**OCA Oracle Certified Associate – Java SE 8 – Recap**

**Chapter 1** : Java Building Blocks

*Java Class Structure*

An object is a runtime instance of a class in memory. All the objects of different classes represent the state of the program.

Members of the class → Fields(Variables) + Methods

Variable → hold the state of the program.

Methods → Operate on variables state.

Keyword: A word with special meaning.

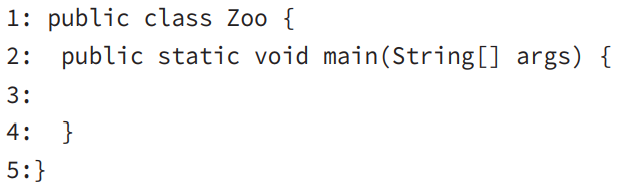
Method: Operation that can be called from other classes.

Method signature: full declaration of a method

*Main method*

A Java program begins execution with its main() method. Main method is the gateway between the startup of a Java process (managed by JVM) and the beginning of the programmer’s code. JVM calls on the underlying system to allocate memory and CPU time, access files and so on.

Example:



To compile and run the code we type it in a file with name **Zoo.java** and we execute:

$ **javac** Zoo.java

$ **java** Zoo

To compile Java Code the file must have the extension: **.java** , while the **name of the file must be the same as the name of the class**. The result of the compilation is a file of ***bytecode*** by the same name, but with the **.class** extension instead of **.java**. **Bytecode** consists of **instructions** that the **JVM** knows how to **execute**.

Main() →

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| public | static | void | main | (String[] args) |  |
| Access modifier, declares the method’s level of exposure to potential callers in the program | Binds the method to the class so it can be called by just the class name ( no objects is needed to be created by Java to call main method)  Presence of a non-static main method will throw an Exception | Return type.  Void return types are preferred when changing an object state | Name | Parameter list. It consists of an array of String objects.  args is only the name of the parameter list. It can be any other name, it just indicates that these are command line arguments.  When giving arguments they are separated by spaces. If you need space inside a word use double quotes: “San Diego” |  |

*Import statements*

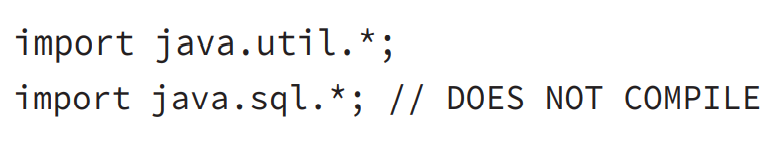
Java classes are grouped into packages. The import statements tell the compiler which package to look in to find a class. This is simlilar to how mailing a letter works. Importing all the classes of a package is done through **wildcard** → **\*. *This kind of approach DOES NOT slow down program. (Wildcard only importes all classes-files, not packages and classes inside nested packages)***

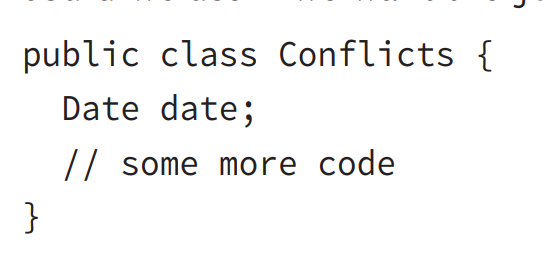
Package: *java.lang* → Is automatically imported in each Java Program.

Classes in the same package are automatically imported.

*Naming conflicts*

When a class (name) used in a program, is present in multiple packages imported , Java will not compile the code. (ambiguous types) for ex. There are 2 Date classes in Java: java.util.Date and java.sql.Date.





If you must use 2 classes with the same name inside one program you can use the qualified names of each or at leas one.

public class Conflicts {

java.util.Date date;

java.sql.Date sqlDate;

}

*Instance initilalizer blocks*

Code Block outside a method → **instance initializer.**

**public class Chicken {**

private String field = “String”;

