Java Building Blocks

1. Creating Objects

En las siguientes secciones, veremos los constructores, los campos de objeto, los inicializadores de instancias y el orden en que se inicializan los valores.

1.1. CALLING CONSTRUCTORS

Para crear una instancia de una clase, todo lo que tiene que hacer es escribir `new` antes del nombre de la clase y agregar paréntesis después. Aquí tienes un ejemplo:

```
Park p = new Park();
```

Primero declara el tipo que va a crear (`Park`) y le da a la variable un nombre (`p`). Esto le da a Java un lugar para almacenar una referencia al objeto. A continuación, escribe nuevo `Park()` para crear el objeto.

`Park()` se parece a un método ya que se sigue con paréntesis, se llama un constructor, que es un tipo especial de método que crea un nuevo objeto. Por ejemplo:

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

Hay dos puntos clave a tener en cuenta sobre el constructor:
(1) el nombre del constructor coincide con el nombre de la clase y
(2) no hay ningún tipo de retorno.

El propósito de un constructor es inicializar campos, aunque usted puede poner cualquier código allí. Otra forma de inicializar los campos es hacerlo directamente en la línea en la que se declaran:

```
public class Chicken {
  int numEggs = 0;// initialize on line
  String name;
  public Chicken() {
    name = "Duke";// initialize in constructor
  } }
```

Para la mayoría de las clases, no es necesario codificar un constructor: el compilador proporcionará un constructor predeterminado "no hacer nada".

1.2. READING AND WRITING MEMBER FIELDS

Es posible leer y escribir variables de instancia directamente de quien lo llama:

```
public class Swan {
  int numberEggs;// instance variable
  public static void main(String[] args) {
    Swan mother = new Swan();
    mother.numberEggs = 1;    // set variable
    System.out.println(mother.numberEggs);  // read variable
  }
}
```

La lectura de una variable se conoce como obtenerla. La clase obtiene `numberEggs` directamente para imprimirlo. Escribir a una variable se conoce como establecerla. Esta clase establece `numberEggs` a `1`.

Incluso puede leer y escribir campos directamente en la línea declarándolos:

```
1: public class Name {
2:   String first = "Theodore";
3:   String last = "Moose";
4:   String full = first + last;
5: }
```

1.3. EXECUTING INSTANCE INITIALIZER BLOCKS

Cuando aprendiste acerca de los métodos, viste llaves (`{}`). El código entre las llaves se denomina bloque de código.

A veces los bloques de código están dentro de un método. Éstos se ejecutan cuando se llama al método. Otras veces, los bloques de código aparecen fuera de un método. Estos se llaman inicializadores de instancia(*initializers*). Por ejemplo:

```
1: public class Bird {
2:    public static void main(String[] args) {
3:      { System.out.println("Feathers"); }
4:    }
5:    { System.out.println("Snowy"); }
6: }
```

Hay tres bloques de código y un inicializador de instancia. Contar bloques de código es fácil: sólo cuenta el número de pares de llaves. No importa que un conjunto de llaves esté dentro del método `main()`, todavía cuenta.

1.4. FOLLOWING ORDER OF INITIALIZATION

Cuando se escribe un código que inicializa campos en varios lugares, debe realizar un seguimiento del orden de inicialización. Usted necesita recordar:

1. Los campos y los bloques de inicialización de instancia se ejecutan en el orden en que aparecen en el archivo.
2. El constructor se ejecuta después de ejecutar todos los campos y bloques de inicialización de instancia.

Por ejemplo:

```
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); } }
```

Al ejecutarlo, obtenemos:

```
setting field
setting constructor
Tiny
```

Comenzamos con el método `main()` porque ahí es donde Java inicia la ejecución. En la línea 9, llamamos al constructor de `Chick`. Java crea un nuevo objeto. Primero inicializa el nombre a "`Fluffy`" en la línea 2. Luego ejecuta la sentencia `println` en el inicializador de instancia en la línea 3. Una vez que todos los campos y los inicializadores de instancia se hayan ejecutado, Java regresa al constructor. La línea 5 cambia el valor del nombre a "`Tiny`" y la línea 6 imprime otra declaración. En este punto, el constructor se ejecuta y vuelve a la instrucción de impresión en la línea 10.

El orden es importante para los campos y bloques de código. **No se puede hacer referencia a una variable antes de que se haya inicializado**:

```
{ System.out.println(name); }  // DOES NOT COMPILE
private String name = "Fluffy";
```

Por ejemplo:

```
public class Egg {
  public Egg() {
    number = 5;
  }
  public static void main(String[] args) {
    Egg egg = new Egg();
    System.out.println(egg.number);
  }
  private int number = 3;
  { number = 4; } }
```

El código anterior nos imprime 5.

2. Understanding Data Types

Las aplicaciones Java contienen dos tipos de datos: tipos primitivos y tipos de referencia.

