# Introducción a la programación Java, parte 2:
# Construcciones para aplicaciones del mundo real

## Funciones más avanzadas del lenguaje Java

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

10-12-2012

En la Parte 1 de este tutorial, el programador Java™ profesional J. Steven Perry presentó la sintaxis del lenguaje Java y las bibliotecas que usted necesita para escribir aplicaciones Java simples. La Parte 2, todavía orientada a desarrolladores nuevos en el desarrollo de aplicaciones Java, presenta las construcciones de programación más sofisticadas requeridas para crear complejas aplicaciones Java del mundo real. Los temas que se cubren incluyen manejo de excepciones, herencia y abstracción, expresiones regulares, genéricos, E/S Java y serialización Java.

Ver más contenido de esta serie

## Antes que comience

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

### Sobre esta serie

El tutorial de dos partes "Introducción a la programación Java" tiene como objetivo conseguir que los desarrolladores de software 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.

### Acerca de este tutorial

Esta segunda mitad del tutorial "Introducción a la programación Java" presenta las capacidades del lenguaje Java que son más sofisticadas que aquellas que se cubrieron en la Parte 1.

### Objetivos

El lenguaje Java es lo suficientemente maduro y sofisticado para ayudarlo a usted a cumplir casi cualquier tarea de programación. En este tutorial, le presentaremos funciones del lenguaje Java que necesitará para manejar escenarios de programación compleja, que incluyen:

- Manejo de excepciones
- Herencia y abstracción
- Interfaces
- Clases anidadas
- Expresiones regulares
- Genéricos
- Tipos de `enum`
- E/S
- Serialización

### Requisitos previos

El contenido de este tutorial está orientado a programadores nuevos en el lenguaje Java que no conocen sus funciones más sofisticadas. El tutorial asume que usted ha trabajado en la "Introducción a la programación Java, parte 1: Conceptos básicos del lenguaje Java" para:

- Obtener un entendimiento de los conceptos básicos de OOP en la plataforma Java.
- Establecer el entorno de desarrollo para los ejemplos del tutorial.
- Comenzar el proyecto de programación que usted continuará desarrollando en la Parte 2.

### Requisitos del sistema

Los ejercicios en este tutorial requieren un entorno de desarrollo que consista en:

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

Las instrucciones de descarga e instalación de ambos se incluyen en la Parte 1.

La configuración recomendada del sistema para este tutorial es la siguiente:

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

## Pasos siguientes con los objetos

La Parte 1 de este tutorial terminó con un objeto `Person` que fue razonablemente útil pero no tan útil como podría serlo. Aquí comenzará a aprender sobre las técnicas para mejorar un objeto como `Person`, comenzando con las siguientes técnicas:

- Sobrecarga de métodos
- Alteración temporal de métodos
- Comparación de un objeto con otro
- Hacer que su código sea más fácil de depurar

## Sobrecarga de métodos

Cuando usted crea dos métodos con el mismo nombre pero con diferentes listas de argumentos (es decir, diferentes números o tipos de parámetros), tiene un método sobrecargado. Los métodos sobrecargados siempre están en la misma clase. Al tiempo de ejecución, el Java Runtime Environment (JRE, también conocido como el tiempo de ejecución Java) decide qué variación de su método sobrecargado llamar en base a los argumentos que se han pasado.

Suponga que `Person` necesita un par de métodos para imprimir una auditoría de su estado actual. Llamaré a esos métodos `printAudit()`. Pegue el método sobrecargado en el Listado 1 en la vista de edición de Eclipse:

### Listado 1. printAudit(): Un método sobrecargado

```
public void printAudit(StringBuilder buffer) {
  buffer.append("Name="); buffer.append(getName());
  buffer.append(","); buffer.append("Age="); buffer.append(getAge());
  buffer.append(","); buffer.append("Height="); buffer.append(getHeight());
  buffer.append(","); buffer.append("Weight="); buffer.append(getWeight());
  buffer.append(","); buffer.append("EyeColor="); buffer.append(getEyeColor());
  buffer.append(","); buffer.append("Gender="); buffer.append(getGender());
}
public void printAudit(Logger l) {
  StringBuilder sb = new StringBuilder();
  printAudit(sb);
  l.info(sb.toString());
}
```

En este caso, usted tiene dos versiones sobrecargadas de `printAudit()` y en realidad una usa a la otra. Al proveer dos versiones, le da al interlocutor la elección sobre cómo imprimir una auditoría de la clase. Dependiendo de los parámetros que se pasen, el tiempo de ejecución Java llamará el método correcto.

### Dos reglas de sobrecarga de métodos

Recuerde estas dos reglas importantes cuando use métodos sobrecargados:

- No puede sobrecargar un método solo al cambiar su tipo de retorno.
- No puede tener dos métodos con la misma lista de parámetros.

Si usted viola estas reglas, el compilador le dará un error.

## Alteración temporal de métodos

Cuando una subclase de otra clase proporciona su propia implementación de un método definido en una clase padre, eso se llama alteración temporal de un método. Para ver cómo la alteración temporal de métodos es útil, necesita trabajar un poco en su clase `Employee`. Una vez que la haya establecido, podré mostrarle dónde la alteración temporal de métodos es de ayuda.

### Employee: Una subclase de Person

Recuerde de la Parte 1 de este tutorial que `Employee` puede ser una subclase (o hijo) de `Person` que tiene algunos atributos adicionales:

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

Para declarar tal clase en un archivo llamado Employee.java, haga clic derecho en el paquete `com.makotogroup.intro` en Eclipse. Elija **New > Class...** y se abrirá el recuadro de diálogo de Clase Java nueva, como se muestra en la Ilustración 1:

## Ilustración 1. Diálogo Clase Java nueva


Escriba `Employee` como el nombre de la clase y `Person` como su superclase, luego haga clic en **Finish**. Verá la clase `Employee` en una ventana de edición. No necesita explícitamente declarar un constructor pero continúe e implemente ambos constructores de todos modos. Primero, asegúrese de que la ventana de edición de la clase `Employee` tenga el foco, luego vaya a **Source > Generate Constructors from Superclass...** y verá un diálogo que se parece a la Ilustración 2:

## Ilustración 2. Genere constructores desde el diálogo de superclase


Verifique ambos constructores (como se muestra en la Ilustración 2) y haga clic en **OK**. Eclipse generará los constructores por usted. Ahora debería tener una clase `Employee` como la del Listado 2:

## Listado 2. La nueva y mejorada clase Employee

```
package com.makotogroup.intro;

public class Employee extends Person {

 public Employee() {
  super();
  // TODO Auto-generated constructor stub
 }

 public Employee(String name, int age, int height, int weight,
            String eyeColor, String gender) {
  super(name, age, height, weight, eyeColor, gender);
  // TODO Auto-generated constructor stub
 }

}
```

## Employee recibe hereda de Person

`Employee` hereda los atributos y el comportamiento de su padre, `Person`, y también tiene algo por su propia cuenta, como puede ver en el Listado 3:

## Listado 3. La clase Employee con los atributos de Person

```
package com.makotogroup.intro;


import java.math.BigDecimal;
```

```
public class Employee extends Person {

  private String taxpayerIdentificationNumber;
  private String employeeNumber;
  private BigDecimal salary;

  public Employee() {
    super();
  }
  public String getTaxpayerIdentificationNumber() {
    return taxpayerIdentificationNumber;
  }
  public void setTaxpayerIdentificationNumber(String taxpayerIdentificationNumber) {
    this.taxpayerIdentificationNumber = taxpayerIdentificationNumber;
  }
  // Other getter/setters...
}
```

