Java 11 1Z0-819 - Chapter 2

Following 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.

Understanding Data Types

For the exam, you should be aware that `short` and `char` are closely related, as both are stored as integral types with the same 16-bit length. The primary difference is that `short` is *signed*, which means it splits its range across the positive and negative integers. Alternatively, `char` is *unsigned*, which means range is strictly positive including `0`. Therefore, `char` can hold a higher positive numeric value than `short`, but cannot hold any negative numbers.

The compiler allows them to be used interchangeably in some cases, as shown here:

```
1 short bird = 'd';
2 char mammal = (short)83;
```

Writing Literals

- Octal (digits `0`–`7`), which uses the number `0` as a prefix—for example, `017`
- Hexadecimal (digits `0`–`9` and letters `A`–`F`/`a`–`f`), which uses `0x` or `0X` as a prefix—for example, `0xFF`, `0xff`, `0XFf`. Hexadecimal is case insensitive so all of these examples mean the same value.
- Binary (digits `0`–`1`), which uses the number `0` followed by `b` or `B` as a prefix—for example, `0b10`, `0B10`

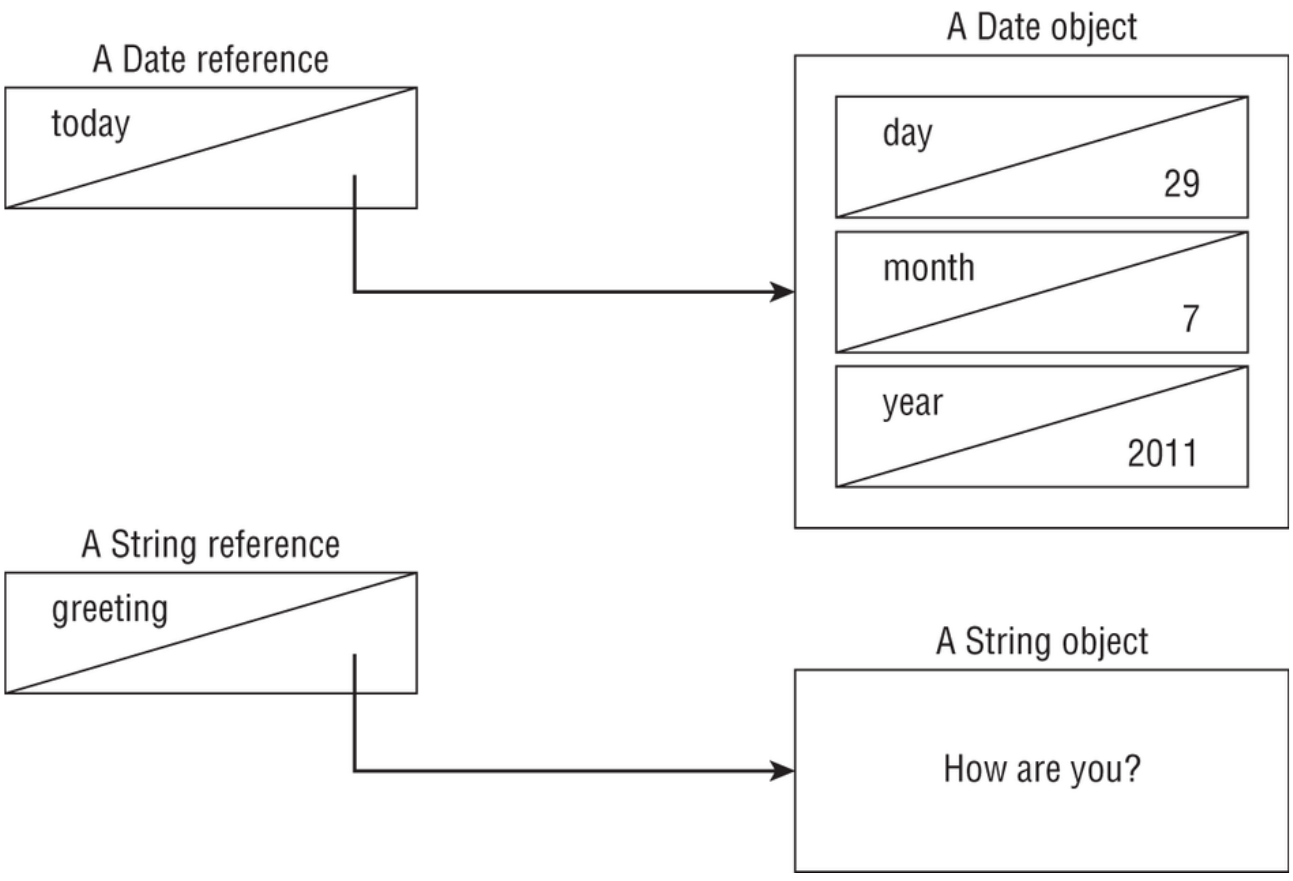


FIGURE 2.1 An object in memory can be accessed only via a reference.

