return *true* when comparing two *String* objects having the same value.

c. Object comparison

The == operator and *equals* method work on objects of other types the same ways as they do on *String* objects.

Given a class declaration:

```
class Data {
   int value = 0;
   Data(int value) {
      this.value = value;
   }
   public boolean equals(Object obj) {
      if (obj == null || getClass() != obj.getClass()) {
         return false;
      }
      return (value == ((Data) obj).value);
   }
}
```

Executing the following code fragment produces *false* and *true* on the console:

```
Data data1 = new Data(1);
Data data2 = new Data(1);
System.out.println(data1 == data2);
```

```
System.out.println(data1.equals(data2));
```

The *data1* and *data2* variables point to different objects, the first comparison thus evaluates to *false*. The second evaluates to *true* since the *equals* method is overridden to compare two *Data* instances based on its enclosed value. Should the *equals* method not be overridden, both comparisons produce the *false* value.

3. Create if and if/else and ternary constructs

a. The if-then construct

The *if-then* construct tells your program to execute a certain section of code only if a particular test evaluates to *true*.

Code example:

```
int i = // an int expression
if (i > 0) {
   System.out.println("Variable i is holding a positive value");
}
```

The string *"Variable i is holding a positive value"* gets printed only if *i* is a positive number. Note that if the *then* block contains only one statement, like in the above fragment, the enclosing curly brackets may be left out.

b. The if-then-else construct

The *if-then-else* construct provides a secondary path of execution to *if-then* when the *if* clause evaluates to *false*.

Code example:

```
int i = // an int expression
if (i > 0) {
   System.out.println("i is a positive number");
} else {
   System.out.println("i is a non-positive number");
}
```

When *i* is a positive number, the string *"i is a positive number"* is printed; otherwise, *"i is a non-positive number"* gets printed. Similar to the *if-then* construct, if the *then* or *else* block contains only one statement, the enclosing curly brackets may be left out.

c. Ternary operator

The ternary operator is shorthand for an *if-then-else* construct with the following syntax:

```
condition ? value1 : value2
```

The above expression should be read as "if the condition is true, the expression evaluates to value1; otherwise, it evaluates to value2".

Code example:

```
int i = // an int expression
String output = i > 0 ? "Positive" : "Non-positive";
System.out.println(output);
```

The above code fragment prints out *"Positive"* if *i* is greater than *0*, *"Non-positive"* otherwise.

4. Use a switch statement

The *switch* statement is a control flow statement that can have several possible execution paths. A *switch* works with the *byte*, *short*, *char*, *int* primitive data types and their wrapper classes, with *enum* types and the *String* class.

The body of a *switch* statement is known as a *switch* block. A statement in the *switch* block can be labeled with one or more cases or default labels. The *switch* statement evaluates its expression, then executes all statements that follow the matching *case* label.
Given a code example:

```
int i = // an int expression
switch (i) {
case -1:
   System.out.println("Negative");
   break;
case 1:
   System.out.println("Positive");
   break;
default:
   System.out.println("Zero");
}
```

The above fragment prints *"Positive", "Negative"* or *"Zero"* if *i* is greater than, less than or equal to *0*, respectively.
The *break* statements are there to prevent the control from falling through. If these statements are removed, all statements after the matching *case* label are executed in sequence.
Switch doesn't allow boolean , long , float, double data types. The values in each case statement must be compile-time constant values of the same data type as the switch value. There is no requirement that the case or default statements be in a particular order.

Objective 4: Creating and Using Arrays

1. Declare, instantiate, initialize and use a one-dimensional array

a. Declaration

An array declaration has two components: the array's type and the array's name. An array's type is written as *type[]*, where *type* is the data type of the contained elements; the empty square brackets are special symbols indicating that this variable holds an array.
Example: *int[] intArray; String[] stringArray; MyObject[] objectArray;*

b. Instantiation without initialization

An array can be instantiated with the *new* operator, followed by the type of elements and the array length enclosed within square brackets.
Example: *intArray = new int[5]; stringArray = new String[10]; objectArray = new Object[20];*

c. Initialization after instantiation

You may initialize an array by assigning values to its elements. This can be done by setting one-by-one, using a loop construct, copying from other arrays, etc.

Code example

```
int[] intArray = new int[5];
for (int i = 0; i < 5; i++) {
   intArray[i] = i;
}
```

Note that the index of array elements starts with *0*.

d. Using array initializers

Alternatively, an array can be instantiated and initialized at the same time using an array initializer. You may list elements of the array within curly brackets when instantiating it as follows:

intArray = {1, 2, 3, 4, 5};

An array initializer may also go with the array type:

intArray = new int[] { 1, 2, 3, 4, 5 };

e. Size of Array :

You can find size of an array using "length" property. For 1D Array, it returns number of elements. For 2D Array, it returns number of

Ex:

```
int[] nums1 = { 4, 6, 8 };
        int[][] nums2 = { { 4, 7, 3 }, { 9, 8, 1 } };
        int[] nums3 = new int[5];
        nums3[0]=2;
        System.out.println(nums1.length);//prints 3
        System.out.println(nums2.length); //prints 2
        System.out.println(nums3.length); //prints 5
```

f. Printing Array Elements :

