developerWorks

Introducción a la programación Java, parte 1: Conceptos básicos del lenguaje Java

Programación orientada a objetos en la plataforma Java

J. Steven Perry
Consultor Director
Makoto Consulting Group, Inc.

Nivel de dificultad: Introductoria

Fecha: 03-12-2012

Este tutorial de dos partes presenta la estructura, la sintaxis y el paradigma de programación del lenguaje y la plataforma Java™. Aprenderá la sintaxis Java que más probablemente encontrará profesionalmente y los modismos de la programación Java que puede usar para desarrollar aplicaciones Java sólidas y plausibles de mantener. En la Parte 1, J. Steven Perry lo guía a través de los puntos fundamentales de la programación orientada a objetos en la plataforma Java, que incluye la sintaxis Java fundamental y su uso. Comenzará con la creación de objetos Java y la adición de comportamiento a ellos y concluirá con una introducción a la Infraestructura de colecciones Java, con un terreno considerable cubierto en el medio.

Ver más contenido de esta serie

Sección 1. Antes que comience

Descubra qué esperar de este tutorial y cómo sacarle el mayor provecho.

Acerca de este tutorial

Desarrolle habilidades de este tema

Este contenido es parte de un knowledge path progresivo para avanzar en sus habilidades. Vea Conviértase en un desarrollador Java

El tutorial de dos partes "Introducción a la programación Java" tiene como objetivo conseguir que los desarrolladores de software que sean nuevos en la tecnología

Java estén listos y en movimiento con la programación orientada a objetos (OOP) y el desarrollo de aplicaciones del mundo real usando el lenguaje y la plataforma Java.

La primera parte es una introducción paso a paso a la OOP con el uso del lenguaje Java. El tutorial comienza con una visión general de la plataforma y el lenguaje Java y le siguen las instrucciones para establecer un entorno de desarrollo que consiste en un Kit de desarrollo de Java (JDK) y el Eclipse IDE. Una vez que se le hayan presentado los componentes de su entorno de desarrollo, comenzará a aprender la sintaxis Java básica en un modo práctico.

La Parte 2 cubre funciones de lenguaje más avanzadas, que incluyen expresiones regulares, genéricos, E/S y serialización. Los ejemplos de programación en la Parte 2 se desarrollan en base al objeto `Person` que usted comienza a desarrollar en la Parte 1.

Objetivos

Cuando haya terminado la Parte 1, estará familiarizado con la sintaxis del lenguaje Java básico y podrá escribir programas Java simples. Debería continuar con "Introducción a la programación Java, parte 2: construcciones para aplicaciones del mundo real" para desarrollarse sobre esta base.

Requisitos previos

Este tutorial es para desarrolladores de software que todavía no tienen experiencia con el código Java o la plataforma Java. El tutorial incluye una visión general de los conceptos de OOP.

Requisitos del sistema

Para completar los ejercicios de este tutorial, instale y establezca un entorno de desarrollo que consista en:

- JDK 6 de Sun/Oracle.
- IDE Eclipse para desarrolladores Java.

En el tutorial, se incluyen instrucciones de descarga e instalación para ambos.

La configuración recomendada del sistema es la siguiente:

- Un sistema que soporte Java SE 6 con al menos 1GB de memoria principal. Java 6 tiene soporte en Linux®, Windows® y Solaris®.
- Al menos 20MB de espacio en disco para instalar los componentes y ejemplos del software.

Sección 2. Visión general de la plataforma Java

La tecnología Java se usa para desarrollar aplicaciones para un amplio alcance de entornos, desde dispositivos del consumidor hasta sistemas empresariales heterogéneos. En esta sección, obtenga una vista de alto nivel de la plataforma Java y sus componentes. Vea Recursos para aprender más acerca de los componentes de la plataforma Java discutidos en esta sección.

El lenguaje Java

Conozca las API de Java

La mayoría de los desarrolladores Java hacen referencia constantemente a la documentación API de Java online oficial, — también llamada el Javadoc (vea Recursos). De forma predeterminada, usted ve tres marcos en el Javadoc. El marco superior izquierdo muestra todos los paquetes en la API y debajo están las clases en cada paquete. El marco principal (a la derecha) muestra detalles del paquete o de la clase seleccionada actualmente. Por ejemplo, si selecciona el paquete `java.util` en el marco superior izquierdo y luego selecciona la clase `ArrayList` que aparece debajo de él, en el marco derecho, verá detalles acerca del `ArrayList`, que incluyen una descripción de lo que hace, cómo usarlo y sus métodos.

Como cualquier lenguaje de programación, el lenguaje Java tiene su propia estructura, reglas de sintaxis y paradigma de programación. El paradigma de programación del lenguaje Java se basa en el concepto de programación orientada a objetos (OOP), que las funciones del lenguaje soportan.

El lenguaje Java es un derivado del lenguaje C, por lo que sus reglas de sintaxis se parecen mucho a C: por ejemplo, los bloques de códigos se modularizan en métodos y se delimitan con llaves (`{` y `}`) y las variables se declaran antes de que se usen.

Estructuralmente, el lenguaje Java comienza con *paquetes*. Un paquete es el mecanismo de espacio de nombres del lenguaje Java. Dentro de los paquetes se encuentran las clases y dentro de las clases se encuentran métodos, variables, constantes, entre otros. En este tutorial, aprenderá acerca de las partes del lenguaje Java.

El compilador Java

Cuando usted programa para la plataforma Java, escribe el código de origen en archivos .java y luego los compila. El compilador verifica su código con las reglas de sintaxis del lenguaje, luego escribe los *códigos byte* en archivos .class. Los códigos byte son instrucciones estándar destinadas a ejecutarse en una Java Virtual Machine (JVM). Al agregar este nivel de abstracción, el compilador Java difiere de los otros compiladores de lenguaje, que escriben instrucciones apropiadas para el chipset de la CPU en el que el programa se ejecutará.

La JVM

Al momento de la ejecución, la JVM lee e interpreta archivos .class y ejecuta las instrucciones del programa en la plataforma de hardware nativo para la que se escribió la JVM. La JVM interpreta los códigos byte del mismo modo en que una CPU interpretaría las instrucciones del lenguaje del conjunto. La diferencia es que la JVM es un software escrito específicamente para una plataforma particular. La JVM es el corazón del principio "escrito una vez, ejecutado en cualquier lugar" del lenguaje Java. Su código se puede ejecutar en cualquier chipset para el cual una implementación apropiada de la JVM está disponible. Las JVM están disponibles para plataformas principales como Linux y Windows y se han implementado subconjuntos del lenguaje Java en las JVM para teléfonos móviles y aficionados de chips.

El recolector de basura

En lugar de forzarlo a mantenerse a la par con la asignación de memoria (o usar una biblioteca de terceros para hacer esto), la plataforma Java proporciona una gestión de memoria lista para usar. Cuando su aplicación Java crea una instancia de objeto al momento de ejecución, la JVM asigna automáticamente espacio de memoria para ese objeto desde el *almacenamiento dinámico*, que es una agrupación de memoria reservada para que use su programa. El *recolector de basura* Java se ejecuta en segundo plano y realiza un seguimiento de cuáles son los objetos que la aplicación ya no necesita y recupera la memoria que ellos ocupan. Este abordaje al manejo de la memoria se llama *gestión de la memoria implícita* porque no le exige que escriba cualquier código de manejo de la memoria. La recogida de basura es una de las funciones esenciales del rendimiento de la plataforma Java.

El kit de desarrollo de Java

Cuando usted descarga un kit de desarrollo de Java (JDK), obtiene, — además del compilador y otras herramientas, — una librería de clase completa de programas de utilidad preconstruidos que lo ayudan a cumplir cualquier tarea común al desarrollo de aplicaciones. El mejor modo para tener una idea del ámbito de los paquetes y bibliotecas JDK es verificar la documentación API JDK (vea Recursos).

El Java Runtime Environment

El Java Runtime Environment (JRE, también conocido como el Java Runtime) incluye las bibliotecas de códigos de la JVM y los componentes que son necesarios para programas en ejecución escritos en el lenguaje Java. Está disponible para múltiples plataformas. Puede redistribuir libremente el JRE con sus aplicaciones, de acuerdo a los términos de la licencia del JRE, para darles a los usuarios de la aplicación una plataforma en la cual ejecutar su software. El JRE se incluye en el JDK.

Sección 3. Configuración de su entorno de desarrollo de Java

En esta sección, tendrá instrucciones para descargar e instalar el JDK 6 y el lanzamiento actual de IDE Eclipse y para configurar su entorno de desarrollo de Eclipse.

Si usted ya ha instalado el JDK e IDE Eclipse, tal vez quiera saltar a la sección Getting started with Eclipse o a la que le sigue, Object-oriented programming concepts.

Su entorno de desarrollo

El JDK incluye un conjunto de herramientas de línea de comandos para compilar y ejecutar su código Java, que incluye una copia completa del JRE. Aunque usted ciertamente puede usar estas herramientas para desarrollar sus aplicaciones, la mayoría de los desarrolladores valoran la funcionalidad adicional, la gestión de tareas y la interfaz visual de un IDE.

Eclipse es un IDE de código abierto popular para el desarrollo Java. Maneja las tareas básicas, tales como la compilación de códigos y la configuración de un entorno de depuración, para que pueda centrase en escribir y probar códigos. Además, puede usar Eclipse para organizar archivos de códigos de origen en proyectos, compilar y probar esos proyectos y almacenar archivos de proyectos en cualquier cantidad de repositorios de origen. Necesita tener instalado un JDK para usar Eclipse para el desarrollo Java.

Instale JDK 6

Siga estos pasos para descargar e instalar JDK 6:

1. Navegue hasta Java SE Downloads y haga clic en el recuadro **Java Platform (JDK)** para visualizar la página de descarga de la última versión del JDK (JDK 6, actualización 21 al momento de escritura).
2. Haga clic en el botón **Download**.
3. Seleccione la plataforma del sistema operativo que necesita.
4. Se le pedirá el nombre de usuario y contraseña de su cuenta. Ingréselos si tiene una cuenta, regístrese si no la tiene o puede hacer clic en **Continue** para saltar este paso y continuar para hacer la descarga.
5. Guarde el archivo en su unidad de disco duro cuando se lo solicite.
6. Cuando se complete la descarga, ejecute el programa de instalación. (El archivo que acaba de descargar es un archivo ZIP autoextraíble que también

es el programa de instalación). Instale el JDK en su unidad de disco duro en un una ubicación fácil de recordar (por ejemplo, C:\home\jdk1.6.0_20 en Windows o ~/jdk1.6.0_20 en Linux). Es una buena idea codificar el número de actualización en el nombre del directorio de instalación que usted elija.

Ahora tiene un entorno Java en su máquina. A continuación, instalará el IDE Eclipse.

Instale Eclipse

Para descargar e instalar Eclipse, siga estos pasos:

1. Navegue hasta Eclipse Galileo Sr2 Packages.
2. Haga clic en **Eclipse IDE for Java Developers**.
3. Bajo los Enlaces de descarga en el lado derecho, seleccione su plataforma.
4. Seleccione la réplica desde la cual quiere hacer la descarga, luego guarde el archivo en su unidad de disco duro.
5. Extraiga los contenidos del archivo .zip a una ubicación en su disco duro que pueda recordar fácilmente (por ejemplo, C:\home\eclipse en Windows o ~/eclipse en Linux).

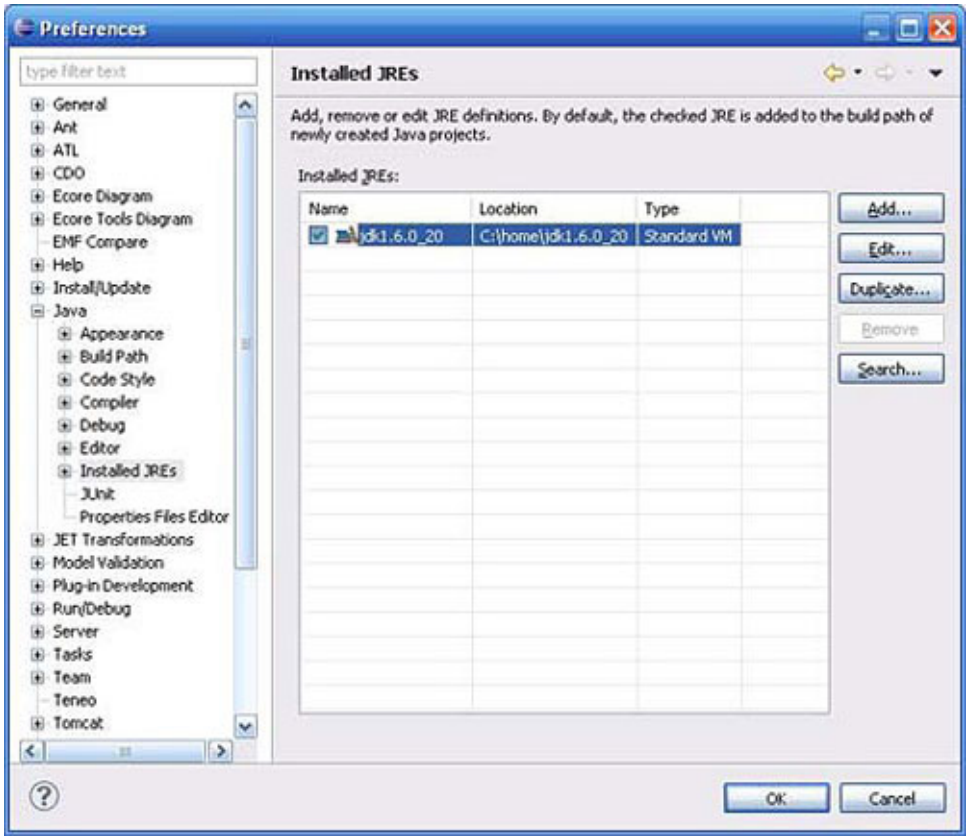
Configure Eclipse

El IDE Eclipse se coloca encima del JDK como una extracción útil pero todavía necesita acceder al JDK y sus diversas herramientas. Antes de que pueda usar Eclipse para escribir el código Java, tiene que indicarle dónde está ubicado el JDK.

Para configurar su entorno de desarrollo Eclipse:

1. Inicie Eclipse al hacer doble clic en eclipse.exe (o el ejecutable equivalente para su plataforma).
2. Aparecerá el Iniciador del espacio de trabajo, que le permitirá seleccionar una carpeta raíz para sus proyectos Eclipse. Elija una carpeta que recuerde fácilmente, por ejemplo C:\home\workspace en Windows o ~/workspace en Linux.
3. Descarte la pantalla Bienvenido a Eclipse.
4. Haga clic en **Window > Preferences > Java > Installed JREs**. La Ilustración 1 muestra la pantalla de configuración para el JRE:

Ilustración 1. Configuración del JDK que Eclipse utiliza.



5. Eclipse indicará un JRE instalado. Necesita asegurarse de usar el que descargó con JDK 6. Si Eclipse no detecta automáticamente el JDK que usted instaló, haga clic en **Add...** y en el siguiente diálogo **Standard VM**, luego haga clic en **Next**.
6. Especifique el directorio de inicio del JDK (por ejemplo, C:\home\jdk1.6.0_20 en Windows), luego haga clic en **Finish**.
7. Confirme que el JDK que quiere usar esté seleccionado y haga clic en **OK**.

Ahora Eclipse está configurado y listo para que usted cree proyectos y compile y ejecute códigos Java. La siguiente sección lo familiarizará con Eclipse.