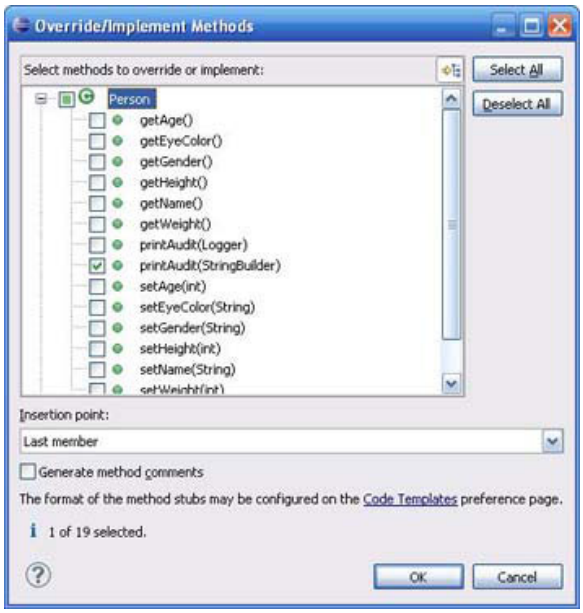
## Alteración temporal de métodos: printAudit()

Ahora, como prometí, usted está listo para un ejercicio de alteración temporal de métodos. Hará una alteración temporal del método printAudit() (vea el Listado 1) que usó para formatear el estado actual de una instancia de Person. Employee hereda ese comportamiento de Person y si usted crea una instancia de Employee, establece sus atributos e invoca a una de las sobrecargas de printAudit(), la llamada tendrá éxito. Sin embargo, la auditoría que se produce no representará totalmente a una Employee. El problema es que no puede formatear los atributos específicos de una Employee porque Person no los conoce.

La solución es alterar temporalmente la sobrecarga de printAudit() que toma un StringBuilder como un parámetro y agregar códigos para imprimir los atributos específicos de Employee.

Para hacer esto en su IDE Eclipse, vaya a **Source > Override/Implement Methods...** y verá un recuadro de diálogo que se parece a la Ilustración 3:

## Ilustración 3. Diálogo de alteración temporal/implementación de métodos



Seleccione la sobrecarga StringBuilder de printAudit, como se muestra en la Ilustración 3 y haga clic en **OK**. Eclipse generará el resguardo de método por usted y luego usted puede simplemente rellenar el resto del siguiente modo:

```
@Override
public void printAudit(StringBuilder buffer) {
  // Call the superclass version of this method first to get its attribute values
  super.printAudit(buffer);
  // Now format this instance's values
  buffer.append("TaxpayerIdentificationNumber=");
    buffer.append(getTaxpayerIdentificationNumber());
  buffer.append(","); buffer.append("EmployeeNumber=");
    buffer.append(getEmployeeNumber());
  buffer.append(","); buffer.append("Salary=");
    buffer.append(getSalary().setScale(2).toPlainString());
}
```

Observe la llamada a super.printAudit(). Lo que está haciendo aquí es pedirle a la superclase (Person) que exponga su comportamiento para printAudit() y luego usted lo aumenta con el comportamiento de printAudit() de tipo Employee.

La llamada a super.printAudit() no necesita ser la primera; solo pareció una buena idea imprimir primero esos atributos. De hecho, no necesita llamar para nada a super.printAudit(). Si usted no lo llama, debe formatear usted mismo los atributos desde Person (en el método Employee.printAudit()) o excluirlos completamente. Hacer la llamada a super.printAudit(), en este caso, es más fácil.

## Miembros de clases

Las variables y los métodos que usted tiene en Person y Employee son variables y métodos de instancia. Para usarlos, usted debe crear instancias de la clase que necesita o tener una referencia a la instancia. Cada instancia de objetos tiene variables y métodos y, para cada uno, el comportamiento exacto (por ejemplo, lo que se genera al llamar a printAudit()) será diferente porque se basa en el estado de la instancia de los objetos.

Las clases mismas también pueden tener variables y métodos, que se llaman miembros de clases. Usted declara miembros de clases con la palabra clave static introducida en la Parte 1 de este tutorial. Las diferencias entre los miembros de clases y los miembros de instancias son las siguientes:

- Cada instancia de una clase comparte una sola copia de una variable de clase.
- Puede llamar a los métodos de clases en la clase misma, sin tener una instancia.
- Los métodos de instancias pueden acceder a las variables de clases pero los métodos de clases no pueden acceder a variables de instancias.
- Los métodos de clases pueden acceder solo a las variables de clases.

### Incorporación de variables y métodos de clases

¿Cuándo tiene sentido agregar variables y métodos de clases? La mejor regla general es hacerlo raras veces, para que no las use excesivamente. Dicho eso, es una buena idea usar variables y métodos de clases:

- para declarar constantes que cualquier instancia de la clase pueda usar (y cuyo valor esté fijo al momento del desarrollo).
- para realizar un seguimiento de "contadores" de instancias de la clase.

- en una clase con métodos de utilidad que nunca necesitan una instancia de la clase (como `Logger.getLogger()`).

## Variables de clases

Para crear una variable de clase, use la palabra clave `static` cuando la declare:

```
accessSpecifier static variableName [= initialValue];
```

**Nota:** Los corchetes aquí indican que sus contenidos son opcionales. No son parte de la sintaxis de declaración.

El JRE crea espacio en la memoria para almacenar cada una de las variables de instancias de una clase para cada instancia de esa clase. En cambio, el JRE crea sólo una copia de cada variable de clase, a pesar de la cantidad de instancias. Lo hace de eso modo la primera vez que se carga la clase (es decir, la primera vez que encuentra la clase en un programa). Todas las instancias de la clase compartirán esa sola copia de la variable. Eso hace que las variables de clases sean una buena elección para las constantes que todas las instancias deberían poder usar.

Por ejemplo, usted declaró el atributo de `Género` de `Person` para que sea una `String` pero no puso ninguna restricción para ella. El Listado 4 muestra un uso común de las variables de clases:

## Listado 4. Uso de variables de clases

```
public class Person {
//. . .
 public static final String GENDER_MALE = "MALE";
 public static final String GENDER_FEMALE = "FEMALE";
// . . .
 public static void main(String[] args) {
   Person p = new Person("Joe Q Author", 42, 173, 82, "Brown", GENDER_MALE);
 // . . .
 }
//. . .
}
```

### Declaración de constantes

Normalmente, las constantes:

- se nombran con todas las letras en mayúsculas.
- se nombran como palabras múltiples, separadas por subrayados.
- se declaran como `finales` (para que sus valores no se puedan modificar).
- se declaran con un especificador de acceso `público` (para que otras clases que necesitan hacer referencia a sus valores por nombre puedan acceder a ellas).

En el Listado 4, para usar la constante para `MALE` en la llamada del constructor `Person`, usted simplemente haría referencia a su nombre. Para usar una constante fuera de la clase, usted la prologaría con el nombre de la clase donde se la declaró, de este modo:

```
String genderValue = Person.GENDER_MALE;
```

## Métodos de clases

Si usted ha estado siguiendo lo establecido desde la Parte 1, ya ha llamado al método estático Logger.getLogger() varias veces, — cada vez que haya recuperado una instancia de Logger para escribir alguna salida a la consola. Observe de todos modos que usted no necesitaba una instancia de Logger para hacer esto; en cambio, hizo referencia a la clase Logger misma. Esta es la sintaxis para hacer una llamada de método de clase. Como con las variables de clases, la palabra clave static identifica a Logger (en este ejemplo) como un método de clase. Los métodos de clases también se llaman a veces métodos estáticos por este motivo.

