2: Todo es un objeto

Aunque se basa en C + +, Java es más que un lenguaje orientado a objetos.

C + + y Java son idiomas híbridos, pero en Java los diseñadores consideraron que la hibridación no era tan importante como lo era en C + +. Un idioma híbrido consiente estilos múltiples de programación; La razón por la que C + + es híbrido es por la compatibilidad hacia atrás con el lenguaje C. Porque C + + es un actualización del lenguaje C, incluye muchas de características indeseables de ese idioma, lo cual puede hacer algunos aspectos de C + + excesivamente complicados.

El idioma Java da por supuesto que usted quiere sólo programar con programación orientada a objetos. Esto quiere decir que antes de que usted pueda comenzar debe desviar su disposición mental a un mundo orientado a objetos (a menos que esté ya allí). El beneficio de este esfuerzo inicial es la habilidad para programar en un idioma que es más simple para aprender y usar que muchos otros idiomas OOP. En este capítulo veremos que los componentes básicos de un programa Java y aprenderemos que todo en Java es un objeto, dentro de un programa Java.

Usted manipula objetos mediante referencias

Cada lenguaje de programación tiene su propia manera de manipular datos. Algunas veces el programador debe estar constantemente atento a que tipo de manipulación está ocurriendo. ¿Está usted manipulando el objeto directamente, o está usted ocupándose de alguna clase de representación indirecta (un puntero en C o C + +) que debe ser tratada con una sintaxis especial?

Todo esto es simplificado en Java. Usted trata todo como un objeto, usando una sintaxis coherente. Aunque usted *trata* todo como un objeto, el identificador que usted manipula es de hecho una "referencia" a un **object [1]**. Usted podría imaginar esta escena como una televisión (el objeto) con su control remoto (la referencia). Mientras usted sostiene esta referencia, usted tiene una conexión con la televisión, pero cuándo alguien dice "cambia el canal" o "baja el volumen", lo que usted manipula es la referencia, lo cual a su vez modifica el objeto. Si usted quiere ir de arriba abajo por el cuarto y todavía controlar la televisión, usted coge el mando / referencia, pero no la televisión.

[1] Este puede ser un punto de conflicto. Hay quien dice "claramente, es un puntero, "pero esto presupone una implementación subyacente. También, las referencias en Java son mucho más semejantes a las referencias en C + + que a los punteros en su sintaxis. En la primera edición de este libro, preferí

inventar un término nuevo, “handle”, porque las referencias C + + y las referencias Java tienen algunas diferencias importantes. Deje fuera a C + + y no quise confundir a los programadores de C + + que creí que sería la audiencia más importante para Java. En la edición 2, me decidí que “la referencia “era el término más comúnmente usado, y que cualquiera que viniese de C + + entendería mejor la terminología de referencias, y podrían entrar de un salto con ambos pies. Sin embargo, hay personas que disienten aun con el término “referencia.” Leí en un libro donde ponía que “totalmente incorrecto que Java soporte paso por referencia” porque los identificadores del objeto Java (según ese autor) son r*ealmente* “las referencias del objeto.” Y (él sigue) *de hecho* todo es pasado por el valor. Así es que no se pasa por referencia, si no que, “se pasa la referencia a un objeto por valor”.
Uno podría argumentar a favor de la precisión de tales explicaciones complejas, pero pienso que mi acercamiento simplifica la comprensión del concepto sin lastimar a ninguna (pues bien, los puristas del lenguaje pueden afirmar que le miento, pero diré que planteo una abstracción apropiada.)

También, el control remoto puede existir por el mismo, sin televisión. Es decir, una referencia no significa necesariamente que haya un objeto asociado a él. Así que si usted quiere tener una palabra o una sentencia, usted crea una referencia **String:**

```
String s;
```

Pero aquí usted ha creado *sólo* la referencia, no un objeto. Si usted envía un mensaje a **s** en este momento, usted obtendrá un error (durante la ejecución) porque **s** no tiene asociado ningún objeto (no hay televisión). Una práctica más segura, por tanto, es siempre inicializar una referencia cuando usted la crea:

```
String s = "asdf";
```

Sin embargo, esto usa un rasgo especial de Java: los Strings pueden ser inicializados con un texto entrecomillado. Normalmente, usted debe usar un tipo de inicialización para los objetos más general.

Usted debe crear todos los objetos

Cuando usted crea una referencia, usted quiere asociarla a un objeto nuevo.
Usted hace eso, en general, con la palabra clave **new. New** dice “créame un nuevo objeto de este tipo.” Así en el ejemplo citado anteriormente, usted puede decir:

```
String s = new String("asdf");
```

No sólo significa “Créame un nuevo **String**” pero también da información de *cómo* crear el **String** suministrando una cadena de caracteres inicial.

Por supuesto, **String** no es el único tipo que existe. Java viene con una colección de tipos prefabricados. Pero lo más importante es que usted puede crear sus propios tipos. De hecho, esa es la actividad fundamental de la programación en Java, y eso lo que usted aprenderá en el resto de este libro.

Donde el almacenamiento vive

Es útil visualizar algunos aspectos de cómo son las cosas ocurren mientras el programa se está ejecutando, en particular como es aprovechada la memoria. Hay seis lugares diferentes para almacenar datos:

- **Los registros.** Éste es el almacenamiento más rápido porque existe en un lugar distinto al resto de los datos almacenados: Dentro del procesador. Sin embargo el número de registros es bastante limitado, por eso los registros son usados por el compilador según sus necesidades. Usted no tiene control de estos ni en sus programas hay evidencias de que estos registros existan.

- **La pila.** Reside en el área de la memoria RAM (acceso aleatorio a memoria), pero tiene acceso directo desde el procesador mediante el *puntero de pila.* El puntero de pila se decrementa para crear El puntero de pila se decrementa para crear memoria nueva y se incrementa para liberar esa memoria. Ésta es una forma sumamente rápida y eficiente para ubicar datos, sólo inferior a los registros. El compilador de Java debe saber, mientras crea el programa, el tamaño exacto y la duración de vida de todos los datos que son almacenados en la pila, porque debe generar el código para mover de arriba abajo el puntero de pila. Esta restricción limita la flexibilidad de sus programas, así es que mientras haya datos en la pila —en particular, referencias a objetos— los objetos Java no pueden colocarse ellos mismos en la pila.