Sección 4. Comenzar a usar Eclipse

Eclipse no es solo un IDE, es todo un ecosistema de desarrollo. Esta sección es una breve introducción práctica para usar Eclipse para el desarrollo Java. Vea Recursos si quiere aprender más acerca de Eclipse.

El entorno de desarrollo de Eclipse

El entorno de desarrollo de Eclipse tiene cuatro componentes principales:

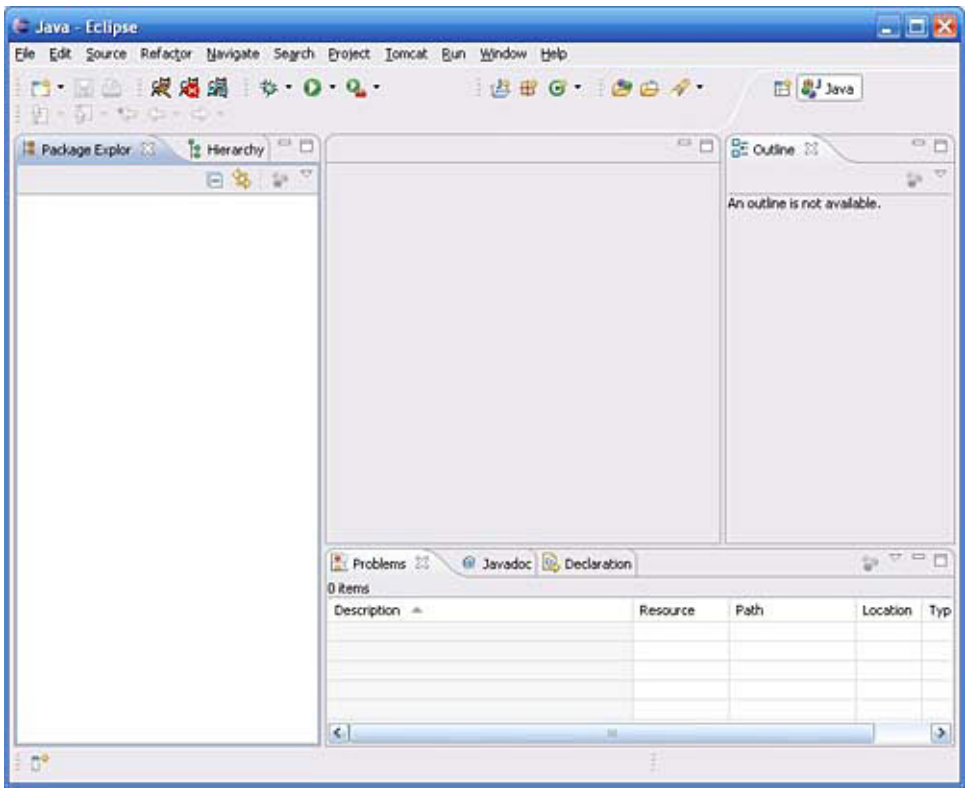
- Espacio de trabajo
- Proyectos
- Perspectivas
- Vistas

La unidad primaria de organización en Eclipse es el *espacio de trabajo*. Un espacio de trabajo contiene todos sus *proyectos*. Una *perspectiva* es un modo de observar cada proyecto (de ahí el nombre) y dentro de una perspectiva hay una o más *vistas*.

La perspectiva Java

La Ilustración 2 muestra la perspectiva Java, que es la perspectiva predeterminada para Eclipse. Debería ver esta perspectiva cuando inicie Eclipse.

Ilustración 2. Perspectiva Java de Eclipse



La perspectiva Java contiene las herramientas que necesita para comenzar a escribir las aplicaciones Java. Cada pestaña que se muestra en la Ilustración 2 es una vista para la perspectiva Java. Package Explorer y Outline son dos vistas particularmente útiles.

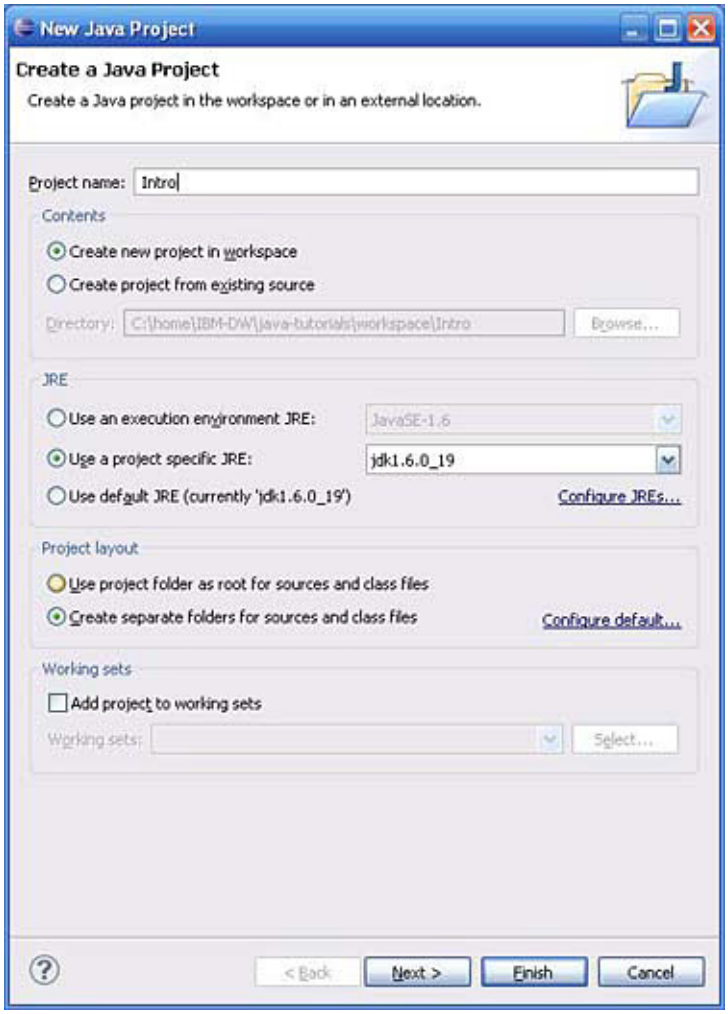
El entorno Eclipse tiene un alto grado de configuración. Cada vista es acoplable, por lo que puede desplazarla alrededor de la perspectiva Java y ubicarla donde quiera. Aunque por ahora, quédese con la configuración predeterminada de la perspectiva y de la vista.

Cree un proyecto

Siga estos pasos para crear un proyecto Java nuevo:

1. Haga clic en **File > New > Java Project ...** y verá un recuadro de diálogo que se abre como el que se muestra en la Ilustración 3:

   Ilustración 3. Asistente para el proyecto Java nuevo



2. Ingrese `Intro` como nombre del proyecto y haga clic en **Finish**.
3. Si quiere modificar los parámetros predeterminados del proyecto, haga clic en **Next**. (Esto se recomienda *solo* si tiene experiencia con el IDE Eclipse).
4. Haga clic en **Finish** para aceptar la configuración del proyecto y crear el proyecto.

Ahora ha creado un nuevo proyecto Java de Eclipse y carpeta de origen. Su entorno de desarrollo está listo para actuar. Sin embargo, una comprensión del paradigma OOP, — que se cubre en las siguientes dos secciones de este tutorial, — es esencial. Si usted está familiarizado con los conceptos y principios de OOP, tal vez quiera saltar a Getting started with the Java language.

Sección 5. Conceptos de programación orientada a objetos

El lenguaje Java está (en su mayor parte) orientado a objetos. Si no ha utilizado un lenguaje orientado a objetos antes, sus conceptos pueden parecer extraños al principio. Esta sección es una breve introducción a los conceptos del lenguaje OOP, que utiliza programación estructurada como punto de contraste.

¿Qué es un objeto?

Los lenguajes de programación estructurada como C y COBOL siguen un paradigma de programación muy diferente de los orientados a objetos. El paradigma de programación estructurada está altamente orientado a datos, lo cual significa que usted tiene estructuras de datos por una parte y luego instrucciones del programa que actúan sobre esos datos. Los lenguajes orientados a objetos, como el lenguaje Java, combinan datos e instrucciones del programa en *objetos*.

Un objeto es una entidad independiente que contiene atributos y comportamientos y nada más. En lugar de tener una estructura de datos con campos (atributos) y pasar esa estructura a toda la lógica del programa que actúa sobre ella (comportamiento), en un lenguaje orientado a objetos, se combinan los datos y la lógica del programa. Esta combinación puede ocurrir en niveles completamente diferentes de granularidad, desde objetos específicos como un `Number` hasta objetos de aplicación general como un servicio de `FundsTransfer` en una gran aplicación bancaria.

Objetos padre e hijo

Un *objeto padre* es aquel que sirve como la base estructural para derivar *objetos hijos* más complejos. Un objeto hijo se parece a su padre pero es más especializado. El paradigma orientado a objetos le permite reutilizar los atributos y comportamientos comunes del objeto padre y le agrega a sus objetos hijos atributos y comportamientos que difieren. (Usted aprenderá más sobre *herencia* en la siguiente sección de este tutorial).

Comunicación y coordinación de objetos

Los objetos se comunican con otros objetos por medio del envío de mensajes (*llamadas de método* en el lenguaje Java). Además, en una aplicación orientada a objetos, el código del programa coordina las actividades entre objetos para realizar tareas dentro del contexto del dominio de aplicación dado. (En el paradigma Modelo-Vista-Controlador, este código de programa de coordinación es el Controlador. Vea Recursos para aprender más acerca de MVC).

Resumen del objeto

Un objeto bien escrito:

- tiene límites nítidos.
- realiza un conjunto limitado de actividades.
- conoce solo lo relacionado a sus datos y cualquier otro objeto que necesite para cumplir sus actividades.

Básicamente, un objeto es una entidad diferenciada que tiene solo las dependencias necesarias de otros objetos para realizar sus tareas.

Ahora verá cómo luce un objeto.

El objeto `Person`

Comenzaré con un ejemplo que está basado en escenario común de desarrollo de aplicación: se representa a un individuo con un objeto `Person`.

Volviendo a la definición de un objeto, usted sabe que un objeto tiene dos elementos primarios: atributos y comportamiento. Verá cómo estos se aplican al objeto `Person`.

Atributos

¿Qué atributos puede tener una persona? Algunos atributos comunes incluyen:

- Nombre
- Edad
- Altura
- Peso
- Color de ojos
- Género

Probablemente a usted se le pueden ocurrir más (y siempre puede agregar más atributos más tarde) pero esta lista es un buen comienzo.

Comportamiento

Una persona real puede hacer todo tipo de actividades pero los comportamientos de los objetos normalmente se relacionan con algún tipo de contexto de aplicación. En un contexto de aplicación de negocio, por ejemplo, puede querer preguntarle a su objeto `Person`: "¿Qué edad tiene?" Como respuesta, `Person` le diría el valor de su atributo de `Edad`.

Una lógica más compleja podría estar oculta dentro del objeto `Person` pero por ahora suponga que esa `Person` tiene el comportamiento de responder estas preguntas:

- ¿Cuál es su nombre?
- ¿Qué edad tiene?
- ¿Cuál es su altura?

- ¿Cuánto pesa?
- ¿Cuál es su color de ojos?
- ¿Cuál es su género?

Estado y cadena

El *Estado* es un concepto importante en OOP. El estado de un objeto se representa en cualquier momento por medio del valor de sus atributos.

En el caso de `Person`, su estado se define por sus atributos, tales como nombre, edad, altura y peso. Si usted quisiera presentar una lista de varios de esos atributos, podría hacerlo utilizando una clase de `String`, sobre la cual hablaré más en el tutorial posteriormente.

Juntos, los conceptos de estado y cadena le permiten decirle a `Person`: cuénteme quién es usted dándome un listado (o una `String`) de sus atributos.

Sección 6. Principios de OOP

Si usted viene de un ambiente de programación estructurada, la proposición de valores de OOP puede que no sea clara todavía. Después de todo, los atributos de una persona y cualquier lógica para recuperar (y convertir) sus valores pueden escribirse en C o COBOL. Esta sección clarifica los beneficios del paradigma OOP al explicar sus principios distintivos: *encapsulamiento*, *herencia* y *polimorfismo*.

Encapsulamiento

Recuerde que un objeto es, sobre todo, diferenciado o independiente. Este es el principio de *encapsulamiento* en funcionamiento. *Ocultación* es otro término que a veces se usa para expresar la naturaleza independiente y protegida de los objetos.

Sin tener en cuenta la terminología, lo que es importante es que el objeto mantiene un límite entre su estado y comportamiento y el mundo exterior. Como los objetos del mundo real, los objetos usados en la programación de computadora tienen diversos tipos de relaciones con diferentes categorías de objetos en las aplicaciones que los utilizan.

En la plataforma Java, puede usar *especificadores de acceso* (los que presentaré más adelante en el tutorial) para variar la naturaleza de las relaciones de los objetos desde lo *público* a lo *privado*. El acceso público tiene una gran apertura, mientras que el acceso privado significa que los atributos del objeto son accesibles solo dentro del objeto mismo.

El límite entre lo público y lo privado hace cumplir el principio orientado a objetos de encapsulamiento. En la plataforma Java, usted puede variar la fortaleza de ese

límite sobre una base de objeto a objeto, al depender de un sistema de confianza. El encapsulamiento es una potente función del lenguaje Java.

Herencia

En la programación estructurada, es común copiar una estructura, darle un nombre nuevo y agregar o modificar los atributos que hacen que la nueva identidad (por ejemplo, un registro de `Cuenta`) sea diferente de su fuente original. Con el tiempo, este abordaje genera una gran cantidad de códigos duplicados, que pueden crear problemas de mantenimiento.

OOP presenta el concepto de *herencia*, por el cual los objetos especializados, — sin ningún código adicional, — pueden "copiar" los atributos y el comportamiento de los objetos de origen en los que se especializan. Si alguno de esos atributos o comportamientos necesitan modificarse, entonces simplemente modifíquelos temporalmente. Solo modifique lo que necesite modificar para crear objetos especializados. Como ya sabe desde la sección Object-oriented programming concepts, el objeto origen se llama el *padre* y la especialización nueva se llama el *hijo*.

Herencia en funcionamiento

Suponga que está escribiendo una aplicación de recursos humanos y quiere usar el objeto `Person` como base para un objeto nuevo llamado `Employee`. Al ser el hijo de `Person`, el `Employee` tendría todos los atributos de un objeto `Person`, junto con los adicionales, tales como:

- Número de identificación del contribuyente
- Fecha de contratación
- Salario

La herencia hace que sea fácil crear la nueva clase de `Employee` del objeto sin necesidad de copiar manualmente todo el código de `Person` o mantenerlo.

Verá muchos ejemplos de herencia en la programación Java más adelante en el tutorial, especialmente en la Part 2.

Polimorfismo

El polimorfismo es un concepto más difícil de entender que el encapsulamiento o la herencia. Básicamente, significa que los objetos que pertenecen a la misma ramificación de una jerarquía, cuando se envía el mismo mensaje (es decir, cuando se le indica que realice lo mismo), pueden manifestar ese comportamiento de modo diferente.

Para entender cómo el polimorfismo se aplica a un contexto de aplicación de negocio, regrese al ejemplo de `Person`. ¿Recuerda indicarle a `Person` que formatee sus atributos en una `String`? El polimorfismo hace que sea posible para `Person`

representar sus atributos en una variedad de formas, dependiendo del tipo de `Person` que sea.

El polimorfismo es uno de los conceptos más complejos con los que se encontrará en OOP en la plataforma Java y no dentro del ámbito de un tutorial introductorio. Vea Recursos si quiere aprender más acerca del polimorfismo.