### Uso de métodos de clases

Ahora combinará lo que ha aprendido sobre variables y métodos estáticos para crear un método estático de Employee. Declarará una variable private static final para tener un Logger, que todas las instancias compartirán y que serán accesibles al llamar a getLogger() en la clase Employee. El Listado 5 muestra cómo:

## Listado 5. Creación de un método de clase (o estático)

```
public class Employee extends Person {
 private static final Logger logger = Logger.getLogger(Employee.class.getName());
//. . .
 public static Logger getLogger() {
   return logger;

 }

}
```

Están sucediendo dos cosas importantes en el Listado 5:

- La instancia Logger se declara con acceso private, por lo que ninguna clase fuera de Employee puede acceder a la referencia de forma directa.
- El Logger se inicializa cuando se carga la clase. Esto sucede porque usted usa la sintaxis del inicializador Java para darle un valor.

Para recuperar el objeto del Logger de la clase Employee, haga la siguiente llamada:

```
Logger employeeLogger = Employee.getLogger();
```

## Comparación de objetos

El lenguaje Java proporciona dos maneras para comparar objetos:

- El operador ==
- El método equals()

### Comparación de objetos con ==

La sintaxis == compara objetos por la igualdad para que a == b retorne verdadero solo si a y b tienen el mismo valor. Para los objetos, esto significa que los dos se refieren a la misma instancia de objeto. Para los primitivos, significa que los valores son idénticos. Considere el ejemplo en el Listado 6:

## Listado 6. Comparación de objetos con ==

```
int int1 = 1;
int int2 = 1;
l.info("Q: int1 == int2?          A: " + (int1 == int2));

Integer integer1 = Integer.valueOf(int1);
Integer integer2 = Integer.valueOf(int2);
l.info("Q: Integer1 == Integer2?   A: " + (integer1 == integer2));

integer1 = new Integer(int1);
integer2 = new Integer(int2);
l.info("Q: Integer1 == Integer2?   A: " + (integer1 == integer2));

Employee employee1 = new Employee();
Employee employee2 = new Employee();
l.info("Q: Employee1 == Employee2? A: " + (employee1 == employee2));
```

Si ejecuta el código del Listado 6 dentro de Eclipse, la salida debería ser:

```
Apr 19, 2010 5:30:10 AM com.makotogroup.intro.Employee main
INFO: Q: int1 == int2?          A: true
Apr 19, 2010 5:30:10 AM com.makotogroup.intro.Employee main
INFO: Q: Integer1 == Integer2?   A: true
Apr 19, 2010 5:30:10 AM com.makotogroup.intro.Employee main
INFO: Q: Integer1 == Integer2?   A: false
Apr 19, 2010 5:30:10 AM com.makotogroup.intro.Employee main
INFO: Q: Employee1 == Employee2? A: false
```

En el primer caso del Listado 6, los valores de los primitivos son los mismos, por lo tanto el operador `==` retorna `verdadero`. En el segundo caso, los objetos `Integer` se refieren a la misma instancia, por lo tanto de nuevo `==` retorna `verdadero`. En el tercer caso, aun cuando los objetos `Integer` envuelven el mismo valor, `==` retorna `falso` porque `integer1` e `integer2` se refieren a objetos diferentes. En base a esto, debería estar claro por qué `employee1 == employee2` retorna `falso`.

### Comparación de objetos con equals()

`equals()` es un método que cada objeto del lenguaje Java obtiene gratis porque se define como un método de instancia de `java.lang.Object` (del que cada objeto Java hereda).

Usted llama `equals()` exactamente como lo haría con cualquier otro método:

```
a.equals(b);
```

Esta sentencia invoca al método `equals()` del objeto `a`, pasándole una referencia al objeto `b`. De modo predeterminado, un programa Java simplemente verificaría para ver que los dos objetos fueran los mismos al usar la sintaxis `==`. Sin embargo, debido a que `equals()` es un método, se puede alterar temporalmente. Considere el ejemplo del Listado 6, modificado en el Listado 7 para comparar los dos objetos que usan `equals()`:

## Listado 7. Comparación de objetos con equals()

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

Integer integer1 = Integer.valueOf(1);
Integer integer2 = Integer.valueOf(1);
l.info("Q: integer1 == integer2?       A: " + (integer1 == integer2));
```

```
l.info("Q: integer1.equals(integer2)?   A: " + integer1.equals(integer2));

integer1 = new Integer(integer1);
integer2 = new Integer(integer2);
l.info("Q: integer1 == integer2?        A: " + (integer1 == integer2));
l.info("Q: integer1.equals(integer2)?   A: " + integer1.equals(integer2));

Employee employee1 = new Employee();
Employee employee2 = new Employee();
l.info("Q: employee1 == employee2 ? A: " + (employee1 == employee2));
*l.info("Q: employee1.equals(employee2) ? A : " +
employee1.equals(employee2));*
Running this code produces:

Apr 19, 2010 5:43:53 AM com.makotogroup.intro.Employee main
INFO: Q: integer1 == integer2?        A: true
Apr 19, 2010 5:43:53 AM com.makotogroup.intro.Employee main
INFO: Q: integer1.equals(integer2)?   A: true
Apr 19, 2010 5:43:53 AM com.makotogroup.intro.Employee main
INFO: Q: integer1 == integer2?        A: false
Apr 19, 2010 5:43:53 AM com.makotogroup.intro.Employee main
INFO: Q: integer1.equals(integer2)?   A: true
Apr 19, 2010 5:43:53 AM com.makotogroup.intro.Employee main
INFO: Q: employee1 == employee2?      A: false
Apr 19, 2010 5:43:53 AM com.makotogroup.intro.Employee main
INFO: Q: employee1.equals(employee2)? A: false
```

## Una observación sobre la comparación de Integers

En el Listado 7, no debería ser una sorpresa que el método equals() del Integer retorne verdadero si == retorna verdadero. Pero observe lo que sucede en el segundo caso, donde usted crea objetos separados que ambos envuelven el valor 1: == retorna falso porque integer1 e integer2 se refieren a diferentes objetos, pero equals() retorna verdadero.

Los escritores del JDK decidieron que para Integer, el significado de equals() sería diferente del predeterminado (que es comparar las referencias del objeto para ver si se refieren al mismo objeto) y, en cambio, retornarían verdadero en casos en los que el valor int subyacente es el mismo.

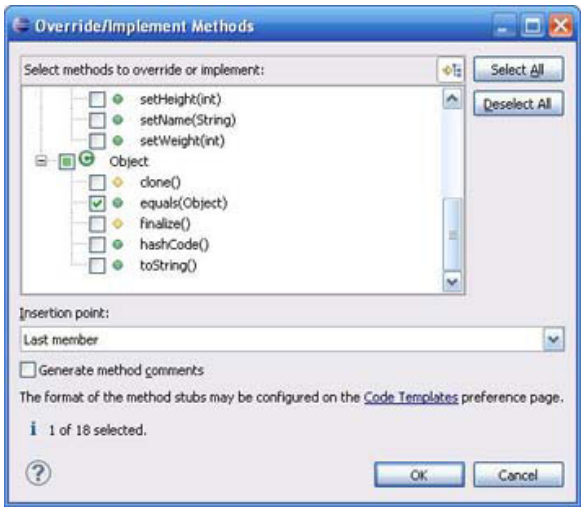
Para Employee, no alteró temporalmente equals(), por lo tanto el comportamiento predeterminado (de usar ==) retorna lo que usted esperaría, dado que employee1 y employee2 de hecho sí se refieren a objetos diferentes.

Básicamente, esto significa que para cualquier objeto que usted escribe, puede definir lo que equals() significa como adecuado para la aplicación que está escribiendo.

## Alteración temporal de equals()

Puede definir lo que equals() significa para los objetos de su aplicación al alterar temporalmente el comportamiento predeterminado de Object.equals(). De nuevo, puede usar Eclipse para hacer esto. Asegúrese de que Employee tenga el foco en la ventana de origen de su IDE Eclipse, luego vaya a **Source > Override/ Implement Methods**. Aparecerá el recuadro de diálogo de la Ilustración 4:

## Ilustración 4. Diálogo de alteración temporal/implementación de métodos



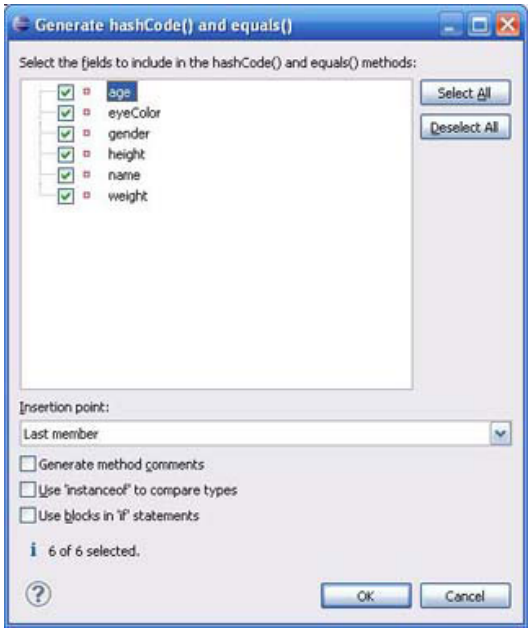

Ha usado este diálogo antes pero, en este caso, usted quiere implementar el método de superclase Object.equals(). Así que, encuentre Object en la lista, verifique el método equals(Object) y haga clic en **OK**. Eclipse generará el código correcto y lo ubicará en su archivo de origen.

Tiene sentido que los dos objetos Employee sean iguales si los estados de esos objetos son iguales. Es decir, son iguales si sus valores, — apellido, nombre, edad, — son iguales.

### Generación automática de equals()

Eclipse puede generar un método equals() por usted en base a las variables de instancias (atributos) que usted ha definido para una clase. Debido a que Employee es una subclase de Person, usted primero generará equals() para Person. En la vista Project Explorer de Eclipse, haga clic derecho en Person y elija **Generate hashCode() and equals()** para acceder al recuadro de diálogo que se muestra en la Ilustración 5:

## Ilustración 5. Diálogo de generación de hashCode() y equals()



Seleccione todos los atributos (como se muestra en la Ilustración 5) y haga clic en **OK**. Eclipse generará un método `equals()` que se parece al del Listado 8:

## Listado 8. Un método equals() generado por Eclipse

