Java Building Blocks

1. Understanding the Java Class Structure

En los programas Java, clases son los bloques básicos de construcción. Al definir una clase, describe todas las partes y características de uno de esos bloques de construcción. Para usar la mayoría de las clases, hay que crear objetos. Un objeto es una instancia en tiempo de ejecución de una clase en memoria. Todos los diversos objetos de todas las clases diferentes representan el estado del programa.

1.1. Fields and Methods

Las clases Java tienen dos elementos primarios: métodos y atributos, más generalmente conocidos como variables. Juntos se llaman los miembros de la clase. Las variables mantienen el estado del programa, y los métodos operan sobre ese estado. Si un cambio es importante mantenerlo, una variable almacena dicho cambio.

Otros bloques de construcción incluyen interfaces y tipos enumerados.

Java llama a una palabra con un significado especial: `keyword`. La palabra clave `class` indica que está definiendo una clase. Animal da el nombre de la clase.

Por ejemplo:

```
1: public class Animal {
2:  String name;
3:  public String getName() {
4:    return name;
5:  }
6:  public void setName(String newName) {
7:    name = newName;
8:  }
9: }
```

Del ejemplo, un método es una operación que se puede llamar. Este tiene un tipo de retorno especial llamado `void`. Este método requiere información y esta información se llama parámetro.

La declaración completa de un método se llama una firma de método (*method signature*).

1.2. Comments

Un comentario de una sola línea comienza con dos barras. A continuación, viene el comentario de varias líneas (también conocido como comentario de varias líneas):

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

Finalmente, tenemos un comentario de `Javadoc`:

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

```
*/
```

Este comentario es similar a un comentario de varias líneas, excepto que comienza con `/ **`. Esta sintaxis especial le dice a la herramienta `Javadoc` que preste atención al comentario. Usted no verá un comentario de `Javadoc` en el examen. Cuidado con lo siguiente:

```
/* */ */
```

Hay `*/` un extra. Eso no es una sintaxis válida.

1.3. Classes vs Files

La mayor parte del tiempo, cada clase Java se define en su propio archivo `*.java`. Es generalmente público, Java no requiere que la clase sea pública.

Incluso puedes poner dos clases en el mismo archivo. Al hacerlo, se permite que una de las clases del archivo sea pública. Por ejemplo:

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

Si tiene una clase pública, debe coincidir con el nombre del archivo. la clase pública *Animal2* no compilaría en un archivo llamado `Animal.java`.

2. Writing a main() Method

Un programa Java comienza la ejecución con su método `main()`. Un método `main()` es la puerta de enlace entre el inicio de un proceso Java, que es gestionado por la máquina virtual Java (JVM), y el comienzo del código del programador. La JVM invita al sistema subyacente a asignar memoria y tiempo de CPU, archivos de acceso, etc.

Dado el siguiente programa:

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

Para compilar y ejecutar este código, escríbalo en un archivo llamado `Zoo.java` y ejecute lo siguiente:

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

Para compilar código Java, el archivo debe tener la extensión `.java`. El nombre del archivo debe coincidir con el nombre de la clase. El resultado es un archivo `bytecode` con el mismo nombre, pero con una extensión de nombre de archivo `.class`. El `bytecode` consta de instrucciones que la JVM sabe ejecutar.

Para mantener las cosas simples por ahora, seguiremos un subconjunto de las reglas:

1. Cada archivo puede contener sólo una clase.

2. El nombre de archivo debe coincidir con el nombre de la clase, *case sensitive*, y tener una extensión `.java`.

Primero revisemos las palabras en la parte `main()`:

- La palabra clave `public` es lo que se denomina modificador de acceso (*access modifier*). Este declara el nivel de exposición de este método a potenciales llamantes en el programa.
- La palabra clave `static` enlaza un método con su clase para que pueda ser llamado simplemente por el nombre de la clase, como en, por ejemplo, `Zoo.main()`.
- Si un método `main()` no está presente en la clase que nombramos con el ejecutable `.java`, el proceso lanzará un error y finalizará. Incluso si un método `main()` está presente, Java lanzará una excepción si no es estática (`static`).
- Un método `main()` no estático también podría ser invisible desde el punto de vista de la JVM.
- La palabra clave `void` representa el tipo de retorno. Un método que no devuelve ningún dato devuelve el control a quien lo llama en silencio.
- Finalmente llegamos a la lista de parámetros del método `main()`, representada como una matriz de objetos `java.lang.String`. En la práctica, puede escribir `String[] args`, `String args[]` o `String ... args`; el compilador acepta cualquiera de estos. El nombre de la variable `args` indica que esta lista contiene valores (argumentos) que se leyeron cuando se inició la JVM. Sin embargo, puedes usar cualquier nombre que quieras. Los caracteres [] son corchetes y representan una matriz.

Por ejemplo:

```
public class Zoo {
  public static void main(String[] args) {
    System.out.println(args[0]);
    System.out.println(args[1]);
} }
```

Compilamos:

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

La salida sería:

```
Bronx
Zoo
```

Los espacios se utilizan para separar los argumentos. Si desea espacios dentro de un argumento, debe usar comillas como en este ejemplo:

```
$ javac Zoo.java
$ java Zoo "San Diego" Zoo
```

Obtenemos:

```
San Diego
Zoo
```

Todos los argumentos de línea de comandos se tratan como objetos `String`, aunque representen otro tipo de datos. Por ejemplo, compilamos y ejecutamos:

```
$ javac Zoo.java
$ java Zoo Zoo 2
```

Obtenemos:

```
Zoo
2
```

Para revisar, necesita tener un JDK para compilar porque incluye un compilador. Usted no necesita tener un JDK para ejecutar el código, un JRE es suficiente. Los archivos de clase Java se ejecutan en la JVM y por lo tanto se ejecutan en cualquier máquina con Java.

3. Understanding Package Declarations and Imports

Java viene con miles de clases integradas. Java pone clases en paquetes. Estos son agrupamientos lógicos para las clases. Por ejemplo:

```
public class ImportExample {
  public static void main(String[] args) {
    Random r = new Random();   // DOES NOT COMPILE
    System.out.println(r.nextInt(10));
  }
}
```

Lo anterior causa de este error es omitir una instrucción de importación necesaria. Las instrucciones de importación indican a Java qué paquetes buscar en las clases. Lo modificamos del siguiente modo:

```
import java.util.Random;  // import tells us where to find Random
public class ImportExample {
  public static void main(String[] args) {
    Random r = new Random();
    System.out.println(r.nextInt(10));  // print a number between 0 and 9
   }
}
```

Los nombres de los paquetes son jerárquicos como la dirección de correo. Si comienza con `java` o `javax`, esto significa que vino con el JDK. Si comienza con otra cosa, es probable que muestre de dónde proviene el uso del nombre del sitio web a la inversa. Por ejemplo, `com.amazon.java8book` nos dice que el código vino de amazon.com. Java llama a los paquetes más detallados paquetes hijos (*child package*). El paquete `com.amazon.java8book` es un paquete hijo de `com.amazon`.

La regla para nombres de paquetes es que son en su mayoría letras o números separados por puntos.

3.1. Wildcards

Las clases en el mismo paquete se importan a menudo juntos. Puede utilizar un acceso directo para importar todas las clases de un paquete:

```
import java.util.*;    // imports java.util.Random among other things
public class ImportExample {
  public static void main(String[] args) {
    Random r = new Random();
    System.out.println(r.nextInt(10));
  }
}
```

El `*` es un comodín que coincide con todas las clases del paquete. Cada clase del paquete `java.util` está disponible para este programa cuando Java lo compila. No importa paquetes hijos, campos o métodos; importa sólo clases.

Podrías pensar que incluir tantas clases ralentiza el programa, pero no lo hace. La computadora calcula lo que realmente se necesita.