Sección 7. Comenzar con el lenguaje Java

El lenguaje Java: no está puramente orientado a objetos

El lenguaje Java le permite crear objetos de primera clase pero no *todo* en el lenguaje es un objeto. Hay dos cualidades que diferencian el lenguaje Java de los lenguajes puramente orientados a objetos como Smalltalk. Primero, el lenguaje Java es una mezcla de objetos y tipos primitivos. Segundo, le permite escribir un código que exponga el funcionamiento interno de un objeto a cualquier otro objeto que lo utilice.

El lenguaje Java sí le da las herramientas necesarias para seguir principios OOP sólidos y producir un código sólido orientado a objetos. Debido a que Java no está puramente orientado a objetos, debe ejercitar alguna disciplina sobre cómo usted escribe un código. — El lenguaje no lo obliga a hacer lo correcto, por lo tanto debe hacerlo usted mismo. (Esta última sección del tutorial, Writing good Java code, proporciona consejos).

Sería imposible introducir toda la sintaxis del lenguaje Java en un solo tutorial. Lo que resta de la Parte 1 se centra en los conceptos básicos del lenguaje, por lo que le da suficiente conocimiento y práctica para escribir programas simples. OOP se trata por completo de los objetos, por lo cual esta sección comienza con dos temas relacionados específicamente con la forma en que el lenguaje Java los maneja: palabras reservadas y la estructura de un objeto Java.

Palabras reservadas

Como cualquier lenguaje de programación, el lenguaje Java designa ciertas palabras que el compilador reconoce como especiales y, como tales, usted no tiene permitido usarlas para nombrar sus construcciones Java. La lista de palabras reservadas es sorprendentemente corta:

- `abstracto`
- `afirmar`
- `booleano`
- `interrupción`
- `byte`
- `caso`
- `capturar`
- `caract.`
- `clase`

- `const.`
- `continuar`
- `predeterminado`
- `hacer`
- `doble`
- `else`
- `enumer.`
- `extiende`
- `final`
- `finalmente`
- `flotante`
- `para`
- `ir a`
- `si`
- `implementa`
- `importar`
- `instancia de`
- `int`
- `interfaz`
- `largo`
- `nativo`
- `nuevo`
- `paquete`
- `privado`
- `protegido`
- `público`
- `retorno`
- `corto`
- `estático`
- `strictfp`
- `súper`
- `conmutador`
- `sincronizado`
- `esto`
- `arrojar`
- `arroja`
- `transitorio`
- `intentar`
- `inválido`
- `volátil`
- `mientras`

Observe que `true`, `false` y `null` técnicamente no son palabras reservadas. Aunque son literales, las incluyo en esta lista porque no puede usarlas para nombrar construcciones Java.

Una ventaja de la programación con un IDE es que puede usar coloreado de sintaxis para palabras reservadas, como verá más adelante en este tutorial.

Estructura de un objeto Java

Recuerde que un objeto es una entidad diferenciada que contiene atributos y comportamiento. Eso significa que tiene un límite nítido y un estado y puede realizar actividades cuando se lo piden correctamente. Cada lenguaje de orientación a objetos tiene reglas sobre cómo definir un objeto.

En el lenguaje Java, los objetos se definen como se demuestra en el Listado 1

Listado 1. Definición de objeto

```
package  packageName;

import  ClassNameToImport;
accessSpecifier  class ClassName {
  accessSpecifier dataType variableName [= initialValue];
  accessSpecifier ClassName([argumentList]) {
    constructorStatement(s)
  }
  accessSpecifier returnType methodName([argumentList]) {
    methodStatement(s)
  }
  // This is a comment
  /* This is a comment too */
  /* This is a
     multiline
     comment */
}
```

El Listado 1 contiene diversos tipos de construcciones, que he diferenciado con formato de fuente. Las construcciones que se muestran en negrita (que encontrará en la lista de palabras reservadas) son literales. En cualquier definición de objeto, deben ser exactamente lo que son aquí. Los nombres que le he dado a las otras construcciones describen los conceptos que representan. Explicaré todas las construcciones en detalle en el resto de esta sección.

Nota: En el Listado 1 y algunos otros ejemplos de códigos en esta sección, los corchetes indican que las construcciones dentro de ellos no se requieren. Los corchetes en sí (a diferencia de `{` y `}`) no son parte de la sintaxis Java.

Comentarios en el código

Observe que el Listado 1 también incluye algunas líneas de comentarios:

```
  // Este es un comentario
  /* Este también es un comentario */
  /* Este es un
     comentario
     múltiple */
```

Casi todos los lenguajes de programación le permiten al programador agregar comentarios para ayudar a documentar el código. La sintaxis Java permite tanto

comentarios de una sola línea como comentarios múltiples. Un comentario de una sola línea debe ocupar una línea, aunque puede usar comentarios de una sola línea adyacentes para formar un bloque. Un comentario de líneas múltiples comienza con `/*`, debe terminar con `*/` y puede distribuirse en cualquier cantidad de líneas.

Aprenderá más sobre los comentarios cuando llegue a la sección Writing good Java code de este tutorial.

Empaquetado de objetos

El lenguaje Java le permite elegir los nombres de sus objetos, tales como `Account`, `Person` o `LizardMan`. En ocasiones, puede que termine usando el mismo nombre para expresar dos conceptos ligeramente diferentes. Esto se llama una *colisión de nombres* y sucede con frecuencia. El lenguaje Java usa *paquetes* para resolver estos conflictos.

Un paquete Java es un mecanismo para proporcionar un espacio de nombres: un área encapsulada en donde los nombres son únicos pero, fuera de esa área, puede que no lo sean. Para identificar una construcción de manera única, debe calificarla totalmente al incluir su espacio de nombres.

Los paquetes también le dan una buena forma para construir aplicaciones más complejas en unidades diferenciadas de funcionalidad.

Definición de paquete

Para definir un paquete, use la palabra clave `paquete` seguida por un nombre de paquete legal y termina con un punto y coma. A menudo, los nombres de los paquetes se separan con puntos y siguen este plan *de facto*:

```
package  orgType.orgName.appName.compName;
```

Esta definición de paquete se divide de la siguiente manera:

- *`orgType`* es el tipo de organización, tales como `com`, `org` o `net`.
- *`orgName`* es el nombre del ámbito de la organización, tales como `makotogroup`, `sun` o `ibm`.
- *`appName`* es el nombre de la aplicación, abreviado.
- *`compName`* es el nombre del componente.

El lenguaje Java no lo obliga a seguir este convenio de paquetes. De hecho, usted no necesita especificar ningún paquete, en cuyo caso todos sus objetos deben tener nombres de clases únicos y residirán en el paquete predeterminado. Como una práctica mejor, recomiendo que defina todas sus clases Java en paquetes. Usted seguirá ese convenio durante este tutorial.

Sentencias de importación

A continuación en la definición de objeto (retomando el Listado 1) se encuentra la *sentencia de importación*. Una sentencia de importación le comunica al compilador Java dónde encontrar las clases a las que usted hace referencia dentro de su código. Cualquier objeto no trivial usa otros objetos para alguna funcionalidad y la sentencia de importación es cómo usted le comunica al compilador Java sobre ellos.

Una sentencia de importación normalmente luce así:

```
import  ClassNameToImport;
```

Especifique la palabra clave `importación` seguida de la clase que quiere importar seguida de un punto y coma. El nombre de la clase debería estar completamente calificado, es decir, debería incluir su paquete.

Para importar todas las clases dentro de un paquete, puede poner `.*` después del nombre del paquete. Por ejemplo, esta sentencia importa cada clase en el paquete `com.makotogroup`:

```
import com.makotogroup.*;
```

Sin embargo, importar todo un paquete puede hacer que su código sea menos legible, por lo tanto recomiendo que importe solo las clases que necesite.

Eclipse simplifica las importaciones

Cuando escribe el código en el editor de Eclipse, puede escribir el nombre de una clase que usted quiera usar, seguido por Ctrl+Shift+O. Eclipse resuelve cuáles son las importaciones que usted necesita y las agrega automáticamente. Si Eclipse encuentra dos clases con el mismo nombre, visualiza un recuadro de diálogo que le pregunta para cuál de las clases quiere agregar importaciones.

Declaración de clase

Para definir un objeto en el lenguaje Java, debe declarar una clase. Considere a una clase como una plantilla para un objeto, como un molde de galletas. La clase define la estructura básica del objeto y, al momento de la ejecución, su aplicación crea una instancia del objeto. La palabra *objeto* a menudo se usa como sinónimo de la palabra *clase*. En sentido estricto, una clase define la estructura de algo de lo cual el objeto es una instancia.

El Listado 1 incluye esta declaración de clase:

```
accessSpecifier  class ClassName {
  accessSpecifier dataType variableName [= initialValue];
  accessSpecifier ClassName([argumentList]) {
    constructorStatement(s)
  }
  accessSpecifier returnType methodName([argumentList]) {
    methodStatement(s)
  }
}
```

Un *accessSpecifier* de una clase podría tener varios valores pero, por lo general, es público. Observará otros valores de *accessSpecifier* pronto.

Convenios de denominación de clases

Puede denominar a las clases prácticamente como quiera pero el convenio es usar *bicapitalización*: comenzar con una letra en mayúscula, usar mayúscula en la primera letra de cada palabra concatenada y dejar todas las otras letras en minúscula. Los nombres de las clases deberían contener solo letras y números. Adherirse a estas guías asegurará que su código sea más accesible para otros desarrolladores que siguen los mismos convenios.

Las clases pueden tener dos tipos de miembros: *variables* y *métodos*.

Variables

Los valores de las variables de una clase dada distinguen cada instancia de esa clase y define su estado. A estos valores a menudo se los denomina *variables de instancia*. Una variable tiene:

- Un *accessSpecifier*
- Un *dataType*
- Un *variableName*
- Opcionalmente, un *initialValue*

Lo posibles valores de *accessSpecifier* son:

> **Variables públicas**
> Nunca es una buena idea usar variables públicas pero, en casos extremadamente raros, podría ser necesario, así que existe la opción. La plataforma Java no restringe sus casos de uso. Por lo tanto, depende de usted ser disciplinado al usar buenos convenios de codificación, incluso si se siente tentado para hacer lo contrario.

- público: Cualquier objeto en cualquier paquete puede ver la variable. (Nunca use este valor).
- protegido: Cualquier objeto definido en el mismo paquete, o una subclase (definida en cualquier paquete), puede ver la variable.
- Ningún especificador (también llamado acceso *amigable* o *privado al paquete*): Solo los objetos cuyas clases se definen en el mismo paquete pueden ver la variable.
- privado: Solo la clase que contiene la variable puede verla

El *dataType* de una variable depende de lo que la variable sea, — podría ser un tipo primitivo u otro tipo de clase (repito, profundizaremos sobre esto más adelante).

El *variableName* depende de usted pero, por convenio, los nombres de las variables usan el convenio de bicapitalización que describí anteriormente, con la excepción de que comienzan con una letra minúscula. (A este estilo a veces se lo llama *lowerCamelCase*).

No se preocupe por el *initialValue* por ahora; solo sepa que puede inicializar una variable de instancia cuando la declara. (De otro modo, el compilador genera una predeterminación por usted que se establecerá cuando se cree una instancia de la clase).

Ejemplo: Definición de clase de `Person`

Antes de seguir con los métodos, aquí hay un ejemplo que resume lo que ha aprendido hasta ahora. El Listado 2 es una definición de clase de `Person`:

Listado 2. Definición de clase básica de `Person`

```
package com.makotogroup.intro;

public class Person {
  private String name;
  private int age;
  private int height;
  private int weight;
  private String eyeColor;
  private String gender;
}
```

La definición de clase básica de `Person` no es muy útil en este momento porque define solo sus atributos (y los privados).

Para ser más interesante, la clase de `Person` necesita comportamiento, — y eso significa métodos.

Métodos

Los métodos de una clase definen su comportamiento. A veces, este comportamiento no es nada más que devolver el valor actual de un atributo. Otras veces, el comportamiento puede ser bastante complejo.

Hay esencialmente dos categorías de métodos: *constructores* y todos los otros métodos, — de los cuales existen muchos tipos. Un método constructor se usa solo para crear una instancia de una clase. Otros tipos de métodos pueden usarse para prácticamente cualquier comportamiento de aplicación.

Al mirar hacia atrás al Listado 1, muestra el modo para definir la estructura de un método, que incluye aspectos como:

- *accessSpecifier*

- *returnType*
- *methodName*
- *argumentList*

La combinación de estos elementos estructurales en la definición de un método se llama su *firma*.

A continuación, verá en más detalle dos tipos de métodos, comenzando con los constructores.

Métodos constructores

Los constructores le permiten especificar cómo crear una instancia de una clase. El Listado 1 muestra la sintaxis de la declaración del constructor en una forma abstracta. Aquí está de nuevo en el Listado 3:

Listado 3. Sintaxis de la declaración del constructor

```
accessSpecifier ClassName([argumentList]) {
  constructorStatement(s)
}
```

Los constructores son opcionales

Si usted no proporciona un constructor, el compilador proporcionará uno por usted, denominado el constructor predeterminado (o *sin argumento*). Si usted proporciona un constructor que no sea un constructor sin argumento (o *no-arg*), el compilador no generará uno por usted.

El *accessSpecifier* de un constructor es el mismo que el de las variables. El nombre del constructor debe coincidir con el nombre de la clase. Por lo tanto, si llama a su clase `Person`, entonces el nombre del constructor también debe ser `Person`.

Para cualquier constructor que no sea el constructor predeterminado, usted pasa una *argumentList*, que es una o más de:

```
argumentType argumentName
```

Los argumentos en una *argumentList* se separan con comas y dos argumentos no pueden tener el mismo nombre. El *argumentType* es un tipo primitivo u otro tipo de clase (lo mismo que sucede con los tipos de variables).

Definición de clase con un constructor

Ahora verá lo que sucede cuando agrega la capacidad de crear un objeto `Person` de dos modos: al usar un constructor sin argumento y al inicializar una lista parcial de atributos.

El Listado 4 muestra cómo crear constructores y también cómo usar la `argumentList`:

Listado 4. Definición de clase de `Person` con un constructor

```
package com.makotogroup.intro;
```

```
public class Person {
  private String name;
  private int age;
  private int height;
  private int weight;
  private String eyeColor;
  private String gender;
  public Person() {
    // Nothing to do...
  }

public Person(String name, int age, int height, String eyeColor, String gender) {
    this.name = name;
    this.age = age;
    this.height = height;
    this.weight = weight;
    this.eyeColor = eyeColor;
    this.gender = gender;
  }
}
```

Observe el uso de la palabra clave `this` al hacer las asignaciones de variables en el Listado 4. Esto es una taquigrafía de Java para "this object" y debe usarse cuando se haga referencia a dos variables con el mismo nombre (como en este caso en el que la edad, por ejemplo, es tanto un parámetro constructor como una variable de clase) y le ayuda al compilador a desambiguar la referencia.

El objeto `Person` se está haciendo más interesante pero necesita más comportamiento. Y para eso, usted necesita más métodos.

Otros métodos

Un constructor es un tipo particular de método con una función particular. De forma similar, muchos otros tipos de métodos desempeñan funciones particulares en los programas Java. La exploración de otros métodos comienza en esta sección y continúa durante todo el tutorial.