Using for loop :

```
int[] nums1 = { 4, 6, 8 };
        for(int i=0;i<nums1.length;i++){
                System.out.print(nums1[i]+" "); //prints 4 6 8
        }

int[][] nums1 = { { 4, 6, 8 }, { 7, 9, 4 } };
        for (int i = 0; i < nums1.length; i++) {
                for (int j = 0; j < nums1[i].length; j++) {
                        System.out.print(nums1[i][j] + " "); // prints 4 6 8 7 9 4
                }
        }
```

g. Using for-each :

```
int[] nums1 = { 4, 6, 8 };
        for(int x :nums1){
                System.out.print(x+" "); //prints 4 6 8
        }


int[][] nums1 = { { 4, 6, 8 }, { 7, 9, 4 } };
        for (int x[] : nums1) {
                for (int y : x) {
                        System.out.print(y + " "); // prints 4 6 8 7 9 4
```

```
                }
            }
```

2. Declare, instantiate, initialize and use multi-dimensional arrays

In the Java programming language, a multi-dimensional array is an array whose components are themselves arrays.

a. Declaration

Similar to the declaration of a one-dimensional array, a multi-dimensional array is declared with the type and name of the array. An array's type is written as *type[]...[]*, where *type* is the data type of the contained elements and the number of square brackets reflects the number of dimensions.

Example: *int[][] intArray; String[][][] stringArray; MyObject[][][][] objectArray;*

b. Instantiation without initialization

An array can be instantiated with the *new* operator, followed by the type of elements and several square brackets. Each couple of brackets represents a dimension. The first bracket must contain the length of the top nesting level, while the length of other levels is not required provided no empty brackets are followed by non-empty ones.

Valid instantiation example:

intArray = new int[2][3][4]; stringArray = new String[2][][]; objectArray = new Object[2][][];

Invalid instantiation example:

intArray = new int[2][][4]; stringArray = new String[][][4]; objectArray = new Object[][3][4];

c. Initialization after instantiation

You may initialize a multi-dimensional array the same way as with a one-dimensional array, except that the assignment must be done in multiple levels.

Code example:

```
int[][] intArray = new int[3][5];
for (int i = 0; i < 3; i++) {
   for (int j = 0; j < 5; j++) {
      intArray[i][j] = i + j;
   }
}
```

d. Using array initializers

An array can also be instantiated and initialized at the same time using an array initializer. You may list elements of the array within curly brackets when instantiating it as follows:

int[][] intArray = { { 1, 2, 3 }, { 4, 5 } };

An array initializer may go with the array type:

int[][] intArray = new int[][] { { 1, 2, 3 }, { 4, 5 } };

Objective 5: Using Loop Constructs

1. Create and use while loops

The *while* statement continually executes a block of statements while a particular condition is *true*. Its syntax can be expressed as:

```
while (condition) {
   statement(s)
}
```

Code example:

```
int i = 0;
while (i < 10) {
   System.out.println(i);
   i++;
}
```

The above code fragment prints integral numbers from *0* to *9* on the console.

2. Create and use for loops including the enhanced for loop

a. The for loop

The *for* loop provides a compact way to iterate over a range of values until a particular condition is satisfied. The general form of this construct can be expressed as follows:

```
for (initialization; termination; increment/decrement) {
   statement(s)
}
```

Notes:

- The initialization expression is executed once, as the loop begins
- When the termination expression evaluates to *false*, the loop terminates
- The increment/decrement expression is invoked after each iteration through the loop
- All three expressions of the for loop are optional

Code example:

```
for (int i = 1; i <= 10; i++) {
   System.out.println("This is iteration " + i);
}
```

The above fragment prints out the phrases *"This is iteration <iteration_order>"* 10 times.

b. The enhanced for loop

The *enhanced for* loop is a form of the *for* construct designed for iteration through collections and arrays. Its syntax is given below:

```
for (type element : collection/array) {
   statement(s);
}
```

Code example:

```
int[] intArray = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
for (int element : intArray) {
   System.out.println(element);
}
```

The above fragment prints all integers from *1* to *10*.

3. Create and use do/while loops

The *do-while* loop continually executes a block of statements while a particular condition is *true*, with syntax being given below:

```
do {
   statement(s)
} while (condition);
```

Code example:

```
int i = 0;
do {
   System.out.println(i);
```

```
    i++;
} while (i < 10);
```

The above code fragment prints integral numbers from *0* to *9* on the console.

4. Compare loop constructs

- The *while* and *do-while* statements both continually execute a block of statements while a particular condition is *true*. The difference between *do-while* and *while* is that *do-while* evaluates its expression at the bottom of the loop instead of the top. Therefore, the statements within the *do* block are always executed at least once.
- The *for* statement provides a compact way to iterate over a range of values. It has two forms, one of which was designed for looping through collections and arrays.

5. Use break and continue

a. The break statement

The *break* statement terminates an enclosing *switch* or loop construct. It has two forms: unlabeled and labeled. The unlabeled form statement terminates the innermost construct, while the labeled may terminate an outer one.

Unlabeled break example:

```
int i = 0, j = 0;
for (i = 1; i <= 5; i++) {
    j = 1;
    while (j <= 5) {
        if (j % 2 == 0) {
            break;
        }
        j++;
    }
}
System.out.println(i + " " + j);
```

The above code fragment prints out numbers *6* and *2* since the *break* statement terminates the *while* loop whenever *j* reaches *2*. As such, number *2* is the final value of this variable. On the other hand, the statement does not make any impact on the *for* loop, allowing *i* to go to its highest possible value.