2.1. USING PRIMITIVE TYPES

Java tiene ocho tipos de datos incorporados, denominados tipo primitivo Java:

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	123L
float	32-bit floating-point value	123.45f
double	64-bit floating-point value	123.456
char	16-bit Unicode value	'a'

Algunos importantes de la tabla anterior:

1. `float` y `double` se utilizan para valores de punto flotante (decimal).
2. Un `float` requiere la letra f que sigue al número para que Java sepa que es del tipo `float`.
3. `byte`, `short`, `int` y `long` se utilizan para números sin puntos decimales.
4. Cada tipo numérico utiliza el doble de bits que el tipo similar más pequeño. Por ejemplo, `short` usa el doble de bits que el `byte`.
5. Todos los tipos numéricos tiene signo en Java. Esto significa que reservan uno de sus bits para cubrir un rango negativo. Por ejemplo, el rango de bytes de -128 a 127.

SIGNED AND UNSIGNED: SHORT AND CHAR

Para el examen, debe tener en cuenta que short y char están estrechamente relacionados, ya que ambos se almacenan como tipos integrales con la misma longitud de 16 bits. La principal diferencia es que short está firmado y char no está firmado. El compilador permite que se usen indistintamente en algunos casos, como se muestra aquí:

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

La impresión de cada variable muestra el valor asociado con su tipo.

```
System.out.println(bird);   // Prints 100
System.out.println(mammal); // Prints S
```

Si intenta establecer un valor fuera del rango de short o char, el compilador informará un error:

```
short reptile = 65535;  // DOES NOT COMPILE
char fish = (short)-1;  // DOES NOT COMPILE
```

Esto se verá en el Capítulo 3, "Operadores".

FLOATING-POINT NUMBERS AND SCIENTIFIC NOTATION

En la mayoría de los sistemas informáticos, los números de coma flotante se almacenan en notación científica. Esto significa que los números se almacenan como dos números, `a` y `b`, de la forma $a \times 10^b$.

Por ejemplo, puede almacenar un valor de 3×10^{200} en un doble, lo que requeriría mucho más de 8 bytes si cada dígito se almacenara sin notación científica (84 bytes). Para lograr esto, solo almacena la primera docena de dígitos del número. El nombre notación científica proviene de la ciencia, donde a menudo solo se requieren los primeros dígitos significativos para un cálculo.

2.1.1. Writing Literals

Cuando un número está presente en el código, se llama literal. De forma predeterminada, Java asume que está definiendo un valor `int` con un literal.

Otra forma de especificar números es cambiar la "base":

1. Octal (dígitos 0-7), que utiliza el número `0` como un prefijo-por ejemplo, `017`.
2. Hexadecimal (dígitos 0-9 y letras A-F), que utiliza el número 0 seguido por `x` o `X` como prefijo -por ejemplo, `0xFF`.
3. Binario (dígitos 0-1), que utiliza el número `0` seguido de `b` o `B` como prefijo, por ejemplo, `0b10`.

Por ejemplo:

```
System.out.println(56);       // 56 - int
System.out.println(0b11);     // 3 - binario
System.out.println(017);      // 15 - octal
System.out.println(0x1F);     // 31 - hexadecimal
```

2.1.2. Literals and the Underscore Character

Lo último que necesita saber acerca de literales numéricos es una característica añadida en Java 7. Puede tener subrayados en números para que sean más fáciles de leer:

```
int million1 = 1000000;
int million2 = 1_000_000;
```

Puede añadir guion bajo en cualquier lugar, excepto al principio de un literal, el final de un literal, justo antes de un punto decimal o justo después de un punto decimal:

```
double notAtStart = _1000.00;         // DOES NOT COMPILE
double notAtEnd = 1000.00_;           // DOES NOT COMPILE
double notByDecimal = 1000_.00;       // DOES NOT COMPILE
double annoyingButLegal = 1_00_0.0_0;  // this one compiles
double reallyUgly = 1__________2;      // Also compiles
```

2.2. USING REFERENCE TYPES

Un tipo de referencia se refiere a un objeto (una instancia de clase). A diferencia de los tipos primitivos que mantienen sus valores en la memoria donde se asigna la variable, las referencias no tienen el valor del objeto al que se refieren. En su lugar, una referencia "apunta" a un objeto almacenando la dirección de memoria donde se encuentra el objeto, un concepto denominado puntero. A diferencia de otros lenguajes, Java no te permite saber cuál es la dirección de la memoria física.