En el Listado 1, le mostré cómo declarar un método:

```
accessSpecifier returnType methodName([argumentList]) {
  methodStatement(s)
}
```

Otros métodos se parecen bastante a los constructores, con un par de excepciones. En primer lugar, usted puede denominar a otros métodos cómo usted quiera (aunque, por supuesto, hay reglas). Recomiendo los siguientes convenios:

- Comenzar con una letra minúscula.
- Evitar números, a menos que sean absolutamente necesarios.
- Usar solo caracteres alfabéticos.

En segundo lugar, a diferencia de los constructores, otros métodos tienen un tipo de retorno opcional.

Otros métodos de `Person`

Con esta información básica de su lado, usted puede ver en el Listado 5 lo que sucede cuando agrega algunos métodos más al objeto `Person`. (He omitido los constructores para ser breve).

Listado 5. `Person` con algunos métodos nuevos

```
package com.makotogroup.intro;

public class Person {
  private String name;
  private int age;
  private int height;
  private int weight;
  private String eyeColor;
  private String gender;

  public String getName() { return name; }
  public void setName(String value) { name = value; }
  // Other getter/setter combinations...
}
```

Observe el comentario en el Listado 5 sobre "combinaciones getter/setter". Usted trabajará más con getters y setters más adelante en el tutorial. Por ahora, todo lo que necesita saber es que un *getter* es un método para recuperar el valor de un atributo y un *setter* es un método para modificar ese valor. Solo le he mostrado una combinación getter/setter (para el atributo de `Nombre`) pero usted podría definir más de manera similar.

Observe en el Listado 5 que si un método no devuelve un valor, debe comunicárselo al compilador especificando el tipo de devolución `inválido` en su firma.

Métodos estáticos y de instancia

Hay generalmente dos tipos de métodos (no constructores): los *métodos de instancia* y los *métodos estáticos*. Los métodos de instancia dependen del estado de una instancia de objeto específico por sus comportamientos. Los métodos estáticos también se denominan a veces *métodos de clase* porque sus comportamientos no dependen del estado de ningún objeto en particular. El comportamiento de un método estático sucede al nivel de la clase.

Los métodos estáticos se usan en gran medida por utilidad; puede considerarlos como un modo de tener métodos globales (*à la* C) mientras mantiene el código mismo agrupado con la clase que lo necesita.

Por ejemplo, en todo este tutorial usará la clase de `Logger` JDK para enviar información a la consola. Para crear una instancia de clase de `Logger`, no cree una instancia de clase de `Logger`; en cambio, invoque un método estático llamado `getLogger()`.

La sintaxis para invocar un método estático es diferente de la sintaxis usada para invocar un método para una instancia de objeto. También use el nombre de la clase que contiene el método estático, como se muestra en esta invocación:

```
Logger l = Logger.getLogger("NewLogger");
```

Por lo cual, para invocar un método estático, no necesita una instancia de objeto, solo el nombre de la clase.

Sección 8. Su primer objeto Java

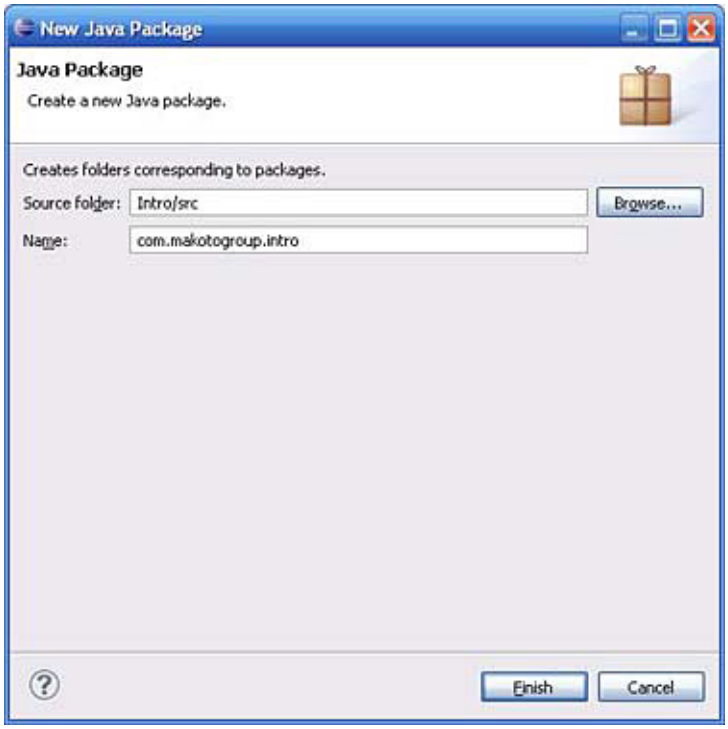
Es tiempo de reunir lo que ha aprendido en las secciones previas y comenzar a escribir algunos códigos. Esta sección lo guía por la declaración de una clase y la adición de variables y métodos a ella para usar el Eclipse Package Explorer. Aprenderá cómo usar la clase de `Logger` para mantener en vista al comportamiento de su aplicación y también cómo usar un método `main()` como un banco de pruebas.

Creación de un paquete

Si todavía no está allí, vaya a la perspectiva Package Explorer en Eclipse. Se lo va a preparar para crear su primera clase Java. El primer paso es crear un lugar para que la clase viva. Los paquetes son construcciones de espacio de nombres pero también se correlacionan convenientemente de forma directa con la estructura del directorio del sistema de archivos.

En lugar de usar el paquete predeterminado (casi siempre una mal idea), creará uno específicamente para el código que estará escribiendo. Haga clic en **File > New > Package** para acceder al asistente del Paquete Java, que se muestra en la Ilustración 4:

Ilustración 4. El asistente del Paquete Java de Eclipse



Escriba `com.makotogroup.intro` en el recuadro de texto del Nombre y haga clic en **Finish**. Verá el nuevo paquete creado en el Package Explorer.

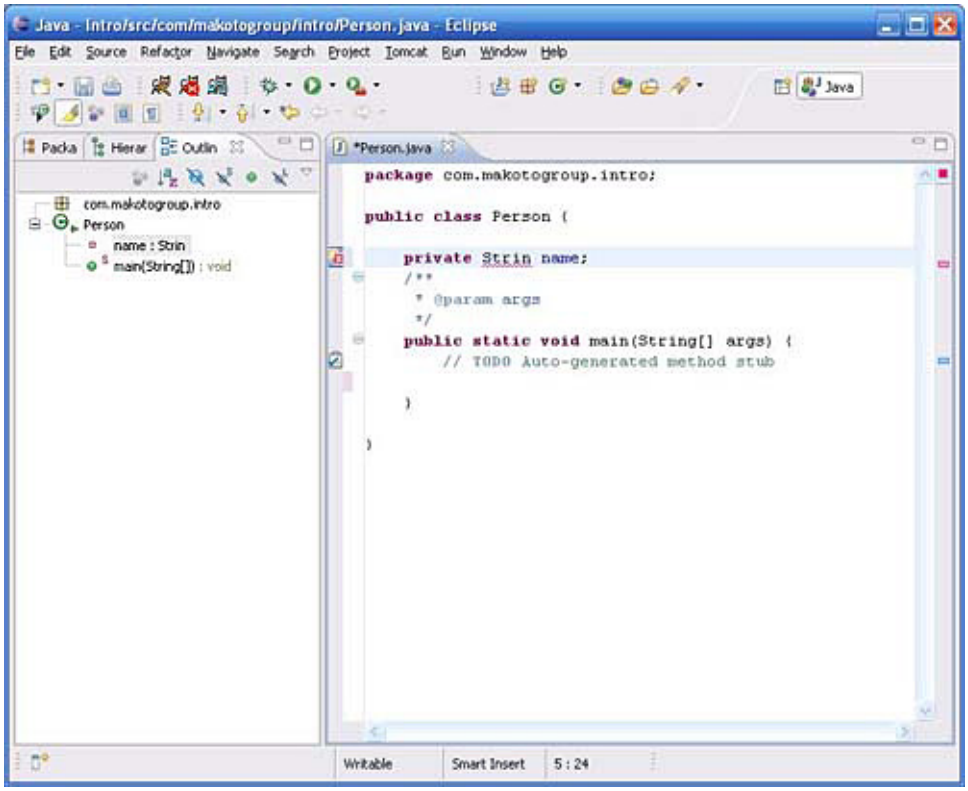
Declaración de la clase

Hay más de un modo para crear una clase desde el Package Explorer pero el modo más fácil es hacer clic derecho en el paquete que acaba de crear y elegir **New > Class...**. Verá el recuadro de diálogo New Class (Clase nueva).

En el recuadro de texto de **Name** escriba `Person`. Bajo **¿Qué resguardos de métodos le gustaría crear?**, marque `public static void main(String[] args)`. (Pronto verá por qué). Luego, haga clic en **Finish**.

La clase nueva aparece en su ventana de edición. Recomiendo cerrar algunas de las vistas de su aspecto predeterminado para que sea más fácil ver su código de origen, como se muestra en la Ilustración 5:

Ilustración 5. Un espacio de trabajo bien ordenado



Eclipse genera una clase de shell para usted e incluye la sentencia del paquete en la parte superior, junto con el método `main()` que usted pidió y los comentarios que ve. Ahora, solo necesita precisar la clase. Puede configurar cómo Eclipse genera clases nuevas por medio de **Window > Preferences > Java > Code Style > Code Templates**. Para simplificar, usted optará por la generación de código simple de instalar de Eclipse.

En la Ilustración 5, observe el asterisco (*) junto al nuevo archivo de código de origen, que indica que he hecho una modificación. Y observe que el código no está guardado. Luego, observe que he cometido un error cuando declaré el atribut `Name`: Declaré que el tipo de `Name` era `Strin`. El compilador no pudo encontrar una referencia para tal clase y la distinguió como un error de compilación (esa es la línea roja ondulada debajo de `Strin`). Por supuesto, puedo corregir mi error al agregar una `g` al final de `Strin`. Esto es solo una pequeña demostración del poder de un IDE sobre el uso de herramientas de líneas de comandos para el desarrollo de software.

Adición de variables de clases

En el Listado 4, usted comienza a precisar la clase de `Person` pero no expliqué mucho sobre la sintaxis. Ahora formalmente definiré cómo agregar variables de clases.

Recuerde que una variable tiene un *`accessSpecifier`*, un *`dataType`*, un *`variableName`* y, opcionalmente, un *`initialValue`*. Anteriormente, observó brevemente cómo definir

el *accessSpecifier* y *variableName*. Ahora verá el *dataType* que una variable puede tener.

Un *dataType* puede ser un tipo primitivo o una referencia a otro objeto. Por ejemplo, observe que `Age` es un `int` (un tipo primitivo) y `Name` es una `String` (un objeto). El JDK viene empaquetado lleno de clases útiles como `java.lang.String` y aquellos en el paquete `java.lang` no necesitan ser importados (una cortesía taquigráfica del compilador Java). Pero ya sea que el *dataType* sea una clase JDK como `String` o una clase definida por el usuario, la sintaxis es esencialmente la misma.

La Tabla 1 muestra los ocho tipos de datos primitivos que es probable que usted vea regularmente, incluidos los valores predeterminados que los primitivos adquieren si usted no inicializa explícitamente el valor de una variable miembro:

Tabla 1. Tipos de datos primitivos

Tipo	Tamaño	Valor predeterminado	Rango de valores
`booleano`	n/d	falso	verdadero o falso
`byte`	8 bits	0	-128 a 127
`caract.`	16 bits	(sin firmar)	\u0000' \u0000' a \uffff' o 0 a 65535
`corto`	16 bits	0	-32768 a 32767
`int`	32 bits	0	-2147483648 a 2147483647
`largo`	64 bits	0	-9223372036854775808 a 9223372036854775807
`flotante`	32 bits	0,0	1.17549435e-38 a 3.4028235e+38
`doble`	64 bits	0,0	4.9e-324 a 1.7976931348623157e+308

Registro incorporado

Antes de avanzar más en la codificación, necesita saber cómo sus programas le dicen lo que están haciendo.

La plataforma Java incluye el paquete `java.util.logging`, un mecanismo de registro incorporado para juntar información de programa en una forma legible. Los registradores son entidades con nombres que usted crea por medio de una llamada de método estático a la clase `Logger`, como la siguiente:

```
import java.util.logging.Logger;
//. . .
Logger l = Logger.getLogger(getClass().getName());
```

Cuando se llama al método `getLogger()`, usted pase una `String`. Por ahora, solo acostúmbrese a pasar al nombre de la clase el código en el que su escritura se encuentra. Desde cualquier método regular (es decir, no estático), el código anterior siempre hará referencia al nombre de la clase y lo pasará al `Logger`.

Si usted está haciendo una llamada de `Logger` dentro de un método estático, solo haga referencia al nombre de la clase en la que se encuentre:

```
Logger l = Logger.getLogger(Person.class.getName());
```

En este ejemplo, el código en el que usted se encuentra es la clase de `Person`, por lo tanto haga referencia a un literal especial denominado `clase` que recupera el objeto `Class` (profundizaremos sobre esto más adelante) y obtiene su atributo `Name`.

La sección Writing good Java code de este tutorial incluye un consejo sobre cómo *no* hacer un registro.

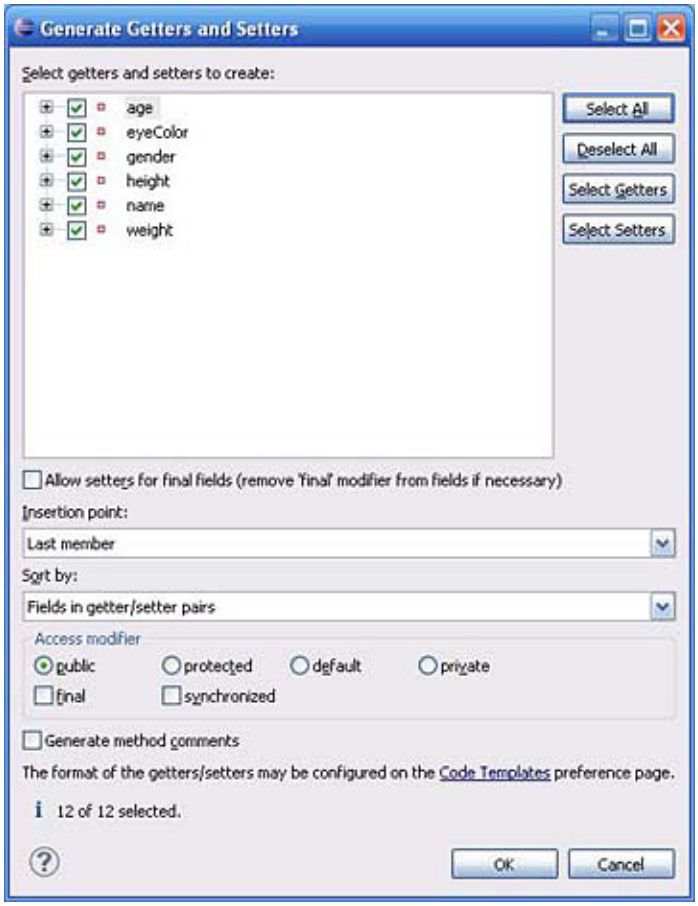
Utilización de `main()` como un banco de pruebas

`main()` es un método especial que usted puede incluir en cualquier clase para que el JRE pueda ejecutar su código. No se requiere que una clase tenga un método `main()`, — de hecho, la mayoría nunca lo tendrá, — y una clase puede tener, como mucho, un método `main()`.

`main()` es un método práctico para tener porque le da un banco de pruebas rápido para la clase. En el desarrollo empresarial, usted usaría bibliotecas de pruebas pero, a los propósitos de este tutorial, usará `main()` como su banco de pruebas.

