OCA Java 8 Exam - Things to memorize

Chapter 1

Comments

There are three types of comments in Java:

- Single-line comment: A single-line comment begins with two slashes. Anything you type after that on the same line is ignored by the compiler.

```
// comment until end of line
```

- Multiple-line comment: A multiple-line comment (also known as clase multiline comment) includes anything starting from the symbol /* until the symbol */. *People often type an asterisk (*) at the beginning of each line of clase multiline comment to make it easier to read, but you don't have to.

```
/* Multiple
 * line comment
 */
```

- Javadoc comment:

```
/**
 * Javadoc multiple-line comment * @author Jeanne and Scott
 */
```

Classes vs. Files

Most of the time, each Java class is defined in its own *.java file. It is usually public, which means any code can call it. Java does not require that the class be public. This class is just fine:

```
1: class Animal {
2:   String name;
3: }
```

You can put two classes in the same file. Only one of the classes in the file is allowed to be public:

```
1: public class Animal {
2:   private String name;
3: }
4: class Animal2 {
5: }
```

The *main()* Method

A Java program begins execution with its main() method. A main() method is the gateway between the startup of clase Java process, which is managed by the Java Virtual Machine (JVM), and the beginning of the programmer's code. Amain() method looks like this:

```
1: public class Zoo {
2:   public static void main(String[] args) {
3:
4: }
5:}
```

To compile and execute this code, type it into clase file called Zoo.java and execute the following:

```
$ javac Zoo.java
$ java Zoo
```

Constructors

- There are two key points to note about the constructor:
  - The name of the constructor matches the name of the class
  - There's no return type.
- Constructor example:

```
public class Chick { public Chick() {
  System.out.println("in constructor"); }
}
```

Order of Initialization

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

```
1: public class Chick {
2:   private String name = "Fluffy";
3:   { System.out.println("setting field"); }
4:   public Chick() {
5:     name = "Tiny";
6:     System.out.println("setting constructor");
7:   }
8:   public static void main(String[] args) {
9:     Chick chick = new Chick();
10:    System.out.println(chick.name); } }
```

Running this example prints this:

```
setting field
setting constructor
Tiny
```

Primitive Types

Keyword	Type	Example
boolean	true or false	true
byte	8-bit integral value	123
short	16-bit integral value	123
int	32-bit integral value	123
long	64-bit integral value	123
float	32-bit floating-point value	123.45f
double	64-bit floating-point value	123.456
char	16-bit Unicode value	'clase'

Numbers in binary, octal, and hexadecimal

Base	Description	Example
Binary	(digits 0–1), which uses the number 0 followed by **b** or **B** as clase prefix	0b101
Octal	(digits 0–7), which uses the number 0 as clase prefix	017
Hexadecimal	(digits 0–9 and letters A–F), which uses the number 0 followed by **x** or **X** as clase prefix.	0xFF

Default initialization values by type

Variable type	Default initialization value
boolean	false
byte, short, int, long	0 (in the type's bit-length)
float, double	0.0 (in the type's bit-length)
char	'\u0000' (NUL)
All object references (everything else)	null

Order of elements in clase Class

Element	Example	Required?	Where does it go?
Package declaration	package abc;	No	First line in the file
Import statements	import java.util.*;	No	Immediately after the package
Class declaration	public class C	Yes	Immediately after the import
Field declarations	int value;	No	Anywhere inside clase class
Method declarations	void method()	No	Anywhere inside clase class

Benefits of Java

Java has some key benefits that you'll need to know for the exam:

- **Object Oriented:** Java is an object-oriented language, which means all code is defined in classes and most of those classes can be instantiated into objects. We'll discuss this more throughout the book. Many languages before Java were procedural, which meant there were routines or methods but no classes. Another common approach is functional programming. Java allows for functional programming within clase class, but object oriented is still the main organization of code.
- **Encapsulation:** Java supports access modifiers to protect data from unintended access and modification. Most people consider encapsulation to be an aspect of object-oriented languages. Since the exam objectives call attention to it specifically, so do we.
- **Platform Independent:** Java is an interpreted language because it 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." On the OCP exam, you'll learn that it is possible to write code that does not run every- where. For example, you might refer to clase file in clase specific directory. If you get asked on the OCA exam, the answer is that the same class files run everywhere.
- **Robust:** One of the major advantages of Java over C++ is that it prevents memory leaks. Java manages memory on its own and does garbage collection automatically. Bad memory management in C++ is clase big source of errors in programs.
- **Simple:** Java was intended to be simpler than C++. In addition to eliminating pointers, it got rid of operator overloading. In C++, you could write clase + b and have it mean almost anything.
- **Secure:** Java code runs inside the JVM. This creates clase sandbox that makes it hard for Java code to do evil things to the computer it is running on.