Labeled break example:

```
int i = 0, j = 0;
myLabel: for (i = 1; i <= 5; i++) {
    j = 1;
    while (j <= 5) {
        if (j % 2 == 0) {
            break myLabel;
        }
        j++;
    }
}
System.out.println(i + " " + j);
```

The above code fragment prints out numbers *1* and *2*. The reason is that this time, the *break* statement terminates the for loop labeled with *myLabel*. When *j* reaches 2 in the first iteration of the *for* loop, the *for* loop is terminated, leaving *i* no chance to go beyond number *1*.

b. The continue statement

The *continue* statement instructs the program to skip the current iteration of an enclosing loop construct. It has two forms: unlabeled and labeled. The unlabeled form skips the innermost loop and evaluates the boolean expression that controls the loop, while the labeled may skip an outer one.

Unlabeled continue example:

```
int i = 0, j = 0;
for (i = 1; i < 3; i++) {
   j = 1;
   while (j < 5) {
      j++;
      if (j % 2 == 0) {
         continue;
      }
      System.out.println("I survived the termination");
   }
}
```

The above code fragment prints *"I survived the termination"* four times. Each iteration of the *for* loop results in two printing operations, when *j* increases to *3* and *5*. The *continue* statement makes the program skip the printing actions when *j* reaches *2* and *4*.

Labeled continue example:

```
int i = 0, j = 0;
myLabel: for (i = 1; i < 3; i++) {
   j = 1;
   while (j < 5) {
      j++;
      if (j % 2 == 0) {
         continue myLabel;
      }
      System.out.println("I survived the termination");
   }
}
```

The above code fragment does not print the provided string even once. The reason is that every iteration of the *for* loop is skipped after the first increment of *j*, making the printing statement unreachable.

Objective 6: Working with Methods and Encapsulation

1. Create methods with arguments and return values; including overloaded methods

a. Method creation

Here is a method that takes two *int* arguments and returns their sum:

```
int sum(int arg1, int arg2) {
   System.out.println("This method adds up two integers");
   return arg1 + arg2;
}
```

A method may take zero, one or more arguments, does something, and then returns a value.

b. Method overloading

Here is another method with the same name as the one shown above:

```
float sum(float arg1, float arg2) {
   System.out.println("This method adds up two floating-point numbers");
   return arg1 + arg2;
}
```

Although both methods have the same name, JVM is still able to distinguish them based on the differences of parameter types. Methods with the same name, but different parameters, are called overloading methods.

c. Differentiation of overloading methods

Assume both *sum* methods introduced above are declared within the same class. The following invocation:

```
sum(1, 2);
```

will print *"This method adds up two integers"* and receive the integer *3*.

Invoking the *sum* method with different arguments:

```
sum(1.2F, 2.1F);
```

will print *"This method adds up two floating-point numbers"* and receive the floating-point number 3.3F.

It can be seen that JVM knows which overloading method must be invoked based on the passed-in arguments.

In method overloading, Order Java uses to choose the right overloaded method as follows.

1. Exact match by type
2. Larger primitive type
3. Autoboxed type
4. Varargs

2. Apply the static keyword to methods and fields

a. Static variables

Fields that have the *static* modifier in their declaration are called static fields, also called class variables. They are associated with the class itself. Every instance of the class shares class variables. Any object can change the value of class variables, but class variables can also be manipulated without creating an instance of the class.

Code example:

Given a class:

```
class Data {
   int instanceValue = 0;
   static int staticValue = 0;
}
```

Let's see how these variables are manipulated:

```
Data data1 = new Data();
Data data2 = new Data();
data1.instanceValue = 1;
data1.staticValue = 1;
System.out.println(data2.instanceValue);
System.out.println(data2.staticValue);
```

The above fragment prints out *0* and *1*. It is apparent that *staticValue* is shared among class instances, while *instanceValue* is not.

b. Static methods

The Java programming language supports static methods as well as static variables. Static methods, which have the *static* modifier in their declarations, should be invoked with the class name, without the need for creating an instance of the class:

ClassName.methodName(args)

Static methods cannot access instance variables or instance methods directly. They are thus usually used to access and mutate static fields.

3. Create and overload constructors; differentiate between default and user defined constructors

a. Create constructors

Constructors are created in a similar way as methods, except that they use the name of the class and have no return type.

Code example:

```
class Person {
   String firstName;
   String lastName;
   Person(String firstName, String lastName) {
      this.firstName = firstName;
      this.lastName = lastName;
   }
}
```

In the above fragment, *Person(String firstName, String lastName)* is a constructors. Invoking it with two *String* arguments will instantiate the class and initialize the newly created instance.

b. Overload constructors

You may declare other constructors for the *Person* class introduced above. E.g.:

```
Person(String lastName) {
   this.firstName = "NoFirstName";
   this.lastName = lastName;
}
```

Both constructors, *Person(String firstName, String lastName)* and *Person(String lastName)*, can be declared in *Person* because they have different argument lists. JVM differentiates constructors on the basis of the number of arguments in the list and their types. Writing two constructors that have the same number and type of arguments for the same class will cause a compile-time error.

c. Default constructors

Both *Person* constructors given above are user defined. In case you do not explicitly declare any constructor for a class, the compiler will automatically provide a no-argument, default one. This default constructor will call the no-argument constructor of the superclass.