- **El montículo (Heap).** Éste es área de memoria multiusos (también en el área de RAM) donde todos los objetos Java viven. Lo interesante de esta área de memoria es que, al contrario que la pila, el compilador no necesita saber que cantidad de memoria es necesaria reservar o cuanto tiempo va a permanecer en esta área. Así, hay una gran flexibilidad en el almacenamiento utilizado esta área de memoria (heap). Cada vez que usted necesita crear un objeto, usted simplemente escribe el código para crearlo usando **new**, y es ubicado en el 'heap' cuando ese código es ejecutado. Por supuesto hay un precio que usted paga para esta flexibilidad: es más costoso en cuanto a tiempo el ubicar en este área que hacerlo en la pila (si *puede crear* objetos en la pila en Java, como lo puede hacer en C + +).

- **Almacenamiento estático.** “estático” es usado aquí en el sentido de “una posición fija” (aunque está también en RAM). La memoria estática contiene datos que están disponible durante todo tiempo entero que dure programa en ejecución. Usted puede usar la palabra clave **static** para

especificar que un elemento particular de un objeto es estático, porque en Java por defecto los objetos no se almacenan en memoria estática.

- **Almacenamiento constante.** Los constantes son a menudo creadas directamente en el código de programa, asegurando que nunca puedan cambiar. Algunas veces las constantes son declaradas de forma aislada a fin de que pueden ser opcionalmente ubicadas en memoria de sólo lectura (ROM), en sistemas empotrados.

- **El almacenamiento secundario (fuera de memoria RAM).** Son los datos que se mantienen fuera de un programa en ejecución que deben de ser almacenados fuera de la memoria RAM. Los dos ejemplos principales de esto son *los objetos corrientes (streamed objects),* donde los objetos son convertidos a una *corriente* o *stream* para ser enviados a otra máquina, y *objetos persistentes*, donde los objetos se guardan en disco, de forma que mantienen su estado una vez que el programa ha terminado. Estos objetos pueden ser de nuevo convertidos a objetos ordinarios en memoria. Java provee soporte para la *persistencia ligera* o *lightweight persistence*. Versiones futuras de Java darán soluciones más completas a la persistencia.

Un caso especial: los tipos primitivos

Un grupo de tipos, que usted usará muy a menudo en su programación, tiene un tratamiento especial. Usted puede pensar en estos como tipos “primitivos”. La razón del tratamiento especial es que crear un objeto con **new** — especialmente una variable pequeña, simple — es poco eficiente porque **new** ubica los objetos en el montículo. Para estos tipos Java recurre a la forma de hacerlo en C y C + +. Es decir, en lugar de crear la variable usando **new**, una variable “automática” es creada sin ser *una referencia.* La variable posee el valor, y es colocada en la pila siendo así mucho más eficiente.

Java determina el tamaño de cada tipo primitivo. Estos tamaños no cambian de una arquitectura de máquina a otra como ocurre en la mayoría de los lenguajes. Que este tamaño sea fijo es una razón por la que los programas de Java son portables entre distintas arquitecturas.

Tipo Primitivo	Tamaño	Mínimo	Máximo	Tipo Envoltorio
boolean	—	—	—	**Boolean**
char	16-bit	Unicode 0	Unicode 2^{16}-1	**Character**
byte	8-bit	-128	+127	**Byte**
short	16-bit	-2^{15}	$+2^{15}$—1	**Short**
int	32-bit	-2^{31}	$+2^{31}$—1	**Integer**
long	64-bit	-2^{63}	$+2^{63}$—1	**Long**
float	32-bit	IEEE754	IEEE754	**Float**

double	64-bit	IEEE754	IEEE754	**Double**
void	—	—	—	**Void**

Todos los tipos numéricos tienen signo, así es que no busque tipos sin signo.

El tamaño del tipo **del boolean** no está explícitamente especificado; está sólo definido para poder tomar los valores literales **true** o **false.**

Las clases "envoltura" (wrappers) de los tipos primitivos de datos permiten que usted ubique un objeto no primitivo en el montículo para representar ese tipo primitivo. Por ejemplo:

```
char c = 'x';
Character C = new Character(c);
```

O usted también podría usar:

```
Character C = new Character('x');
```

Las razones para hacer esto serán mostradas en un capítulo posterior.

Los números de alta precisión

Java incluye dos clases para realizar aritmética de alta precisión: **BigInteger** y **BigDecimal.** Aunque estos aproximadamente están dentro de la misma categoría como las clases "envoltura", ninguno tiene similitudes primitivas.

Ambas clases tienen métodos que tienen similitudes con las operaciones que usted realiza con tipos primitivos. Es decir, usted puede hacer las mismas cosas con un **BigInteger** o **BigDecimal** de las que pueden hacer con un **int** o un **float**, pero debe usar llamadas a métodos en vez de operadores.

BigInteger soporta enteros de precisión arbitraria. Esto quiere decir que usted puede representar valores numéricos de cualquier tamaño sin perder precisión durante las operaciones.

BigDecimal es para números de coma flotante de precisión arbitraria; Usted puede usar este tipo, por ejemplo, para cálculos monetarios precisos.

Consulte la documentación JDK para los detalles acerca de los constructores y los métodos que puede usar en estas dos clases.

Arrays en Java

Supuestamente todos los lenguajes de programación soportan arrays. Usar arrays en C y C + + es peligroso porque esos arrays son sólo bloques de memoria. Si un programa accede a un array fuera de su bloque de memoria o

usa la memoria antes de la inicialización (errores comunes de programación) los resultados serán imprevisibles.

Una de las metas principales de Java es la seguridad, para que los quebraderos de cabeza de los programadores en C y C + + no los tengan repetidos en Java. Se garantiza que un array en Java es inicializado y no puede ser accedido fuera de su rango.

El precio a pagar es el usar un poquito de memoria de más en cada array así como también verificar el índice en tiempo de ejecución, pero se asume que el aumento de seguridad y la productividad lo vale.

Cuando usted crea un array de objetos, usted realmente crea un array de referencias, y cada uno de esas referencias son automáticamente inicializadas a un valor especial con su propia palabra clave: **null.** Cuando Java ve **null**, reconoce que la referencia no es un puntero a un objeto.

Usted debe asignar un objeto a cada referencia antes de que usted lo use, y si usted trata de usar una referencia que todavía esta a **null,** el problema será reportado en tiempo de ejecución.

Así, los errores típicos con arrays son prevenidos en Java. Usted también puede crear a un array de tipos primitivos. De nuevo, el compilador garantiza la inicialización porque pone en el cero la memoria para ese array. Los arrays serán estudiados en detalle en capítulos posteriores.

Usted nunca necesita destruir un objeto

En la mayoría de lenguajes de programación, el concepto de la duración de una vida de una variable ocupa una porción significativa del esfuerzo programador. ¿Cuánto tiempo dura la variable? Si se supone que usted debe destruirla, ¿cuándo debería hacerlo? La confusión sobre la vida de las variables puede conducir para un montón de problemas, y esta sección enseña cómo Java simplifica el asunto haciendo todo el trabajo de limpieza por usted.