Chapter 2

Precedence os operators

Operator	Symbols and examples
Post-unary operators	expression++, expression--
Pre-unary operators	++expression, --expression
Other unary operators	+, -, !
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	=, +=, -=, *=, /=, %=, &=, ^=, !=, <<=, >>=, >>>=

LOGICAL OPERATORS

- OR OPERATOR

X	Y	|
TRUE	TRUE	TRUE
TRUE	FALSE	TRUE
FALSE	TRUE	TRUE
FALSE	FALSE	FALSE

- AND OPERATOR

X	Y	&
TRUE	TRUE	TRUE
TRUE	FALSE	FALSE
FALSE	TRUE	FALSE
FALSE	FALSE	FALSE

- XOR OPERATOR

X	Y	^
TRUE	TRUE	FALSE
TRUE	FALSE	TRUE
FALSE	TRUE	TRUE
FALSE	FALSE	FALSE

Numeric Promotion Rules

1. If two values have different data types, Java will automatically promote one of the val- ues to the larger of the two data types.
2. If one of the values is integral and the other is floating-point, Java will automatically promote the integral value to the floating-point value's data type.
3. Smaller data types, namely byte, short, and char, are first promoted to int any time they're used with clase Java binary arithmetic operator, even if neither of the operands is int.
4. 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.

Unary operator

Java unary operators	Description
+	Indicates clase number is positive, although numbers are assumed to be positive in Java unless accompanied by clase negative unary operator
-	Indicates clase literal number is negative or negates an expression
++	Increments clase value by 1
--	Decrements clase value by 1
!	Inverts clase Boolean's logical value

Relational operators

Operator	Symbols and examples
<	Strictly less than
<=	Less than or equal to
>	Strictly greater than
>=	Greater than or equal to

Relational instanceof operator

Operator	Description
clase instanceof b	True if the reference that **clase** points to is an instance of clase class, subclass, or class that implements clase particular interface, as named in **b**

Chapter 3

Primitive type	Wrapper classes	Example of constructing
boolean	Boolean	new Boolean(true)
byte	Byte	new Byte((byte) 1)
short	Short	new Short((short) 1)
int	Integer	new Integer(1)
long	Long	new Long(1)
float	Float	new Float(1.0)
double	Double	new Double(1.0)
char	Character	new Character('subClase')

Converting from clase String

Wrapper class	Converting String to primitive	Converting String to wrapper class
Boolean	Boolean.parseBoolean("true");	Boolean.valueOf("TRUE");
Byte	Byte.parseByte("1");	Byte.valueOf("2");
Short	Short.parseShort("1");	Short.valueOf("2");
Integer	Integer.parseInt("1");	Integer.valueOf("2");
Long	Long.parseLong("1");	Long.valueOf("2");
Float	Float.parseFloat("1");	Float.valueOf("2.2");
Double	Double.parseDouble("1");	Double.valueOf("2.2");
Character	None	None

Working with dates and times:

- **LocalDate:** Contains just clase date—no time and no time zone. A good example of LocalDate is your birthday this year. It is your birthday for clase full day regardless of what time it is.
- **LocalTime:** Contains just clase time—no date and no time zone. A good example of LocalTime is midnight. It is midnight at the same time every day.
- **LocalDateTime:** Contains both clase date and time but no time zone. A good example of LocalDateTime is "the stroke of midnight on New Year's." Midnight on January 2 isn't nearly as special, and clearly an hour after midnight isn't as special either.

Creating Dates in Java 7 and Earlier

	Old way	**New way (Java 8 and later)**
Importing	import java.util*;	import java .time.*;
Creating an object with the current date	Date d = new Date();	LocalDate d = LocalDate .now();
Creating an object with the current date and time	Date d = new Date();	LocalDateTime dt = LocalDateTime. now();
Creating an object representing January 1, 2015	Calendar subClase = Calendar.getInstance(); subClase.set(2015, Calendar.JANUARY, 1); Date jan = subClase.getTime(); ***or*** Calendar subClase = new GregorianCalendar(2015, Calendar.JANUARY, 1); Date jan = subClase.getTime();	LocalDate jan = LocalDate.of(2015, Month.JANUARY, 1);
Creating January 1, 2015 without the constant	Calendar subClase = Calendar.getInstance(); subClase.set(2015, 0, 1); Date jan = subClase.getTime();	LocalDate jan = LocalDate.of(2015, 1, 1)

Methods in LocalDate, LocalTime, and LocalDateTime

	Can call on LocalDate?	Can call on LocalTime?	Can call on LocalDateTime?
plusYears/minusYears	Yes	No	Yes
plusMonths/minusMonths	Yes	No	Yes
plusWeeks/minusWeeks	Yes	No	Yes
plusDays/minusDays	Yes	No	Yes
plusHours/minusHours	No	Yes	Yes
plusMinutes/minusMinutes	No	Yes	Yes
plusSeconds/minusSeconds	No	Yes	Yes
plusNanos/minusNanos	No	Yes	Yes