Por ejemplo, dado el siguiente código:

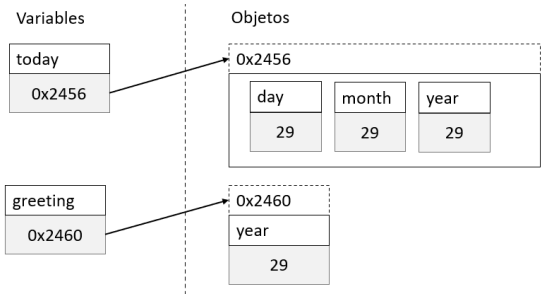
```
java.util.Date today;
String greeting;
```

La variable `today` es una referencia de tipo `Date` y sólo puede apuntar a un objeto `Date`. La variable de `greeting` es una referencia que sólo puede apuntar a un objeto `String`. Se asigna un valor a una referencia de dos maneras:

1. Una referencia puede ser asignada a otro objeto del mismo tipo.
2. Se puede asignar una referencia a un nuevo objeto utilizando la palabra clave `new`.

Por ejemplo:
```
today = new java.util.Date();
greeting = "How are you?";
```

La variable `today` se puede utilizar para acceder a los diversos campos y métodos del objeto `Date`. Gráficamente:



2.3. DISTINGUISHING BETWEEN PRIMITIVES AND REFERENCE TYPES

Hay algunas diferencias importantes que debe saber entre primitivos y tipos de referencia:

1. En primer lugar, se pueden asignar `null` a los tipos de referencia, lo que significa que actualmente no se refieren a un objeto. Tipos primitivos le dará un error de compilador si intenta asignarlos a `null`. Por ejemplo:

```
int value = null;    // DOES NOT COMPILE
String s = null;
```

2. A continuación, los tipos de referencia se pueden utilizar para llamar a métodos cuando no apuntan a `null`:
```
String reference = "hello";
int len = reference.length();
int bad = len.length(); // DOES NOT COMPILE
```

3. Finalmente, observe que todos los tipos primitivos tienen nombres en minúscula. Todas las clases en Java comienzan con mayúsculas.

3. Declaring Variables

Una variable es un nombre para una pieza de memoria que almacena datos. Cuando declara una variable, debe indicar el tipo de variable junto un nombre. Por ejemplo, el código siguiente declara dos variables. Uno se llama `zooName` y es del tipo `String`. El otro se llama `numberAnimals` y es del tipo `int`:

```
String zooName;
int numberAnimals;
```

Ahora que hemos declarado una variable, podemos darle un valor. Esto se llama inicializar una variable:

```
zooName = "The Best Zoo";
numberAnimals = 100;
```

Puesto que a menudo desea inicializar una variable de inmediato, puede hacerlo en la misma declaración que la declaración:

```
String zooName = "The Best Zoo";
```

```
int numberAnimals = 100;
```

3.1. IDENTIFYING IDENTIFIERS

Sólo hay tres reglas a tener en cuenta para los identificadores legales:

1. El nombre debe comenzar con una letra o el símbolo $ o _.
2. Los caracteres subsiguientes también pueden ser números.
3. No puede utilizar el mismo nombre que una palabra Java reservada. Recuerde que Java distingue entre mayúsculas y minúsculas, por lo que puede utilizar versiones de las palabras clave que sólo difieren en el caso. Por favor, no, aunque

La siguiente es una lista de todas las palabras reservadas en Java. `const` y `goto` no se utilizan realmente en Java:

abstract	assert	boolean	break	byte
case	catch	char	class	const*
continue	default	do	double	else
enum	extends	false	final	finally
float	for	goto*	if	implements
import	instanceof	int	interface	long
native	new	null	package	private
protected	public	return	short	static
strictfp	super	switch	synchronized	this
throw	throws	transient	true	try
void	volatile	while	_(guion bajo)	

Los siguientes ejemplos son legales:

```
okidentifier
$OK2Identifier
_alsoOK1d3ntifi3r
__SStillOkbutKnotsonice$
```

Los siguientes no son legales:

```
int 3DPointClass;     // identifiers cannot begin with a number
byte hollywood@vine; // @ is not a letter, digit, $ or _
String *$coffee;      // * is not a letter, digit, $ or _
double public;        // public is a reserved word
short _;              // a single underscore is not allowed
```

3.1.1. Style: camelCase

Aunque puedes hacer cosas descabelladas con nombres de identificadores, no deberías. Java tiene convenciones para que el código sea legible y coherente. Esta consistencia incluye `CamelCase`. En `CamelCase`, cada palabra comienza con una letra mayúscula. ¿Qué le gustaría leer: `Thisismyclass` o `ThisIsMyClass` nombre? El examen usará sobre todo convenciones comunes para los identificadores, pero no siempre.

IDENTIFIERS IN THE REAL WORLD

La mayoría de los desarrolladores de Java siguen estas convenciones para los nombres de identificador:

1. Los nombres de método y variables comienzan con una letra minúscula seguida por `CamelCase`.

2. Los nombres de las clases comienzan con una letra mayúscula seguida por `CamelCase`. No inicie ningún identificador con $. El compilador utiliza este símbolo para algunos archivos.

Además, sepa que las letras válidas en Java no son sólo caracteres del alfabeto inglés. Java admite el conjunto de caracteres Unicode, por lo que hay más de 45.000 caracteres que pueden iniciar un identificador Java legal.