**{** System.out.println(“Set field to String”); **}**

**void someMethod(){ }**

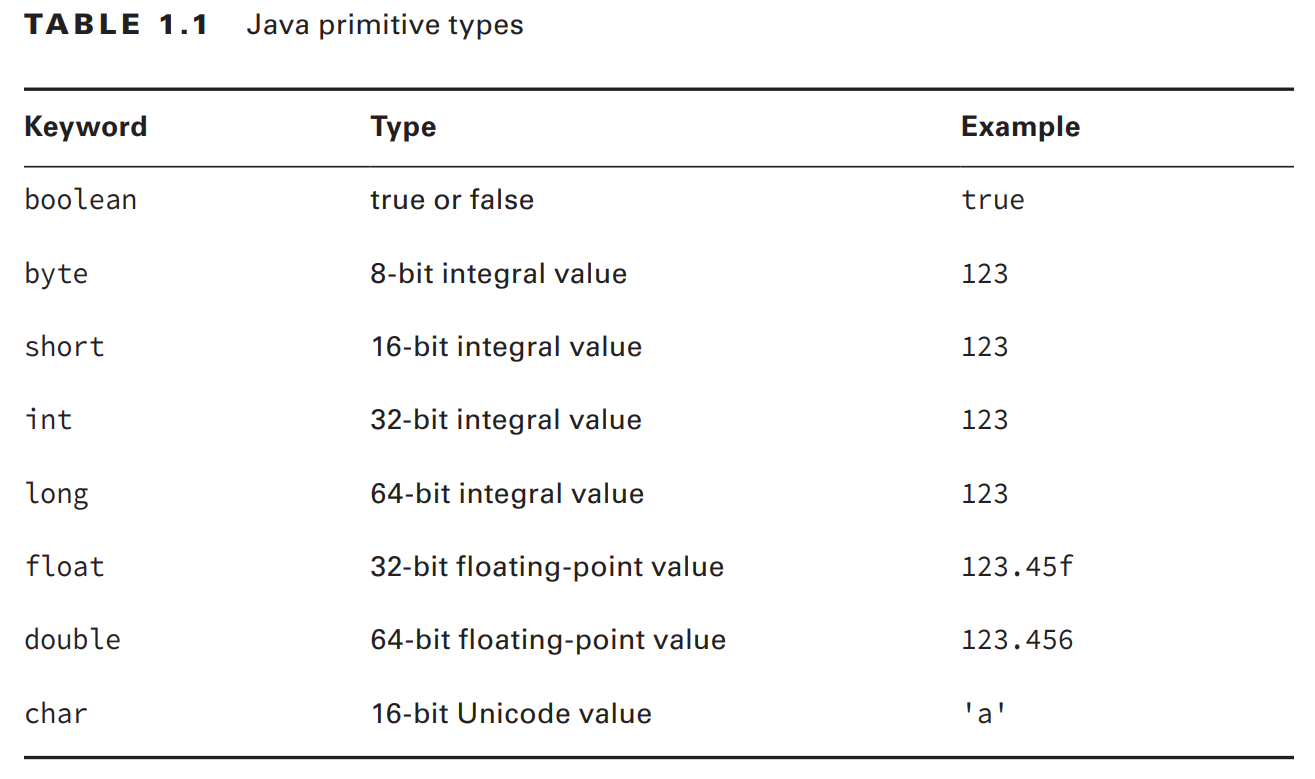
**}**

*Fields and instance initializers are run*

**1)** *In the order they appear in*

**2)** *Before the constructor*

***Object references vs Primitives***

**

When a number is present in the code, it is called a literal.

Java allows you to specify digits in several other formats:

* octal
* hexadecimal (0xFF)
* binary

Number literals can have multiple underscores in order to make them easier to read:

for example:

int million = 1000000;

int million = 1\_000\_000;

Underscores can **NOT** be added in this places :

* in the beginning of the literals
* in the end of the literals
* right before a decimal point
* right after a decimal point

**Reference types:**

A reference type refers to an object. Unlike primitive types that hold their values in memory where the variable is allocated,references do not hold the value of the object they refer to. Instead they hold a reference which points to an object by storing the memory address where the object is located. Unlike other languages, Java does not allow to learn what the physical memory address is. You can only use the reference to refer to the object.

A value is assigned to a reference in one of 2 ways:

* 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

**An object in memory can be accessed only via reference.**

***Key differences***

Reference types can be assigned to null (they don’t currently point to an object) , while primitive values can’t. Trying to set a primitive type to null will give a compiler error and not compile.

**Declaring and initializing variables**

A variable is a name for a piece of memory that stores data. When you declare a variable you need to state the variable type along with giving it a name.

After declaring a variable we can give it a value. This is called initializing.

**Identifiers**

Legal identifiers:

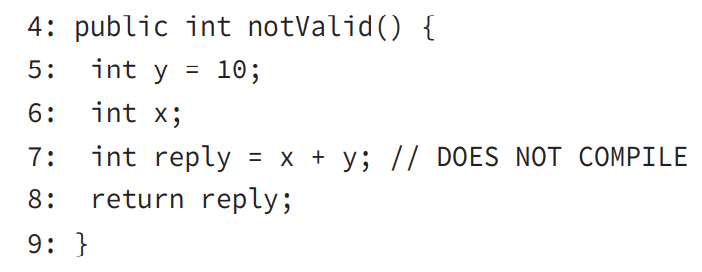
* The name must begin with a letter or symbol $ , \_ ;
* Subsequent characters may also have numbers
* The same name cannot be used as a Java reserved word , so you are not allowed to use it.