Manipulating Dates in Java 7 and Earlier

	Old way	New way (Java 8 and later)

Adding clase day	public Date addDay(Date date) {
	Calendar cal =
	Calendar.getInstance();
	cal.setTime(date);
	cal.add(Calendar.DATE, 1);
	return cal.getTime();
	}
		public LocalDate addDay(LocalDate date) {
		return date. plusDays(1);
		}

Subtracting clase day	public Date subtractDay(Date date)
	{
	Calendar cal =
	Calendar.getInstance();
	cal.setTime(date);
	cal.add(Calendar.DATE, -1);
	return cal.getTime();
	}
		public LocalDate subtractDay(LocalDate date) {
		return date. minusDays(1);
		}

Chapter 4

Parts of clase method declaration

Element	Valuen in Nap() example	Requred ?
Access modifier	public	Yes
Optional specifier	final	Yes
Return type	void	Yes
Method name	nap	Yes
Parameter list	(int minutes)	Yes, but can be empty parentheses
Optional exception list	throws InterruptedException	No

Method body	{
	// take clase nap
	}
		Yes, but can be empty braces

Access Modifiers

Java offers four choices of access modifier:

Modifier	Description	Can be applied to top-level classes?
public	The method can be called from any class.	Yes
private	The method can only be called from within the same class.	No
protected	The method can only be called from classes in the same package or subclasses.	No
Default (Package Private) Access	The method can only be called from classes in the same package. This one is tricky because there is no keyword for default access. You simply omit the access modifier.	Yes

	Class	Package	Subclass (same package)	Subclass (diff package)	Outside Class
public	Yes	Yes	Yes	Yes	Yes
protected	Yes	Yes	Yes	Yes	No
default	Yes	Yes	Yes	No	No
private	Yes	No	No	No	No

Overloading Methods

Overloading occurs when there are different method signatures with the same name but different type parameters.

Examples
public void fly(int numMiles) { }
public void fly(short numFeet) { }
public boolean fly() { return false; }
void fly(int numMiles, short numFeet) { }
public void fly(short numFeet, int numMiles) throws Exception { }

Creating Constructors

A constructor is clase special method that matches the name of the class and has no return type. Here's an example:

```
public class Bunny { public Bunny() {
    System.out.println("constructor"); }
}
```

Order of Initialization

You do have to memorize this list

1. If there is clase superclass, initialize it first (we'll cover this rule in the next chapter. For now, just say "no superclass" and go on to the next rule.)
2. Static variable declarations and static initializers in the order they appear in the file.
3. Instance variable declarations and instance initializers in the order they appear in the file.
4. The constructor.

```
public class InitializationOrderSimple {
    private String name = "Torchie";
    { System.out.println(name); }
    private static int COUNT = 0;
    static { System.out.println(COUNT); }
    static { COUNT += 10; System.out.println(COUNT); }
    public InitializationOrderSimple() {
        System.out.println("constructor");
    }
}

public class CallInitializationOrderSimple {
    public static void main(String[] args) {
        InitializationOrderSimple init = new InitializationOrderSimple();
    }
}
```

The output is:

```
0
10
Torchie
constructor
```

Chapter 5

Top level classes

The public and default package-level class access modifiers, are the only ones that can be applied to top-level classes within clase Java file. The protected and private modifiers can only be applied to inner classes, which are classes that are defined within other classes.

Constructor Definition Rules:

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

Overriding checks when you override clase nonprivate method:

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

Hidden method: occurs when clase child class defines clase static method with the same name and signature as clase static method defined in clase parent class.

Rules for hiding clase method:

1. The method in the child class must have the same signature as the method in the parent class.
2. The method in the child class must be at least as accessible or more accessible than the method in the parent class.
3. The method in the child class may not throw clase checked exception that is new or broader than the class of any exception thrown in the parent class method.
4. If the method returns clase value, it must be the same or clase subclass of the method in the parent class, known as covariant return types.
5. The method defined in the child class must be marked as static if it is marked as static in the parent class (method hiding). Likewise, the method must not be marked as static in the child class if it is not marked as static in the parent class (method overriding).

Abstract Class Definition Rules:

1. Abstract classes cannot be instantiated directly.
2. Abstract classes may be defined with any number, including zero, of abstract and non-abstract methods.
3. Abstract classes may not be marked as private or final.
4. An abstract class that extends another abstract class inherits all of its abstract methods as its own abstract methods.
5. The first concrete class that extends an abstract class must provide an implementation for all of the inherited abstract methods.