```
@Override
public boolean equals(Object obj) {
 if (this == obj)
  return true;
 if (obj == null)
  return false;
 if (getClass() != obj.getClass())
  return false;
 Person other = (Person) obj;
 if (age != other.age)
  return false;
 if (eyeColor == null) {
  if (other.eyeColor != null)
   return false;
 } else if (!eyeColor.equals(other.eyeColor))
  return false;
 if (gender == null) {
  if (other.gender != null)
   return false;
 } else if (!gender.equals(other.gender))
  return false;
 if (height != other.height)
  return false;
 if (name == null) {
  if (other.name != null)
   return false;
 } else if (!name.equals(other.name))
  return false;
 if (weight != other.weight)
  return false;
 return true;
}
```

No se preocupe por el `hashCode()` por ahora, — puede dejarlo o suprimirlo. El método `equals()` generado por Eclipse parece complicado pero lo que hace es muy simple: si el objeto que pasó es el mismo objeto que el del Listado 8, entonces `equals()` retornará `verdadero`. Si el objeto que pasó es nulo, retornará `falso`.

Luego de eso, el método verifica para ver si los objetos de la `Class` son los mismos (es decir que el objeto que pasó debe ser un objeto `Person`). Si eso es verdad, entonces cada valor de atributo del objeto que pasó se verifica para ver si coincide valor por valor con el estado de la instancia de `Person` dada. Si los valores de los atributos son nulos (es decir, que faltan), entonces el `equals()` verificará tantos como pueda y, si esos coinciden, los objetos se considerarán iguales. Tal vez no quiera este comportamiento para cada programa pero funciona para la mayoría de los propósitos.

## Ejercicio: Genere un equals() para Employee

Intente seguir los pasos en Autogenerating `equals()` para generar un `equals()` para `Employee`. Una vez que tiene su `equals()` generado, agregue el siguiente código arriba de él:

```
public static void main(String[] args) {
 Logger l = Logger.getLogger(Employee.class.getName());

 Employee employee1 = new Employee();
 employee1.setName("J Smith");
 Employee employee2 = new Employee();
 employee2.setName("J Smith");
 l.info("Q: employee1 == employee2?      A: " + (employee1 == employee2));
 l.info("Q: employee1.equals(employee2)? A: " + employee1.equals(employee2));
}
```

Si usted ejecuta el código, debería ver la siguiente salida:

```
Apr 19, 2010 17:26:50 AM com.makotogroup.intro.Employee main
INFO: Q: employee1 == employee2?      A: false
Apr 19, 2010 17:26:50 AM com.makotogroup.intro.Employee main
INFO: Q: employee1.equals(employee2)? A: true
```

En este caso, una coincidencia solo en el Nombre fue suficiente para convencer a equals() de que los dos objetos eran iguales. Siéntase libre para agregar más atributos a este ejemplo y vea lo que obtiene.

### Ejercicio: Altere temporalmente toString()

¿Recuerda el método printAudit() del comienzo de esta sección? Si pensaba que estaba trabajando bastante duro, ¡tenía razón! Formatear el estado de un objeto en una String es un patrón tan común que los diseñadores del lenguaje Java lo desarrollaron justo en el Object mismo, en un método llamado (no es ninguna sorpresa) toString(). La implementación predeterminada de toString() no es muy útil pero cada objeto tiene una. En este ejercicio, usted hará la toString() predeterminada un poco más útil.

Si sospecha que Eclipse puede generar un método toString() por usted, está en lo correcto. Retroceda en su Project Explorer y haga clic derecho en la clase Employee, luego elija **Source > Generate toString()...**. Verá un recuadro de diálogo similar al de la Ilustración 5. Elija todos los atributos y haga clic en **OK**. El código generado por Eclipse para Employee se muestra en el Listado 9:

## Listado 9. Un método toString() generado por Eclipse

```
@Override
public String toString() {
 return "Employee [employeeNumber=" + employeeNumber + ", salary=" + salary
 + ", taxpayerIdentificationNumber=" + taxpayerIdentificationNumber
 + "]";
}
```

El código que Eclipse genera para toString no incluye la toString() de la superclase (al ser la superclase de Employee una Person). Puede arreglar eso en solo un momento, usando Eclipse, con esta alteración temporal:

```
@Override
public String toString() {
 return super.toString() +
   "Employee [employeeNumber=" + employeeNumber + ", salary=" + salary
   + ", taxpayerIdentificationNumber=" + taxpayerIdentificationNumber
   + "]";
}
```

La incorporación de toString() hace que printAudit() sea mucho más simple:

```
@Override
public void printAudit(StringBuilder buffer) {
 buffer.append(toString());
}
```

toString() ahora hace el trabajo pesado de formatear el estado actual del objeto y usted simplemente coloca lo que retorna en el StringBuilder y regresa.

Recomiendo implementar toString() siempre en sus clases, aunque sea solo por propósitos de soporte. Es prácticamente inevitable que, en algún punto, usted quiera ver cuál es el estado de un objeto mientras su aplicación se está ejecutando y toString() es un gran modo para hacer eso.

# Excepciones

Ningún programa funciona nunca el 100 por ciento del tiempo y los diseñadores del lenguaje Java sabían esto. En esta sección, aprenda acerca del mecanismo incorporado de la plataforma Java para manejar situaciones donde su código no funcione exactamente como se planeó.

## Conceptos básicos del manejo de excepciones

Una excepción es un evento que ocurre durante la ejecución de un programa y que interrumpe el flujo normal de las instrucciones del programa. El manejo de excepciones es una técnica esencial de la programación Java. En esencia, usted corta su código en un bloque try (que significa "intenta esto y avísame si causa una excepción") y lo usa para catch diversos tipos de excepciones.

Para comenzar con el manejo de excepciones, observe el código en el Listado 10:

## Listado 10. ¿Ve el error?