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. A *reserved word* is special word that Java has held aside so that you are not allowed to use it. Remember that Java is case sensitive, so you can use versions of the keywords that only differ in case. Please don't, though.

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. To use this feature, you just type `var` instead of the primitive or reference type. Here's an example:

```
public void whatTypeAmI() {
   var name = "Hello";
   var size = 7;
}
```

The formal name of this feature is *local variable type inference*. Let's take that apart. First comes *local variable*. This means just what it sounds like. You can only use this feature for local variables. The exam may try to trick you with code like this:

```
public class VarKeyword {
   var tricky = "Hello"; // DOES NOT COMPILE
}
```

Examples with *var*

Let's go through some more scenarios so the exam doesn't trick you on this topic! Do you think the following compiles?

```
3:  public void doesThisCompile(boolean check) {
4:     var question;
5:     question = 1;
6:     var answer;
7:     if (check) {
8:        answer = 2;
9:     } else {
10:       answer = 3;
11:    }
12:    System.out.println(answer);
13: }
```

The code does not compile. Remember that for local variable type inference, the compiler looks only at the line with the declaration. Since `question` and `answer` are not assigned values on the lines where they are defined, the compiler does not know what to make of them. For this reason, both lines 4 and 6 do not compile.

```
5:    var a = 2, b = 3;  // DOES NOT COMPILE
```

In other words, Java does not allow `var` in multiple variable declarations.

the following does compile:

```
17: var o = (String)null;
```

Remember that `var` is only used for local variable type inference!

Time for two more examples. Do you think this is legal?

```
package var;

public class Var {
   public void var() {
      var var = "var";
   }
   public void Var() {
      Var var = new Var();
   }
}
```

Believe it or not, this code does compile. Java is case sensitive, so `Var` doesn't introduce any conflicts as a class name. Naming a local variable `var` is legal. Please don't write code that looks like this at your job! But understanding why it works will help get you ready for any tricky exam questions Oracle could throw at you!

There's one last rule you should be aware of. While `var` is not a reserved word and allowed to be used as an identifier, it is considered a reserved type name. A *reserved type name* means it cannot be used to define a type, such as a class, interface, or `enum`. For example, the following code snippet does not compile because of the class name:

```
public class var {  // DOES NOT COMPILE
   public var() {
   }
}
```

Review of *var* Rules

We complete this section by summarizing all of the various rules for using `var` in your code. Here's a quick review of the `var` rules:

1. A `var` is used as a local variable in a constructor, method, or initializer block.

   A `var` cannot be used in constructor parameters, method parameters, instance variables, or class variables.

   A `var` is always initialized on the same line (or statement) where it is declared.

   The value of a `var` can change, but the type cannot.

   A `var` cannot be initialized with a `null` value without a type.

   A `var` is not permitted in a multiple-variable declaration.

   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

```
public void eat(int piecesOfCheese) {
   int bitesOfCheese = 1;
}
```

There are two local variables in this method. The `bitesOfCheese` variable is declared inside the method. The `piecesOfCheese` variable is a method parameter and, as discussed earlier, it also acts like a local variable in terms of garbage collection and scope. Both of these variables are said to have a *scope* local to the method. This means they cannot be used outside of where they are defined.

Destroying Objects

Remember from Chapter 1, your code isn't the only process running in your Java program. Java code exists inside of a Java Virtual Machine (JVM), which includes numerous processes independent from your application code. One of the most important of those is a built-in garbage collector.

All Java objects are stored in your program memory's *heap*. The heap, which is also referred to as the *free store*, represents a large pool of unused memory allocated to your Java application. The heap may be quite large, depending on your environment, but there is always a limit to its size. After all, there's no such thing as a computer with infinite memory. If your program keeps instantiating objects and leaving them on the heap, eventually it will run out of memory and crash.

Garbage Collector

Calling *System.gc()*

Java includes a built-in method to help support garbage collection that can be called at any time.