3.1.2. Style: snake_case

Otro estilo que puede ver tanto en el examen como en el mundo real o en otros idiomas se llama *snake case*, a menudo escrito como snake_case para enfatizar. Simplemente usa un guion bajo (_) para separar palabras, a menudo completamente en minúsculas. El ejemplo anterior se escribiría como `this_is_my_class_name` en snake_case.

Note
Si bien tanto camelCase como snake_case tienen una sintaxis perfectamente válida en Java. Teniendo esto en cuenta, Oracle (y Sun antes) recomienda que todos usen camelCase para los nombres de clases y variables. Sin embargo, existen algunas excepciones. Los valores constantes `final static` a menudo se escriben en snake_case, como `THIS_IS_A_CONSTANT`. Además, los valores de enumeración tienden a escribirse con snake_case, como en `Color.RED`, `Color.DARK_GRAY`, etc.

3.2. DECLARING MULTIPLE VARIABLES

También puede declarar e inicializar varias variables en la misma sentencia:

```
void sandFence() {
   String s1, s2;
   String s3 = "yes", s4 = "no";
}
```

Se declararon cuatro variables `String`: `s1`, `s2`, `s3` y `s4`. Puede declarar muchas variables en la misma declaración siempre que sean del mismo tipo. También puede inicializar cualquiera o todos estos valores en línea. ¿Cuántas variables cree que se declaran e inicializan en este código?

```
void paintFence() {
   int i1, i2, i3 = 0;
}
```

Como era de esperar, se declararon tres variables: `i1`, `i2` e `i3`. Sin embargo, sólo uno de esos valores se inicializó: `i3`. Cada fragmento separado por una coma es una pequeña declaración propia.

Dado:
```
int num, String value; // DOES NOT COMPILE
```
Este código no compila porque intenta declarar varias variables de diferentes tipos en la misma instrucción. El acceso directo para declarar varias variables en la misma sentencia sólo funciona cuando comparten un tipo.

Note
"Legal", "válido" y "compila" son sinónimos en el mundo de los exámenes Java

Por ejemplo:

```
4: boolean b1, b2;
```

```
5: String s1 = "1", s2;
6: double d1, double d2;
7: int i1; int i2;
8: int i3; i4;
```

1. La primera afirmación es legal. Declara dos variables sin inicializarlas.
2. La segunda afirmación también es legal. Declara dos variables e inicializa sólo una de ellas.
3. La tercera afirmación no es legal. Java no le permite declarar dos tipos diferentes. Si desea declarar varias variables en la misma sentencia, deben compartir la misma declaración de tipo y no repetirla. `doble d1, d2;` habría sido legal. Un punto y coma (;) separa las sentencias en Java. Simplemente sucede que hay dos declaraciones completamente diferentes en la misma línea.
4. La cuarta declaración de la línea 7 es legal. Aunque int aparece dos veces, cada uno está en una declaración separada. Un punto y coma (;) separa las declaraciones en Java. Da la casualidad de que hay dos declaraciones completamente diferentes en la misma línea.
5. La quinta declaración de la línea 8 no es legal. Nuevamente, tenemos dos declaraciones completamente diferentes en la misma línea. El segundo en la línea 8 no es una declaración válida porque omite el tipo.

4. Initializing Variables

Antes de poder usar una variable, necesita un valor. Algunos tipos de variables obtienen este valor automáticamente, y otros requieren que el programador lo especifique.

4.1. CREATING LOCAL VARIABLES

Una variable local es una variable definida dentro de un método. Las variables locales deben ser inicializadas antes de su uso. No tienen un valor predeterminado y contienen datos de basura hasta que se inicializan. Por ejemplo:

```
4: public int notValid() {
5:  int y = 10;
6:  int x;
7:  int reply = x + y; // DOES NOT COMPILE
8:  return reply;
9: }
```

El compilador también es lo suficientemente inteligente como para reconocer las inicializaciones que son más complejas. Por ejemplo:

```
public void findAnswer(boolean check) {
  int answer;
  int onlyOneBranch;
  if (check) {
    onlyOneBranch = 1;
    answer = 1;
  } else {
    answer = 2;
  }
  System.out.println(answer);
  System.out.println(onlyOneBranch); // DOES NOT COMPILE
}
```

Hay dos ramas de código. `answer` se inicializa en ambos de modo que el compilador es perfectamente feliz. `onlyOneBranch` sólo se inicializa si `check` es verdadera. El compilador sabe que existe la posibilidad de que la comprobación sea falsa, lo que resulta en un código no inicializado, y da un error de compilación.

4.2. PASSING CONSTRUCTOR AND METHOD PARAMETERS

Las variables que se pasan a un constructor o método se denominan parámetros de constructor o parámetros de método, respectivamente. Estos parámetros son variables locales que se han inicializado previamente. Las reglas para inicializar los parámetros del método y del constructor son las mismas, por lo que nos centraremos principalmente en los parámetros del método.