Ámbito

La mayoría de idiomas procedurales tienen el concepto de *alcance.* Esto determina la visibilidad y la duración de vida de los nombres definidos dentro de ese alcance. En C, C + +, y Java, el alcance es determinado por la colocación de corchetes **{ }.** Por ejemplo:

```
{
  int x = 12;
  // Only x available
  {
```

```
    int q = 96;
    // Both x & q available
  }
  // Only x available
  // q “out of scope”
}
```

Una variable definida dentro de un bloque está disponible sólo dentro de ese bloque. Cualquier texto después de '//' hasta el fin de la línea es un comentario.

La sangría simplifica leer código Java. Java es un lenguaje donde los espacios adicionales, etiquetas, y retornos de carro no afectan el programa resultante. Nótese que usted *no puede hacer* lo siguiente, aunque si es correcto en C y C + +:

```
{
  int x = 12;
  {
    int x = 96; // Illegal
  }
}
```

El compilador reportará un error diciendo que la variable **x** ya estaba definida. Así la habilidad de C y C + + para “silenciar” una variable de un bloque superior no es admitida porque los diseñadores de Java pensaron que esto da lugar a programas confusos.

El ámbito de objetos

Los objetos en Java no tienen las mismas duraciones de vida como los primitivos. Cuando usted crea un objeto Java usando **new**, sigue existiendo incluso fuera de su bloque. Así si usted usa:

```
{
  String s = new String("a string");
} // End of scope
```

La referencia **s** deja de existir al final del bloque. Sin embargo, el objeto **String** hacia el que **s** apuntaba, todavía ocupa memoria. En esta porción de código, no hay forma de acceder al objeto porque la única referencia para él no tiene alcance. En capítulos posteriores usted verá cómo la referencia al objeto puede ser reutilizada y duplicada durante el curso de un programa.

El resultado es que los objetos creados con **new** permanecen durante el tiempo que usted quiera, un montón de problemas de la programación en C + + simplemente desaparecen en Java. Los problemas más duros que ocurren en C + + es porque no se obtiene ninguna ayuda del idioma en lo referente a que los objetos estén disponibles cuando son necesarios. Y más importante, en C + + es que usted debe asegurarse de que destruye los objetos cuando ha terminado con ellos.

Eso trae a colación una pregunta interesante. ¿Si Java deja los objetos ocupando memoria, quién limpia esta memoria y detiene su programa? Éste es exactamente el tipo de problema que ocurriría en C + +.

Aquí es donde ocurre un poquito de 'magia'. Java tiene a un *colector de basura (garbage collector)*, el cual busca todos los objetos que se creó con **new** y no serán referenciadas de nuevo. Luego libera la memoria de esos objetos, así esta la memoria puede servir para nuevos objetos. Esto quiere decir que usted nunca necesita preocuparse por limpiar la memoria. Usted simplemente crea objetos, y cuando usted ya no los necesita se eliminarán por ellos mismos. Esto elimina algunos problemas para el programador: "falta de memoria" cuando un programador olvida liberar la memoria que ya no es usada.

Creando tipos nuevos de datos: clases

¿Si todo es un objeto, qué determina a una clase particular de objeto y su comportamiento? De otra forma, ¿qué establece el *tipo* de un objeto? Usted podría esperar que hubiera una palabra reservada "**type**" y eso ciertamente tendría sentido. Históricamente, sin embargo, la mayoría de lenguajes orientados a objetos han usado la **palabra reservada class** para querer decir "estoy a punto de decirle de que tipo de objeto soy" La palabra reservada **class** esta seguida por el nombre del tipo nuevo. Por ejemplo:

```
class ATypeName { /* Class body goes here */ }
```

Esto introduce un tipo nuevo, aunque el cuerpo de clase conste sólo de un comentario (las estrellas y barras serán explicadas más adelante en este capítulo), por tanto no se puede hacer mucho más con él. Sin embargo, usted puede crear un objeto de este tipo usando **new:**

```
ATypeName a = new ATypeName();
```

Pero usted no le puede decir que haga nada más (es decir, usted no le puede enviar ningún mensaje interesante) hasta que usted defina algunos métodos para esta clase.

Atributos y métodos

Cuando usted define una clase (y todo lo que usted hace en Java es definir clasifica, crear objetos de esas clases, y enviar mensajes a esos objetos), usted puede poner dos tipos de elementos en su clase: *Atributos* (algunas veces llamados datos miembro), y *métodos* (algunas veces llamados funciones miembro).

Un atributo es un objeto de cualquier tipo con el que usted puede comunicarse mediante su referencia. También puede ser uno de los tipos primitivos (el cuál no es una referencia). Si es una referencia a un objeto, usted debe inicializarla asociándola a un objeto real (usando **new**, como se vio anteriormente)

mediante un método especial llamado *constructor* (descrito completamente en el capítulo 4).

Si es un tipo primitivo usted lo puede inicializar directamente en el momento de la definición en la clase. (Como verá más tarde, las referencias también pueden ser inicializadas en el momento de la definición.)

Tipo Primitivo	Valor por Defecto
boolean	**false**
char	**'\u0000' (null)**
byte	**(byte)0**
short	**(short)0**
int	**0**
long	**0L**
float	**0.0f**
double	**0.0d**

Estos son los valores predeterminados que Java garantiza cuando la variable es usada *como un miembro de una clase.* Esto asegura que las variables miembro de tipos primitivos siempre serán inicializadas (algo que C + + no hace), reduciendo una fuente de problemas. Sin embargo, este valor inicial puede no ser correcto o el necesario para su programa. Es aconsejable inicializar explícitamente sus variables.

Esto no se garantiza en las variables "locales" — las que no son atributos de una clase. Así, si dentro de una definición de un método usted tiene:

```
int x;
```

X obtendrá algún valor arbitrario (como en C y C + +); y no será automáticamente inicializado a cero. Usted es responsable de asignar un valor apropiado antes de usar **x.** Si se olvida, Java definitivamente mejora a C + +: se genera un error en fase de compilación diciéndole que la variable no ha sido inicializada. (Muchos compiladores C + + le advertirán sobre la falta de inicialización de variables, pero en Java éstos son errores.)

Métodos, argumentos y valores de retorno

En muchos lenguajes (como C y C + +), el término *función* se usa para describir una subrutina. El término que es más comúnmente usado en Java es *método,* como "una forma para hacer algo." Si lo desea, usted puede continuar pensando en términos de funciones. Realmente es sólo una diferencia sintáctica, pero este libro sigue el uso común de Java con el término "método."