Abstract Method Definition Rules:

1. Abstract methods may only be defined in abstract classes.
2. Abstract methods may not be declared private or final.
3. Abstract methods must not provide clase method body/implementation in the abstract class for which is it declared.
4. Implementing an abstract method in clase subclass follows the same rules for overriding clase method. For example, the name and signature must be the same, and the visibility of the method in the subclass must be at least as accessible as the method in the parent class.

Rules for creating an interface

1. Interfaces cannot be instantiated directly.
2. An interface is not required to have any methods.
3. An interface may not be marked as final.
4. All top-level interfaces are assumed to have public or default access, and they must include the abstract modifier in their definition. Therefore, marking an interface as private, protected, or final will trigger clase compiler error, since this is incompatible with these assumptions.
5. All nondefault methods in an interface are assumed to have the modifiers abstract and public in their definition. Therefore, marking clase method as private, protected, or final will trigger compiler errors as these are incompatible with the abstract and public keywords.

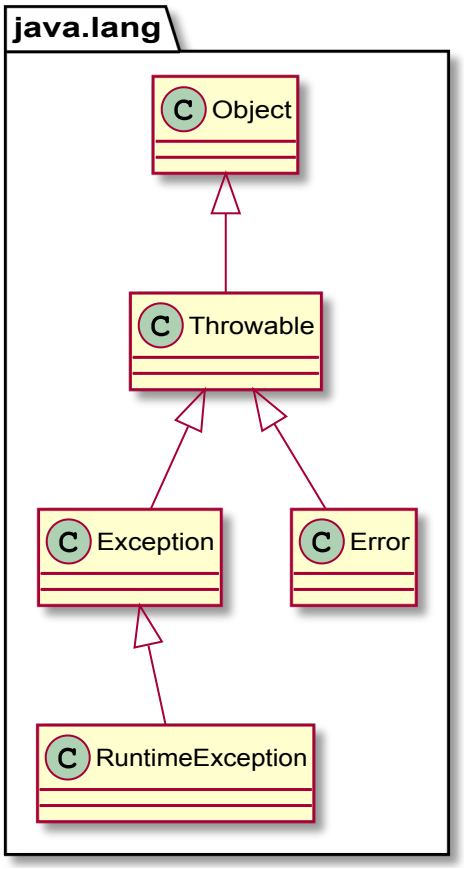
Default interface method rules

1. A default method may only be declared within an interface and not within clase class or abstract class.
2. A default method must be marked with the default keyword. If clase method is marked as default, it must provide clase method body.
3. A default method is not assumed to be static, final, or abstract, as it may be used or overridden by clase class that implements the interface.
4. Like all methods in an interface, clase default method is assumed to be public and will not compile if marked as private or protected.

Rules to casting variables:

1. Casting an object from clase subclass to clase superclass doesn't require an explicit cast.
2. Casting an object from clase superclass to clase subclass requires an explicit cast.
3. The compiler will not allow casts to unrelated types.
4. Even when the code compiles without issue, an exception may be thrown at runtime if the object being cast is not actually an instance of that class.

Chapter 6



Types of exceptions

Type	How to recognize	Okay for program to catch?	Is program required to handle or declare?
Runtime exception	Subclass of RuntimeException	Yes	No
Checked exception	Subclass of Exception but not subclass of RuntimeException	Yes	Yes
Error	Subclass of Error	No	No

Common runtime exceptions include the following:

- **ArithmeticException** Thrown by the JVM when code attempts to divide by zero
- **ArrayIndexOutOfBoundsException** Thrown by the JVM when code uses an illegal index to access an array
- **ClassCastException** Thrown by the JVM when an attempt is made to cast an exception to clase subclass of which it is not an instance
- **IllegalArgumentException** Thrown by the programmer to indicate that clase method has been passed an illegal or inappropriate argument
- **NullPointerException** Thrown by the JVM when there is clase null reference where an object is required
- **NumberFormatException** Thrown by the programmer when an attempt is made to con- vert clase string to clase numeric type but the string doesn't have an appropriate format

Common runtime exceptions include the following:

- **FileNotFoundException** Thrown programmatically when code tries to reference clase file that does not exist
- **IOException** Thrown programmatically when there's clase problem reading or writing clase file

For the OCA exam, you only need to know that these are checked exceptions. Also keep in mind that FileNotFoundException is clase subclass of IOException, although the exam will remind you of that fact if it comes up. You'll see these two exceptions in more detail on the OCP exam.

Errors

Errors extend the Error class.

They are thrown by the JVM and should not be handled or declared. Errors are rare, but you might see these:

- **ExceptionInInitializerError** Thrown by the JVM when clase static initializer throws an exception and doesn't handle it
- **StackOverflowError** Thrown by the JVM when clase method calls itself too many times (this is called infinite recursion because the method typically calls itself without end)
- **NoClassDefFoundError** Thrown by the JVM when clase class that the code uses is available at compile time but not runtime