You may override behavior of the default constructor by declaring a constructor with the exact same signature:

```
MyClass() {
   // do something
}
```

4. Apply access modifiers

a. Introduction

Access level modifiers determine whether other classes can use a particular field or invoke a particular method. There are two levels of access control:

- Type level: *public*, or package-private (no explicit modifier)
- Member level: *public*, *private*, *protected*, or package-private (no explicit modifier)

b. Explanation of modifiers

- *public:* visible to all classes in the application
- *default*, also known as package-private: visible only within its own package
- *protected*: can only be accessed within its own package (as with package-private) and, in addition, by subclasses of its class in other packages
- *private*: can only be accessed in its own class

5. Apply encapsulation principles to a class

Encapsulation is a mechanism of wrapping the data (variables) and code acting on the data (methods) together as a single unit. In encapsulation, variables of a class are hidden from other classes, and can only be accessed through methods of their enclosing class.

In order to achieve encapsulation, you may declare variables of a class with the *private* access modifier and provide *public* methods to access and mutate these variables.

Code example:

```
public class Data {
   private int value;
   public int getValue() {
      return this.value;
   }
   public void setValue(int value) {
      this.value = value;
   }
}
```

6. Determine the effect upon object references and primitive values when they are passed into methods that change the values

a. Passing primitive values

Assume you have a method that change the input value:

```
void changeValue(int value) {
   value = 2 * value;
}
```

Consider the following code fragment:

```
int i = 1;
changeValue(i);
System.out.println(i);
```

The above code prints out number *1*, implying that the change made in the method body does not have any effect on the original argument. This is an example of pass-by-value, applying to arguments of primitive data types.

b. Passing object references

Given a class declaration:

```
class Data {
```

```
    int value;
    Data(int value) {
        this.value = value;
    }
}
```

A method is declared to change value of a *Data* instance:

```
void changeValue(Data data) {
    data.value = 2 * data.value;
}
```

Now, execute the following fragment:

```
Data data = new Data(1);
changeValue(data);
System.out.println(data.value);
```

The above code fragment prints out number *2*, meaning that the change made in the method body is reflected in the original argument. So Java always uses pass by value.

c. Explanation

The difference in behaviors of pass-by-value and pass-by-reference results from the fact that an object variable does not store the object itself. Instead, it keeps the address where the object is located in the memory. Passing an object is, therefore, copying the address stored in the argument to the parameter. Both the argument and the parameter then point to the same object and any changes made to the parameter is reflected in the argument.

On the other hand, primitive variables store values rather than addresses. When passing a primitive value to a method, the value is copied from the argument to the parameter. As such, the argument and the parameter are independent of each other, although they have the same value right before the method is executed.

Objective 7: Working with Inheritance

1. Describe inheritance and its benefits

a. Description

Inheritance is a mechanism allowing one type to acquire fields and methods of another. The type that inherits members from the other is called subtype and the type whose members are inherited is called supertype.

b. Benefits

Inheritance allows members of a supertype to be reused in all of its subtypes, while still lets subtypes declare their own fields and methods. Note that a supertype can be inherited by any number of subtypes, but a subtype can only inherit from a supertype.

Code example:

Given two classes:

```
public class SuperClass {
    String field = "I am a field of supertype";
    void print() {
        System.out.println(field);
    }
}
class SubClass extends SuperClass { }
```

Let's run the following fragment:

```
SubClass object = new SubClass();
object.print();
```
The above code prints *"I am a field of supertype"* on the console, although *SubClass* does not define any *print* method of its own. This method is, in fact, inherited from *SuperClass*.

2. Develop code that makes use of polymorphism; develop code that overrides methods; differentiate between the type of a reference and the type of an object

a. Polymorphism

In the Java programming language, polymorphism is the ability for a method to behave differently based on the actual objects it is invoked on.

Code example:

Given three classes:

```
class MyClass {
   void print() {
      System.out.println("Printed from MyClass");
   }
}
class SubClass1 extends MyClass {
   void print() {
      System.out.println("Printed from SubClass1");
   }
}
class SubClass2 extends MyClass {
   void print() {
      System.out.println("Printed from SubClass2");
   }
}
```

Let's execute the following code fragment:

```
MyClass obj1 = new MyClass();
MyClass obj2 = new SubClass1();
MyClass obj3 = new SubClass2();
obj1.print();
obj2.print();
obj3.print();
```

The above fragment prints out *"Printed from MyClass"*, *"Printed from SubClass1"*, and *"Printed from SubClass2"* in sequence, even though all variables *obj1*, *obj2* and *obj3* are of the same type. This discrepancy in method behaviors is called polymorphism.

1. The method in the sub class must have the same signature as the method in the super class.
2. The method in the sub class must be at least as accessible or more accessible than the method in the super class.
3. The method in the sub class may not throw a checked exception that is new or broader than the class of any exception thrown in the super class method.
4. If the method returns a value, it must be the same or a subclass of the method in the parent class, known as covariant return types.

b. Method overriding

Method overriding is a mechanism when an instance method in a subtype with the same name and return type as an instance method in a supertype replaces the behavior of the inherited method with its own.