Los métodos en Java determinan los mensajes que un objeto puede recibir. En esta sección usted aprenderá lo fácil que es definir un método.

Las partes fundamentales de un método son el nombre, los argumentos, el tipo de retorno, y el cuerpo. Aquí está la forma básica:

```
tipoDeRetorno nombreDelMetodo( /* Lista de argumentos */ ) {
/* Cuerpo del método */
}
```

El tipo de retorno es el tipo del valor que se devuelve por el método después de que usted lo llame. La lista de argumentos da los tipos y los nombres de la información que usted quiere pasar al método. El nombre de método y lista de argumentos conjuntamente identifican inequívocamente al método.

Los métodos en Java pueden ser creados sólo como parte de una clase. Un método puede ser invocado sólo desde un **object [2]**, y ese objeto debe poder realizar la llamada a ese método. Si usted trata de llamar a un método equivocado para un objeto, usted obtendrá un mensaje de error en la fase de compilación. Para llamar a un método desde un objeto se nombra el objeto seguido por un punto, seguido por el nombre del método y su lista de argumentos, de la siguiente forma:

[2] métodos **estáticos**, que se explican más adelante, se pueden invocar desde la *clase*, sin un objeto.

```
NombreDelObjeto.nombreDelMetodo (arg1, arg2, arg3);
```

Por ejemplo, suponga que usted tiene un método **f()** que no tiene argumentos y devuelve un valor de tipo **int** . Entonces, si usted tiene un objeto que se llama **a** que puede invocar el método **f()**, usted puede decir esto:

```
int x = a.f();
```

El tipo del valor de regreso debe ser compatible con el tipo de **x.**

Este acto de invocar un método se denomina comúnmente como *enviar un mensaje a un objeto.* En el ejemplo citado anteriormente, el mensaje es **f ()** y el objeto lo es **a.** La programación orientada a objetos está a menudo resumida como simplemente “enviar mensajes *a* objetos.”

La lista de argumentos

La lista de argumentos de un método especifica qué información usted pasa en el método. Como usted podría adivinar, esta información — como todo lo demás en Java — se plasma en objetos. Entonces, lo que usted debe especificar en la lista de argumentos son los tipos de los objetos y su nombre que tiene cada uno. Como en cualquier situación en Java dónde a usted le parece que lo que usa son objetos, usted realmente usa referentes **[3]**. El tipo de la referencia debe ser correcta, sin embargo. Si el argumente debe de ser

supuestamente un **String**, usted debe pasar un **String** o el compilador dará un error.

[3] Con la excepción usual de tipos datos "especiales" **boolean, char**, **byte**, **short**, **int**, **long**, **float,** y **double .** En general, sin embargo, usted pasa objetos, lo que realmente usted pasa son las referencias a los objetos.

Considere un método que tomo a un **String** como argumento. Aquí está la definición, que debe ser colocada dentro de una definición de clase para para que sea compilada:

```
int storage(String s) {
return s.length() * 2;
}
```

Este método devuelve cuántos bytes son necesarios para almacenar un **String** particular. (Cada **char** en un **String** usa 16 bits, o dos bytes, para representar los caracteres Unicode.) El argumento es de tipo **String** y su nombre es **s**. Una vez **s** es pasada al método, usted lo puede usarla como cualquier otro objeto. (Usted puede enviarle mensajes.) Aquí, el método **length( )**, que es uno de los métodos de **Strings;** devuelve el número de caracteres en un **string**.

Usted también puede ver el uso de la palabra reservada **return**, la cual hace dos cosas.

Primero, que quiere decir "dejar el método, terminar." En segundo lugar, si el método devuelve un valor, ese valor es colocado inmediatamente después de la palabra **return.** En este caso, el valor de retorno se produce evaluando la expresión:

s.length () * 2.

Usted puede devolver el tipo que quiera, sino si usted no quiere devolver nada, usted lo hace eso señalando que el método devuelve **void.** Aquí hay algunos ejemplos:

```
boolean flag() { return true; }
float naturalLogBase() { return 2.718f; }
void nothing() { return; }
void nothing2() {}
```

Cuando el tipo de retorno es **null**, la palabra **return** es usada sólo para salir del método, y está por consiguiente innecesaria cuando usted alcanza el fin del método. Usted puede abandonar un método en cualquier momento, pero si usted devuelve un tipo no nulo el compilador le obligará (con mensajes de error) a devolver el tipo apropiado de valor que se espera.

En este punto, puede parecer que un programa es simplemente un montón de objetos con métodos que otros objetos toman como argumentos y envían mensajes a esos otros objetos. Eso es ciertamente lo que pasa, pero en el

siguiente capítulo aprenderá a hacer el trabajo de bajo nivel tomando decisiones dentro de un método. Para este capítulo, los envíos de mensajes son suficientes.

Construyendo un programa Java

Hay varios otros asuntos que usted debe entender antes de escribir su primer programa en Java.

Nombre visibilidad

Un problema en cualquier lenguaje de programación es el control de nombres. Si usted usa un nombre en un módulo del programa, y otros programadores usan el mismo nombre en otro módulo, ¿cómo hace para distinguir un nombre de otro e impedir que los dos nombres "colisionen"?
En C este es uno de los problemas particulares porque un programa sea a menudo un mar inmanejable de nombres. Las clases en C + + (en las que se basa Java) anidan funciones dentro de clases entonces no se pueden llamar con nombres de funciones que anidó dentro otras clases. Sin embargo, C + + todavía consiente datos globales y funciones globales. Para solucionar este problema, C + + introdujo loss *namespaces* usando palabras reservadas adicionales.

Java fue capaz de evitar todo esto haciendo un nuevo refinamiento. Para producir un nombre inequívoco para una biblioteca, el identificador usado no es diferente a un nombre de dominio de Internet. De hecho, los creadores Java quieren que usted use su nombre de dominio de Internet al revés para garantizar la unicidad.

Desde que mi nombre de dominio es **BruceEckel.com**, mi biblioteca de utilidades de fechas pasó a llamarse **com.bruceeckel.utility.fechas**. Después de su nombre de dominio puesto al revés, los puntos son usados para representar subdirectorios.

En Java 1.0 y Java 1.1 las extensiones de dominio **com**, **edu**, **org**, **produzcan**, etc., fueron capitalizado por convención, así es que la biblioteca aparecería:

COM.bruceeckel.utility.foibles.

Tras el desarrollo de Java 2, sin embargo, fue descubierto que esto causó problemas, y por eso ahora el nombre entero del paquete se escribe con letras minúscula.

Este mecanismo hace que todos sus archivos automáticamente pertenezcan a sus **namespaces**, y cada clase dentro de un archivo tiene un identificador único. Así es que usted no necesita aprender el lenguaje especial para solucionar este problema — el lenguaje se encarga de esto por usted.