(You can’t use **const** or **goto** since java thinks they might be used in the future)

**Local Variables**

A local variable is a variable defined within a method. Local variables must be initialized before use. They do not have default value and contain garbage data until initialized. The compiler will not let you read an uninitialized value.

*Following code generates an error:*



Be careful of uninitialized variables with conditions ex :

**int** i ;

**if** (condition) {

i = 0;

}

**return** i; // does not compile – compiler knows that variable i might not be initialized if the condition //evaluates to false

**Instance and class variables**

Variables that are **not local variables** are known as **instance** **variables** or class variables. **Instance** **variables** are also called fields. However, **class variables** are **shared** across multiple objects.

A class variable has the keyword **static** before it.

Instance and class variables do not require initialization. As soon as they are declared they are given a default value.

Default values of instance/class variables:

|  |  |
| --- | --- |
| **Type** | **Default initialization value** |
| boolean | false |
| byte,short,int,long | 0 (type’s bit length) |
| float,double | 0.0 |
| char | ‘\u000’ (NUL) |
| All object references (including String and everything else ) | null |

**Variables scope**

Each block of code ({…}) has its own scope and variables declared within a scope cannot be used outside of it.

Nested blocks:

Variables declared in a block which contains smaller blocks inside , can be used inside the smaller blocks.

**Local** variables : in **scope** from declaration to **end of block**

**Instance** variables: in **scope** from declaration until object **garbage is collected**

**Class** variables: in **scope** from declaration until **program ends**

**Ordering elements in a Class**

|  |  |  |  |
| --- | --- | --- | --- |
| **Element** | **Example** | **Required ?** | **Where does it go ?** |
| Package declaration | package abc; | NO | First line |
| Import statement | import java.util.\*; | NO | Immediately after package |
| Class declaration | public class C | YES | Immediately after import |
| Field declaration | int value; | NO | Anywhere inside class |
| Method declaration | void method() | NO | Anywhere inside class |

Multiple classes can be defined inside a file , but ONLY ONE can be public.

**Destroying objects**

Java provides a garbage collector to automatically look for objects that aren’t needed anymore. All Java objects are stored in the program’s memory heap. Heap is a pool of unused memory that is allocated to our Java program. The heap may be quite large, depending on your environment, but there is always a limit to its size. If your program keeps instantiating objects and leaving them on the heap, eventually it will run out of memory.

**Garbage Collection**

Garbage collection refers to the process of automatically freeing memory on the heap by deleting objects that are no longer reachable in your program.

System.gc() → Calls the garbage collector, but IS NOT GUARANTEED to run. It only suggests that now (when the method is called) might be a good time to garbage collect.

*When is garbage collection performed ? When is an object eligible for garbage collection ?*

An object will remain on the heap until it is no longer reachable. An object is no longer reachable when one of 2 situations occur:

* **The object no longer has any reference pointing on it**
* **All references to the object have gone out of scope**

Do not confuse a reference with the object that it refers to; they 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.

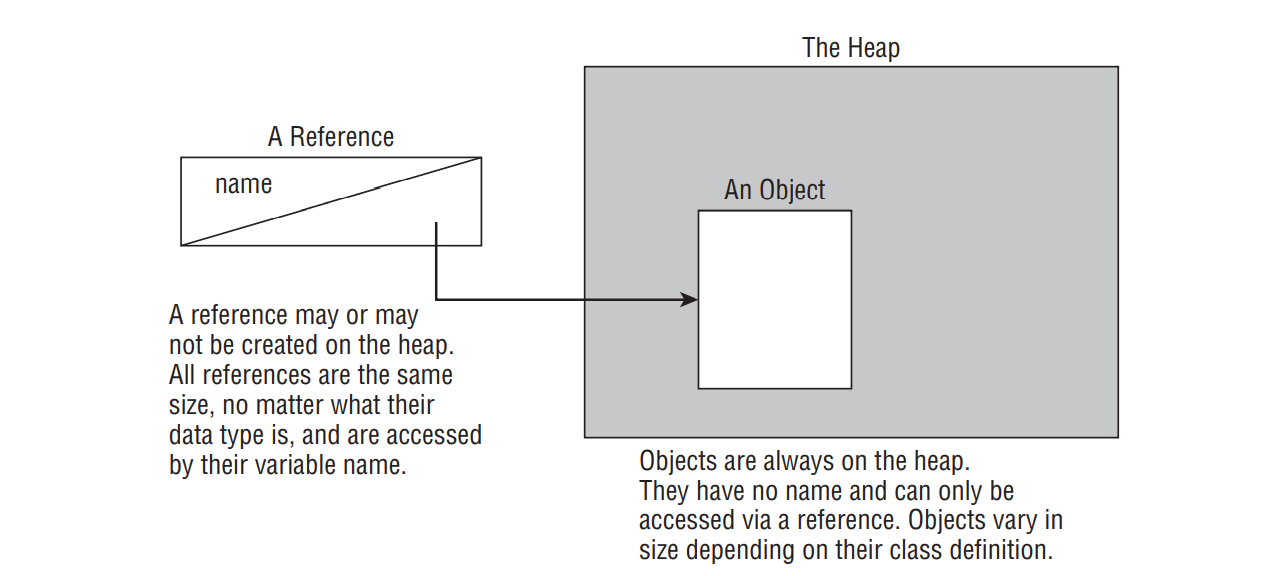


Figure out when each object is eligible for gc.

