OCP – Preparation

Inhoud

[Chapter 1 (Welcome to Java) 3](#_Toc107950463)

[Major Components 3](#_Toc107950464)

[Identifying Benefits of Java 3](#_Toc107950465)

[Understanding the Java Class Structure 4](#_Toc107950466)

[Understanding package declarations and imports 5](#_Toc107950467)

[Ordering Elements in a Class 7](#_Toc107950468)

[Code Formatting on the exam 7](#_Toc107950469)

[Exam Essentials 8](#_Toc107950470)

[Chapter 2 (Java Building Blocks) 9](#_Toc107950471)

[Creating Objects 9](#_Toc107950472)

[Understanding Data Types 9](#_Toc107950473)

[Declaring Variables 10](#_Toc107950474)

[Managing variable scope 12](#_Toc107950475)

[Destroying Objects 12](#_Toc107950476)

[Exam Essentials 13](#_Toc107950477)

[Chapter 3 (Operators) 14](#_Toc107950478)

[Understanding Java Operators 14](#_Toc107950479)

[Applying Unary Operators 14](#_Toc107950480)

[Working with Binary Arithmetic Operators 15](#_Toc107950481)

[Assignment Values 15](#_Toc107950482)

[Comparing Values 16](#_Toc107950483)

[Chapter 4 (Making Decisions) 16](#_Toc107950484)

[Chapter 5 (Core Java APIs) 16](#_Toc107950485)

[Chapter 6 (Lambdas and Functional Interfaces) 16](#_Toc107950486)

[Chapter 7 (Methods and Encapsulation) 16](#_Toc107950487)

[Chapter 8 (Class design) 16](#_Toc107950488)

[Chapter 9 (Advanced Class Design) 16](#_Toc107950489)

[Chapter 10 (Exceptions) 16](#_Toc107950490)

[Chapter 11 (Modules) 17](#_Toc107950491)

[Chapter 12 (Java Fundamentals) 17](#_Toc107950492)

[Chapter 13 (Annotations) 17](#_Toc107950493)

[Chapter 14 (Generics and Collections) 17](#_Toc107950494)

[Chapter 15 (Functional Programming) 17](#_Toc107950495)

[Chapter 16 (Exceptions, Assertions and Localizations) 17](#_Toc107950496)

[Chapter 17 (Modular Applications) 17](#_Toc107950497)

[Chapter 18 (Concurrency) 17](#_Toc107950498)

[Chapter 19 (I/O) 17](#_Toc107950499)

[Chapter 20 (NIO.2) 17](#_Toc107950500)

[Chapter 21 (JDBC) 17](#_Toc107950501)

[Chapter 22 (Security) 17](#_Toc107950502)

# Chapter 1 (Welcome to Java)

## Major Components

* JDK – (Java Development Kit) Contains the minimum software you need to do Java Development.
* Compiler – (javac) Converts .java files to .class files, and the launcher java, which creates the virtual machine and executes the program.
* JDK also contains other tools including the archiver (jar) command, which can package files together, and the API documentation (Javadoc) command for generating documentation.
* Javac program generates instructions in a special format that the java command can run called bytecode, then java launches the Java Virtual Machine (JVM) before running the code.
* The JVM knows how to run your .class files and knows how to run bytecode.
* Java comes with a large suite of **application programming interfaces (APIs)** that you can use from the start, such as a StringBuilder class to create large String and a method in Collections

## Identifying Benefits of Java

**Object Oriented –** Java is an object-oriented language. Means all code is defined in classes, and most of those classes can be instantiated into objects.

**Encapsulation –** Java supports access modifiers to protect data from unintended access and modification.

**Platform Independent –** Java is an interpreted language that gets compiled to bytecode. A key benefit is that Java code gets compiled once rather than needing to be recompiled for different operating systems. This is known as “write once, run everywhere.”

**Robust –** Java prevents memory leaks. Java manages memory on its own and does garbage collection automatically.

**Simple –** Java was intended to be simpler to understand than C++ removal of pointers and operator overloading.

**Secure –** Java code runs inside the JVM. This creates a sandbox that makes it hard for Java code to do evil things to the computer it is running on.