Usando otros componentes

Cada vez que usted quiere usar una clase predefinida en su programa, el compilador debe saber cómo hallarla. Por supuesto, la clase ya podría existir en el mismo archivo de código fuente desde el que está siendo invocada. En ese caso, usted simplemente usa la clase — aun si la clase está definida posteriormente en el archivo (Java elimine el problema "referencias posteriores" por lo que no se debe preocupar de él).

¿Qué pasa con las clases definidas en otros ficheros? Usted podría pensar que el compilador debería ser lo suficientemente listo para simplemente ir y encontrarlo, pero hay un problema. Imagine que usted quiere usar una clase con un nombre particular, pero hay más de clase definida con ese nombre (probablemente definiciones diferentes). O peor, se imagina que usted escribe un programa, y cuando usted lo construye agrega una nueva clase a su biblioteca que está en conflicto con el nombre de una clase existente.

Para solucionar este problema, usted debe eliminar todas las ambigüedades potenciales.

Esto se resuelve diciendo al compilador Java exactamente qué clases quiere usar con la palabra reservada **import. Import** dice al compilador que importe un paquete, que es una biblioteca de clases. (En otros idiomas, una biblioteca podría constar de funciones y datos así como también clases, pero recuerde que todo código en Java debe estar escrito dentro de una clase.)

La mayoría de las veces usted estará usando componentes de las bibliotecas estándar Java que vienen con su compilador. Con estos, no necesita preocuparse de nombres de dominio puestos al revés; Usted solamente escribe, por ejemplo:

```
import java.util.ArrayList
```

Para decir al compilador que usted quiere usar la clase **ArrayList** de Java. Sin embargo, **util** contiene un gran número de clases y usted podría querer usar varias de ellas sin pronunciarlos a todos ellos explícitamente. Esto se facilita usando **‘ * ’** para indicar un comodín:

```
import java.util.*;
```

Es más común importar una colección de clases de esta manera que importar clases individualmente.

La palabra reservada **static**

Normalmente, cuando usted crea una clase usted describe cómo serán los objetos de esa clase y cómo se comportarán. Usted realmente no obtiene nada

hasta que crea un objeto de esa clase con **new**, y en ese momento se reserva la memoria para el objeto y los métodos se hacen disponibles.

Pero hay dos situaciones en las cuales este requisito no es suficiente. Uno es si usted quiere tener una reserva de memoria para un dato particular, a pesar de que varios objetos están creados, o si no hay ningún objeto creado. El otro es si usted necesita un método que no esté asociado a ningún objeto particular de esta clase. Es decir, usted necesita un método que usted puede llamar incluso si no hay objetos creados. Usted puede lograr ambos propósitos con la palabra reservada **static.** Cuando usted dice que algo es **static**, quiere decir esa información o el método no está asociado con ninguna instancia particular objeto de esa clase. Por lo tanto si usted nunca ha creado un objeto de esa clase usted puede llamar a un método **static** o puede acceder a datos **static.**

Con datos y métodos no estáticos usted debe crear un objeto y debe usar el objeto para acceder a los datos o el método, los datos y los métodos no estáticos deben conocer el objeto particular al que están asociados. Por supuesto, como los métodos **static** no necesitan que ningún objeto sea creado antes de que sean usados, no pueden acceder *directamente* a los métodos o miembros no estáticos simplemente llamándolos sin referenciar aun objeto creado (los miembros y los métodos **static** deben estar asociados a un objeto particular).

Algunos lenguajes orientados a objetos usan los *datos de clase* de términos y *métodos de clase*, queriendo decir que los datos y métodos existen sólo para la clase como un todo, y no para objetos particulares de la clase. Algunas veces la literatura Java usa estos términos también.

Para hacer un campo o método **static**, usted simplemente pone la palabra clave antes de la definición. Por ejemplo, las siguientes líneas declaran un campo **static** y lo inicializa:

```
class StaticTest {
static int i = 47;
}
```

Ahora si usted crea dos objetos **StaticTest**, habrá sólo un área de memoria reservada para **StaticTest.i.** Ambos objetos compartirán lo mismo. Considere:

```
StaticTest st1 = new StaticTest();
StaticTest st2 = new StaticTest();
```

En este punto, ambos **st1.i** y **st2.i** tienen el mismo valor 47 y apuntan a la isma zona de memoria. Hay dos formas para referirse a una variable **estática.** Como se indicó anteriormente, la puede referenciar por un objeto, diciendo, por ejemplo, **st2.i.** También puede referirse a eso directamente a través de su nombre de clase, algo que usted no puede hacer con un miembro no estático. (Ésta es la forma preferida para referirse a una variable **estática** ya que enfatiza la naturaleza **estática** de esa variable.)

```
StaticTest.i + +;
```

El operador **+ +** incrementa la variable. En este punto, ambos **st1.i** y **st2.i** tendrán el valor 48.Una lógica similar se aplica a los métodos estáticos. Usted puede referirse a un método estático también a través de un objeto como con cualquier otro método, o con la sintaxis especial **ClassName.method ().** Usted define un método estático similarmente:

```
class StaticFun {
static void incr() { StaticTest.i++; }
}
```

Usted puede ver que el método **StaticFun incr ()** incrementa los datos **estáticos** usando el operador **+**. Usted puede llamar a **incr ()** en la forma típica, a través de un objeto:

```
StaticFun sf = new StaticFun();
sf.incr();
```

O, como **incr ()** es un método estático, usted lo puede llamar directamente a través su clase:

```
StaticFun.incr();
```

Mientras **static**, estando aplicada a un campo, definitivamente cambia la forma de que los datos son creados (uno para cada clase vs. un para cada objeto), estando aplicada a un método esto no es tan dramático. Un uso importante de **static** para los métodos es permitir que usted llame ese método sin crear un objeto.

Esto es esencial, como veremos, en definir el método **principal ()** que es el punto de inicio para arrancar una aplicación.

Como cualquier método, un método **estático** puede crear o usar objetos de su tipo, así es que un método **estático** es a menudo usado como un “pastor” para una bandada de instancias de su propio tipo.