Code example:

Given two classes:

```
class SuperClass {
   void print() {
      System.out.println("Printed from SuperClass");
   }
}
class SubClass extends SuperClass {
   void print() {
      System.out.println("Printed from SubClass");
   }
}
```

When the following code fragment:

```
SubClass object = new SubClass();
object.print();
```

is executed, the phrase *"Printed from SubClass"* gets printed out. You can say that the *print* method in *SubClass* overrides the *print* method in *SuperClass*.

c. Reference types vs object types

When a variable references an object, the type of the variable is specified when it is declared and this type can never be changed. At the same time, the variable may be assigned another object with a different type.

Let's have a look at an example to better understand the distinction between reference type and object type:

```
Number number = new Integer(0); //1
number = new Float(0.0F); //2
```

The *number* variable is always of type *Number*. This is reference type. At line *//1*, *number* points to an instance of type *Integer*, and to an instance of type *Float* after the reassignment at line *//2*. *Integer* and *Float* in this case are object types.

Reference and object are two different entities. The reference is a variable that has a name and can be used to access the contents of an object. A reference can be assigned to another reference, passed to a method, or

returned from a method. All references are the same size, no matter what their type is. An object sits on the heap and does not have a name. Therefore, you have no way to access an object except through a reference. Objects come in all different shapes and sizes and consume varying amounts of memory. An object cannot be assigned to another object, nor can an object be passed to a method or returned from a method. It is the object that gets garbage collected, not its reference.

3. Determine when casting is necessary

When an object type is in the same type hierarchy with the type of a variable, the object is assignable to the variable. It is simple if the object and variable are of the same type:

```
Integer integer;
```

integer = new Integer(0);

You may have known that *Integer* is a subtype of *Number*, and of *Object* as well. Thus, you can say *integer IS A Number*, and also *integer IS A*(n) *Object*. This allows the object referenced by variable *integer* to be reassigned to variables of type *Number* and *Object*:

```
Integer integer = new Integer(0);
Number number = integer;
Object object = integer;
```

It can be seen that if the object type is a subtype of the variable type, it is assigned as if the object and variable types were the same.

The story becomes more complicated if the object type is a supertype of the variable type. Consider a code fragment:

```
Integer integer = new Integer(0);
Object object = integer;
integer = object; //3
```

Line *//3* looks valid as both *integer* and *object* are pointing to the same instance and the statement at line *//3* just reassigns the *integer* variable the same object. However, the compiler just cares about the declared type of an object instead of its actual type. The *object* variable, an instance of *Object*, may be an *Integer*, but not necessarily. This absurd can be avoided with the help of type casting:

```
integer = (Integer) object;
```

Conclusion: If the type of an object is the same or a subtype of the variable type, the object can be directly assigned to the variable; on the other hand, if the object type is a supertype of the variable type, type casting must be used.

4. Use super and this to access objects and constructors

a. The this keyword

Within an instance method or a constructor, *this* is a reference to the enclosing object. You can refer to any member of the current object from within an instance method or a constructor by using *this*.

Using this with fields or methods

Given a class:

```
class MyClass {
   String field = "I am an instance field";
   void print(String string) {
      System.out.println(string);
   }
   void execute() {
      String field = "I am a local variable";
      this.print(this.field); //8
   }
}
```

The *execute* method refers to the *field* and *print* instance members of *MyClass* using *this* and prints out *"I am an instance field"* when invoked.

The most common reason for using *this* is to access a shadowed instance field. In the above example, if the *this* keyword is removed from line *//8*, the phrase *"I am a local variable"* will be printed instead.

Using this with constructors

From within a constructor, you can use the *this* keyword to call another constructor in the same class, like the following:

```
class Person {
   String name;
   Person() {
      this("NoName");
   }
   Person(String name) {
      this.name = name;
   }
}
```

Important constructor rules :

1. The first statement of every constructor is a call to another constructor within the class using this(), or a call to a constructor in the direct parent class using super().
2. The super() call may not be used after the first statement of the constructor.
3. If no super() call is declared in a constructor, Java will insert a no-argument super() as the first statement of the constructor.
4. If the parent doesn't have a no-argument constructor and the child doesn't define any constructors, the compiler will throw an error and try to insert a default no-argument constructor into the child class.
5. If the parent doesn't have a no-argument constructor, the compiler requires an explicit call to a parent constructor in each child constructor.

b. The super keyword

Accessing superclass members

The *super* keyword can be used within a subclass to refer to hidden fields or overridden methods in a superclass:

```
class SuperClass {
   String string = "I am in SuperClass";
   void print(String string) {
      System.out.println(string + " - printed from SuperClass");
   }
}
class SubClass extends SuperClass {
   String string = "I am in SubClass";
   void print(String string) {
      System.out.println(string + " - printed from SubClass");
   }
   void execute() {
      super.print(super.string);
   }
}
```

The string *field* and *print* method of *SuperClass* are hidden and overridden by members of the same name in *SubClass*. These *SuperClass* members can be accessed with the help of the *super* keyword.

The invocation of the *execute* method causes *"I am in SuperClass - printed from SuperClass"* to be printed out. If the *super* keyword is removed, method *execute* will *print "I am in SubClass - printed from SubClass"* instead.

Invoking superclass constructors

From within a constructor of a subclass, you can use the *super* keyword to call a constructor in a superclass:

```
class SuperClass {
   String string;
   SuperClass() {
      this.string = "I am a string";
   }
}
class SubClass extends SuperClass {
   SubClass() {
      super();
      System.out.println(string);
   }
}
```

When you instantiate the *SubClass* class, the phrase *"I am a string"* will be printed out on the console as the *string* field has been initialized in the *SuperClass* constructor.