**Multithreaded –** Java is designed to allow multiple pieces of code to run at the same time. There are also many APIs to facilitate this task.

**Backward Compatibility –** The Java Language architects pay careful attention to making sure old programs will work with later versions of Java. By using the Deprecation technique, they accomplish this where code is flagged to indicate it shouldn’t be used.

## Understanding the Java Class Structure

**Object** – is a runtime instance of a class in memory, also referred to as an instance since it represents a single representation of the class.

**State of the program** – all the various objects of all the different classes.

**Reference** – is a variable that points to an object.

**Fields and Methods**

Java classes have two primary elements: **methods** – often called functions or procedures in other languages, and **fields**, more generally known as variables. Together they are called the members of the class.

* **Variables hold** the state of the program,
* and **methods operate** on that state.

If a change is important to remember, a variable store that change.

**Classes vs. Files**

Most of the time Java classes are defined in their own .java file. Generally, they are public meaning any other class can call the class. Interestingly, Java does not require that the class be public. You can put two or more classes in a single .java file BUT only one class may be public, and it must have the same name as the file name else it will not compile!

**Writing a main() method**

* each file can contain only one public class but can contain multiple classes.
* The filename must match the class name, including case, and have a .java extension.
* The main() method signature is:
  + public static void main(String[] args){}
* java file is compiled with fist with command javac filename.java
* secondly its then run with the command java filename (omitting the .class JDK knows what file to use)
* if you want to pass parameters as arguments then you can add it after the filename example:
  + java filename 1 2 3 4
    - (parameters 1 2 3 4 is 4 different STRING INPUT PARAMETERS
    - These parameters are always of type String []

**Running a program in one line**

* Adding the .java to the filename using the java command gives us this function since Java 11 ONLY for single .java file programs. When you have more than one .java file you still need to use the javac filename.java -> java filename process to launch your program.
* This helps when creating small one file applications ideal for testing small single file programs.

## Understanding package declarations and imports

Java comes with thousands of built-in classes, and there are countless more from developers.

* Java puts classes in packages
* import java.util.Random; //example import tells us where to find Random class

**Wildcards (\*)**

import java.util.\* // this imports all classes included in the package java.util.\*

* doesn’t import child packages, fields, or methods; it imports only classes.
* There is a special type of import called the static import that imports other types

**Redundant imports**

* Java.lang package is automatically imported
* import java.lang.System
* import java.lang.\*

**naming conflicts**

* main reason to use packages is so that class names don’t need to be unique across all of Java.
* An example is the Date class. Java provides the java.util.Data and java.sql.Data
* Error : reference to Data is ambiguous
* If you import both imports with same name then the class named import takes precedence over wildcards. Yet still you can’t have two same name default classes imported, thus one default and one explicit reference.

**Creating a new package**

* The directory structure on your computer is related to the package name.
  + package packagea;
    - public class ClassA
  + package packageb;
    - import packagea.ClassA
* when running a java program, java knows where to look for those package names. In this case running from C:\temp works because both packagea and packageb are underneath it.
  + Running it from C:\temp\packageb/ClassB.java – this doesn’t work because of the java command to run a file directly only when that program is contained within a single file. Here ClassB.java relies on ClassA

**Compiling and running code with packages**

1. First compile all the .java files with command javac
   1. Example
      1. javac packagea/ClassA.java packageb/ClassB.java
      2. if you successful then two .class files created and can be run by java packageb.ClassB
2. Compiling with Wildcards
   1. You can use an asterisk to specify that you’d like to include all java files in a directory.
      1. example – javac packagea/\*.java packageb/\*.java