Vaya al editor de código de origen de Eclipse para `Person` y agregue un código para hacer que se parezca al Listado 4. Eclipse tiene un práctico generador de códigos para generar getters y setters (entre otros). Para probarlo, ponga el cursor del ratón en la definición de clase de `Person` (es decir, en la palabra `Person` en la definición de clase) y vaya a **Source > Generate Getters and Setters...**. Cuando el recuadro de diálogo se abra, haga clic en **Select All**, como se muestra en la Ilustración 6:

Ilustración 6. Eclipse genera getters y setters



Para el punto de inserción, elija **Last member** y haga clic en **OK**. Observe que los getters y setters aparecen luego del método `main()`.

Hay más sobre `main()`

Ahora agregará algunos códigos a `main()` para que le permita crear una instancia de una `Person`, establecer algunos atributos y luego imprimirlos en la consola.

Comience agregando un constructor a `Person`. Escriba el código en el Listado 6 en su ventana de origen debajo de la parte superior de la definición de clase (la línea inmediatamente debajo de `public class Person ()`):

Listado 6. Constructor de `Person`

```
public Person(String name, int age, int height, int weight, String eyeColor,
                String gender) {
    this.name = name;
    this.age = age;
    this.height = height;
    this.weight = weight;
    this.eyeColor = eyeColor;
    this.gender = gender;
  }
```

Asegúrese de no tener líneas onduladas que indiquen errores de compilación.

Luego, vaya al método `main()` y haga que se parezca al Listado 7:

Listado 7. El método `main()`

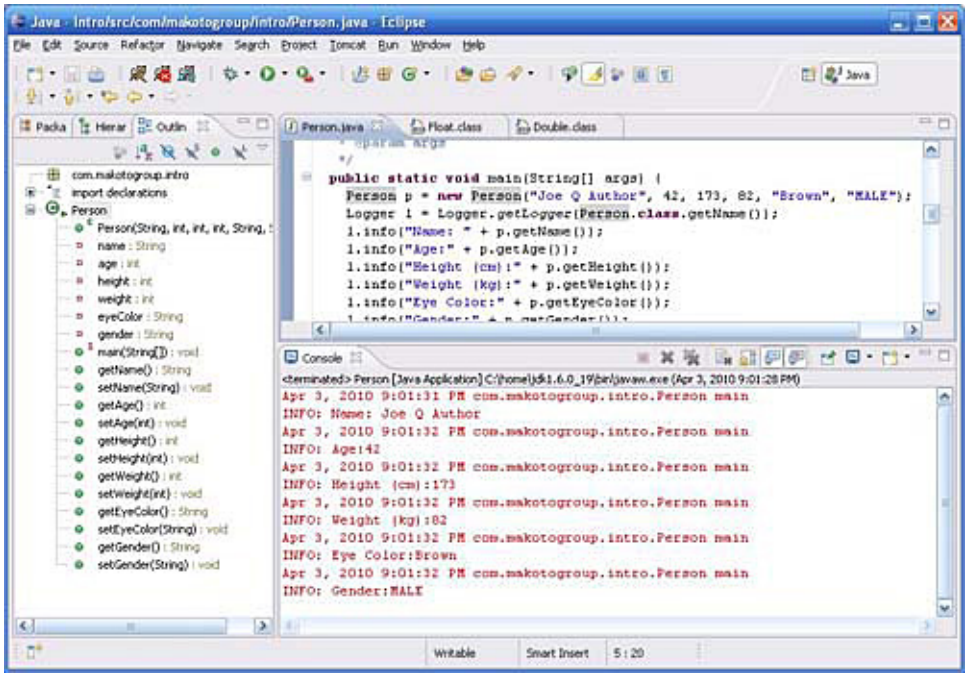
```
public static void main(String[] args) {
  Person p = new Person("Joe Q Author", 42, 173, 82, "Brown", "MALE");
  Logger l = Logger.getLogger(Person.class.getName());
  l.info("Name: " + p.getName());
  l.info("Age:" + p.getAge());
  l.info("Height (cm):" + p.getHeight());
  l.info("Weight (kg):" + p.getWeight());
  l.info("Eye Color:" + p.getEyeColor());
  l.info("Gender:" + p.getGender());
}
```

No se preocupe por la clase de `Logger` por ahora. Solo ingrese el código como lo ve en el Listado 7. Ahora está listo para ejecutar su primer programa Java.

Ejecución de código en Eclipse

Para ejecutar una aplicación Java desde dentro de Eclipse, seleccione la clase que quiere ejecutar, que debe tener un método `main()`. Actualmente, usted tiene solo una clase, — y sí tiene un método `main()`, — por lo tanto seleccione `Person`, luego haga clic en el icono **Run** (que es verde y tiene una pequeña flecha triangular que apunta a la derecha). Cuando se lo pida, seleccione ejecutar `Person` como una aplicación Java, luego siéntese y obsérvelo funcionar. Debería ver algo parecido a la captura de pantalla de la Ilustración 7:

Ilustración 7. Vea ejecución de `Person`



Observe que la vista de Consola se abre automáticamente y muestra la salida de `Logger`. También he seleccionado la vista de Esquema en el panel izquierdo, que revela la estructura básica de la clase de `Person` en una vista rápida.

Sección 9. Adición de comportamiento a un objeto Java

`Person` se ve bien hasta el momento pero le podría servir algún comportamiento adicional para que sea más interesante. Como ya ha aprendido, crear un comportamiento significa agregar métodos. Esta sección ve más de cerca los *métodos accessor,* — a saber, los getters y setters que usted ya ha visto en acción. También aprenderá la sintaxis para llamar métodos.

Métodos accessor

Para encapsular los datos de una clase desde otros objetos, declare que sus variables sean `privadas` y luego proporcione métodos accessor. Como sabe, un getter es un método accessor para recuperar el valor de un atributo; un setter es un método accessor para modificar ese valor. La denominación de accessor sigue un convenio estricto conocido como el patrón JavaBeans, por el que cualquier atributo `Foo` tiene un getter llamado `getFoo()` y un setter llamado `setFoo()`.

El patrón JavaBeans es tan común que su soporte está construido justo en el IDE Eclipse. Incluso, usted ya lo ha visto en acción, — cuando generó getters y setters para `Person` en la sección anterior.

Los accessor siguen estas guías:

- El atributo en sí siempre se declara con acceso `privado`.
- El especificador de acceso para los getters y setters es `público`.
- Los getters no toman ningún parámetro y devuelven un valor cuyo tipo es el mismo que el atributo al que accede.
- Los setters solo toman un parámetro, del tipo del atributo, y no devuelven un valor.

Declaración de accessor

El modo más fácil, por mucho, de declarar accessor es dejar que Eclipse lo haga por usted, como se mostró en la Ilustración 6. Pero también debería saber cómo codificar manualmente un par de getters y setters Suponga que usted tiene un atributo, `Foo`, cuyo tipo es `java.lang.String`. Una declaración completa para aquel (siguiendo las guías de accessor) sería:

```
private String foo;
public String getFoo() {
  return foo;
}
public void setFoo(String value) {
  foo = value;
}
```

Tal vez usted observe enseguida que el valor del parámetro que pasó al setter se denomina de modo diferente que si lo hubiera generado Eclipse. Este es mi propio convenio y uno que recomiendo a otros desarrolladores. En la rara ocasión que yo codifique manualmente un setter, siempre uso el nombre `value` como el valor de parámetro para el setter. Este captador de miradas me recuerda que he codificado manualmente el setter. Debido a que normalmente permito que Eclipse genere los getters y setters por mí, cuando no lo hago es por una buena razón. Usar `value` como valor de parámetro del setter me recuerda que este setter es especial. (Los comentarios de código también lo podrían hacer).

Llamado de métodos

Invocar, o llamar, métodos es fácil. Usted vio en el Listado 7 cómo invocar los diversos getters de `Person` para devolver sus valores. Ahora formalizaré el mecanismo de realizar llamadas de métodos.

La invocación de un método con y sin parámetros

Para invocar un método para un objeto, necesita una referencia a ese objeto. La sintaxis de la invocación de métodos comprende la referencia del objeto, un punto literal, el nombre del método y cualquier parámetro que necesite pasarse:

```
objectReference.someMethod();
objectReference.someOtherMethod(parameter);
```

Aquí hay una invocación de método sin parámetros:

```
Person p = /*obtain somehow */;
p.getName();
```

Y aquí hay una invocación de método con parámetros (con acceso al atributo `Name` de `Person`):

```
Person p = new Person("Joe Q Author", 42, 173, 82, "Brown", "MALE");
```

Recuerde que los constructores también son métodos. Y puede separar los parámetros con espacios y saltos de línea. Al compilador Java no le interesa. Las siguientes dos invocaciones de métodos son idénticas:

```
new Person("Joe Q Author", 42, 173, 82, "Brown", "MALE");
```

```
new Person("Joe Q Author",
        42,
        173,
        82,
        "Brown",
        "MALE");
```

Invocación de método anidada

Las invocaciones de métodos también pueden ser anidadas:

```
Logger l = Logger.getLogger(Person.class.getName());
l.info("Name: " + p.getName());
```

Aquí usted está pasando el valor de retorno de `Person.class.getName()` al método `getLogger()`. Recuerde que la llamada del método `getLogger()` es una llamada de método estático, por lo cual su sintaxis difiere un poco. (Usted no necesita una referencia de `Logger` para hacer la invocación; en lugar de eso, solo use el nombre de la clase misma como el lado izquierdo de la invocación).

Eso es todo realmente sobre la invocación de métodos.

Sección 10. Cadenas y operadores

El tutorial ha presentado hasta el momento varias variables de tipo `String` pero sin mucha explicación. Aprenda más acerca de las cadenas en esta sección y también descubra cuándo y cómo usar los operadores.

Cadenas

El manejo de cadenas en C es un trabajo intensivo porque son matrices de finalización nula de caracteres de 8 bits que usted tiene que manipular. En el lenguaje Java, las cadenas son objetos de primera clase de tipo `String`, con métodos que le ayudan a manejarlas. (Lo más cerca que el código Java llega al mundo C con respecto a las cadenas es el tipo de datos primitivos `char`, que puede tener un solo carácter Unicode, por ejemplo *a*).

Ya ha visto cómo crear una instancia de un objeto `String` y establecer su valor (en el Listado 5) pero existen varias otras formas de hacer eso. Aquí hay un par de formas para crear una instancia `String` con un valor de `hello`:

```
String greeting = "hello";
```

```
greeting = new String("hello");
```

Debido a que las `String`s son objetos de primera clase en el lenguaje Java, puede usar `new` para crear una instancia de ellas. Establecer una variable de tipo `String` tiene el mismo resultado porque el lenguaje Java crea un objeto `String` para tener el literal, luego asigna ese objeto a la variable de instancia.

Concatenación de cadenas

Puede hacer muchas cosas con `String` y la clase tiene muchos métodos útiles. Sin siquiera usar un método, ya ha hecho algo interesante con dos `String`s al concatenarlas o combinarlas:

```
l.info("Name: " + p.getName());
```

El signo más (+) es taquigráfico para concatenar `String`s en el lenguaje Java. (Existe una sanción de rendimiento por hacer este tipo de concatenación dentro de un bucle pero por ahora no tiene que preocuparse por eso).

Ejemplo de concatenación

Intentemos concatenar `String`s dentro de la clase `Person`. En este momento, tiene una variable de instancia de `name` pero sería bueno tener un `firstName` y un `lastName`. Entonces podría concatenarlas cuando otro objeto solicite el nombre completo de `Person`.

Lo primero que necesita hacer es agregar las nuevas variables de instancias (en la misma ubicación en el código de origen donde se define actualmente el `name`):

```
//private String name;
private String firstName;
private String lastName;
```

Ya no necesita el `name`; lo ha reemplazado con el `firstName` y el `lastName`.

Encadenamiento de llamadas de métodos

Ahora puede generar getters y setters para el `firstName` y el `lastName` (como se muestra en la Ilustración 6), eliminar el método `setName()` y cambiar el `getName()` para que se vea así:

```
public String getName() {
  return firstName.concat(" ").concat(lastName);
}
```

Este código ilustra el *encadenamiento* de las llamadas de métodos. Esta es una técnica comúnmente usada con objetos inmutables como `String`, donde una modificación a un objeto inmutable siempre devuelve la modificación (pero no cambia el original). Entonces, usted opera sobre el valor cambiado de retorno.

Operadores

Como puede esperar, el lenguaje Java puede calcular y ya ha visto como asignar variables. Ahora, le daré un breve vistazo a algunos de los operadores del lenguaje Java que necesitará mientras sus aptitudes mejoran. El lenguaje Java usa dos tipos de operadores:

- *Unario*: Solo se necesita un operando.
- *Binario*: Se necesitan dos operandos.

Los operadores aritméticos del lenguaje Java se resumen en la Tabla 2:

Tabla 2. Operadores aritméticos del lenguaje Java

Operador	Uso	Descripción

`+`	`a + b`	Suma `a` y `b`
`+`	`+a`	Potencia `a` a `int` si es un `byte`, `short`, o `char`
`-`	`a - b`	Resta `b` de `a`
`-`	`-a`	Aritméticamente niega `a`
`*`	`a * b`	Multiplica `a` y `b`
`/`	`a / b`	Divide `a` por `b`
`%`	`a % b`	Devuelve el resto de la división de `a` por `b` (el operador de módulo)
`++`	`a++`	Incrementa `a` por 1; calcula el valor de `a` antes de incrementarlo
`++`	`++a`	Incrementa `a` por 1; calcula el valor de `a` después de incrementarlo
`--`	`a--`	Disminuye `a` por 1; calcula el valor de `a` antes de disminuirlo
`--`	`--a`	Disminuye `a` por 1; calcula el valor de `a` después de disminuirlo
`+=`	`a += b`	Taquigrafía para `a = a + b`
`-=`	`a -= b`	Taquigrafía para `a = a - b`
`*=`	`a *= b`	Taquigrafía para `a = a * b`
`%=`	`a %= b`	Taquigrafía para `a = a % b`

Operadores adicionales

Además de los operadores en la Tabla 2, ha visto varios otros símbolos que se llaman operadores en el lenguaje Java. Por ejemplo:

- El punto (`.`), que califica los nombres de los paquetes e invoca métodos.
- Los paréntesis (`()`), que delimitan una lista separada por comas de parámetros para un método.
- `new`, que (cuando le sigue un nombre de constructor) crea una instancia de un objeto.

La sintaxis del lenguaje Java también incluye una cantidad de operadores que se usan específicamente para programación condicional, es decir, programas que responden de forma diferente en base a una entrada diferente. Los verá en la siguiente sección.

Sección 11. Operadores condicionales y sentencias de control

En esta sección, aprenderá acerca de las diversas sentencias y operadores que usará para comunicarle a sus programas Java cómo quiere que actúen en base a una entrada diferente.

Operadores relacionales y condicionales

El lenguaje Java le da operadores y sentencias de control que le permiten tomar decisiones en su código. Muy a menudo, una decisión en el código comienza con una *expresión booleana* (es decir, una que evalúa ya sea verdadera o falsa). Tales expresiones usan *operadores relacionales*, que comparan un operando o una expresión con otra, y *operadores condicionales*.