En el ejemplo anterior, `check` es un parámetro de método.

```
public void findAnswer(boolean check) {}
```

Eche un vistazo al siguiente método `checkAnswer()` en la misma clase:

```
public void checkAnswer() {
   boolean value;
   findAnswer(value);  // DOES NOT COMPILE
}
```

4.3. DEFINING INSTANCE AND CLASS VARIABLES

Las variables que no son variables locales se conocen como variables de instancia o variables de clase. Las variables de instancia también se denominan campos/atributos. Las variables de clase se comparten entre varios objetos. Puede decir que una variable es una variable de clase porque tiene la palabra clave `static` antes de ella.

Las variables de instancia y clase no requieren que se inicialicen. Tan pronto como se declaran estas variables, se les da un valor predeterminado. La siguiente tabla muestra los valores predeterminados:

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' (NULL)`
`All object references (everything else)`	`null`

4.4. INTRODUCING VAR

A partir de Java 10, tiene la opción de utilizar la palabra clave `var` en lugar del tipo para las variables locales en determinadas condiciones. Para usar esta función, simplemente escriba `var` en lugar del tipo primitivo o de referencia. Aquí tienes un ejemplo:

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

El nombre formal de esta característica es inferencia de tipo de variable local (*local variable type inference*). Vamos a dividir esto. Primero vamos con variable local.

4.4.1. Type Inference of var

Échale un vistazo a este ejemplo:

```
7:  public void reassignment() {
8:     var number = 7;
9:     number = 4;
10:    number = "five";  // DOES NOT COMPILE
11: }
```

En la línea 8, el compilador determina que queremos una variable `int`. En la línea 9, no tenemos problemas para asignarle un `int` diferente. En la línea 10, Java tiene un problema. Es equivalente a escribir esto:

```
int number = "five";
```

Entonces, el tipo de `var` no puede cambiar en tiempo de ejecución, pero ¿qué pasa con el valor? Eche un vistazo al siguiente fragmento de código:

```
var apples = (short)10;
apples = (byte)5;
apples = 1_000_000;  // DOES NOT COMPILE
```

...es equivalente a:

```
short apples = (short)10;
apples = (byte)5;
apples = 1_000_000;  // DOES NOT COMPILE
```

siguiente ejemplo:

```
7:  public void breakingDeclaration() {
8:     var silly
9:        = 1;
10: }
```

Este ejemplo es válido y compila, pero consideramos que la declaración y la inicialización de `silly` están sucediendo en la misma línea.

4.4.2. Examples with var

¿Crees que lo siguiente compila?

```
3:  public void doesThisCompile(boolean check) {
4:     var question;
5:     question = 1;
6:     var answer;
7:     if (check) {
8:        answer = 2;
9:     } else {
10:       ¿Crees que lo siguiente compila? = 3;
```

```
11:     }
12:     System.out.println(answer);
13: }
```

Dado que a la `question` y la `answer` no se les asignan valores en las líneas donde se definen, el compilador no sabe qué hacer con ellas.

Otro ejemplo:

```
4: public void twoTypes() {
5:     int a, var b = 3;  // DOES NOT COMPILE
6:     var n = null;      // DOES NOT COMPILE
7: }
```

Todos los tipos declarados en una sola línea deben ser del mismo tipo y compartir la misma declaración. No pudimos escribir `int a, int v = 3;`. Asimismo, esto no está permitido:

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

En otras palabras, Java no permite `var` en múltiples declaraciones de variables.

VAR AND NULL
Si bien una `var` no se puede inicializar con un valor `null` sin un tipo, se le puede asignar un valor `null` después de declararse. Por ejemplo:

```
13: var n = "myData";
14: n = null;
15: var m = 4;
16: m = null;  // DOES NOT COMPILE
```

La línea 14 se compila sin problemas porque `n` es de tipo `String`, que es un objeto. Por otro lado, la línea 16 no compila ya que el tipo de m es un primitivo int, al que no se le puede asignar un valor nulo. Lo siguiente se compila:

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

Dado que se proporciona el tipo, el compilador puede aplicar la inferencia de tipos y establecer el tipo de `var` en `String`.

Probemos con otro ejemplo:

```
public int addition(var a, var b) {  // DOES NOT COMPILE
   return a + b;
}
```

En este ejemplo, a y b son parámetros de método. Estas no son variables locales.

¿Crees que esto es legal?

```
package var;

public class Var {
```

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

Lo crea o no, este código compila. Si bien `var` no es una palabra reservada y se permite su uso como identificador, se considera un nombre de tipo reservado. Un nombre de tipo reservado (*reserved type name*) significa que no se puede utilizar para definir un tipo, como una clase, interfaz o enumeración.