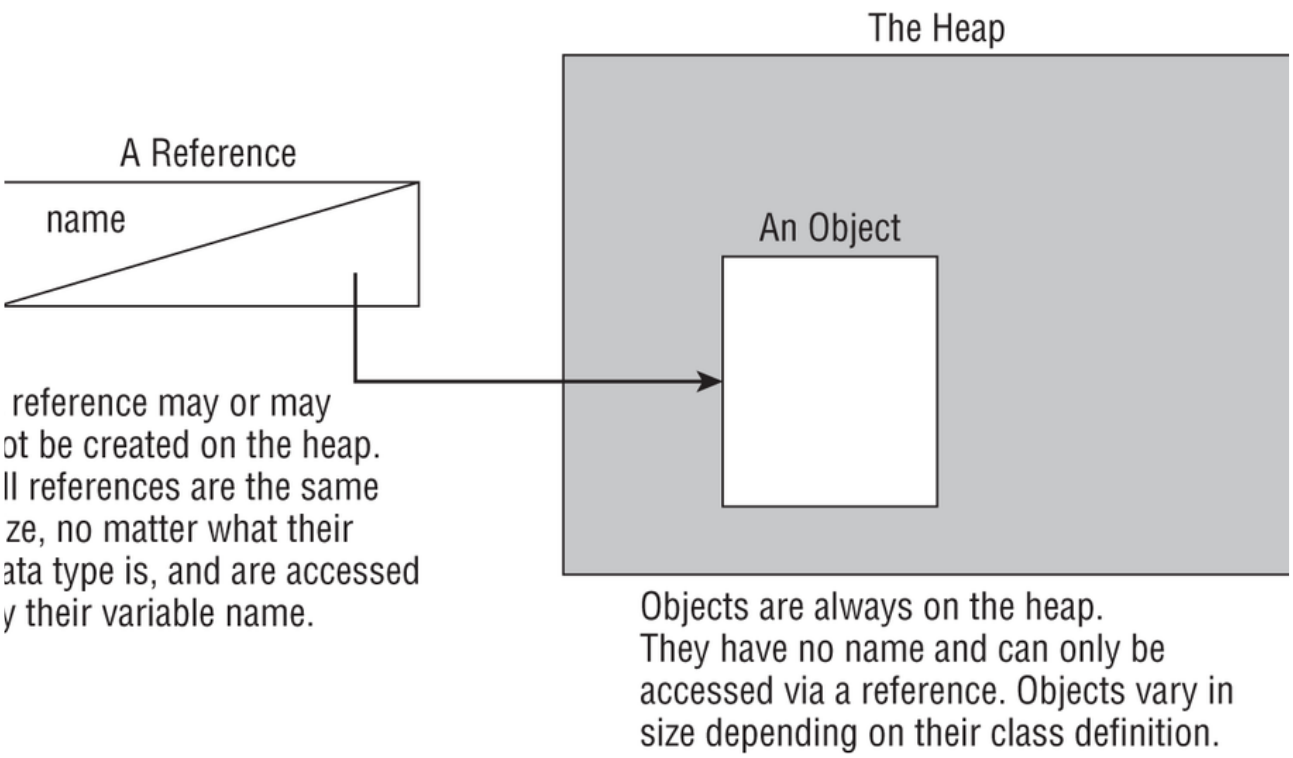
```
public static void main(String[] args) {
   System.gc();
}
```

What is the `System.gc()` command *guaranteed* to do? Nothing, actually. It merely *suggests* that the JVM kick off garbage collection. The JVM may perform garbage collection at that moment, or it might be busy and choose not to. The JVM is free to ignore the request.

When is `System.gc()` *guaranteed* to be called by the JVM? Never, actually. While the JVM will likely run it over time as available memory decreases, it is not guaranteed to ever actually run. In fact, shortly before a program runs out of memory and throws an `OutOfMemoryError`, the JVM will *try* to perform garbage collection, but it's not guaranteed to succeed.

An object is no longer reachable when one of two situations occurs:

- The object no longer has any references pointing to it.



FINALIZE()

Java allows objects to implement a method called `finalize()`. This feature can be confusing and hard to use properly. In a nutshell, the garbage collector would call the `finalize()` method once. If the garbage collector didn't run, there was no call to `finalize()`. If the garbage collector failed to collect the object and tried again later, there was no second call to `finalize()`.

This topic is no longer on the exam. In fact, it is deprecated in `Object` as of Java 9, with the official documentation stating, "The finalization mechanism is inherently problematic." We mention the `finalize()` method in case Oracle happens to borrow from an old exam question. Just remember that `finalize()` can run zero or one times. It cannot run twice.