Su primer programa Java

Finalmente, aquí está su primer programa completo. Comienza por imprimir un String, y luego la fecha, usando la clase **Date** de la biblioteca estándar Java.

```
// HelloDate.java
import java.util.*;

public class HelloDate {
  public static void main(String[] args) {
    System.out.println("Hello, it's: ");
    System.out.println(new Date());
  }
```

```
}
```

En el comienzo de cada programa, debe colocar la declaración **import** para importar las clases adicionales que usted necesitará para el código de ese archivo. El decir extra es porque hay alguna biblioteca de clases que es automáticamente traída en cada archivo Java: **Java.lang.** Abra su navegador de Internet y mire la documentación de Sun. (Si usted no se la ha descargado de *java.sun.com* la documentación Java, hágalo ahora **[4]**). Si usted mira la lista de los paquetes, usted verá todas las bibliotecas de clases que vienen incluidas con Java. Seleccione **java.lang.** Esto mostrará una lista de todas las clases que tiene esa biblioteca. Como **java.lang** está implícitamente incluido en cada archivo de código Java, estas clases están automáticamente disponibles. La clase **Date** no existe en **java.lang**, lo cual quiere decir que debe importar otra biblioteca para usar esa funcionalidad. Si usted no conoce la biblioteca donde se encuentra una clase particular, o si usted quiere ver todas las clases, usted puede seleccionar "Tree" en la documentación Java. Ahora usted puede encontrar cualquier clase que viene con Java. Luego usted puede usar la función "buscar" del navegador para encontrar **Date.** Cuándo la encuentre verá como pertenece a **java.util.Date**, por lo que para usar **Date** se debe importar la librería pertinente con **import java.util.***.

[4] El compilador Java y la documentación de Sun no fué incluida en el CD de este libro porque tiende a cambiar regularmente. Si lo descarga usted mismo, puede obtener la versión mas reciente.

Retroceda al principio, seleccione **java.lang** y luego **System**, verá que la clase **System** tiene varios campos, y si usted indaga usted descubrirá que es un objeto **static PrintStream.** Como es **static** no necesita crear nada. El objeto **out** existe siempre y usted lo puede usar cuando quiera. Lo que usted puede hacer con el objeto **out** es determinado por el tipo que es: **PrintStream.** Convenientemente, **PrintStream** es mostrado en la descripción como un hiperenlace, así que si usted hace clic verá una lista de todos los métodos que usted puede llamar para **PrintStream.** Un buen número de estos estarán descritos más tarde en este libro. Por ahora todo en lo que nos interesa es **println (),** que lo que hace es "imprimir en la consola y acabe con una línea nueva." Así, en cualquier programa Java usted puede escribir **System.out.println ("las cosas");** Cada vez que usted quiere imprimir algo en pantalla.

El nombre de la clase es idéntico al nombre del archivo. Cuando usted crea un programa autónomo como éste, una de las clases en el archivo debe tener el mismo nombre que el archivo. (El compilador se queja si usted no hace esto.) Esta clase debe contener un método que se llame **main ()** con esta apraiencia:

```
public static void main(String[] args) {
```

La palabra reservada **public** quiere decir que el método está disponible para el mundo exterior (descrito en detalle en el capítulo 5). Los argumentos para **main()** es un array de objetos **String.** Los **args** no serán usados en este

programa, pero el compilador Javanecesita que esten allí porque almacena los parámetros desde la línea de comando.

La línea que escribe la fecha es muy interesante:

```
System.out.println (el nuevo Date ());
```

El argumento es un objeto **Date** que es creado para enviar su valor a **println ().** Tan pronto como esta declaración se ejecute, el objeto **Date** es innecesario, y el colector de basuras puede limpiarlo. No necesitamos preocuparnos por limpiarlo.

Compilando y ejecutando

Para compilar y ejecutar el programa, y todos los de este libro, usted primero debe tener un entorno de programación Java. Hay un gran número de entornos de desarrollo de terceros, pero en este libro que daremos por supuesto que usted usa al JDK de Sun, el cual es gratis. Si usted usa otro sistema de desarrollo **[4]**, necesitará mirar en la documentación de ese sistema cómo compilar y dirigir programas.

[5] El compilador IBM's "**jikes**" es una alternativa común, este es significativamente más rápido que el **javac** de Sun.

Vaya a *java.sun.com.* Allí encontrará información y enlaces que le ayudarán a descargar e instalar el JDK en su ordenador.

Una vez que el JDK es instalado, y usted ha establecido el path de su ordenador para que encuentre **javac** y **java**, descargue y descomprima el código fuente para este libro (lo puede encontrar en el CD-ROM incluido con este libro, o en *www.BruceEckel.com*). Esto creará un subdirectorio por cada capítulo de este libro. Vaya al subdirectorio **c02** y escriba:

```
javac HelloDate.java
```

Esta orden no debería producir respuesta. Si usted obtiene cualquier clase de mensaje de error quiere decir que usted no ha instalado el JDK correctamente y necesita investigar esos problemas. Por otra parte, si usted vuelve a recibir el prompt, escriba:

```
Java HelloDate
```

Y usted obtendrá el mensaje y la fecha como salida.

Este es el proceso que seguirá cada vez que compile y ejecute un programa de este libro. Sin embargo, usted verá que además del código fuente de este libro también tiene un archivo llamado **build.xml** en cada capítulo, que contiene "ant " órdenes para compilar automáticamente los archivos para ese capítulo. Ficheros compilados y **ant** (incluyendo el enlace para descargarlo) están

descritos que más completamente en el capítulo 15, una vez usted tiene instalado **'ant'** (http://jakarta.apache.org/ant) ya puede escribir **'ant'** en la ventana de comandos para compilar y ejecutar los programas en cada capítulo. Si no ha instalado **ant** aún, puede escribir **javac** y **java** las órdenes manuales.

Los comentarios y la documentación embebida

Hay dos tipos de comentarios en Java. La primera es el comentario tradicional estilo C que fue heredado de C++. Estos comentarios comienzan con un **/ *** y continúan, posiblemente a través de muchas líneas, hasta un ***/.** Muchos programadores empiezan cada línea de un comentario continuado con un *, así es más fácil reconocerlos:

```
/* éste es un comentario
 * que continúa
 * a través de líneas
 */
```

Recuerde, sin embargo, que todo dentro de **/ *** y ***/** será ignorado, por lo que hay diferencia en decir:

```
* éste es un comentario que
Continúa a través de líneas */
```

La segunda forma de comentarios proviene de C + +. Es el comentario de una línea sola, se empieza por // y continúa hasta el fin de la línea. Este carácter de imprenta de comentario es conveniente y comúnmente usado por su simplicidad. Usted no necesita buscar en el teclado /y luego * (en lugar de eso, usted presiona la misma tecla dos veces), y no hay necesidad de cerrar el comentario. Así es que usted a menudo verá:

```
// éste es un comentario de un línea
```

La documentación del comentario

Una de las mejores ideas en Java es que escribir código no es la unica actividad importante, documentar el código al menos es igual de importante. Posiblemente el problema más grande con documentar código ha sido mantener esa documentación. Si la documentación y el código están separados, se convierte en una molestia cambiar la documentación cada vez que usted cambia el código. La solución parece simple: Asocie el código a la documentación. La forma más fácil para hacer esto es poner todo en el mismo archivo. Para completar la descripción, sin embargo, usted necesita que una sintaxis especial de comentario para marcar la documentación, y una

herramienta para extraer esos comentarios y almacenarlos de una forma útil. Esto es lo que ha hecho Java.