3.2. Redundant Imports

Hay un paquete especial en el mundo Java llamado `java.lang`. Este paquete es especial en que se importa automáticamente. Todavía puede escribir este paquete en una instrucción de importación, pero no es necesario. ¿Cuántas importaciones cree que son redundantes?:

```
1: import java.lang.System;
2: import java.lang.*;
3: import java.util.Random;
4: import java.util.*;
5: public class ImportExample {
6:  public static void main(String[] args) {
7:    Random r = new Random();
8:    System.out.println(r.nextInt(10));
9:  }
10: }
```

La respuesta es que tres de las importaciones son redundantes: Las líneas 1, 2 y las líneas 3/4 son redundantes.

Para el ejemplo siguiente, Archivos y rutas están en el paquete `java.nio.file`. ¿Qué importaciones cree que funcionaría para obtener este código para compilarlo ?:

```
public class InputImports {
  public void read(Files files) {
    Paths.get("name");
  }
}
```

De forma corta:

```
import java.nio.file.*;
```

Otra respuesta sería:

```
import java.nio.file.Files;
import java.nio.file.Paths;
```

Considere los siguientes `import` que no funcionan:

```
import java.nio.*;   // NO GOOD - a wildcard only matches
                     //class names, not "file.*Files"
import java.nio.*.*;   // NO GOOD - you can only have one wildcard
                       //and it must be at the end
import java.nio.files.Paths.*; // NO GOOD - you cannot import methods
                               //only class names
```

3.3. Naming Conflicts

Una de las razones para usar paquetes es que los nombres de clase no tienen que ser únicos en todo Java. Un ejemplo común de esto es la clase Date. Java proporciona implementaciones de `java.util.Date` y `java.sql.Date`:

```
public class Conflicts {
  Date date;
  // some more code
}
```

Puede escribir: `import java.util.*;` o `import java.util.Date ;`:

```
import java.util.*;
import java.sql.*; // DOES NOT COMPILE
```

El compilador nos muestra le siguiente mensaje:

```
The type Date is ambiguous
```

En nuestro ejemplo, la solución es fácil: elimine la importación de `java.sql.Date` la cual no necesitamos:

```
import java.util.Date;
import java.sql.*;
```

Ah, ahora funciona. Si importa explícitamente un nombre de clase, tendrá prioridad sobre cualquier comodín presente. Un ejemplo más:

```
import java.util.Date;
import java.sql.Date;
```

El compilador nos muestra:

```
The import java.sql.Date collides with another import statement
```

¿Y si realmente necesitamos ambas clases? Puede escoger uno para utilizar en la importación y utilizar el nombre de clase completo cualificado de otros para especificar que es especial:

```
import java.util.Date;
public class Conflicts {
  Date date;
  java.sql.Date sqlDate;
}
```

O también:

```
public class Conflicts {
  java.util.Date date;
  java.sql.Date sqlDate;
}
```

3.4. Creating a New Package

Todo el código que hemos escrito en este capítulo ha estado en el paquete *default*. Este es un paquete especial sin nombre que debe usar sólo para el código desechable. Puede decir que el código está en el paquete *default*, porque no hay nombre de paquete.

Supongamos que tenemos dos clases:

```
C:\temp\packagea\ClassA.java
package packagea;
public class ClassA {
}
```

```
C:\temp\packageb\ClassB.java
package packageb;
import packagea.ClassA;
public class ClassB {
  public static void main(String[] args) {
    ClassA a;
    System.out.println("Got it");
  }
}
```

Cuando ejecuta un programa Java, Java sabe dónde buscar esos nombres de paquetes. En este caso, ejecutar desde `C:\temp` funciona porque tanto `packagea` como `packageb` están debajo de él.

Por ejemplo. Creamos dos archivos:
`C:\temp\packagea\ClassA.java`
`C:\temp\packageb\ClassB.java`

Luego ejecutamos:
```
cd C:\temp
```

Y compilamos:
```
javac packagea/ClassA.java packageb/ClassB.java
```

Lo anterior funciona bien. Para el case del `classpath` y JARs. Por ejemplo, ejecutamos:
```
java -cp ".;C:\temp\someOtherLocation;c:\temp\myJar.jar" myPackage.MyClass
```

El punto indica que desea incluir el directorio actual en la ruta de clase. El resto del comando dice que debe buscar archivos de clase (o paquetes) sueltos en `someOtherLocation` y dentro de `miJar.jar`.