```
// . . .
public class Employee extends Person {
// . . .
 private static Logger logger;// = Logger.getLogger(Employee.class.getName());

 public static void main(String[] args) {
   Employee employee1 = new Employee();
   employee1.setName("J Smith");
   Employee employee2 = new Employee();
   employee2.setName("J Smith");
   logger.info("Q: employee1 == employee2?      A: " + (employee1 == employee2));
   logger.info("Q: employee1.equals(employee2)? A: " + employee1.equals(employee2));

 }
```

Observe que se ha comentado el inicializador para la variable estática que tiene la referencia de Logger. Ejecute este código y obtendrá la siguiente salida:

```
Excepción en el subproceso "main" java.lang.NullPointerException
 at com.makotogroup.intro.Employee.main(Employee.java:54)
```

Esta salida le está diciendo que usted está intentando hacer referencia a un objeto que está allí, lo cual es un error de desarrollo bastante serio. Afortunadamente, puede usar los bloques try y catch para obtenerlo (junto con un poco de ayuda de finally).

### try, catch y finally

El Listado 11 muestra el código con error del Listado 10 limpio con los bloques de códigos estándar para el manejo de excepciones try, catch y finally:

## Listado 11. Obtener una excepción

```
// . . .
public class Employee extends Person {
// . . .
 private static Logger logger;// = Logger.getLogger(Employee.class.getName());

 public static void main(String[] args) {
   try {
    Employee employee1 = new Employee();
    employee1.setName("J Smith");
    Employee employee2 = new Employee();
    employee2.setName("J Smith");
    logger.info("Q: employee1 == employee2?      A: " + (employee1 == employee2));
    logger.info("Q: employee1.equals(employee2)? A: " + employee1.equals(employee2));
   } catch (NullPointerException npe) {
    // Handle...
    System.out.println("Yuck! Outputting a message with System.out.println() " +
                       "because the developer did something dumb!");
   } finally {
    // Always executes
   }
 }
```

Juntos, los bloques try, catch y finally forman una red para obtener las excepciones. Primero, la sentencia try recorta el código que puede arrojar una excepción. Si lo hace, la ejecución baja inmediatamente al bloque catch o manejador de excepciones. Cuando termina los procesos de probar y obtener, la ejecución continúa con el bloque finally, ya sea que se haya arrojado una excepción o no. Cuando obtiene una excepción, puede intentar recuperarse con gracia de ella o puede salir del programa (o método).

En el Listado 11, el programa se recupera del error, luego imprime un mensaje para reportar lo que sucedió.

## La jerarquía de excepciones

El lenguaje Java incorpora toda una jerarquía de excepciones que consiste en muchos tipos de excepciones agrupados en dos categorías importantes:

- A las **Excepciones verificadas** las verifica el compilador (es decir, el compilador se asegurará de que se manejen en algún lugar de su código).
- A las **Excepciones sin verificar** (también llamadas excepciones de tiempo de ejecución) no las verifica el compilador.

Cuando un programa causa una excepción, se dice que arroja la excepción. Una excepción verificada se declara al compilador por cualquier método con la palabra clave throws en su firma de método. A esto le sigue una lista de excepciones separadas por comas que el método podría arrojar potencialmente durante el curso de su ejecución. Si su código llama a un método que especifica que arroja uno o más tipos de excepciones, debe manejarlo de algún modo o agregar un throws a su firma de método para pasar ese tipo de excepción.

En el caso de una excepción, el tiempo de ejecución del lenguaje Java busca un manejador de excepciones en algún lugar de la pila. Si no encuentra uno al momento en que llega al inicio de la pila, detendrá abruptamente el programa, como vio en el Listado 10.

### Bloques `catch` múltiples

Usted puede tener bloques `catch` múltiples pero se deben estructurar en un modo particular. Si alguna excepción es una subclase de otra excepción, entonces la clase hija se ubica delante de la clase padre en el orden del bloque `catch`. Aquí hay un ejemplo:

```
try {
 // Code here...
} catch (NullPointerException e) {
 // Handle NPE...
} catch (Exception e) {
 // Handle more general exception here...
}
```

En este ejemplo, la `NullPointerException` es una clase hija (eventualmente) de `Exception`, por lo tanto debe ubicarse delante del bloque `Exception catch` más general.

Ha visto solo un pequeño atisbo del manejo de excepciones de Java en este tutorial. El tema podría constituir un tutorial por sí mismo. Vea Temas relacionados para aprender más acerca del manejo de excepciones en programas Java.

## Desarrollo de aplicaciones Java

En esta sección, continuará desarrollando `Person` como una aplicación Java. A lo largo del proceso, tendrá una mejor idea de cómo un objeto, o colección de objetos, evoluciona en una aplicación.

### Elementos de una aplicación Java

Todas las aplicaciones Java necesitan un punto de entrada donde el tiempo de ejecución de Java sepa comenzar a ejecutar códigos. Ese punto de entrada es el método `main()`. Los objetos del dominio normalmente no tienen métodos `main()` pero por lo menos una clase en cada aplicación debe tenerlo.

Ha estado trabajando desde la Parte 1 en el ejemplo de una aplicación de recursos humanos que incluye `Person` y sus subclases `Employee`. Ahora verá lo que sucede cuando agrega una clase nueva a la aplicación.

### Creación una clase controladora

El propósito de una clase controladora (como el nombre insinúa) es "controlar" una aplicación. Observe que este simple controlador para la aplicación de recursos humanos contiene un método `main()`:

```
package com.makotogroup.intro;
public class HumanResourcesApplication {
 public static void main(String[] args) {
 }
}
```

Cree una clase controladora en Eclipse, usando el mismo procedimiento que usó para crear `Person` y `Employee`. Nombre la clase `HumanResourcesApplication`, asegurándose de seleccionar la opción para agregar un método `main()` a la clase. Eclipse generará la clase por usted.

Agregue algunos códigos a su nuevo `main()` para que luzca del siguiente modo:

```
public class Person {
. . .
private final Logger log = Logger.getLogger(Person.class);
. . .
   public static void main(String[] args) {
    Employee e = new Employee();
    e.setName("J Smith");
    e.setEmployeeNumber("0001");
    e.setTaxpayerIdentificationNumber("123-45-6789");
    e.printAudit(log);
   }
. . .
}
```

Ahora inicie la clase `HumanResourcesApplication` y obsérvela ejecutarse. Debería ver esta salida (aquí las barras inclinadas invertidas indican una continuación de línea):

```
Apr 29, 2010 6:45:17 AM com.makotogroup.intro.Person printAudit
INFO: Person [age=0, eyeColor=null, gender=null, height=0, name=J Smith,\
weight=0]Employee [employeeNumber=0001, salary=null,\
taxpayerIdentificationNumber=123-45-6789]
```

Eso es todo lo que hay realmente sobre la creación de una aplicación Java simple. En la siguiente sección, comenzará a observar algunas de las sintaxis y bibliotecas que le ayudarán a desarrollar aplicaciones más complejas.

# Herencia

Usted se ha encontrado con ejemplos de herencia algunas veces ya en este tutorial. Esta sección revisa algunos de los materiales de la Parte 1 sobre la herencia y explica en más detalle cómo funciona la herencia, — incluye jerarquía de herencia, constructores y herencia y abstracción de herencia.

## Cómo funciona la herencia

Las clases en el código Java existen en jerarquías. Las clases que están por encima de una clase dada en una jerarquía son las superclases de esa clase. Esa clase en particular es una subclase de cada clase que está más arriba en la jerarquía. Una subclase hereda de sus superclases. La clase `java.lang.Object` está en la parte superior de la jerarquía de clases, es decir que cada clase Java es una subclase de, y hereda del, `Object`.

Por ejemplo, suponga que usted tiene una clase `Person` que se parece a la del Listado 12:

## Listado 12. Clase Person pública

```
package com.makotogroup.intro;

// . . .
public class Person {
 public static final String GENDER_MALE = "MALE";
 public static final String GENDER_FEMALE = "FEMALE";
 public Person() {
 //Nothing to do...
 }
 private String name;
 private int age;
 private int height;
 private int weight;
 private String eyeColor;
 private String gender;
// . . .