4.4.3. Review of var Rules

Aquí hay una revisión rápida de las reglas de `var`:

1. Una `var` se utiliza como variable local en un constructor, método o bloque inicializador.
2. Una `var` no se puede utilizar en parámetros de constructor, parámetros de método, variables de instancia o variables de clase.
3. Una `var` siempre se inicializa en la misma línea (o declaración) donde se declara.
4. El valor de una `var` puede cambiar, pero el tipo no.
5. Una `var` no se puede inicializar con un valor nulo sin un tipo.
6. No se permite una `var` en una declaración de variables múltiples.
7. Una `var` es un nombre de tipo reservado, pero no una palabra reservada, lo que significa que se puede usar como un identificador excepto como un nombre de clase, interfaz o enumeración.

VAR IN THE REAL WORLD
Considere usar `var`:

```
PileOfPapersToFileInFilingCabinet pileOfPapersToFile =
     new PileOfPapersToFileInFilingCabinet();
```

Puedes ver cómo acortar esto sería una mejora sin perder ninguna información:

```
var pileOfPapersToFile = new PileOfPapersToFileInFilingCabinet();
```

5. Managing Variable Scope

¿Cuántas variables locales puedes ver en el ejemplo?:

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

Hay dos variables locales en este método. `bitesOfCheese` se declara dentro del método. `piecesOfCheese` se llama un parámetro del método. También es local al método. Se dice que ambas variables tienen un ámbito local para el método. Esto significa que no pueden utilizarse fuera del método.

5.1. LIMITING SCOPE

Las variables locales nunca pueden tener un alcance mayor que el método en el que están definidas. Sin embargo, pueden tener un ámbito más pequeño. Considere este ejemplo:

```
3: public void eatIfHungry(boolean hungry) {
4:  if (hungry) {
5:   int bitesOfCheese = 1;
6:  }  // bitesOfCheese goes out of scope here
7:   System.out.println(bitesOfCheese);// DOES NOT COMPILE
8: }
```

La variable `hunger` tiene un alcance de todo el método. `bitesOfCheese` tiene un alcance más pequeño. Sólo está disponible para su uso en la sentencia `if` porque se declara dentro de ella. Cuando vea un conjunto de llaves (`{}`) en el código, significa que ha ingresado un nuevo bloque de código. Cada bloque de código tiene su propio ámbito.

5.2. NESTING SCOPE

Recuerde que los bloques pueden contener otros bloques. Estos bloques contenidos más pequeños pueden hacer referencia a variables definidas en los bloques de gran alcance, pero no viceversa. Por ejemplo:

```
16: public void eatIfHungry(boolean hungry) {
17:   if (hungry) {
18:     int bitesOfCheese = 1;
19:     {
20:        boolean teenyBit = true;
21:        System.out.println(bitesOfCheese);
22:     }
23:   }
24:   System.out.println(teenyBit);  // DOES NOT COMPILE
25: }
```

5.3. TRACING SCOPE

No se preocupe si aún no está familiarizado con las declaraciones `if` o los bucles `while`. No importa lo que haga el código, ya que estamos hablando de alcance. Vea si puede averiguar en qué línea entra y sale del alcance cada una de las cinco variables locales:

```
11: public void eatMore(boolean hungry, int amountOfFood) {
12:    int roomInBelly = 5;
13:    if (hungry) {
14:       var timeToEat = true;
15:       while (amountOfFood > 0) {
16:          int amountEaten = 2;
17:          roomInBelly = roomInBelly - amountEaten;
18:          amountOfFood = amountOfFood - amountEaten;
19:       }
20:    }
21:    System.out.println(amountOfFood);
22: }
```

El primer paso para determinar el alcance es identificar los bloques de código. En este caso, hay tres bloques. Puede darse cuenta de esto porque hay tres juegos de llaves. Gráficamente tenemos:

```
11: public void eatMore(boolean hungry, int amountOfFood) {
12:    int roomInBelly = 5;
13:    if (hungry) {
14:       var timeToEat = true;
15:       while (amountOfFood > 0) {
16:          int amountEaten = 2;
17:          roomInBelly = roomInBelly - amountEaten;
18:          amountOfFood = amountOfFood - amountEaten;
19:       }
20:    }
21:    System.out.println(amountOfFood);
22: }
```

Entonces sobre los bloques tenemos:

1. Bloque `while`: 15 – 19
2. Bloque `If`: 13 -20
3. Bloque de método: 11 – 22

El alcance de las variables:

1. `hungry` y `amountOfFood`: 11 – 22 (todo el método)
2. `roomInBelly`: 12 -22
3. `timeToEat`: 14 -20
4. `amountEaten`: 16-19

5.4. APPLYING SCOPE TO CLASSES

Todo eso fue para variables locales. Por suerte, la regla para variables de instancia es más fácil: están disponibles tan pronto como se definen y duran toda la vida del propio objeto. La regla para las variables de clase (estáticas) es aún más fácil: entran en el ámbito cuando se declaran como los otros tipos de variables. Sin embargo, permanecen en el alcance para la vida entera del programa. Por ejemplo:

```
1:  public class Mouse {
2:    static int MAX_LENGTH = 5;
3:    int length;
4:    public void grow(int inches) {
5:      if (length < MAX_LENGTH) {
6:        int newSize = length + inches;
7:        length = newSize;
8:      }
9:     }
10: }
```

En esta clase, tenemos una variable de clase (`MAX_LENGTH`), una variable de instancia (`length`) y dos variables locales (`inches` y `newSize`).