Puede utilizar un comodín (*) para que coincida con todos los JAR en un directorio:
```
java -cp "C:\temp\directoryWithJars\*" myPackage.MyClass
```

4. Creating Objects

Recuerde que un objeto es una instancia de una clase.

4.1. Constructors

Por ejemplo:
```
Random r = new Random();
```

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

`Random()` 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: el nombre del constructor coincide con el nombre de la clase y 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".

4.2. Reading and Writing Object 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: }
```

4.3. 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:

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

```
6: { System.out.println("Snowy"); }
```

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.

4.4. 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.

5. Distinguishing Between Object References and Primitives

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

5.1. 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`	`123`
`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 un flotador.
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`.

No es necesario memorizar los valores máximos posibles, estos tienen constantes pre definidas. Por ejemplo:
`System.out.println(Integer.MAX_VALUE);`

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
System.out.println(0b11);     // 3
System.out.println(017);      // 15
System.out.println(0x1F);     // 31
```

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 subrayados 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
```

5.2. 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 nueva palabra clave.

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`.

5.3. Key differences

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 de tipo minúscula. Todas las clases que vienen con Java comienzan con mayúsculas.

6. Declaring and Initializin 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 con darle 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;
```

6.1. Declaring Multiple Variables

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

```
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?

```
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.

"Legal", "válido" y "compila" son sinónimos en el mundo de los exámenes Java. Por ejemplo:

```
boolean b1, b2;
String s1 = "1", s2;
double d1, double d2;
int i1; int i2;
```

```
int i3; i4;
```

La primera afirmación es legal. Declara dos variables sin inicializarlas. La segunda afirmación también es legal. Declara dos variables e inicializa sólo una de ellas. 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.

6.2. 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		

Los siguientes ejemplos son legales:

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

Los siguientes no son legales:

```
3DPointClass  // identifiers cannot begin with a number
hollywood@vine // @ is not a letter, digit, $ or _
*$coffee // * is not a letter, digit, $ or _
public   // public is a reserved word
```

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.

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.

7. Understanding Default Initialization of 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.

7.1. 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 compilador

7.2. 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' (NUL)`
`All object references (everything else)`	`null`

8. Understanding 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.

Sin embargo, pueden tener un ámbito más pequeño. Considere este ejemplo:

```
3: public void eatIfHungry(boolean hungry) {
4:  if (hungry) {
5:   int bitesOfCheese = 1;32
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.

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: }
```

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`).

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

9. Ordering Elements in a Class

Echemos un vistazo al orden correcto para escribirlos en un archivo. Los comentarios pueden ir a cualquier parte del código. Más allá de eso, es necesario memorizar las reglas de la tabla:

Elemento	Ejemplo	¿Requerido?	¿Dónde se ubica?
Package declaration	package abc;	No	Primera línea del archivo
Import statements	import java.util.*;	No	Inmediatamente después del paquete
Class declaration	public class C	Si	Inmediatamente después del import
Field declarations	int value;	No	En cualquier lugar dentro de la clase
Method declarations	void method()	No	En cualquier lugar dentro de la clase

Por ejemplo:

```
package structure;          // package must be first non-comment
import java.util.*;         // import must come after package
public class Meerkat {      // then comes the class
  double weight;            // fields and methods can go in either order
  public double getWeight() {
    return weight; }
  double height;            //  another field – they don't need to be together
}
```

Otro ejemplo:

```
/* header */
package structure;
// class Meerkat
public class Meerkat { }
```

En el siguiente no es correcto:

```
import java.util.*;
package structure;   // DOES NOT COMPILE
String name;  // DOES NOT COMPILE
public class Meerkat { }
```

Para el examen OCA se pueden definir varias clases en el mismo archivo, pero sólo se permite que una de ellas sea pública. La clase pública coincide con el nombre del archivo. Por ejemplo, estas dos clases deben estar en un archivo denominado `Meerkat.java`:

```
1: public class Meerkat { }
2: class Paw { }
```

10. 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.