La Tabla 3 enumera los operadores relacionales y condicionales del lenguaje Java:

Tabla 3. Operadores relacionales y condicionales

Operador	Uso	Retorna verdadero si...
`>`	`a > b`	`a` es mayor que `b`
`>=`	`a >= b`	`a` es mayor que o igual a `b`
`<`	`a < b`	`a` es menor que `b`
`<=`	`a <= b`	`a` es menor que o igual a `b`
`==`	`a == b`	`a` es igual a `b`
`!=`	`a != b`	`a` no es igual a `b`
`&&`	`a && b`	Ambos `a` y `b` son verdaderos, evalúa condicionalmente a `b` (si `a` es falso, `b` no se evalúa)
`||`	`a || b`	`a` o `b` es verdadero, evalúa condicionalmente a `b` (si `a` es verdadero, `b` no se evalúa)
`!`	`!a`	`a` es falso
`&`	`a & b`	Ambos `a` y `b` son verdaderos, siempre evalúa a `b`
`|`	`a | b`	`a` o `b` es verdadero, siempre evalúa a `b`
`^`	`a ^ b`	`a` y `b` son diferentes

La sentencia `if`

Ahora que tiene un grupo de operadores, es momento de usarlos. Este código muestra lo que sucede cuando agrega algo de lógica al accessor `getHeight()` del objeto `Person`:

```
public int getHeight() {
  int ret = height;
  // If locale of the machine this code is running on is U.S.,
  if (Locale.getDefault().equals(Locale.US))
    ret /= 2.54;// convert from cm to inches
  return ret;
}
```

Si el entorno local actual está en los Estados Unidos (donde no se usa el sistema métrico), entonces tal vez tenga sentido convertir el valor interno de altura (en

centímetros) a pulgadas. Este ejemplo ilustra el uso de la sentencia `if`, que evalúa una expresión booleana en paréntesis. Si esa expresión evalúa como verdadera, ejecuta la siguiente sentencia.

En este caso, solo necesita ejecutar una sentencia si el `Locale` de la máquina en la que se ejecuta el código es `Locale.US`. Si necesita ejecutar más de una sentencia, puede usar llaves para formar una *sentencia compuesta*. Una sentencia compuesta agrupa muchas sentencias en una, — y las sentencias compuestas también pueden contener otras sentencias compuestas.

Ámbito variable

Cada variable en una aplicación Java tiene un *ámbito*, o espacio de nombres localizado, al cual usted puede acceder por nombre dentro del código. Fuera de ese espacio, la variable está *fuera de ámbito* y usted obtendrá un error de compilación si intenta acceder a ella. Los niveles de ámbitos en el lenguaje Java se definen de acuerdo a dónde se declare una variable, como se muestra en el Listado 8:

Listado 8. Ámbito variable

```
public class SomeClass {

  private String someClassVariable;
  public void someMethod(String someParameter) {
    String someLocalVariable = "Hello";
    if (true) {
      String someOtherLocalVariable = "Howdy";
    }
    someClassVariable = someParameter; // legal
    someLocalVariable = someClassVariable; // also legal
    someOtherLocalVariable = someLocalVariable;// Variable out of scope!
  }
  public void someOtherMethod() {
    someLocalVariable = "Hello there";// That variable is out of scope!
  }
}
```

Dentro de `SomeClass`, `someClassVariable` es accesible por medio de todos los métodos de instancia (es decir, no estáticos). Dentro de `someMethod`, `someParameter` es visible pero, fuera de ese método, no lo es y lo mismo sucede para `someLocalVariable`. Dentro del bloque `if`, `someOtherLocalVariable` se declara y, fuera de ese bloque `if`, está fuera de ámbito.

El ámbito tiene muchas reglas pero el Listado 8 muestra las más comunes. Tómese algunos minutos para familiarizarse con ellas.

La sentencia `else`

Hay momentos en el flujo de control de un programa en los que usted quiere participar solo si una expresión particular falla al evaluar como verdadero. En ese momento es cuando `else` resulta de ayuda:

```
public int getHeight() {
  int ret;
  if (gender.equals("MALE"))
    ret = height + 2;
  else {
    ret = height;
    Logger.getLogger("Person").info("Being honest about height...");
  }
  return ret;
}
```

La sentencia `else` funciona del mismo modo que `if` en el sentido que ejecuta solo la siguiente sentencia que encuentra. En este caso, dos sentencias se agrupan en una sentencia compuesta (observe las llaves), que luego el programa ejecuta.

También puede usar `else` para realizar una verificación `if` adicional, del siguiente modo:

```
if (conditional) {
  // Block 1
} else if (conditional2) {
  // Block 2
} else if (conditional3) {
  // Block 3
} else {

  // Block 4
} // End
```

Si *`conditional`* evalúa como verdadero, entonces el *`Block 1`* se ejecuta y el programa salta a la siguiente sentencia luego de la llave final (lo que se indica con `// End`). Si *`conditional`* *no* evalúa como verdadero, entonces se evalúa *`conditional2`*. Si es verdadero, entonces el *`Block 2`* se ejecuta y el programa salta a la siguiente sentencia luego de la llave final. Si *`conditional2`* no es verdadero, entonces el programa sigue con *`conditional3`* y así sucesivamente. Solo si las tres condicionales fallan, se ejecutaría el *`Block 4`*.

El operador ternario

El lenguaje Java proporciona un operador práctico para hacer simples verificaciones de sentencias `if`/`else`. Su sintaxis es:

```
(conditional) ? statementIfTrue : statementIfFalse;
```

Si *`conditional`* evalúa como verdadero, entonces se ejecuta *`statementIfTrue`*; caso contrario, se ejecuta *`statementIfFalse`*. Las sentencias compuestas no se permiten para ninguna de las sentencias.

El operador ternario es de ayuda cuando usted sabe que necesitará ejecutar una sentencia como el resultado de la condicional que evalúa como verdadero, y otra si no lo hace. Los operadores ternarios a menudo se usan para inicializar una variable (como un valor de retorno), como de la siguiente manera:

```
public int getHeight() {
  return (gender.equals("MALE")) ? (height + 2) : height;
}
```

Los paréntesis que siguen luego del signo de interrogación anterior no se requieren estrictamente pero sí hacen que el código sea más legible.

Sección 12. Bucles

Además de poder aplicar condiciones a sus programas y ver diferentes resultados en base a diversos escenarios `if`/`then`, a veces quiere que su código solo haga lo mismo una y otra vez hasta que se haga el trabajo. En esta sección, aprenda acerca de dos construcciones usadas para iterar el código o ejecutarlo más de una vez. Bucles `for` y bucles `while` loops.

¿Qué es un bucle?

Un *bucle* es una construcción de programación que se ejecuta repetidamente mientras se cumple alguna condición (o conjunto de condiciones). Por ejemplo, puede pedirle a un programa que lea todos los registros hasta el final de un archivo o que realice un bucle por todos los elementos de una matriz, procesando cada uno. (Aprenderá acerca de las matrices en la sección Java Collections de este tutorial).

Bucles `for`

La construcción de bucle básico en el lenguaje Java es la sentencia `for`, que le permite iterar un rango de valores para determinar cuántas veces ejecutar un bucle. La sintaxis abstracta para un bucle `for` es:

```
for (initialization; loopWhileTrue; executeAtBottomOfEachLoop) {
  statementsToExecute
}
```

Al comienzo del bucle, se ejecuta la sentencia de inicialización (las sentencias de inicialización múltiples se pueden separar con comas). Mientras *`loopWhileTrue`* (una expresión condicional de Java que debe evaluar ya sea como verdadero o falso) sea verdadero, el bucle se ejecutará. Al final del bucle, se ejecuta *`executeAtBottomOfEachLoop`*.

Ejemplo de un bucle `for`

Si usted quisiera cambiar a un método `main()` para que se ejecute tres veces, podría usar un bucle `for`, como se muestra en el Listado 9:

Listado 9. Un bucle `for`

```
public static void main(String[] args) {
  Logger l = Logger.getLogger(Person.class.getName());
  for (int aa = 0; aa < 3; aa++) {
    Person p = new Person("Joe Q Author", 42, 173, 82, "Brown", "MALE");
    l.info("Loop executing iteration# " + aa);
    l.info("Name: " + p.getName());
    l.info("Age:" + p.getAge());
    l.info("Height (cm):" + p.getHeight());
    l.info("Weight (kg):" + p.getWeight());
    l.info("Eye Color:" + p.getEyeColor());
    l.info("Gender:" + p.getGender());
  }
}
```

La variable local `aa` se inicializa en cero al comienzo del listado. Esta sentencia se ejecuta solo una vez, cuando se inicializa el bucle. Luego, el bucle continúa tres veces y cada vez se incrementa `aa` en uno.

Como verá más adelante, una sintaxis alternativa de bucle `for` está disponible para realizar un bucle por las construcciones que implementan la interfaz `Iterable` (tales como matrices y otras clases de programas de utilidad Java). Por ahora, solo observe el uso de la sintaxis del bucle `for` en el Listado 9.

Bucles `while`

La sintaxis para un bucle `while` es:

```
while (loopWhileTrue) {
  statementsToExecute
}
```

Como puede sospechar, `while` *`loopWhileTrue`* evalúa como verdadero, por lo tanto el bucle se ejecutará. En la parte superior de cada iteración (es decir, antes de que se ejecute cualquier sentencia), se evalúa la condición. Si es verdadero, el bucle se ejecuta. Por lo que es posible que un bucle `while` nunca se ejecute si su expresión condicional no es verdadera por lo menos una vez.

Observe de nuevo el bucle `for` en el Listado 9. Por comparación, el Listado 10 lo codifica usando un bucle `while`:

Listado 10. Un bucle `while`

```
public static void main(String[] args) {
  Logger l = Logger.getLogger(Person.class.getName());
  int aa = 0;
  while (aa < 3) {
    Person p = new Person("Joe Q Author", 42, 173, 82, "Brown", "MALE");
    l.info("Loop executing iteration# " + aa);
    l.info("Name: " + p.getName());
    l.info("Age:" + p.getAge());
    l.info("Height (cm):" + p.getHeight());
    l.info("Weight (kg):" + p.getWeight());
    l.info("Eye Color:" + p.getEyeColor());
    l.info("Gender:" + p.getGender());
    aa++;
  }
```

Como puede ver, un bucle `while` requiere un poco más de mantenimiento que un bucle `for`. Usted debe inicializar la variable `aa` y también recordar incrementarla al final del bucle.

Bucles `do...while`

Si usted quiere un bucle que se ejecute siempre una vez y luego verifique su expresión condicional, pruebe usar un bucle `do...while`, como se muestra en el Listado 11:

Listado 11. Un bucle `do...while`

```
int aa = 0;
do {
  Person p = new Person("Joe Q Author", 42, 173, 82, "Brown", "MALE");
  l.info("Loop executing iteration# " + aa);
  l.info("Name: " + p.getName());
  l.info("Age:" + p.getAge());
  l.info("Height (cm):" + p.getHeight());
  l.info("Weight (kg):" + p.getWeight());
  l.info("Eye Color:" + p.getEyeColor());
  l.info("Gender:" + p.getGender());
  aa++;
} while (aa < 3);
```

La expresión condicional (`aa < 3`) no se verifica hasta el final del bucle.

Ramificación del bucle

Hay momentos en los que necesita retirarse de un bucle antes de que la expresión condicional evalúe como falso. Esto podría suceder si usted estuviera buscando una matriz de `String`s para un valor particular y, una vez que lo encontrara, no le importara los otros elementos de la matriz. Para aquellos momentos en que usted solo quiere retirarse, el lenguaje Java proporciona la sentencia `break`, como se muestra en el Listado 12:

Listado 12. Una sentencia `break`

```
public static void main(String[] args) {
  Logger l = Logger.getLogger(Person.class.getName());
  int aa = 0;
  while (aa < 3) {
    if (aa == 1)
      break;
    Person p = new Person("Joe Q Author", 42, 173, 82, "Brown", "MALE");
    l.info("Loop executing iteration# " + aa);
    l.info("Name: " + p.getName());
    l.info("Age:" + p.getAge());
    l.info("Height (cm):" + p.getHeight());
    l.info("Weight (kg):" + p.getWeight());
    l.info("Eye Color:" + p.getEyeColor());
    l.info("Gender:" + p.getGender());
    aa++;
  }
```

La sentencia `break` lo lleva a la siguiente sentencia ejecutable fuera del bucle en el que se ubica.

Continuación del bucle

En el ejemplo (simplista) del Listado 12, usted solo quiere ejecutar el bucle una vez y retirarse. También puede saltar una sola iteración de un bucle pero continúa ejecutando el bucle. Para eso, necesita la sentencia `continue`, que se muestra en el Listado 13:

Listado 13. Una sentencia `continue`

```
public static void main(String[] args) {
Logger l = Logger.getLogger(Person.class.getName());
int aa = 0;
while (aa < 3) {
  if (aa == 1)
    continue;
  else
    aa++;
  Person p = new Person("Joe Q Author", 42, 173, 82, "Brown", "MALE");
  l.info("Loop executing iteration# " + aa);
  l.info("Name: " + p.getName());
  l.info("Age:" + p.getAge());
  l.info("Height (cm):" + p.getHeight());
  l.info("Weight (kg):" + p.getWeight());
  l.info("Eye Color:" + p.getEyeColor());
  l.info("Gender:" + p.getGender());
}
```

En el Listado 13, usted salta la segunda iteración de un bucle pero continúa con la tercera. `continue` es de ayuda cuando está, digamos, procesando registros y se encuentra con un registro que definitivamente no quiere procesar. Solo salte ese registro y siga con el siguiente.

Sección 13. Colecciones Java

La mayoría de las aplicaciones del mundo real lidian con colecciones de elementos: archivos, variables, registros de archivos, conjuntos de resultados de bases de datos, entre otros. El lenguaje Java tiene una sofisticada Infraestructura de colecciones que le permite crear y manejar colecciones de objetos de diversos tipos. Esta sección no le enseñará todo acerca de las Colecciones Java pero le presentará las clases de colecciones más comúnmente usadas y le hará comenzar a usarlas.

Matrices

La mayoría de los lenguajes de programación incluyen el concepto de una *matriz* para tener una colección de elementos y el lenguaje Java no es ninguna excepción. Una matriz no es nada más que una colección de elementos del mismo tipo.

> **Nota**: Los corchetes en los ejemplos de códigos de esta sección son parte de la sintaxis requerida para las Colecciones Java, *no* indicadores de elementos opcionales.

Puede declarar una matriz en una de dos formas:

- Crearlo con un cierto tamaño, que se fija por la vida de la matriz.
- Crearlo con un cierto conjunto de valores iniciales. El tamaño de este conjunto determina el tamaño de la matriz, — será exacta y suficientemente grande para tener todos esos valores y su tamaño se fija por la vida de la matriz.

Declaración de una matriz

En general, usted declara una matriz del siguiente modo:

```
new elementType [arraySize]
```

Existen dos formas para crear una matriz entera de elementos. Esta sentencia crea una matriz con espacio para cinco elementos pero está vacía:

```
// crea una matriz vacía de 5 elementos:
int[] integers = new int[5];
```

Esta sentencia crea la matriz y la inicializa, todo a la vez:

```
// crea una matriz de 5 elementos con valores:
int[] integers = new int[] { 1, 2, 3, 4, 5 };
```

Los valores iniciales van entre las llaves y se separan con comas.

Matrices, al modo difícil

Un modo más difícil para crear una matriz sería crearla y luego codificar un bucle para inicializarlo:

```
int[] integers = new int[5];
for (int aa = 0; aa < integers.length; aa++) {
  integers[aa] = aa;
}
```

Este código declara la matriz entera de cinco elementos. Si usted intenta poner más de cinco elementos en la matriz, el tiempo de ejecución de Java reclamará y arrojará una *excepción*. Aprenderá acerca de las excepciones y cómo manejarlas en la Part 2.