5.5. REVIEWING SCOPE

Revisemos las reglas sobre el alcance:

1. Variables locales: En el ámbito de aplicación desde la declaración hasta el final del bloque.
2. Variables de instancia: En el ámbito desde la declaración hasta la recogida de objetos.
3. Variables de la clase: En el ámbito desde la declaración hasta el final del programa

6. Destroying Objects

Java proporciona un recolector de basura para buscar automáticamente objetos que ya no son necesarios. Todos los objetos Java se almacenan en el montón (*heap*) de su memoria de programa. El `heap`, que también se conoce como el almacén libre, representa un gran grupo de memoria no utilizada asignada a su aplicación Java. El *heap* puede ser bastante grande, dependiendo de su entorno, pero siempre hay un límite a su tamaño.

6.1. UNDERSTANDING GARBAGE COLLECTION

Recopilación de basura (*Garbage collection*) se refiere al proceso de liberación automática de memoria en el *heap* eliminando los objetos que ya no son accesibles en su programa. Hay muchos algoritmos diferentes para la recolección de basura, pero no es necesario conocer ninguno de ellos para el examen. Lo que debe saber es que `System.gc()` no está garantizado para ejecutarse, y debería ser capaz de reconocer cuándo los objetos son elegibles para la recolección de basura.

6.1.1. Eligible for Garbage Collection

¿Significa esto que un objeto que es elegible para la recolección de basura será recolectado inmediatamente? Definitivamente no. Cuando el objeto realmente se descarta no está bajo su control, pero para el examen, necesitará saber en cualquier momento qué objetos son elegibles para la recolección de basura.

Como programador, lo más importante que puede hacer para limitar los problemas de falta de memoria es asegurarse de que los objetos sean elegibles para la recolección de basura una vez que ya no sean necesarios. Es responsabilidad de la JVM realizar la recolección de basura.

6.1.2. Calling System.gc()

Java incluye un método integrado para ayudar a admitir la recolección de basura que se puede llamar en cualquier momento.

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

¿Qué garantiza el comando `System.gc()`? En realidad, nada. Simplemente sugiere que la JVM inicie la recolección de basura. La JVM es libre de ignorar la solicitud.

¿Cuándo se garantiza que `System.gc()` será llamado por la JVM? Nunca. De hecho, poco antes de que un programa se quede sin memoria y arroje un `OutOfMemoryError`, la JVM intentará realizar la recolección de basura, pero no se garantiza que tenga éxito.

6.2. TRACING ELIGIBILITY

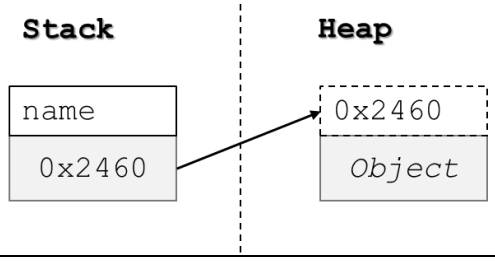
Un objeto permanecerá en el *heap* hasta que no pueda ser alcanzado. Un objeto ya no es accesible cuando se produce una de dos situaciones:

1. El objeto ya no tiene referencias que apuntan a él.
2. Todas las referencias al objeto han salido del alcance.

OBJECTS VS. REFERENCES

No confunda una referencia con el objeto al que se refiere; son dos entidades diferentes. La referencia es una variable que tiene un nombre y que se puede usar para acceder al contenido de un objeto. Una referencia puede ser asignada a otra referencia, pasada a un método o devuelta de un método. Todas las referencias son del mismo tamaño, sin importar su tipo.

Un objeto se sienta en el montón y no tiene un nombre. Por lo tanto, no tiene ninguna manera de acceder a un objeto excepto a través de una referencia.