10.1. Garbage Collection

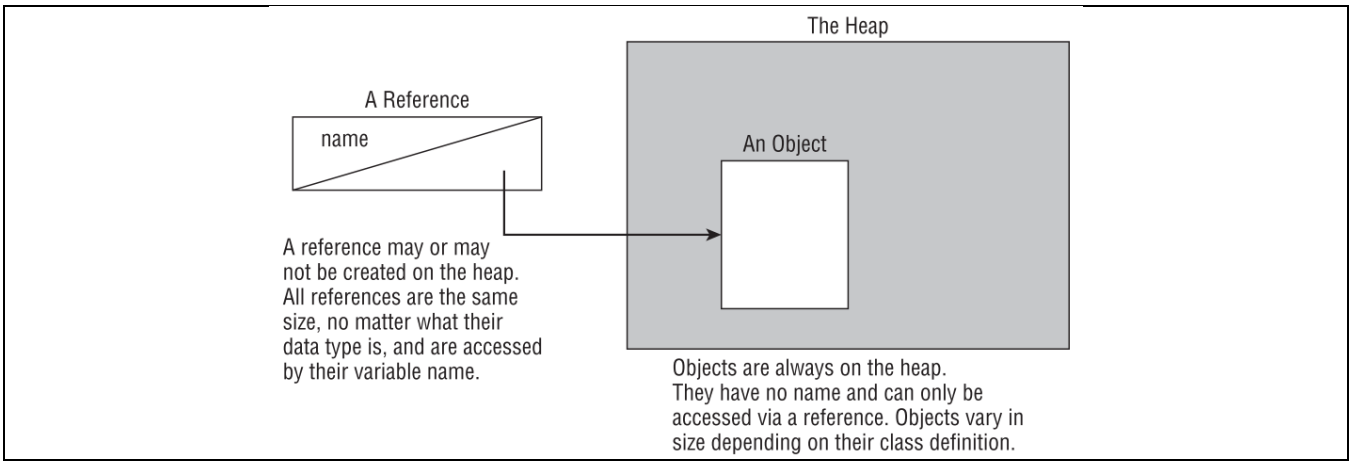
Recopilación de basura se refiere al proceso de liberación automática de memoria en el *heap* eliminando los objetos que ya no son accesibles en su programa. Hay muchos algoritmos diferentes para la recolección de basura, pero no es necesario conocer ninguno de ellos para el examen. Lo que debe saber 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.

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.

Objetos vs. Referencias
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.

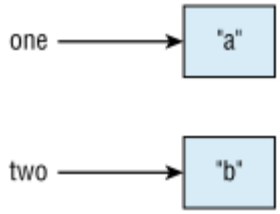


Por ejemplo:

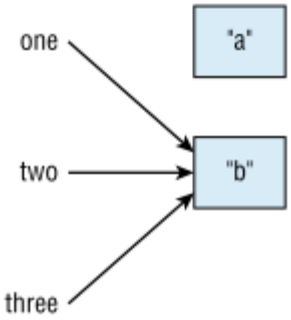
```
1: public class Scope {
2:  public static void main(String[] args) {
3:    String one, two;
4:    one = new String("a");
5:    two = new String("b");
6:    one = two;
7:    String three = one;
8:    one = null;
9:  } }
```

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.

10.2. 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.

Para el examen sólo tenga en cuenta que no puede ser llamado y que definitivamente no será llamado dos veces. Por ejemplo, el siguiente código no genera ninguna salida:

```
public class Finalizer {
  protected void finalize()  {
    System.out.println("Calling finalize");
  }
  public static void main(String[] args) {
    Finalizer f = new Finalizer();
  } }
```

La razón es que el programa sale antes de que haya alguna necesidad de ejecutar el recolector de basura. Un ejemplo adicional:

```
public class Finalizer {
  private static List objects = new ArrayList();
  protected void finalize()  {
    objects.add(this);  // Don't do this
  } }
```

Aquí es que al final del método, el objeto ya no es elegible para la recolección de basura porque una variable estática se refiere a ella y las variables estáticas permanecen en el ámbito hasta que el programa termina. Java es lo suficientemente inteligente como para darse cuenta de esto y aborta el intento de tirar el objeto.