Notes:

- The invocation of a superclass constructor must be the first line in the subclass constructor.
- If a constructor does not explicitly invoke a superclass constructor, the compiler automatically inserts a call to the no-argument constructor of the superclass.

5. Use abstract classes and interfaces

a. Comparison

- Similarities: Abstract classes are similar to interfaces. You cannot instantiate them, and they may contain a mix of methods declared with or without an implementation.
- Differences: With abstract classes, you can declare fields that are not static and final, and define public, protected, and private concrete methods. With interfaces, all fields are automatically public, static, and final, and methods are public. In addition, you can extend only one class, whether it is abstract, whereas you can implement any number of interfaces.

b. Abstract class use cases

- You want to share code among several closely related classes.
- You expect that classes that extend your abstract class have many common methods or fields, or require access modifiers other than *public*.
- You want to declare non-static or non-final fields. This enables you to define methods that can access and modify the state of the object to which they belong.

c. Interface use cases

- You expect that unrelated classes would implement your interface.
- You want to specify the behavior of a particular data type, but not concerned about who implements its behavior.
- You want to take advantage of multiple inheritance of type.

Static Interface Methods :
Java 8 also now includes support for static methods within interfaces. Like all methods in an interface, a static method is assumed to be public and will not compile if marked as private or protected. To reference the static method, name of the interface must be used.
Default Interface Methods :
A default method is a method defined within an interface with the default keyword in which a method body is provided.
The following are the default interface method rules you need to be familiar with:
1. A default method may only be declared within an interface and not within a class or abstract class.
2. A default method must be marked with the default keyword. If a method is marked as default, it must provide a method body.
3. A default method is not assumed to be static, final, or abstract, as it may be used or overridden by a class that implements the interface.
4. Like all methods in an interface, a default method is assumed to be public and will not compile if marked as private or protected.

Objective 8: Handling Exceptions

1. Differentiate among checked exceptions, unchecked exceptions, and Errors

a. Checked exceptions

Checked exceptions are exceptional conditions that a well-written application should anticipate and recover from. These are subject to the *Catch or Specify Requirement*, meaning that code that might throw a checked exception must be enclosed by either of the following:

- A *try* block that catches the exception, followed by a *catch* or a *finally* block or both that handles the exception
- A method that specifies that it can throw the exception; the method must provide a *throws* clause that lists the exception

All exceptions are checked exceptions, except for those indicated by *Error*, *RuntimeException*, and their subclasses.

b. Unchecked exceptions

Unchecked exceptions are exceptional conditions that are internal to the application, and that the application usually cannot anticipate or recover from. These usually indicate programming bugs, such as logic errors or improper use of an API.
Runtime exceptions are indicated by *RuntimeException* and its subclasses.

c. Errors

Errors are exceptional conditions that are external to the application, and that the application usually cannot anticipate or recover from. For example, *OutOfMemoryError* occurs when JVM runs out of memory or *StackOverflowError* occurs when stack overflows.
Errors are represented by *Error* and its subclasses.

2. Create a try-catch block and determine how exceptions alter normal program flow

Given a code fragment:

```
try {
   // do something
   throw new Exception("I am an exception");
} catch(Exception e) {
```

```
    System.out.println(e.getMessage());
}
System.out.println("Normal program flow");
```

The above code prints out *"I am an exception"* and *"Normal program flow"* when executed. The exception message gets printed since the thrown exception directs the program flow to the *catch* block after *try* block. If the *throw* statement is removed, the program will not visit the *catch* block and the exception message will not be printed.

3. Describe the advantages of Exception handling

Exception handling is used to recover an application from a failure and maintain normal workflow of the application.

Take a look at a method declaration:

```
void myMethod() {
    int[] array = new int[5];
    try {
        for (int i = 0; i <= 5; i++) {
            array[i] = i;
        }
    } catch (Exception e) {
        System.out.println("Something wrong happened");
    }
    for (int element : array) {
        System.out.println(element);
    }
}
```

Due to a bug when initializing the array, an *ArrayIndexOutOfBoundsException* is thrown. This exception is handled by the *catch* block and the program continues its normal workflow after a message is printed on the console. All elements of the array are printed out when the last statement is executed.

Without exception handling in the *myMethod* method, two possibilities may occur:

- The exception is thrown up further the call stack and handled somewhere else. The program continues without executing remaining statements in the *myMethod* method. Unexecuted statements may lead to unexpected consequences.
- The exception is never caught, resulting in the program crashing.