La herramienta para extraer los comentarios es *javadoc*, y es una parte de la instalación JDK. Usa una parte de la tecnología del compilador Java para buscar etiquetas especiales del comentario que usted pone en sus programas. No sólo extrae la información marcada por estas etiquetas, si no que también extrae el nombre de clase y de los métodos que se anexa al comentario. Así por un incremento mínimo de trabajo puede conseguir una adecuada documentación de sus programas.

La salida de **javadoc** es un archivo de HTML que usted puede ver con su navegador de Internet. Por lo tanto, **javadoc** le permite crear y mantener un único archivo fuente y generar documentación útil. Gracias a **javadoc** tenemos un estándar para crear documentación.

Además, puede escribir sus propios manipuladores del **javadoc**, llamados a *doclets*, si usted quiere realizar operaciones especiales en la información tramitada por **javadoc** (devuelva en un formato diferente, por ejemplo). Doclets son explicados en el capítulo 15.

Lo que sigue es sólo una introducción y visión general de los fundamentos de **javadoc**. Una amplia descripción se puede encontrar en la documentación JDK descargable desde *java.sun.com* (Se trata de una descarga independiente a JDK). Cuando usted descomprima la documentación, mire en los subdirectorios "tooldocs" (o sigue a los enlaces " tooldocs ").

La sintaxis

Todas las órdenes del **javadoc** ocurren sólo dentro de comentarios ***/ * *.*** Los comentarios acaban con ***** / como siempre. Hay dos formas primarias para usar **javadoc**: Incruste HTML, o uso "doc tags." *Las etiquetas doc* son órdenes que empiezan con '**@**' y son colocadas al principio de la línea del comentario. Las etiquetas doc puede aparecer en cualquier parte de un comentario **javadoc**, siempre empezando con '**@**' pero dentro de llaves.

Hay tres " tipos " de comentarios de documentación, que corresponde al elemento que se va a comentar: clase, variable, o método. Es decir, un comentario de clase aparece antes de la definición de la clase; un comentario de variable aparece justo delante de la definición de una variable, y un comentario de método aparece delante de la definición de un método. Como un ejemplo simple:

```
/** A class comment */
public class docTest {
/** A variable comment */
public int i;
/** A method comment */
public void f() {}
```

```
}
```

Hay que destacar que javadoc sólo procesará documentación de comentario para miembros **públicos** y **protegidos.** Los comentarios para **private** y los miembros de acceso de paquete (vea a capítulo 5) son ignorados y usted obtendrá su salida. (Sin embargo, usted puede usar la opción **-private** para incluir a miembros **private** también.) Esto tien sentido, ya que sólo los miembros **públicos** y **protegidos** están disponible fuera del archivo. Sin embargo, todos los comentarios **de clase** son incluidos en la salida.

La salida para el código citado anteriormente es un archivo de HTML que tiene el mismo formato del estándar como el resto de la documentación Java, así es que los usuarios estarán acostumbrados al formato y fácilmente podrán navegar sus clases. Es digno de entrar en el código citado anteriormente, enviarlo a través de javadoc y mirar el archivo resultante de HTML para ver los resultados.

El HTML embebido

Javadoc pasa órdenes de HTML al documento HTML generado. Esto permite que pueda usar toda la funcionalidad de HTML; Sin embargo, el motivo primario es dejarle formatear código:

```
/**
* <pre>
* System.out.println(new Date());
* </pre>
*/
```

Usted también puede usar HTML tal como usted lo haría en cualquier otro documento de Web para formatear el texto en sus descripciones:

```
/**
* You can <em>even</em> insert a list:
* <ol>
* <li> Item one
* <li> Item two
* <li> Item three
* </ol>
*/
```

Notese que dentro de la documentación, los asteriscos a principio de una línea son ignorados, junto con espacios principales. Javadoc *110 Pensar en Java Www.BruceEckel.com* reformatea todo a fin de que se conforme a la apariencia estándar de la documentación. No use encabezamientos HTML como **< h1** > o **< hr** > porque javadoc inserta sus propios encabezamientos y el interferiría con ellos.

Todos los tipos de documentación comentario — clase, variable, y método — pueden soportar HTML embebido.

Algunas etiquetas de ejemplo

Aquí hay alguna de las etiquetas del javadoc disponibles para documentación de código. Antes de tratar de hacer algo serio usando javadoc, usted debería consultar la referencia de javadoc en la documentación de JDK para tener un mayor conocimiento para usar javadoc.

@**see:** referenciando otras clases

Las etiquetas **@see** permiten referenciar la documentación en otras clases. Javadoc generará HTML con las etiquetas **@see** enlacadas a la otra documentación. Las formas son:

```
@see classname
@see fully-qualified-classname
@see fully-qualified-classname#method-name
```

Cada uno agrega un hiper-enlace “See Also” a la documentación generada. Javadoc no comprobará que los enlaces que usted lo da hacer seguro son válidos.

{@link *package.class#member label*}

Muy similar para **@see** , excepto que puede ser usada como el texto del hiper-enlace en vez de “ See Also.”

{ @ docRoot }

Produce el camino relativo al directorio raíz de la documentación. Útil para crear hiper-enlaces a páginas del árbol de la documentación.

{ @ inheritDoc }

Hereda la documentación de la clase más próxima de este comentario de doctor de la corriente del intothe de clase.

@version

Así es su uso:

```
@version información de la versión
```

@versión en la cual **la información de versión** es la información más significativa usted escoge incluir. Cuando lo la opción **-version** es colocada en la línea de comando del javadoc, la información de versión será generada para el fichero HTML resultante.

@author

Así es su uso:

```
@author información del autor
```

Lógicamente es el nombre del autor, pero también podría incluir su dirección de correo electrónico o cualquier otra información apropiada. Con la opción **-author** en la línea de comando del javadoc, la información del autor será generada en la salida de la documentación generada de HTML.

Usted puede tener etiquetas múltiples de autor para una lista de autores, pero deben estar consecutivas. Toda la información del autor será tratada de forma conjunta en un párrafo solo en el HTML generado.

@since

Esta etiqueta permite que usted indique la versión de este código que tiene un rasgo particular. Usted lo verá aparecer en la documentación de HTML Java para indicar qué versión del JDK es usada.