11. Benefits of Java

Java tiene algunos beneficios clave que necesitará k ahora para el examen:

1. **Orientado a objetos**: Java es un lenguaje orientado a objetos, lo que significa que todo el código se define en las clases y la mayoría de esas clases se pueden instanciar en objetos. Otro enfoque común es la programación funcional. Java permite la programación funcional dentro de una clase, pero orientada a objetos sigue siendo la principal organización de código.
2. **Encapsulación**: Java soporta modificadores de acceso para proteger los datos de acceso no intencionado y modificación.
3. **Independiente de la plataforma**: Java es un lenguaje interpretado porque se compila a bytecode. Un beneficio clave es que el código Java se compila una vez en vez de necesitar ser recompilado para diferentes sistemas operativos. Esto se conoce como "escribir una vez, correr en todas partes". En el examen OCP, aprenderá que es posible escribir código que no se ejecuta en todas partes.
4. **Robusto**: Una de las principales ventajas de Java sobre C ++ es que evita fugas de memoria. Java gestiona la memoria por sí solo y realiza la recolección de basura automáticamente. La mala gestión de la memoria en C ++ es una gran fuente de errores en los programas.
5. **Simple**: Java estaba destinado a ser más sencillo que C ++. Además de eliminar punteros, se deshizo de la sobrecarga del operador.
6. **Seguro**: el código Java se ejecuta dentro de la JVM. Esto crea una caja de seguridad que dificulta el código Java para hacer cosas malas al equipo en el que se está ejecutando.

12. Review Questions

12.1.

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

```
4: short numPets = 5;
5: int numGrains = 5.6;
6: String name = "Scruffy";
7: numPets.length();
8: numGrains.length();
9: 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.

12.2.

Given the following classes, what is the maximum number of imports that can be removed and have the code still compile?

```
package aquarium;
public class Water { }

package aquarium;
import java.lang.*;
import java.lang.System;
import aquarium.Water;
import aquarium.*;
public class Tank {
  public void print(Water water) {
   System.out.println(water); } }
```

A. 0
B. 1
C. 2
D. 3
E. 4
F. Does not compile.

12.3.

Which of the following legally fill in the blank so you can run the `main()` method from the command line? (Choose all that apply)

```
public static void main(______)
```

A. `String[] _names`
B. `String[] 123`
C. `String abc[]`
D. `String _Names[]`
E. `String... $n`

F. `String names`
G. None of the above.

12.4.

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

A. A local variable of type `boolean` defaults to `null`.
B. A local variable of type `float` defaults to `0`.
C. A local variable of type `Object` defaults to `null`.
D. A local variable of type `boolean` defaults to `false`.
E. A local variable of type `boolean` defaults to `true`.
F. A local variable of type `float` defaults to `0.0`.
G. None of the above.

12.5.

Which of the following lines of code compile? (Choose all that apply)

A. `int i1 = 1_234;`
B. `double d1 = 1_234_.0;`
C. `double d2 = 1_234._0;`
D. `double d3 = 1_234.0_;`
E. `double d4 = 1_234.0;`
F. None of the above.

12.6.

Which represent the order in which the following statements can be assembled into a program that will compile successfully? (Choose all that apply)

```
A: class Rabbit {}
B: import java.util.*;
C: package animals;
```

A. A, B, C
B. B, C, A
C. C, B, A
D. B, A
E. C, A
F. A, C
G. A, B

12.7.

What does the following code output?

```
1: public class Salmon {
2:  int count;
3:  public void Salmon() {
4:    count = 4;
5:  }
6: public static void main(String[] args) {
```

```
7:  Salmon s = new Salmon();
8:  System.out.println(s.count);
9: } }
```

A. 0
B. 4
C. Compilation fails on line 3.
D. Compilation fails on line 4.
E. Compilation fails on line 7.
F. Compilation fails on line 8.