Advantages of Exceptions:

1. Separating Error-Handling Code from "Regular" Code
2. Propagating Errors Up the Call Stack
3. Grouping and Differentiating Error Types

4. Create and invoke a method that throws an exception

a. Methods throwing checked exceptions

Given a method that throws a checked exception:

```
void checkedExceptionMethod() throws IOException {
    // do something
    throw new IOException();
}
```

When the above method is called by another, the calling method must either surround the invocation with a *try-catch* block:

```
void anotherMethod() {
   try {
      checkedExceptionMethod();
   } catch (IOException e) {
      e.printStackTrace();
   }
}
```

or specify the exception in a *throws* clause to re-throw the exception further up the call stack:

```
void anotherMethod() throws IOException {
   checkedExceptionMethod();
}
```

b. Methods throwing unchecked exceptions

Here is a method that throws an unchecked exception:

```
void uncheckedExceptionMethod() throws RuntimeException {
   // do something
   throw new RuntimeException();
}
```

There are no requirements on calling methods of the above. These callers may be declared as if the called method did not specify any exception:

```
void anotherMethod() {
   uncheckedExceptionMethod();
}
```

5. Recognize common exception classes (such as NullPointerException, ArithmeticException, ArrayIndexOutOfBoundsException, ClassCastException)

a. NullPointerException

Thrown when an application attempts to use *null* where an object is required, e.g.:

- Calling the instance method of a *null* object
- Accessing or modifying the field of a *null* object
- Taking the length of *null* as if it were an array
- Accessing or modifying the slots of *null* as if it were an array
- Throwing *null* as if it were a *Throwable* value

b. ArithmeticException

Thrown when an exceptional arithmetic condition has occurred. For example, an integer "divide by zero" throws an instance of this class:

```
int i = 0;
int j = 1 / i;
```

c. ArrayIndexOutOfBoundsException

Thrown to indicate that an array has been accessed with an illegal index. The index is either negative or greater than or equal to the size of the array. Example:

```
int[] array = { 1, 2, 3, 4 };
int i = array[4];
```

d. ClassCastException

Thrown to indicate that the code has attempted to cast an object to a subclass of which it is not an instance. Example:

```
Number number = new Integer(0);
Float floatingPoint = 2 * (Float) number;
```

Objective 9: Working with Selected classes from the Java API

1. Manipulate data using the StringBuilder class and its methods

a. Append

An *append* overloading method appends the string representation of the argument to this sequence.

Code example:

```
StringBuffer stringBuffer = new StringBuffer("OCA");
stringBuffer.append("JP");
System.out.println(stringBuffer);
```

The above code fragment prints out *"OCAJP"* when executed.

b. Insert

An *insert* overloading method inserts the string representation of the argument into this sequence at a specified position.

Code example:

```
StringBuffer stringBuffer = new StringBuffer("OJP");
stringBuffer.insert(1, "CA");
System.out.println(stringBuffer);
```

The above code fragment also prints out *"OCAJP"* when executed.

c. Replace

The *replace* method replaces the characters in a substring of this sequence with characters in the specified *String*.

Code example:

```
StringBuffer stringBuffer = new StringBuffer("OCAJP");
stringBuffer.replace(2, 3, "P");
System.out.println(stringBuffer);
```

The above code fragment prints out *"OCPJP"* when executed.

d. Delete

Character(s) of this sequence may be removed using the *delete* or *deleteCharAt* method.

- *delete(int start, int end*): Removes the characters in a substring of this sequence.
- *deleteCharAt(int index):* Removes the char at the specified position in this sequence.

Code example:

```
StringBuffer stringBuffer = new StringBuffer("OCAJP");
stringBuffer.delete(0, 3);
System.out.println(stringBuffer);
```

The above code fragment prints out *"JP"* when executed.

e. Reverse

The *reverse* method causes this character sequence to be replaced by the reverse of the sequence.

Code example:

```
StringBuffer stringBuffer = new StringBuffer("OCAJP");
stringBuffer.reverse();
System.out.println(stringBuffer);
```

The above code fragment prints out *"PJACO"* when executed.

2. Create and manipulate Strings

Note that *String*s are immutable. Methods manipulating a *String* create new ones instead of making changes to the existing.

a. Create

The most common way to create a *String* is to use a *String* literal:

```
String string = "Whizlabs";
```

You can also use a *String* constructor:

```
String string = new String("Whizlabs");
```

Other constructors may take a *StringBuffer* instance, a *char* array or a *byte* array to create a new *String*.

b. Replace

Character(s) of this string may be replaced using the *replace*, *replaceAll* or *replaceFirst* method:

- *replace(char oldChar, char newChar)*: returns a string resulting from replacing all occurrences of *oldChar* in this string with *newChar*.
- *replace(CharSequence target, CharSequence replacement)*: replaces each substring of this string that matches the literal target sequence with the specified literal replacement sequence.
- *replaceAll(String regex, String replacement)*: replaces each substring of this string that matches the given regular expression with the given replacement.
- *replaceFirst(String regex, String replacement)*: replaces the first substring of this string that matches the given regular expression with the given replacement.

Code example:

```
String oldString = "Whizlabs Exam Prepration";
String newString = oldString.replaceAll("\\s", "-");
System.out.println(newString);
```

The above code prints out *"Whizlabs-Exam-Prepration"* when executed.

c. Split

The *split* method splits this string around matches of the given regular expression:

```
String string = "Whizlabs Exam Prepration";
String[] sArray= string.split("\\s");
```

The *sArray* array will contain three *String* elements with contents being *"Whizlabs"*, *"Exam"* and *"Preparation"*.

d. Concat

The *concat* method concatenates the specified string to the end of this string:

```
String oldString = "Whizlabs";
String newString = oldString.concat("Exam Preparation");
System.out.println(newString);
```

The above code prints out "Whizlabs Exam Preparation" when executed.