Carga de una matriz

Para cargar una matriz, serpentee por los enteros desde 1 a través de la longitud de la matriz (a la cual llega al llamar `.length` en la matriz, — más sobre eso en un minuto). En este caso, usted se detiene cuando llegue a 5

Una vez que se carga la matriz, puede acceder a ella como antes:

```
Logger l = Logger.getLogger("Test");
for (int aa = 0; aa < integers.length; aa++) {
  l.info("This little integer's value is: " + integers[aa]);
}
```

Esta sintaxis (nueva desde el JDK 5) también funciona:

```
Logger l = Logger.getLogger("Test");
for (int i : integers) {
  l.info("This little integer's value is: " + i);
}
```

Encuentro la sintaxis más nueva más simple para trabajar y la usaré a lo largo de esta sección.

El índice de elementos

Considere a una matriz como una serie de grupos y en cada grupo va un elemento de cierto tipo. Se gana el acceso a cada grupo al usar un *índice*:

```
element = arrayName [elementIndex];
```

Para acceder a un elemento, necesita la referencia a la matriz (su nombre) y el índice donde reside el elemento que usted quiere.

El método `length`

Un método útil, como ya ha visto, es `length`. Es un método incorporado, por lo tanto su sintaxis no incluye los paréntesis habituales. Solo escriba la palabra `length` y devolverá, — cómo es de esperar, — el tamaño de la matriz.

Las matrices en el lenguaje Java están basadas en cero. Por lo tanto, para alguna matriz llamada `array`, el primer elemento en la matriz reside siempre en `array[0]` y el último reside en `array[array.length - 1]`.

Una matriz de objetos

Ha visto cómo las matrices pueden tener tipos primitivos pero vale la pena mencionar que también pueden tener objetos. En ese sentido, la matriz es la colección más utilitaria del lenguaje Java.

Crear una matriz de objetos `java.lang.Integer` no es muy diferente de crear una matriz de tipos primitivos. Una vez más, tiene dos modos de hacerlo:

```
// crea una matriz vacía de 5 elementos:
Integer[] integers = new Integer[5];
```

```
// crea una matriz de 5 elementos con valores:
Integer[] integers = new Integer[] { Integer.valueOf(1),
                                     Integer.valueOf(2)
                                     Integer.valueOf(3)
                                     Integer.valueOf(4)
                                     Integer.valueOf(5));
```

Embalaje y desembalaje

Cada tipo primitivo en el lenguaje Java tiene una clase homóloga JDK, que puede ver en la Tabla 4:

Tabla 4. Primitivos y homólogos JDK

Primitivo	Homólogo JDK
`booleano`	`java.lang.Boolean`
`byte`	`java.lang.Byte`
`caract.`	`java.lang.Character`
`corto`	`java.lang.Short`
`int`	`java.lang.Integer`
`largo`	`java.lang.Long`
`flotante`	`java.lang.Float`
`doble`	`java.lang.Double`

Cada clase JDK proporcionar métodos para analizar y convertir desde su representación interna hasta un tipo primitivo correspondiente. Por ejemplo, este código convierte el valor decimal 238 a un `Integer`:

```
int value = 238;
Integer boxedValue = Integer.valueOf(value);
```

Esta técnica se conoce como *embalaje* porque está poniendo el primitivo en un envoltorio o caja.

De forma similar, para convertir la representación de `Integer` de nuevo a su homólogo `int`, usted lo *desembalaría* de la siguiente forma:

```
Integer boxedValue = Integer.valueOf(238);
int intValue = boxedValue.intValue();
```

Embalaje automático y desembalaje automático

En un sentido estricto, usted no necesita embalar y desembalar los primitivos de forma explícita. En cambio, podría usar las funciones de embalaje automático y desembalaje automático del lenguaje Java, del siguiente modo:

```
int intValue = 238;
Integer boxedValue = intValue;
//
intValue = boxedValue;
```

Sin embargo, recomiendo que evite usar el embalaje automático y desembalaje automático porque puede llevar a tener problemas de códigos. El código en los fragmentos de embalaje y desembalaje es más evidente, y de ese modo más legible, que el código embalado automáticamente y creo que vale la pena hacer el esfuerzo extra.

Análisis y conversión de tipos embalados

Ha visto cómo obtener un tipo embalado pero, ¿qué tal si se analiza una `String` que usted sospecha que tiene un tipo embalado en la caja misma? Las clases de envoltorio JDK tienen métodos para eso también:

```
String characterNumeric = "238";
Integer convertedValue = Integer.parseInt(characterNumeric);
```

También puede convertir los contenidos de un tipo de envoltorio JDK a `String`:

```
Integer boxedValue = Integer.valueOf(238);
String characterNumeric = boxedValue.toString();
```

Observe que cuando usa el operador de concatenación en una expresión de `String` (ya ha visto esto en las llamadas al `Logger`), el tipo primitivo se embala automáticamente y los tipos de envoltorio tienen automáticamente a `toString()` invocada sobre ellos. Muy útil.

Listas

Una `Lista` es una construcción de colección que, por definición, es una colección ordenada, también conocida como una *secuencia*. Debido a que una `Lista` es ordenada, usted tiene completo control sobre el lugar en donde van los elementos en la `Lista`. Una colección de `Lista` Java solo puede tener objetos y define un contrato estricto sobre cómo se comporta.

La `Lista` es una interfaz, por lo que usted no puede crear una instancia de ella directamente. Trabajará con su implementación más comúnmente usada, `ArrayList`:

```
List<Object> listOfObjects = new ArrayList<Object>();
```

Observe que hemos asignado el objeto `ArrayList` a una variable de tipo `Lista`. La programación Java le permite asignar una variable de un tipo a otro, siempre y

cuando la variable a la que se asigna sea una superclase o interfaz implementada por la variable desde la cual se asigna. Veremos más sobre cómo se afectan las asignaciones de variables en la Part 2 en la sección Inheritance.

Tipo formal

¿Qué sucede con el `<Object>` en el recorte anterior del código? Se llama el *tipo formal* y le comunica al compilador que esta `Lista` contiene una colección de tipo `Object`, lo que significa que puede poner prácticamente lo que quiera en la `Lista`.

Si usted quisiera intensificar las restricciones sobre lo que pudiera o no ir en la `Lista`, la definiría de modo diferente:

```
List<Person> listOfPersons = new ArrayList<Person>();
```

Ahora su `Lista` solo puede tener instancias de `Person`.

Uso de las `Listas`

Usar las `Lista`s es muy fácil, como las colecciones Java en general. Aquí hay algunas de las cosas que querrá hacer con las `Lista`s:

- Poner algo en la `Lista.`
- Preguntarle a la `Lista` cuán grande es actualmente.
- Obtener algo de la `Lista`.

Intentemos algunas de estas. Ya ha visto cómo crear una instancia de la `Lista` al crear una instancia de su tipo de implementación de `ArrayList`, por lo que comenzará desde allí.

Para poner algo en una `Lista`, llame al método `add()`:

```
List<Integer> listOfIntegers = new ArrayList<Integer>();
listOfIntegers.add(Integer.valueOf(238));
```

El método `add()` agrega el elemento al final de la `Lista`.

Para preguntarle a la `Lista` cuán grande es, llame al `size()`:

```
List<Integer> listOfIntegers = new ArrayList<Integer>();
listOfIntegers.add(Integer.valueOf(238));
Logger l = Logger.getLogger("Test");
l.info("Current List size: " + listOfIntegers.size());
```

Para recuperar un elemento de la `Lista`, llame al `get()` y pase el índice del elemento que usted quiere:

```
List<Integer> listOfIntegers = new ArrayList<Integer>();
listOfIntegers.add(Integer.valueOf(238));
Logger l = Logger.getLogger("Test");
l.info("Item at index 0 is: " listOfIntegers.get(0));
```

En una aplicación del mundo real, una `Lista` contendría registros, u objetos de negocios, y usted posiblemente querría examinarlos todos como parte de su proceso. ¿Cómo hace eso de un modo genérico? Tiene que iterar por la colección, lo cual usted puede hacer porque la `Lista` implementa la interfaz `java.lang.Iterable`. (Aprenderá acerca de las interfaces en la Part 2).

`Iterable`

Si una colección implementa `java.lang.Iterable`, se llama una *colección iterable*. Eso significa que usted puede comenzar por un extremo y recorrer la colección elemento por elemento hasta que se le acaben los elementos.

Ya ha visto la sintaxis especial para iterar por las colecciones que implementan la interfaz `Iterable`, en la sección Loops. Aquí está de nuevo:

```
for (objectType varName : collectionReference) {
  // Start using objectType (via varName) right away...
}
```

Iteración en una `Lista`

El ejemplo anterior fue abstracto. Ahora, aquí tiene uno más realista:

```
List<Integer> listOfIntegers = obtainSomehow();
Logger l = Logger.getLogger("Test");
for (Integer i : listOfIntegers) {
  l.info("Integer value is : " + i);
}
```

Ese pequeño recorte de código hace lo mismo que este más largo:

```
List<Integer> listOfIntegers = obtainSomehow();
Logger l = Logger.getLogger("Test");
for (int aa = 0; aa < listOfIntegers.size(); aa++) {
  Integer I = listOfIntegers.get(aa);
  l.info("Integer value is : " + i);
}
```

El primer recorte usa sintaxis taquigráfico: no hay una variable de `índice` (`aa` en este caso) para inicializar y ninguna llamada al método `get()` de la `Lista`.

Debido a que la `Lista` extiende la `java.util.Collection`, que implementa `Iterable`, usted puede usar la sintaxis taquigráfica para iterar por cualquier `Lista`.

Conjuntos

Un `Conjunto` es una construcción de colecciones que por definición contiene elementos únicos, — es decir, ningún duplicado. Mientras que una `Lista` puede contener el mismo objeto cientos de veces, un `Conjunto` solo puede contener cierta instancia una vez. Una colección de `Conjunto` Java solo puede tener objetos y define un contrato estricto sobre cómo se comporta.

Debido a que el `Conjunto` es una interfaz, usted no puede crear una instancia de él directamente, por lo tanto le mostraré una de mis implementaciones favoritas: `HashSet`. `HashSet` es fácil de usar y es similar a la `Lista`.

Aquí hay algunas cosas que querrá hacer con un `Conjunto`:

- Poner algo en el `Conjunto`.
- Preguntarle al `Conjunto` cuán grande es actualmente.
- Obtener algo del `Conjunto`.

Uso de los `Conjuntos`

Un atributo característico de un `Conjunto` es que garantiza la singularidad entre sus elementos pero no le interesa el orden de los elementos. Considere el siguiente código:

```
Set<Integer> setOfIntegers = new HashSet<Integer>();
setOfIntegers.add(Integer.valueOf(10));
setOfIntegers.add(Integer.valueOf(11));
setOfIntegers.add(Integer.valueOf(10));
for (Integer i : setOfIntegers) {
  l.info("Integer value is: " + i);
}
```

Puede que usted espere que el `Conjunto` tenga tres elementos en él pero de hecho solo tiene dos porque el objeto `Integer` que contiene el valor `10` solo se agregará una vez.

Tenga este comportamiento en mente cuando haga la iteración por un `Conjunto`, como en el siguiente modo:

```
Set<Integer> setOfIntegers = new HashSet();
setOfIntegers.add(Integer.valueOf(10));
setOfIntegers.add(Integer.valueOf(20));
setOfIntegers.add(Integer.valueOf(30));
setOfIntegers.add(Integer.valueOf(40));
setOfIntegers.add(Integer.valueOf(50));
Logger l = Logger.getLogger("Test");
for (Integer i : setOfIntegers) {
  l.info("Integer value is : " + i);
}
```

Es posible que los objetos se impriman en un orden diferente del que usted los agregó porque un `Conjunto` garantiza la singularidad, no el orden. Verá esto por usted mismo si pega el código anterior en el método `main()` de su clase `Person` y lo ejecuta.

Mapas

Un `Mapa` es una construcción de colección útil porque le permite asociar un objeto (la *clave*) con otro (el *valor*). Como puede imaginar, la clave para el `Mapa` debe ser única y se usa para recuperar el valor en un momento posterior. Una colección de `Mapa` Java solo puede tener objetos y define un contrato estricto sobre cómo se comporta.

Debido a que el `Mapa` es una interfaz, usted no puede crear una instancia de él directamente, por lo tanto le mostraré una de mis implementaciones favoritas: `HashMap`.

Aquí hay algunas de las cosas que querrá hacer con los `Mapa`s:

- Poner algo en el `Mapa`.
- Obtener algo del `Mapa`.
- Obtener un `Conjunto` de claves para el `Mapa` — para hacer la iteración en él.

Uso de los `Mapa`s

Para poner algo en un `Mapa`, necesita tener un objeto que represente su clave y un objeto que represente su valor:

```
public Map<String, Integer> createMapOfIntegers() {
  Map<String, Integer> mapOfIntegers = new HashMap<String, Integer>();
  mapOfIntegers.put("1", Integer.valueOf(1));
  mapOfIntegers.put("2", Integer.valueOf(2));
  mapOfIntegers.put("3", Integer.valueOf(3));
  // . . .
  mapOfIntegers.put("168", Integer.valueOf(168));
}
```

Es este ejemplo, el `Mapa` contiene `Integer`s, con la clave hecha por una `String`, que resulta ser su representación de `String`. Para recuperar un valor `Integer` particular, necesita su representación `String`:

```
mapOfIntegers = createMapOfIntegers();
Integer oneHundred68 = mapOfIntegers.get("168");
```

Uso del `Conjunto` con el `Mapa`

En ocasiones, usted puede encontrarse con una referencia a un `Mapa` y simplemente quiere recorrer todo el conjunto de contenidos. En este caso, necesitará un `Conjunto` de las claves para el `Mapa`:

```
Set<String> keys = mapOfIntegers.keySet();
Logger l = Logger.getLogger("Test");
for (String key : keys) {
  Integer value = mapOfIntegers.get(key);
  l.info("Value keyed by '" + key + "' is '" + value + "'");
}
```

Observe que el método `toString()` del `Integer` recuperado desde el `Mapa` se llama automáticamente cuando se usa en la llamada del `Logger`. El `Mapa` no devuelve una `Lista` de sus claves porque se hace una clave para el `Mapa` y cada clave es única; la singularidad es la característica distintiva de un `Conjunto`.

Sección 14. Archivo del código Java

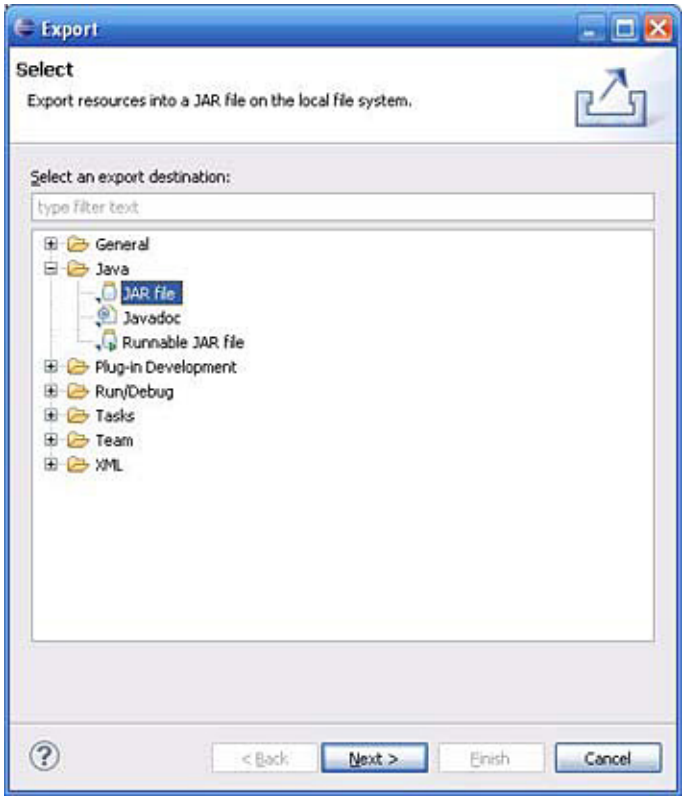
Ahora que ha aprendido un poco sobre la escritura de aplicaciones Java, puede que se esté preguntando cómo empaquetarlas para que otros desarrolladores puedan usarlas o cómo importar códigos de otros desarrolladores en sus aplicaciones. Esta sección le muestra cómo.