**Using an alternate Directory**

* by default javac command places the compiled classes in the same directory as the source code.
* An alternative option
  + javac packagea/ClassA.java packageb/ClassB.java
    - packagea
      * ClassA.java
      * ClassA.class
    - packageb
      * ClassB.java
      * ClassB.class
  + javac -d classes packagea/ClassA.java packageb/ClassB.java
    - packagea
      * ClassA.java
    - packageb
      * ClassB.java
    - classes
      * packagea
        + ClassA.class
      * Packageb
        + ClassB.class
  + java -cp classes packageb.ClassB
  + java -classpath classes packageb.ClassB
  + java --class-path classes packageb.ClassB
  + options for javac (location of classes needed to compile the program)
    - -cp <classpath>
    - -classpath <classpath>
    - --class-path <classpath>
    - -d <dir> (// directory to place generate class files)
  + Options for java (location of classes needed to run program)
    - -cp <classpath>
    - -classpath <classpath>
    - --class-path <classpath>

**Compiling with JAR Files**

Java archive (JAR) is like a zip file of mainly Java class files

java -cp “.;C:\temp\someOtherLocation;c:\temp\myJar.jar” myPackage.MyClass

java -cp “.:/tmp/someOtherLocation:/tmp/myJar.jar” myPackage.MyClass

java -cp “C:\temp\directoryWithJars\\*” myPackage.MyClass

this command will add all the JARs to the classpath that are in directoryWithJars.

**Creating a JAR File**

jar -cvf myNewFile.jar

jar –create –verbose –file myNewFile.jar

(alternatively specify the dir)

jar -cvf myNewFile.jar -C dir

|  |  |
| --- | --- |
| Option | Description |
| -c  --create | Creates a new JAR file |
| -v  --verbose | Prints details when working with JAR files |
| -f <filename>  --file <filename> | JAR Filename |
| -c <directory> | Directory containing files to be used to create the JAR |

**Running a program in one line with Packages**

You can use single-file source-code programs from within a package as long as they rely only on classes supplied by the JDK.

You can run it by:

* java Learning.java // from within the singleFile directory
* java singleFile/Learning.java //from the directory above singleFile

## Ordering Elements in a Class

**Order for declaring a class**

1. Package declaration (First line of the file)
2. Import statements
3. Class declaration (Required)
4. Field declarations
5. Method declarations

## Code Formatting on the exam

Common cases where you don’t need to check the imports:

* Code that begins with a class name
* Code that begins with a method declaration
* Code that begins with a code snippet that would normally be inside a class or method
* Code that has line numbers that don’t begin with 1

## Exam Essentials

**Identify benefits of Java**

* Object-oriented design
* Encapsulation
* Platform independence
* Robustness
* Simplicity
* Security
* Multithreading
* And backward compatibility

**Define common acronyms**

* JDK -> Java Development Kit and contains the compiler and JVM launcher
* JVM stands for Java Virtual Machine, and it runs bytecode.
* API is an application programming interface, which is code that you can call

**Be able to write code using a main() method**

public static void main(String[] args){

}

Arguments are referenced starting with args[0]. Accessing an argument that wasn’t passed in will cause the code to throw and exception.

**Understand the effect of using packages and imports**

* Packages contain Java classes
* Classes can be imported by class name or wildcard.
* Wildcards do not look at subdirectories
* In event of a conflict, class name imports take precedence

**Be able to recognize misplaced statements in a class**

* Package and import statements are optional
* If present both go before the class declaration in that order
* Fields and methods are also optional and are allowed in any order within the class declaration

# Chapter 2 (Java Building Blocks)

## Creating Objects

**Calling constructors**

Create an instance of a class

* Write “new” before the class name and add parentheses after it
* Example –> Park = new Park();

Key points:

* The name of the constructor matches the name of the class
* And there’s no return type

**Reading and Writing member fields**

* Create an instance of the class and use the reference of the instance to get the field/method (members) of the class.

**Executing instance initializer blocks**

{} – code block

{} outside a method in the class -> instance initializers

Balanced parentheses problem -> you can’t use the enclose brace “}” if there’s no corresponding open brace “{“that it matches written earlier in the code

**Following Order of Initialization**

1. Fields and instance initializer blocks are run in the order in which they appear in the file
2. The constructor runs after all fields and instance initializer blocks have run

## Understanding Data Types

**Using Primitive types**

Java has 8 built =-in data types, referred to as the Java primitive types.