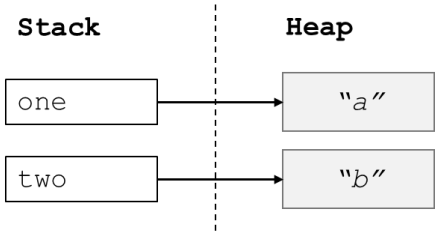


Darse cuenta de la diferencia entre una referencia y un objeto contribuye en gran medida a comprender la recolección de basura, el operador new y muchas otras facetas del lenguaje Java. Por ejemplo:

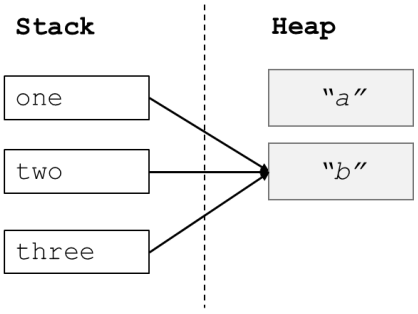
```
1: public class Scope {
2:  public static void main(String[] args) {
3:    String one, two;
4:    one = new String("a");
5:    two = new String("b");
6:    one = two;
7:    String three = one;
8:    one = null;
9:  } }
```

En el examen, le recomendamos que dibuje lo que está pasando. Por ejemplo:

La gráfica después de la línea 5:



La gráfica después de la línea 7:



Ahora, estábamos tratando de averiguar cuándo los objetos fueron elegibles para la recolección de basura. En la línea 6, nos deshicimos de la única flecha apuntando a "a", haciendo que el objeto sea elegible para la recolección de basura. "b" tiene flechas que apuntan a ella hasta que sale del alcance. Esto significa que "b" no sale del alcance hasta el final del método en la línea 9.

FINALIZE()
Java permite a los objetos implementar un método llamado `finalize()` que podría ser llamado. Este método se llama si el recolector de basura intenta recopilar el objeto. Si el recolector de basura no se ejecuta, el método no se llama. Si el recolector de elementos no consigue recopilar el objeto e intenta volver a ejecutarlo más tarde, el método no se llama por segunda vez.

Este tema ya no está en el examen. De hecho, está obsoleto en `Object` a partir de Java 9, y la documentación oficial indica: "El mecanismo de finalización es intrínsecamente problemático". Mencionamos el método `finalize()` en caso de que Oracle tome prestado de una pregunta del examen anterior. Solo recuerde que `finalize()` puede ejecutarse cero o una vez. No se puede ejecutar dos veces.

7. Review Questions

7.1.

Which of the following are valid Java identifiers? (Choose all that apply.)

```
A. _
B. _helloWorld$
C. true
D. java.lang
E. Public
F. 1980_s
G. _Q2_
```

7.2.

Which of the following statements about the code snippet are true? (Choose all that apply.)

```
4: short numPets = 5L;
5: int numGrains = 2.0;
6: String name = "Scruffy";
7: int d = numPets.length();
8: int e = numGrains.length;
9: int f = name.length();
```

A. Line 4 generates a compiler error.
B. Line 5 generates a compiler error.
C. Line 6 generates a compiler error.
D. Line 7 generates a compiler error.
E. Line 8 generates a compiler error.
F. Line 9 generates a compiler error.

7.3.

Which of the following are correct? (Choose all that apply.)

A. A local variable of type `boolean` defaults to `null`.
B. A local variable of type `float` defaults to `0.0f`.
C. A local variable of type `double` defaults to `0`.
D. A local variable of type `Object` defaults to `null`.

E. A local variable of type `boolean` defaults to `false`.
F. A local variable of type `float` defaults to `0.0`.
G. None of the above

7.4.

Suppose we have a class named Rabbit. Which of the following statements are true? (Choose all that apply.)

```
01: public class Rabbit {
02:    public static void main(String[] args) {
03:       Rabbit one = new Rabbit();
04:       Rabbit two = new Rabbit();
05:       Rabbit three = one;
06:       one = null;
07:       Rabbit four = one;
08:       three = null;
09:       two = null;
10:      two = new Rabbit();
11:      System.gc();
12: } }
```

A. The Rabbit object created on line 3 is first eligible for garbage collection immediately following line 6.
B. The Rabbit object created on line 3 is first eligible for garbage collection immediately following line 8.
C. The Rabbit object created on line 3 is first eligible for garbage collection immediately following line 12.
D. The Rabbit object created on line 4 is first eligible for garbage collection immediately following line 9.
E. The Rabbit object created on line 4 is first eligible for garbage collection immediately following line 11.
F. The Rabbit object created on line 4 is first eligible for garbage collection immediately following line 12.
G. The Rabbit object created on line 10 is first eligible for garbage collection immediately following line 11.
H. The Rabbit object created on line 10 is first eligible for garbage collection immediately following line 12.

7.5.

Which statements about the following code snippet are correct? (Choose all that apply.)

```
03: var squirrel = new Object();
04: int capybara = 2, mouse, beaver = -1;
05: char chipmunk = -1;
06: squirrel = "";
07: beaver = capybara;
08: System.out.println(capybara);
09: System.out.println(mouse);
10: System.out.println(beaver);
11: System.out.println(chipmunk);
```

A. The code prints `2`.
B. The code prints `-1`.
C. The code prints the empty String.
D. The code prints: `null`.
E. Line 4 contains a compiler error.
F. Line 5 contains a compiler error.
G. Line 9 contains a compiler error.
H. Line 10 contains a compiler error.