@param

Esto sirve para la documentación de los métodos, y su uso es el siguiente:

```
@param parameter-name description
```

Por cada **nombre de parámetro** es el identificador en la lista de parámetros del método, y **la descripción** es del texto que puede continuar en subsiguientes líneas. La descripción es considerada acabada cuando se encuentra una etiqueta nueva de la documentación. Usted puede tener cualquier número de estos, probablemente un para cada parámetro.

@return

Esto sirve para la documentación de los métodos, y su uso es el siguiente:

```
@return description
```

La descripción le da el significado del valor de regreso. Puede continuar en subsiguientes líneas.

@throws

Las excepciones serán explicadas en el capítulo 9, pero brevemente son objetos que pueden ser lanzados desde un método si ese método falla. Aunque sólo un objeto de excepción puede ser lanzado cuando usted llama un método, un método particular podría producir varios de tipos diferentes de excepciones. El uso de la etiqueta de excepción es:

```
@throws fully-qualified-class-name description
```

Da un nombre inequívoco de clase, y **la descripción** (que pueda continuar en subsiguientes líneas) le dice por qué este tipo particular de excepción puede emerger de la llamada de método.

@deprecated

Esto se usa para indicar características que se han mejorado en versiones posteriores. La etiqueta deprecated es una sugerencia para no usar este rasgo particular, y que en el futuro tiene probabilidad de estar borrado. Un método que es marcado con **@deprecated** hace que el compilador de una advertencia si es usado.

Ejemplo de documentación

Aquí está el primer programa Java otra vez, esta vez con comentarios añadidos:

```
//: c02:HelloDate.java
import java.util.*;
/** The first Thinking in Java example program.
* Displays a string and today's date.
* @author Bruce Eckel
* @author www.BruceEckel.com
* @version 2.0
*/

public class HelloDate {
     /** Sole entry point to class & application
     * @param args array of string arguments
     * @return No return value
     * @exception exceptions No exceptions thrown
     */

     public static void main(String[] args) {
          System.out.println("Hello, it's: ");
          System.out.println(new Date());
     }
} ///:~
```

La primera línea del archivo usa la técnica de escribir ‘ ***//:***’ Como un indicador especial para la línea del comentario conteniendo el nombre del archivo fuente. Esa la línea contiene la información de la ruta del archivo (en este caso, **c02** indica a capítulo 2) seguido del nombre del archivo **[6]**. La última línea también termina con un comentario, y éste (‘ ***///:***~’) indica el fin del código fuente, lo cual permite que automáticamente se actualice en el texto de este libro después de que ser ejecutado con un compilador y ser ejecutado.

[6] Originalmente, creé que una herramienta usando Python (vea a www.Python.org) usa esta información para extraer los archivos de código, ponerlos en subdirectorios apropiados, y crear makefiles. En esta edición, todos los archivos se guardan en CVS y automáticamente incorporados en este libro usando un macro VBA (Visual Basic for Applications). Esta nueva mejora parece ser mucho mejor en términos del mantenimiento de código, en su mayor parte por CVS.

Codificando con estilo

El estilo descrito en el *'Code Conventions for the Java Programming Language* **[7]** *'* es capitalizar la primera letra de un nombre de clase. Si el nombre de clase consta de varias palabras, se escriben de forma concatenada (es decir, no se usan guiones para separar los nombres), y la primera letra de cada palabra es capitalizada, de la siguiente forma:

```
class AllTheColorsOfTheRainbow { // ...
```

Esto se denomina "escritura de camello." Para casi todo lo demás: métodos, campos (variables del miembro), y nombres de referencias a objetos, el estilo aceptado es como para las clases *excepto que* la primera letra del identificador es letra minúscula. Por ejemplo:

```
class AllTheColorsOfTheRainbow {
     int anIntegerRepresentingColors;

     void changeTheHueOfTheColor(int newHue) {
          // ...
     }

     // ...
}
```

Por supuesto, usted debería recordar que el usuario también debe escribir todo estos nombres largos, y por tanto hay que ser compasivos. El código Java que usted verá en las bibliotecas de Sun también sigue la colocación de llaves como se hace en este libro.

[7] Http://java.sun.com/docs/codeconv/index.html. Por el espacio físico en este libro como en diapositivas de cursos es imposible seguir todas las directivas que se exponen.

Resumen

La meta de este capítulo es aprender los mínimos conceptos básicos de Java cómo escribir un programa simple. Usted también ha recibido una visión general del lenguaje y algunas de sus ideas básicas. Sin embargo, todos los ejemplos hasta ahora han sido de la clase "haga esto, luego haga eso, luego si

no haga algo.” Qué ocurre si usted quiere que el programa haga elecciones, tales como “si el resultado de hacer esto es rojo, haga eso; En caso de que no, entonces hace otra cosa”. Los conocimientos de Java para esta actividad fundamental de programación se estudia en el siguiente capítulo.

Ejercicios

1. Siguiendo el ejemplo **HelloDate.java** de este capítulo, crear un programa "hola, mundo" que simplemente imprima esa frase. Solo necesitamos un método en nuestra clase ("**main**", que se ejecuta cuando empieza el programa). Recordemos hacerlo **static** e incluir la lista de argumentos, aunque no la utilicemos. Compilar el programa con **javac** y ejecutarlo con java. Si **alguién** está utilizando un entorno de desarrollo distinto del JDK, deberá aprender a compilar y ejecutar en él.

2. Encontrar el fragmento de código donde aparece **UnNombredeTipo** y convertirlo en un programa Java que se compile y ejecute.

3. Convertir el fragmento de código de **SoloDatos** en un programa que se compile y ejecute.

4. Modificar el ejercicio 3 de modo que los valores de las variables en **SoloDatos** se asignen e impriman en el **main**()

5. Escribir un programa que incluya y utilice el método **storage**() que definimos en este capítulo

6. Convertir el código de **StaticFun** en un programa funcional.

7. Escribir un programa que imprima tres argumentos tomados de la línea de comandos. Para esto, necesitaremos indicar en la línea de comandos un array de **strings**.

8. Convertir el ejemplo de **TodosLosColoresDelArcoIris** en un programa que se compile y ejecute.

9. Buscar el código de la segunda versión de **HelloDate.java** , que es un ejemplo sencillo de los comentarios de documentación. Ejecutar **javadoc** sobre ese fichero y ver los resultados con el navegador Web.

10. Convertir **docTest** en un fichero que se compile y pasarlo por **javadoc** . Verificar la documentación resultante con el navegador Web.

11. Añadir una lista de items HTML a la documentación en el ejercicio 10.

12. Coger el programa del ejercicio 1 y añadirle comentarios de documentación. Extraer esos comentarios a un fichero HTML utilizando **javadoc** y verlos en el navegador web.