The 8 building blocks of Java objects, because all java objects are just a complex collection of these primitive data types.

Primitive types are JUST a value and not a type of object.

|  |  |  |
| --- | --- | --- |
| boolean | True or false | True |
| byte | 8-bit integral value (-128->127) | 123 |
| short | 16-bit integral value | 123 |
| int | 32-bit integral value | 123 |
| long | 64-bit integral value | 123L |
| float | 32-bit floating value | 123.45f |
| double | 64-bit floating value | 123.46 |
| char | 16-bit Unicode value | ‘a’ |

**Writing Literals**

Literals are actual values that is assigned to a variable/ field for example:

longValue = 32432L; (L -> specify that it’s a long type)

VALID digits in several other formats:

- OCTAL (digits 0-7), which uses the number 0 as a prefix -- for example 017

- hexadecimal (digits 0-9 and letters A-F)

- binary (digits 0/1)

**Literals and the underscore character**

int million1 = 1000000; same as:

int million2 = 1\_000\_000;

invalid double values with INVALID USE OF \_

\_1000.00

3200\_.00

3300.00\_

These does NOT COMPILE

**Using Reference Types**

- a reference can be assigned to another object of the same type  
- a reference can be assigned to a new object using the new keyword  
  
A Reference type refers to an object (an instance of a class).  
A reference "points" to an object by storing the memory address where the object is located, a consept refeerred to as a pointer.

**Key Differences**

1. Reference types can be assigned NULL, (currently not referencing an object in memory)  
2. Primitive types will give you compiler error if you attempt to assign them null.  
3. Reference types can be used to call methods when they do not point to null.  
4. Primitives do not have methods declared on them they are just a value.  
5. Primitives types have lowercase type names. All classes that come with Java begin with uppercase.

## Declaring Variables

**Identifying identifiers**

* Identifiers must begin with a letter, a $ symbol, or a \_ symbol
* Identifiers can include numbers but not start with them
* Since Java 9, a single underscore \_is not allowed as an identifier
* You cannot use the same name as a java reserved word.

Style: camelCase

* First letter of each word is capitalized

Style: snake\_case

* Each word is separated by an \_

**Multiple variables in one line**  
// creates 4 string variables referencing NULL object  
public String s1**,**s2**,**s3**,**s4**;**// declared multiple variables with different string values all in one line, generally not good practice.  
public String s5 = "test1"**,**s6 = "test2"**,**s7 = "test3"**,**s8 = "test4"**;**// Catch question, only the i3 is given value 3;  
public int i1**,**i2**,**i3 = **3;**// int i2, String s2; does not compile for multiple line variables the same type needs to be shared.

**local variables :**

* Is a variable defined within a constructor, method or initializer block
* must be initialized before use
* do not have a default value and contain garbage data until initialized.

**Instance Variables** are variables that’s not local and are defined within the class like these below  
 DEFAULT initialization for instance and class variables only

instance variables can be initialized in an instance blocks

Boolean false  
byte,short,int,long 0  
float,double 0.0  
char '\u0000' (NUL)  
all object references (everything else) null

**Introducing var**

Starting in Java 10, you have the option of using the keyword var instead of the type for local variables under certain conditions.

* Example: var name = “hello”;
* Formal name of this feature is local variable type inference

**Type Inference of var**

**Type inference** -> when you type var, you are instructing the compiler to determine the type for you.

var number = 7;

number = 4;

number = “five”; // this will not compile. Initial var type inference is set to type of int thus a string is not of type int;

Type inference of the keyword var variable can’t be changes at runtime thus type is set at compile time.

**Var and null**

While a var cannot be initialized with a null value without a type, it can be assigned a null value after it is declared, provided that the underlying data type of the var is an object.