A more concise way to concatenate strings is to use the concatenation operator (*+*):

```
String oldString = "Whizlabs";
String newString = oldString + "Exam Preparation";
```

e. Join

The *join* method returns a new *String* composed of copies of the *CharSequence* elements joined together with a copy of the specified delimiter:

```
String myString = String.join(".", "Whizlabs", "com");
System.out.println(myString);
```
The above fragment prints out *"Whizlabs.com"* when executed.

f. Format

The *format* method returns a formatted string using the specified format string and arguments:

```
String myString = String.format("This is a revision note for %s %d exam", "OCAJP", 8);
System.out.println(myString);
```
The above code fragment prints out *"This is a revision note for OCAJP 8 exam"* when executed.

3. Create and manipulate calendar data using classes from java.time.LocalDateTime, java.time.LocalDate, java.time.LocalTime, java.time.format.DateTimeFormatter, java.time.Period

a. LocalDateTime, LocalDate and LocalTime

LocalDateTime is an immutable date-time object that represents a date-time, often viewed as year-month-day-hour-minute-second. Other date and time fields, such as day-of-year, day-of-week and week-of-year, can also be accessed. Time is represented to nanosecond precision.

LocalDate is an immutable date-time object that represents a date, often viewed as year-month-day. Other date fields, such as day-of-year, day-of-week and week-of-year, can also be accessed.

LocalTime is an immutable date-time object that represents a time, often viewed as hour-minute-second. Time is represented to nanosecond precision.

Useful methods:

- Create a new instance: *now*, *of**, *parse*, *with**
- Get date/time information: *get**
- Compute another instance of the same type: *plus**, *minus**

Note: The asterisk (*) represents zero or more missing characters. There are many methods starting with the characters prior to this symbol. You just need to remember the shown characters. The rest may be inferred from the context of an exam question.

b. DateTimeFormatter

The *DateTimeFormatter* class helps to print and parse date-time objects. It provides the main application entry point for printing and parsing and provides common implementations of *DateTimeFormatter*:

- Using predefined constants, such as *ISO_LOCAL_DATE*
- Using pattern letters, such as *uuuu-MMM-dd*
- Using localized styles, such as *long* or *medium*

Code example:

```
DateTimeFormatter formatter = DateTimeFormatter.ISO_DATE;
LocalDate date = LocalDate.now();
String text = date.format(formatter);
LocalDate parsedDate = LocalDate.parse(text, formatter);
```

c. Period

This class models a quantity or amount of time in terms of years, months and days.

Useful methods:

- Create a new instance: *of**, *parse*, *with**

- Get Period information: *get**
- Compute another Period: *plus**, *minus**

Note: The asterisk (*) represents zero or more missing characters. There are many methods starting with the characters prior to this symbol. You just need to remember the shown characters. The rest may be inferred from the context of an exam question.

4. Declare and use an ArrayList of a given type

a. Create

An *ArrayList* instance may be created with either an empty list:

```
ArrayList list = new ArrayList();
```

or with elements from a collection:

```
ArrayList list2 = new ArrayList(list1); // list1 is a collection
```

b. Add

The *add* method appends the specified element to the end of this list if no position is specified:

```
ArrayList myList = new ArrayList();
myList.add("a");
```

myList will have string *"a"* as its unique element after the above addition.

You can specify the position for the new element to be inserted into:

```
myList.add(0, "b");
```

myList now contains *"b"* as the first element and *"a"* as the second.

c. Get

The *get* method returns the element at the specified position in this list:

```
Object myElement = myList.get(1);
```

The *myElement* variable now points to the second element of *myList*.

d. Remove

The *remove* method removes the element at the specified position in this list:

```
myList.remove(0);
```

The above statement removes the first element of *myList*.

An overloading method removes the first occurrence of the specified element from this list, if it is present:

```
myList.remove("b");
```

The above statement remove the first *"b"* element of *myList*.

e. Replace

The *set* method replaces the element at the specified position in this list with the specified element:

```
myList.set(0, "c");
```

The above statement replaces the first element of *myList* with *"c"*.

5. Write a simple Lambda expression that consumes a Lambda Predicate expression

a. Lambda expression syntax

A lambda expression consists of three elements:

- A comma-separated list of formal parameters enclosed in parentheses
- The arrow token (->)
- A body, which consists of a single expression or a statement block

Code example:

```
(Integer i, Integer j) -> { return i == j; };
```

Predicate :

Predicate is a functional interface in java.util.function package. It has a abstract method as follows

boolean test(T t);

Ex:

```
Predicate start = s -> s.startsWith("O");
System.out.println(start.test("OCAJP8")); // prints true
```

b. Passing a Lambda Predicate

Given an interface declaration:

```
interface NumberComparator {
   boolean compare(Integer first, Integer second);
}
```

and a method that has a parameter of type *NumberComparator* introduced above:

```
public static void compareEquality(Map<Integer, Integer> map, NumberComparator comparator) {
   for (Map.Entry<Integer, Integer> entry : map.entrySet()) {
      if (comparator.compare(entry.getKey(), entry.getValue())) {
         System.out.println(entry);
      }
   }
}
```

You can pass a lambda expression to the above method as follows:

```
Map<Integer, Integer> map = new HashMap<>();
map.put(0, 1);
map.put(2, 2);
compareEquality(map, (Integer i, Integer j) -> {
   return i == j;
});
```

In the given scenario, the compiler may infer parameter types of the lambda expression from the context. In addition, the expression body contains only a single statement. The invocation of *compareEquality* can therefore be rewritten as:

```
compareEquality(map, (i, j) -> i == j);
```