public class Scope {

public static void main(String[] args) {

String one , two;

one = new String(“a”);

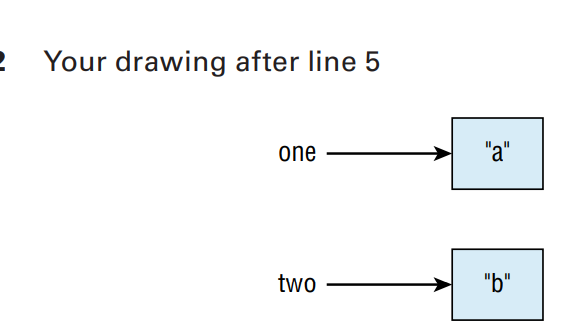
5: two = new String(“b”);

one = two; // *object [a] is now eligible for GC*

String three = one;

one = null; // *object [b] is now eligible for GC*

}

}

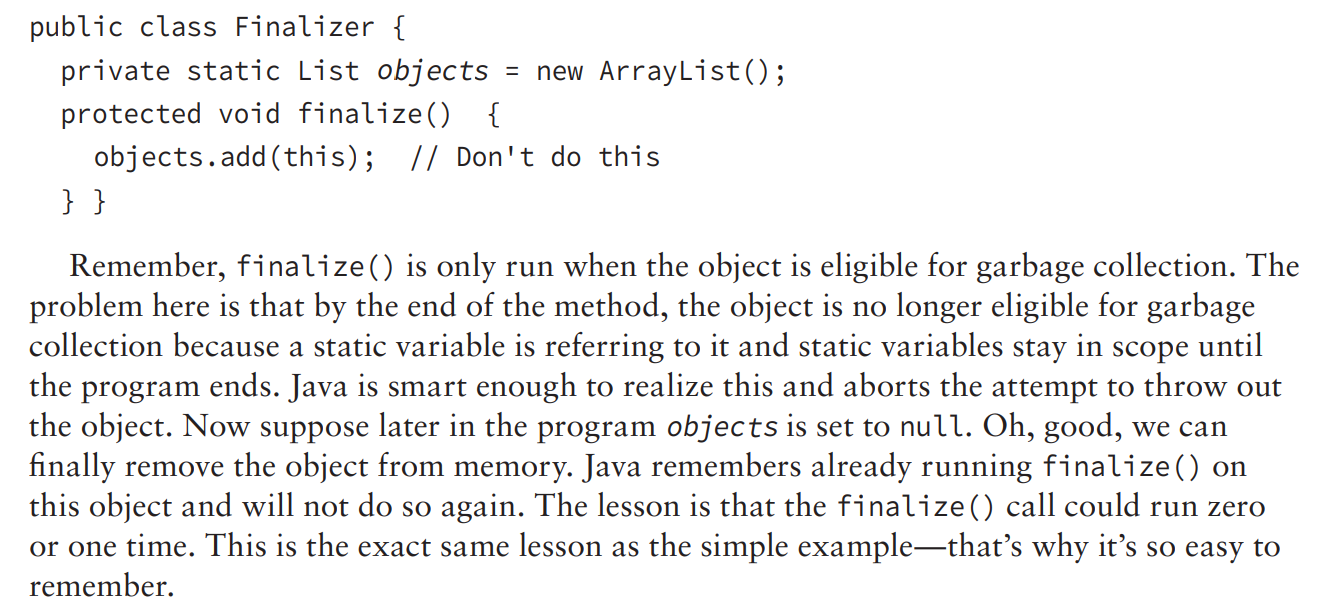
**finalize() method**

Java allows objects to implement a method called finalize() that might get called. This method gets called if the garbage collector tries to collect the object. If the garbage collector doesn’t run, the method does not get called. If gc fails to collect the object and tries to run it again later the method does not get called a second time.

So finalize() :

**1) Might not get called**

**2) Never gonna be called twice**

****

So if an object is referred inside a static variable it will stay in scope until the end of program (recall static variable scope). When Java goes to call the GC this object is referenced in a static List so gc does not run. When the object later is set to null , Java does not call finalize() a second time. Cause Java is smart :)