**Review of var Rules**

1. A var is used as a local variable in a constructor, method or initializer block
2. A var cannot be used in constructor parameters, method parameters, instance variables or class variables
3. A var is always initialized on the same line (or statement0 where it is declared
4. The value of a var can change but the type cannot
5. A var cannot be initialized with a null value without a type
6. A var is not permitted in a multiple-variable declaration
7. A var is a reserved type name but not a reserved word, meaning it can be used as an identifier except as a class, interface or enum name.

## Managing variable scope

**RULES OF SCOPE**  
1. local variables - in scope from declarations to end of block  
2. instance variables - in scope from declaration until object garbage collected  
3. class variables - in scope from declaration until program ends

## Destroying Objects

**GARBAGE Collection**  
- the process of automatically freeing memory on the heap by deleting objects that are  
 no longer reachable in your program  
- System.gc() is not guaranteed to run  
 + this method java provides merely suggests that now might be a good time for java to kick of GC.  
 java can ignore this suggestion  
- A object is ready for GC when its is no longer reachable by the program in two situations  
 + the object no longer has any references pointing to it  
 + all reference to the object have gone out of scope.  
- it is the object that gets GC not the reference

**finalize()**java allows objects to implement a method called finalize() that might get called.  
- gets called if the Garbage collector tries to collect the object  
- if garbage collector doesn't run this method doesn't run  
- will never be called twice!!

## Exam Essentials

**Be able to recognize a constructor**

* A constructor has the same name as the class, looks like a method without a return type

**Be able to identify legal and illegal declarations and initialization**

* Multiple variables can be declared and initialized in the same statement when they share a type
* Local variables require an explicit initialization, others use the default value for that type
* Identifiers may contain letters, numbers $, \_ although they may not begin with numbers
* Cannot define an identifier that is just a single underscore character \_
* Numeric literals may contain underscores between two digits, such as 1\_000, but not other places, such as \_100\_.0\_
* Numeric literals can begin with 1-9, 0,0x,0x,0b, and 0B, with the latter four indicating a change of numeric base.

**Be able to use var correctly**

* A var is used for a local variable inside a constructor, a method, or an initializer block
* It cannot be used for constructor parameters, method parameters, instance variables, or class variables.
* A var is initialized on the same line where it is declared, and while it can change value, it cannot change type.
* A var cannot be initialized with a null value without a type, nor can it be used in multiple variable declarations
* Var is not a reserved word in Java and can be used as a variable name

**Be able to determine where variables go into and out of scope**

* All variables go into scope when they are declared
* Local variables go out of scope when the block they are declared in ends
* Instance variables go out of scope when the object is eligible for garbage collection.
* Class variables remain in scope as long as the program is running

**Know how to identify when an object is eligible for garbage collections**

* Draw a diagram to keep track of references and objects as you trace the code.
* When no arrows point to a box (object), it is eligible for garbage collection

# Chapter 3 (Operators)

## Understanding Java Operators

**Types of Operators**

1. Unary
2. Binary
3. Ternary

These three operators can be applied to one, two or three operands respectively.

**Operator Precedence**

**TABLE 2.1 Order of operator precedence  
Operator Symbols and examples**

Post-unary operators expression++, expression--  
Pre-unary operators ++expression, --expression  
Other unary operators +, -, !, ~, (type)  
Multiplication/Division/Modulus \*, /, %  
Addition/Subtraction +, -  
Shift operators <<, >>, >>>  
Relational operators <, >, <=, >=, instanceof  
Equal to/not equal to ==, !=  
Logical operators &, ^, |  
Short-circuit logical operators &&, ||  
Ternary operators boolean expression ? expression1 : expression2  
Assignment operators =, +=, -=, \*=, /=, %=, &=, ^=, !=, <<=, >>=, >>>=

## Applying Unary Operators

**Unary operators**

* ! -> Inverts a boolean’s logical value
* + -> Indicates a number is positive, although numbers are assumed to be positive in java unless accompanied by a negative unary operator
* - -> Indicates a literal number is negative or negates an expression
* ++ -> Increments a value by 1
* -- -> Decrements a value by 1
* (type) -> Casts a value to a specific type