JAR

El JDK se envía con una herramienta llamada JAR, que significa Java Archive (archivo Java). Use esta herramienta para crear archivos JAR. Una vez que haya empaquetado su código en un archivo JAR, otros desarrolladores pueden simplemente dejar el archivo JAR en sus proyectos y configurar sus proyectos para usar su código.

Crear un archivo JAR en Eclipse es facilísimo. En su espacio de trabajo, haga clic derecho en el paquete `com.makotogroup.intro` y seleccione **Export**. Verá el diálogo que se muestra en la Ilustración 8. Elija **Java > JAR file**.

Ilustración 8. Recuadro de diálogo de exportación.



Cuando se abra el siguiente recuadro de diálogo, navegue hasta la ubicación donde quiere almacenar su archivo JAR y coloque el nombre que quiera al archivo. La extensión .jar es la predeterminada, que recomiendo usar. Haga clic en **Finish**.

Verá su archivo JAR en la ubicación que seleccionó. Puede usar las clases en él desde su código si lo pone en su ruta de desarrollo en Eclipse. Hacer eso es facilísimo también, como verá a continuación.

Uso de aplicaciones de terceros

Mientras usted se siente más y más cómodo al escribir aplicaciones Java, querrá usar más y más aplicaciones de terceros para soportar su código. A modo de ejemplo, digamos que usted quiere usar `joda-time`, una biblioteca de sustitución JDK para hacer manejo de fecha/tiempo, manipulaciones y cálculos.

Supongamos que ya ha descargado `joda-time`, que se almacena en un archivo JAR. Para usar las clases, su primer paso es crear un directorio lib en su proyecto y dejar el archivo JAR en él:

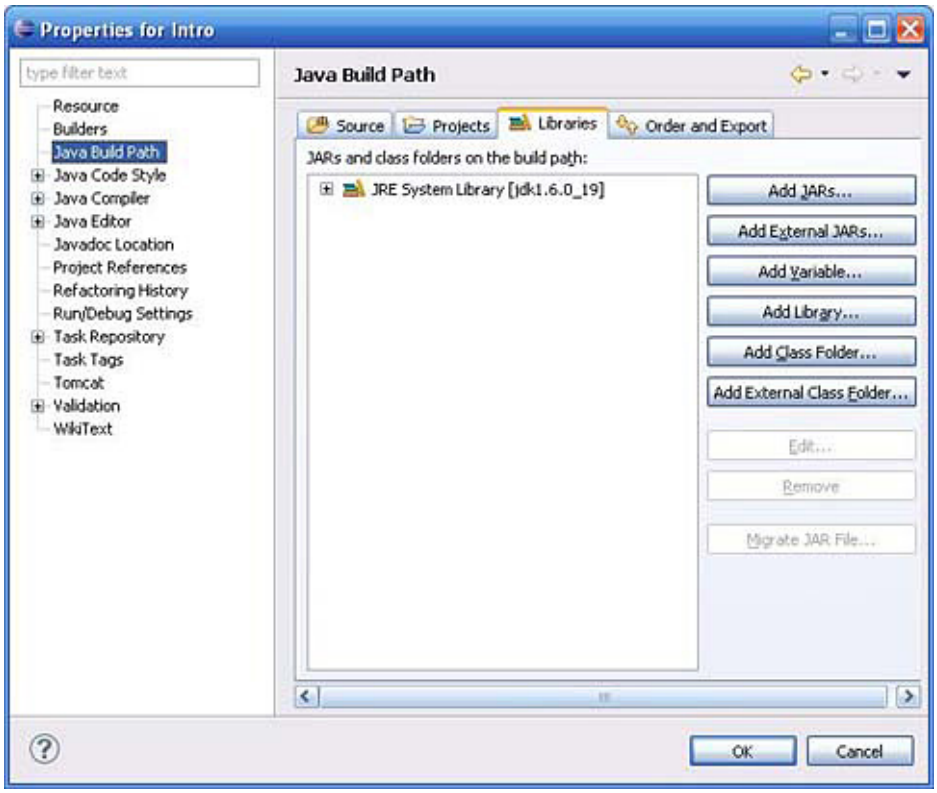
1. Haga clic derecho en la carpeta raíz Intro en la vista Project Explorer.
2. Elija **New > Folder** y llame la carpeta lib.
3. Haga clic en **Finish**.

La carpeta nueva aparece al mismo nivel que `src`. Ahora copie el archivo .jar `joda-time` en su nuevo directorio `lib`. Para este ejemplo, el archivo se llama `joda-time-1.6.jar`. (Al nombrar un archivo JAR es común incluir el número de la versión).

Ahora todo lo que necesita hacer es comunicarle a Eclipse que incluya las clases del archivo `joda-time-1.6.jar` en su proyecto:

1. Haga clic derecho en el proyecto Intro en su espacio de trabajo, luego seleccione **Properties**.
2. En el recuadro de diálogo de Propiedades, seleccione la pestaña de Bibliotecas, como se muestra en la Ilustración 9:

Ilustración 9. Properties > Java Build Path



3. Haga clic en el botón **Add External JARs**, luego navegue hasta el directorio lib del proyecto, seleccione el archivo joda-time-1.6.jar y haga clic en **OK**.

Una vez que Eclipse haya procesado los códigos (es decir, los archivos de clases) en el archivo JAR, están disponibles para hacer referencia (importar) desde su código Java. Observe en Project Explorer que hay una carpeta nueva llamada Bibliotecas referenciadas que contiene el archivo joda-time-1.6.jar.

Sección 15. Escribir un buen código Java

Usted tiene suficiente sintaxis Java acumulada en su haber para escribir programas Java básicos, lo que significa que la primera mitad de este tutorial está a punto de concluir. Esta sección final presenta algunas de las mejores prácticas que le ayudarán a escribir códigos Java más limpios y más plausibles de ser mantenidos.

Mantenga pequeñas las clases

Usted creó algunas clases en este tutorial. Luego de generar pares de getters y setters incluso para la pequeña cantidad (de acuerdo a los estándares de una clase Java del mundo real) de atributos, la clase `Person` tiene 150 líneas de código. Esta

es una clase pequeña. No es raro ver clases con 50 o 100 métodos y mil líneas de origen (o más). El punto de los métodos es mantener solo los que necesita. Si necesita varios métodos ayudantes que hagan esencialmente lo mismo pero que tomen parámetros diferentes (tales como el método `printAudit()`), esa es una buena elección. Solo asegúrese de limitar la lista de métodos a lo que necesita, no más.

En general, las clases representan alguna entidad conceptual en su aplicación y sus tamaños deberían reflejar solo la funcionalidad para hacer lo que sea que la entidad necesite hacer. Deberían permanecer muy centradas para hacer una pequeña cantidad de actividades y hacerlas bien.

Nombre a los métodos con cuidado

Un buen patrón de codificación cuando se trata de nombres de métodos es el patrón de nombres de métodos *revelador de intenciones*. Este patrón es más fácil de entender con un simple ejemplo. ¿Cuál de los siguientes nombres de métodos es más fácil de descifrar a la vista?

- `a()`
- `computeInterest()`

La respuesta debería ser evidente, sin embargo por alguna razón, los programadores tienen una tendencia a darle a los métodos (y variables, para tal caso) nombres pequeños, abreviados. Sin duda, un nombre ridículamente largo puede ser poco práctico pero un nombre que transmite lo que un método hace no necesita ser ridículamente largo. Seis meses después de que escriba un grupo de códigos, tal vez no recuerde lo que quiso hacer con un método llamado `a()` pero es evidente que un método llamado `computeInterest()`, bueno, probablemente compute interés.

Mantenga pequeños los métodos

Los métodos pequeños son tan preferibles como las clases pequeñas y por razones similares. Un modismo que intento seguir es mantener el tamaño de un método en *una página* como lo veo en mi pantalla. Esto hace que mis clases de aplicaciones sean más plausibles de ser mantenidas.

Si un método crece más allá de una página, lo refactorizo. *Refactorear* significa cambiar el diseño de un código existente sin cambiar sus resultados. Eclipse tiene un maravilloso conjunto de herramientas de refactorización. Normalmente, un método largo contiene subgrupos de funcionalidad agrupados. Tome esta funcionalidad y muévala a otro método (nombrándola como corresponde) y pase los parámetros como se necesite.

Limite cada método a un solo trabajo. He descubierto que un método que hace solo una cosa bien, normalmente no toma más de alrededor de 30 líneas de códigos.

Use los comentarios

Por favor, use los comentarios. Las personas que siguen luego de usted (o incluso usted mismo, seis meses más adelante) le agradecerán. Tal vez haya escuchado el antiguo refrán *Un código bien escrito es de autodocumentación, por lo tanto, ¿quién necesita comentarios?* Le daré dos razones por lo que esto es falso:

- La mayoría de los códigos no están bien escritos.
- Su código probablemente no está tan bien escrito como usted piensa.

Así que haga comentarios en su código. Punto.

Use un estilo consistente.

La codificación del estilo es realmente una cuestión de preferencia personal pero me gustaría ofrecer este consejo: use una sintaxis Java estándar para las llaves:

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

No use este estilo:

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

Ni este:

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

¿Por qué? Bueno, es estándar, por lo tanto la mayoría de los códigos con los que usted se encuentre (como códigos que usted no escribió pero que se pueden pagar para mantener) muy probablemente estarán escritos de ese modo. Una vez dicho eso, Eclipse *sí* le permite definir estilos de código y formatear su código de cualquier modo que a usted le guste. Lo principal es que elija un estilo y lo mantenga.

Use un registro incorporado

Antes de que Java 1.4 introdujera el registro incorporado, el modo canónico para descubrir lo que su programa estaba haciendo era hacer una llamada de sistema como la siguiente:

```
public void someMethod() {
  // Do some stuff...
  // Now tell all about it
  System.out.println("Telling you all about it:");
  // Etc...
}
```

El recurso de registro incorporado del lenguaje Java (consulte de nuevo Your first Java object) es una mejor alternativa. Yo *nunca* uso `System.out.println()` en mi código y sugiero que usted tampoco lo use.

Siguiendo los pasos de Fowler

El mejor libro en la industria (en mi opinión, y no lo digo solo yo) es *Refactoring: Improving the Design of Existing Code* de Martin Fowler et al. (Vea Recursos). Incluso es divertido leerlo. Fowler y sus coautores hablan sobre "olores de códigos" que piden refactorización y profundizan mucho sobre las diversas técnicas para arreglarlos.

En mi opinión, la refactorización y la habilidad para escribir códigos de prueba son las aptitudes más importantes para que los programadores nuevos aprendan. Si todos fueran buenos en ambos aspectos, revolucionaría la industria. Si usted se vuelve bueno en ambos, en última instancia producirá códigos más limpios y aplicaciones más funcionales que muchos de sus colegas.

Sección 16. Conclusión para la Parte 1

En este tutorial, ha aprendido acerca de la programación orientada a objetos, ha descubierto una sintaxis Java que le permite crear objetos útiles y se ha familiarizado con un IDE que le ayuda a controlar su entorno de desarrollo. Sabe cómo crear y ejecutar objetos Java que pueden hacer una buena cantidad actividades, que incluyen hacer cosas diferentes en base a entradas diferentes. También sabe cómo hacer que sus aplicaciones admitan archivos JAR para que los otros desarrolladores las usen en sus programas y cuenta con algunas de las mejores prácticas básicas de programación Java en su haber.

Lo que sigue

En la segunda mitad de este tutorial, comenzará a aprender acerca de algunas de las construcciones más avanzadas de la programación Java, aunque el debate general todavía será de alcance introductorio. Los temas de programación Java que se cubren en ese tutorial incluyen:

- Herencia y abstracción
- Interfaces
- Clases anidadas
- Expresiones regulares
- Genéricos
- Tipos de enumeración

- E/S
- Serialización

Lea "Introducción a la programación Java, parte 2: Construcciones para aplicaciones del mundo real".

Recursos

Aprender

- Java technology homepage: El sitio oficial de Java tiene enlaces a todo lo relacionado con la plataforma Java, incluida la especificación del lenguaje Java y documentación Java API.
- Java 6: Aprenda más sobre JDK 6 y las herramientas que incluye.
- Javadoc homepage: Aprenda los detalles del uso de Javadoc, incluido cómo usar la herramienta de línea de comandos y cómo escribir sus propios Doclets que le permiten crear formatos personalizados para su documentación.
- *Refactorización: Improving the Design of Existing Code* (Martin Fowler et al., Addison-Wesley, 1999): Este libro es un recurso excelente para aprender cómo escribir un código más limpio y más plausible de mantener.
- New to Java technology: Verifique este compendio de recursos de developerWorks para desarrolladores Java novatos.
- *5 things you didn't know about...*: Esta serie de developerWorks proporciona introducciones breves a consejos y tradiciones de programación Java menos conocidos (pero a menudo introductorios).
- "Struts, an open-source MVC implementation" (Malcolm Davis, developerWorks, febrero del 2001): Este artículo presenta el patrón de diseño modelo-vista-controlador como se implementa en una de las infraestructuras de desarrollo web más antiguas de la plataforma Java.
- "*Java theory and practice*: Garbage collection and performance" (Brian Goetz, developerWorks, enero del 2004): Este artículo es una visión general de la recogida de basura, que incluye consejos para novatos para escribir clases amigables de GC.
- Eclipse IDE project resources from developerWorks: Aprenda para qué es bueno Eclipse, por qué es importante, cómo puede comenzar a usarlo y dónde aprender más sobre él.
- The Java Tutorials: Obtenga una introducción detallada al lenguaje Java.
- La developerWorks Java technology zone: Cientos de artículos acerca de cada aspecto de la programación Java.

Obtener los productos y tecnologías

- JDK 6: Descargue el JDK 6 desde Sun (Oracle).
- Eclipse: Descargue el IDE Eclipse para desarrolladores Java.
- IBM developer kits: IBM proporciona una cantidad de kits para desarrolladores Java para utilizar en plataformas populares.

Comentar

- Participe en My developerWorks community. Conéctese con otros usuarios de developerWorks mientras explora los blogs, foros, grupos y wikis realizados por el desarrollador.

Sobre el autor

J. Steven Perry

J. Steven Perry es desarrollador de software, arquitecto y fanático general de Java que ha estado desarrollando software profesionalmente desde 1991. Sus intereses profesionales abarcan desde el funcionamiento interno de la JVM hasta el modelo UML y todo lo que está en el medio. Steve tiene una pasión por escribir y ser mentor. Es el autor de *Java Management Extensions* (O'Reilly), *Log4j* (O'Reilly) y los artículos de developerWorks de IBM "Joda-Time" y OpenID for Java Web applications". Pasa tiempo libre con sus tres hijos, anda en bicicleta y enseña yoga.