**Logical Complement (!) and negation operators (-)**

*Boolean isAnimalAsleep = false;*

*System.out.println(isAnimalAsleep); // false*

*isAnimalAsleep = !isAnimalAsleep;*

*System.out.println(isAnimalAsleep); // true*

*double dValue = 2.2;*

*System.out.println(dValue); // 2.2*

*System.out.println(-dValue); // -2.2*

**Increment and Decrement Operators**

*int parkAttendance = 0;*

*System.out.println(parkAttendance); // 0*

*System.out.println(++parkAttendance); // 1*

*System.out.println(parkAttendance); // 1*

*System.out.println(parkAttendance--); // 1*

*System.out.println(parkAttendance); // 0*

CATCH on the exam:

Multiple on the same line on same variable!

Example:

*int lion = 3;*

*int tiger = ++lion \* 5 / lion--; // ++lion(4) \* 5 / lion--(value = 4 but lion after value is 3)*

*System.out.println(“lion is “ + lion); // lion = 3*

*System.out.println(“tiger is “ + tiger); // tiger = 5*

## Working with Binary Arithmetic Operators

**Binary arithmetic operators**

* + -> Adds two numeric values
* - -> Subtracts two numeric values
* \* -> Multiplies two numeric values
* / -> Divides one numeric value by another
* % -> Modulus operator returns the remainder after division of one numeric value by another

**Numeric promotion**

* if two values have different data types one is promoted to the larger of the two data types:
* if an int and a float is added they will be promoted to float and result will be float.
* the result is the same data type as the promoted type.
* byte short and char are first promoted to int even if no int is in the equation

## Assignment Values

**Assignment Operator**

‘=’ Assigns the value on the right to the variable on the left

**Casting Values**

Casting is performed by placing the data type, enclosed in parentheses, to the left of the value you want to cast.

*int fur = (int)5;*

*int hair = (short)2;*

*short tail = (short)(4 + 10);*

* during the exam, remember to keep track of parentheses and return types any time casting is involved

**Compound Assignment Operators**

Besides the = simple assignment operator java supports numerous compound assignment operators:

* += -> Adds the value on the right to the variable on the left and assigns the sum to the variable
* -= -> Subtracts the value on the right from the variable on the left and assigns the difference to the variable
* \*= -> Multiplies the value on the right with the variable on the left and assigns the product to the variable
* /= -> Divides the variable on the left by the right and assigns the quotient to the variable

**Assignment Operator Return Value**

One final thing to know about assignment operators is that the result of an assignment is an expression in and of itself, equal to the value of the assignment. For example

*long wolf = 5;*

*long coyote = (wolf=3);*

*System.out.println(wolf); // 3*

*System.out.println(coyote); //3*

## Comparing Values

**Equality Operators**

* ==
  + Returns true if the two values represent the same value
  + Returns true if the two values reference the same object
* !=
  + Returns true if the two values represent different values
  + Returns true if the two values do not reference the same object

The equality operators are used in one of three scenarios:

1. Comparing two numeric or character primitive types. If the numeric values are of different types, the values are automatically promoted. For example 5 == 5.0 // true the int 5 gets promoted to 5.0
2. Comparing two boolean values
3. Comparing two objects, including null and String values

# Chapter 4 (Making Decisions)

# Chapter 5 (Core Java APIs)

# Chapter 6 (Lambdas and Functional Interfaces)

# Chapter 7 (Methods and Encapsulation)

# Chapter 8 (Class design)

# Chapter 9 (Advanced Class Design)

# Chapter 10 (Exceptions)

# Chapter 11 (Modules)

# Chapter 12 (Java Fundamentals)

# Chapter 13 (Annotations)

# Chapter 14 (Generics and Collections)

# Chapter 15 (Functional Programming)

# Chapter 16 (Exceptions, Assertions and Localizations)

# Chapter 17 (Modular Applications)

# Chapter 18 (Concurrency)

# Chapter 19 (I/O)

# Chapter 20 (NIO.2)

# Chapter 21 (JDBC)

# Chapter 22 (Security)