}
```

La clase Person en el Listado 12 hereda implícitamente del Object. Debido a que eso se asume para cada clase, no necesita escribir extends Object para cada clase que usted define. Pero, ¿qué significa decir que una clase hereda de su superclase? Significa simplemente que Person tiene acceso a las variables y métodos expuestos en sus superclases. En este caso, Person puede ver y usar métodos y variables públicas de Object y métodos y variables protegidas de Object.

## Definición de una jerarquía de clases

Ahora suponga que tiene una clase Employee que hereda de Person. Su definición de clase (o gráfico de herencia) se parecería a algo como esto:

```
public class Employee extends Person {

 private String taxpayerIdentificationNumber;
 private String employeeNumber;
 private BigDecimal salary;
// . . .
}
```

### Herencia múltiple versus simple

Los lenguajes como C++ soportan el concepto de herencia múltiple: en cualquier punto de la jerarquía, una clase puede heredar de una o más clases. El lenguaje Java soporta solo la herencia simple, que significa que solo puede usar la palabra clave extends con una sola clase. Por lo tanto, la jerarquía de clases para cualquier clase Java dada consiste siempre en un línea derecha hasta el java.lang.Object.

Sin embargo, el lenguaje Java soporta la implementación de interfaces múltiples en una sola clase, lo que le da un método alternativo de tipos para la herencia simple. Le presentaré las interfaces múltiples más adelante en el tutorial.

El gráfico de herencia Employee insinúa que Employee tiene acceso a todas las variables y métodos públicos y protegidos en Person (porque lo extiende directamente), así como también en Object (porque en realidad extiende también esa clase, aunque indirectamente). Sin embargo, debido a que Employee y Person están en el mismo paquete, Employee también tiene acceso a las variables y métodos privados del paquete (a veces denominados amigables) en Person.

Para profundizar un paso más en la jerarquía de clases, podría crear una tercera clase que extiende Employee:

```
public class Manager extends Employee {
// . . .
}
```

En el lenguaje Java, cualquier clase puede tener, como mucho, una superclase pero una clase puede tener cualquier cantidad de subclases. Eso es lo más importante de recordar sobre la jerarquía de herencia en el lenguaje Java.

## Constructores y herencia

Los constructores no son plenos miembros orientados a objetos, por lo tanto no son heredados. En cambio, usted debe implementarlos explícitamente en subclases. Antes de eso, revisaré algunas reglas básicas sobre cómo se definen e invocan los constructores.

### Conceptos básicos sobre los constructores

Recuerde que un constructor siempre tiene el mismo nombre que la clase que suele construir y no tiene un tipo de retorno. Por ejemplo:

```
public class Person {
 public Person() {
 }
}
```

Cada clase tiene por lo menos un constructor y, si no define explícitamente un constructor para su clase, el compilador generará uno por usted, denominado el constructor predeterminado. La definición de la clase anterior y ésta son idénticas en el modo en que funcionan:

```
public class Person {
}
```

### Invocación de un constructor de superclase

Para invocar a un constructor de superclase distinto del constructor predeterminado, debe hacerlo explícitamente. Por ejemplo, suponga que `Person` tiene un constructor que toman el nombre del objeto `Person` que se está creando. Desde el constructor predeterminado de `Employee`, usted podría invocar al constructor `Person` que se muestra en el Listado 13:

## Listado 13. Inicialización de un nuevo `Employee`

```
public class Person {
 private String name;
 public Person() {
 }
 public Person(String name) {
   this.name = name;
 }
}

// Meanwhile, in Employee.java
public class Employee extends Person {
 public Employee() {
   super("Elmer J Fudd");
 }
}
```

Sin embargo, probablemente usted nunca quiera inicializar un nuevo objeto `Employee` de este modo. Hasta que se sienta más cómodo con los conceptos orientados a objetos, y la sintaxis Java en general, es una buena idea implementar constructores de superclases en subclases, si piensa que los necesitará, e invocarlos homogéneamente El Listado 14 define un constructor en `Employee` que se parece al de `Person` para que coincidan. Es mucho menos confuso desde el punto de vista del mantenimiento.

## Listado 14. Invocación homogénea de una superclase

```
public class Person {
 private String name;
 public Person(String name) {
   this.name = name;
 }
}
// Meanwhile, in Employee.java
public class Employee extends Person {
 public Employee(String name) {
   super(name);
 }
}
```

## Declaración de un constructor

Lo primero que hace un constructor es invocar al constructor predeterminado de su superclase inmediata, a menos que usted, — en la primera línea de código del constructor, — invoque un constructor diferente. Por ejemplo, estas dos declaraciones son funcionalmente idénticas, así que elija una:

```
public class Person {
 public Person() {
 }
}
// Meanwhile, in Employee.java
public class Employee extends Person {
 public Employee() {
 }
}
```

O:

```
public class Person {
 public Person() {
 }
}
// Meanwhile, in Employee.java
public class Employee extends Person {
 public Employee() {
   super();
 }
}
```

## Constructores sin argumentos

Si usted proporciona un constructor alternativo, debe proporcionar explícitamente el constructor predeterminado, sino no estará disponible. Por ejemplo, el siguiente código le daría un error de compilación:

```
public class Person {
 private String name;
 public Person(String name) {
   this.name = name;
 }
}
// Meanwhile, in Employee.java
public class Employee extends Person {
 public Employee() {
 }
}
```

Este ejemplo no tiene un constructor predeterminado porque proporciona un constructor alternativo sin incluir explícitamente el constructor predeterminado. Esta es la razón por la que el constructor predeterminado a veces se denomina el constructor sin argumento (o no-arg); debido a que hay condiciones bajo las cuales no se incluye, no es realmente predeterminado.

### Cómo los constructores invocan a constructores

Un constructor puede invocar a otro constructor de adentro de una clase al usar la palabra clave `this`, junto con una lista de argumentos. Del mismo modo que `super()`, la llamada `this()` debe ser la primera línea del constructor. Por ejemplo:

```
public class Person {
 private String name;
 public Person() {
   this("Some reasonable default?");
 }
 public Person(String name) {
   this.name = name;
 }
}
// Meanwhile, in Employee.java
```

Verá este modismo frecuentemente, en el que un constructor delega a otro, al pasar algún valor predeterminado si se invoca ese constructor. También es una excelente forma de agregar un constructor nuevo a una clase mientras se minimiza el impacto en el código que ya usa un constructor más antiguo.

### Niveles de acceso de los constructores

Los constructores pueden tener cualquier nivel de acceso que usted quiera y se imponen ciertas reglas de visibilidad. La Tabla 1 resume las reglas de acceso de los constructores:

## Tabla 1. Reglas de acceso de los constructores

| Modificador de acceso de los constructores | Descripción |
|---|---|
| `público` | Cualquier clase puede invocar un constructor |
| `protegido` | Cualquier clase en el mismo paquete o cualquier subclase puede invocar un constructor. |
| Sin modificador (privado del paquete) | Cualquier clase en el mismo paquete puede invocar un constructor. |
| `privado` | Solo la clase en que se definió el constructor puede invocar a un constructor. |

Tal vez pueda pensar en casos de uso en los que los constructores se declararían `protegidos` o incluso privados del paquete pero, ¿de qué modo es un constructor `privado` útil? He usado constructores privados cuando no

quería permitir una creación directa de un objeto por medio de la palabra clave `new` al implementar, digamos, el patrón Factory (vea Temas relacionados). En ese caso, se usaría un método estático para crear instancias de la clase y se le permitiría a ese método, al estar incluido en la clase misma, invocar el constructor privado:

## Herencia y abstracción

Si una subclase altera temporalmente a un método de una superclase, ese método está esencialmente oculto porque llamar a ese método por medio de una referencia a la subclase invoca a la versión de subclase del método, no a la versión de la superclase. Esto no quiere decir que el método de la superclase ya no sea accesible. La subclase puede invocar el método de superclase al prologuear el nombre del método con la palabra clave `super` (y a diferencia de lo que sucede con las reglas del constructor, esto se puede hacer desde cualquier línea en el método de subclase o incluso en un método completamente diferente). De forma predeterminada, un programa Java llamará al método de subclase si se invoca por medio de una referencia a la subclase.

Lo mismo corresponde para las variables, siempre que el interlocutor tenga acceso a la variable (es decir, la variable es visible al código que intenta acceder a ella). Esto puede causarle un sinfín de dolor mientras gana competencia en la programación Java. Sin embargo, Eclipse proporcionará varios avisos de que está ocultando una variable de una superclase o de que una llamada de método no llamará lo que usted piensa que hará.

En un contexto OOP, abstracción se refiere a generalizar datos o comportamientos a un tipo más alto en la jerarquía de herencia que la clase actual. Cuando usted mueve variables o métodos de una subclase a una superclase, se dice que está abstrayendo esos miembros. La razón principal para hacer esto es reutilizar un código común al empujarlo tanto alto como se pueda en la jerarquía. Tener un código común en un lugar hace que sea más fácil de mantener.

### Clases y métodos abstractos

Hay momentos en los que querrá crear clases que solo sirven como abstracciones y no necesariamente tienen que crearse alguna vez como instancias. Dichas clases se denominan clases abstractas. Del mismo modo, descubrirá que hay momentos en los que ciertos métodos tienen que implementarse de forma diferente para cada subclase que implemente la superclase. Dichos métodos son métodos abstractos. Aquí hay algunas reglas básicas para clases y métodos abstractos:

- Cualquier clase se puede declarar `abstracta`.
- No se pueden crear instancias de las clases abstractas.
- Un método abstracto no puede contener un cuerpo del método.
- Cualquier clase con un método abstracto debe declararse `abstracta`.

### Uso de la abstracción

Suponga que usted no quiere permitir que la clase `Employee` se cree como instancia de modo directo. Simplemente la declara con la palabra clave `abstract` y listo:

```
public abstract class Employee extends Person {
// etc.
}
```

Si usted intenta ejecutar este código, obtendrá un error de compilación:

```
public void someMethodSomwhere() {
 Employee p = new Employee();// compile error!!
}
```

El compilador reclama que el `Employee` es abstracto y no se puede crear como instancia.

### El poder de la abstracción

Suponga que necesita un método para examinar el estado de un objeto `Employee` y que se asegure que sea válido. Esta necesidad parecería ser común a todos los objetos `Employee` pero se comportaría de modo bastante diferente entre todas las subclases potenciales que no habría potencial para la reutilización. En ese caso, se declara el método `validate()` como `abstracto` (lo que obliga a todas las subclases a implementarlo):

```
public abstract class Employee extends Person {
 public abstract boolean validate();
}
```

A cada subclase directa de `Employee` (tal como `Manager`) ahora se le requiere implementar el método `validate()`. Sin embargo, una vez que una subclase ha implementado el método `validate()`, ninguna de sus subclases necesita implementarlo.

Por ejemplo, suponga que tiene un objeto `Executive` que amplía a `Manager`. Esta definición sería perfectamente válida:

```
public class Executive extends Manager {
 public Executive() {
 }
}
```

### Cuándo (no) abstraer: Dos reglas

Como primera regla general, no abstraiga en su diseño inicial. Usar clases abstractas prematuramente en el diseño lo obliga a recorrer cierta ruta y eso puede restringir su aplicación. Recuerde, el comportamiento común (que es todo el punto de tener clases abstractas) siempre puede refactorizarse aun más en el gráfico de herencia. Casi siempre es mejor hacer esto una vez que haya descubierto que sí lo necesita. Eclipse tiene un maravilloso soporte para la refactorización.

En segundo lugar, con lo poderosos que son, resístase al uso de las clases abstractas cuando pueda. A menos que sus superclases contengan muchos comportamientos comunes, y por sí solos no son realmente significativos, deje que permanezcan no abstractas. Los profundos gráficos de herencia pueden hacer que el mantenimiento de códigos sea difícil. Considere el equilibrio entre las clases que son muy grandes y los códigos plausibles de ser mantenidos.

## Asignaciones: Clases

Cuando asigne una referencia de una clase a una variable de un tipo que pertenece a otra clase, puede hacerlo pero hay reglas. Observemos este ejemplo:

```
Manager m = new Manager();
Employee e = new Employee();
Person p = m; // okay
p = e; // still okay
Employee e2 = e; // yep, okay
e = m; // still okay
e2 = p; // wrong!
```

La variable de destino debe ser de un supertipo de la clase que pertenece a la referencia de origen o sino el compilador le dará un error. Básicamente, lo que sea que se encuentre al lado derecho de la asignación debe ser una subclase o la misma clase de lo que está a la izquierda. Si no es así, es posible que las asignaciones de los objetos con diferentes gráficos de herencia (tales como `Manager` y `Employee`) se asignen a una variable del tipo incorrecto. Considere este ejemplo:

```
Manager m = new Manager();
Person p = m; // so far so good
Employee e = m; // okay
Employee e = p; // wrong!
```

Mientras un `Employee` es una `Person`, definitivamente no es un `Manager` y el compilador fuerza esto.

# Interfaces

En esta sección, comience a aprender acerca de las interfaces y comience a usarlas en su código Java.

## Definición de una interfaz

Una interfaz es un conjunto denominado de comportamientos (o elementos de datos contantes) para el cual un implementador debe proporcionar códigos. Una interfaz especifica el comportamiento que la implementación proporciona pero no cómo se realiza.

Definir una interfaz es sencillo:

```
public interface interfaceName {
 returnType methodName( argumentList );
}
```

Una declaración de interfaz parece una declaración de clases, excepto que usted usa la palabra clave `interface`. Puede nombrar la interfaz del modo que quiera (sujeto a las reglas del lenguaje) pero, por convenio, los nombres de interfaces se parecen a los nombres de clases.

Los métodos definidos en una interfaz no tienen cuerpo de método. El implementador de la interfaz es responsable de proporcionar el cuerpo de método (del mismo modo que con los métodos abstractos).

Defina las jerarquías de las interfaces, como lo hace para las clases, con la excepción de que una sola clase puede implementar tantas interfaces como quiera. (Recuerde, una clase puede ampliar solo una clase). Si una clase amplía a otra e implementa una interfaz o interfaces, entonces las interfaces se enumeran luego de la clase ampliada, como a continuación:

```
public class Manager extends Employee implements BonusEligible, StockOptionRecipient {
// Etc...
}
```

### Interfaces marcadoras

Una interfaz no necesita tener ningún tipo de cuerpo. De hecho, la siguiente definición es perfectamente aceptable:

```
public interface BonusEligible {
}
```

En términos generales, dichas interfaces se denominan interfaces marcadoras porque marcan una clase que implementa esa interfaz pero no ofrece ningún comportamiento explícito especial.

Una vez que sabe todo eso, en realidad definir una interfaz es fácil:

```
public interface StockOptionRecipient {
 void processStockOptions(int numberOfOptions, BigDecimal price);
}
```

## Implementación de interfaces

Para usar una interfaz, la implementa, que simplemente significa proporcionar un cuerpo de método, que a su vez proporciona el comportamiento para cumplir el contrato de la interfaz. Eso se hace con la palabra clave `implements`:

```
public class className extends superclassName implements interfaceName {
// Class Body
}
```

Suponga que usted implementa la interfaz `StockOptionRecipient` en la clase `Manager`, como se muestra en el Listado 15:

## Listado 15. Implementación de una interfaz

```
public class Manager extends Employee implements StockOptionRecipient {
 public Manager() {
 }
 public void processStockOptions (int numberOfOptions, BigDecimal price) {
   log.info("I can't believe I got " + number + " options at $" +
          price.toPlainString() + "!");  }
}
```

Cuando implementa la interfaz, usted proporciona el comportamiento para el o los métodos en la interfaz. Debe implementar los métodos con firmas que coincidan con las de la interfaz, con la incorporación del modificador de acceso `public`.

### Generación de interfaces en Eclipse

Eclipse puede generar fácilmente la firma correcta del método por usted si decide que una de sus clases debería implementar una interfaz. Solo cambie la firma de la clase para implementar la interfaz. Eclipse pone una línea ondulante roja debajo de la clase y la distingue así como un error porque la clase no proporciona el o los métodos en la interfaz. Haga clic en el nombre de la clase con su ratón, presione **Ctrl + 1** y Eclipse le sugerirá "arreglos rápidos". De estos, elija **Add Unimplemented Methods** y Eclipse generará los métodos por usted, ubicándolos al final del archivo de origen.

Una clase abstracta puede declarar que implementa una interfaz particular pero no se le requiere que implemente todos los métodos en esa interfaz. Esto sucede porque no se requieren clases abstractas para proporcionar implementaciones para todos los métodos que afirman implementar. Sin embargo, la primera clase concreta (es decir, la primera que se puede crear como instancia) debe implementar todos los métodos que la jerarquía no implementa.

### Uso de interfaces

Una interfaz define un nuevo tipo de datos de referencia, lo que significa que usted puede referirse a una interfaz en cualquier lugar en el que usted se referiría a una clase. Esto incluye los casos en que usted declara una variable de referencia o proyecta de un tipo a otro, como se muestra en el Listado 16.

### Listado 16. Asignación de una nueva instancia Manager a una referencia StockOptionEligible

```
public static void main(String[] args) {
 StockOptionEligible soe = new Manager();// perfectly valid
 calculateAndAwardStockOptions(soe);
 calculateAndAwardStockOptions(new Manager());// works too
}
. . .
public static void calculateAndAwardStockOptions(StockOptionEligible soe) {
 BigDecimal reallyCheapPrice = BigDecimal.valueOf(0.01);
 int numberOfOptions = 10000;
 soe.processStockOptions(numberOfOptions, reallyCheapPrice);
}
```

Como puede ver, es perfectamente válido asignar una nueva instancia Manager a una referencia StockOptionEligible, así como pasar una nueva instancia Manager a un método que espera una referencia StockOptionEligible.

### Asignaciones: Clases

Cuando asigne una referencia de una clase que implemente una interfaz a una variable de un tipo de interfaz, puede hacerlo pero hay reglas. Desde el Listado 16, vemos que asignar una instancia Manager a una referencia de variable StockOptionEligible es perfectamente válido. La razón es que la clase Manager implementa esa interfaz. Sin embargo, la siguiente asignación no sería válida:

```
 Manager m = new Manager();
 StockOptionEligible soe = m; //okay
 Employee e = soe; // Wrong!
```

Debido a que Employee es un supertipo de Manager, al principio esto puede parecer bien, pero no lo es. Debido a que Manager es una especialización de Employee, es *diferente* y, en este caso particular, implementa una interfaz que Employee no implementa.

Las asignaciones como estas siguen las reglas de asignación que vimos en Inheritance. Y del mismo modo que sucede con las clases, solo puede asignar una referencia de interfaz a una variable del mismo tipo o a un tipo de superinterfaz.

## Clases anidadas

En esta sección, aprenda acerca de las clases anidadas y dónde y cómo usarlas.

## Dónde usar las clases anidadas

Como su nombre lo sugiere, una clase anidada es una clase que se define dentro de otra clase. Aquí hay una clase anidada:

```
public class EnclosingClass {
. . .
 public class NestedClass {
 . . .

 }
}
```

Al igual que las variables y los métodos miembro, las clases Java también se pueden definir en cualquier ámbito, incluidos los public, private o protected. Las clases anidadas pueden ser útiles cuando usted quiere manejar procesos internos dentro de su clase de un modo orientado a objetos, pero esta funcionalidad está limitada a la clase en donde la necesita.

Normalmente, usted usará una clase anidada para casos en los que necesite una clase que esté asociada firmemente con la clase en que se define. Una clase anidada tiene acceso a los datos privados dentro de su clase cerrada pero esto conlleva algunos efectos secundarios que no son evidentes cuando comienza a trabajar con clases anidadas (o internas).

## Ámbito en las clases anidadas

Debido a que una clase anidada tiene un ámbito, está sujeta a las reglas del ámbito. Por ejemplo, solo se puede acceder a una variable miembro por medio de una instancia de la clase (un objeto). Lo mismo sucede con una clase anidada.

Suponga que tiene la siguiente relación entre un Manager y una clase anidada llamada DirectReports, que es una colección de los Employees que presentan informes a ese Manager:

```
public class Manager extends Employee {
 private DirectReports directReports;
 public Manager() {
   this.directReports = new DirectReports();
 }
. . .
 private class DirectReports {
 . . .
 }
}
```

Al igual que cada objeto Manager representa a un ser humano único, el objeto DirectReports representa una colección de las personas reales (empleados) que presentan informes a un gestor. DirectReports diferirá de un Manager a otro. En este caso, tiene sentido que uno sólo haga referencia a la clase anidada DirectReports en el contexto de su instancia de inclusión de Manager, por lo tanto la he hecho private.

### Clases anidadas públicas

Debido a que es private, solo el Manager puede crear una instancia de DirectReports. Pero suponga que usted quisiera darle a una entidad externa la habilidad para crear instancias de DirectReports. En este caso, parece

que usted podría darle a la clase DirectReports un ámbito public y entonces cualquier código externo podría crear instancias de DirectReports, como se muestra en el Listado 17:

## Listado 17. Creación de instancias de DirectReports: Primer intento

```
public class Manager extends Employee {
 public Manager() {
 }
. . .
 private class DirectReports {
 . . .
 }
}
//
public static void main(String[] args) {
 Manager.DirectReports dr = new Manager.DirectReports();// This won't work!
}
```

El código en el Listado 17 no funciona y probablemente se está preguntando por qué. El problema (y también su solución) está en el modo en que se define DirectReports dentro de Manager y en las reglas del ámbito.

### Revisión de las reglas del ámbito

Si usted tuviera una variable miembro de Manager, esperaría que el compilador le solicitara que tuviera una referencia a un objeto Manager antes de que pudiera hacer referencia a él, ¿verdad? Bueno, lo mismo se sucede con DirectReports, por lo menos como usted lo definió en el Listado 17.

Para crear una instancia de una clase anidada pública, se usa una versión especial del operador new. En combinación con una referencia a alguna instancia de inclusión de una clase externa, new le permite crear una instancia de la clase anidada:

```
public class Manager extends Employee {
 public Manager() {
 }
. . .
 private class DirectReports {
 . . .
 }
}
// Meanwhile, in another method somewhere...
public static void main(String[] args) {
 Manager manager = new Manager();
 Manager.DirectReports dr = manager.new DirectReports();
}
```

Observe que la sintaxis exige una referencia a la instancia de inclusión, más un punto y la palabra clave new, seguida de la clase que usted quiere crear.

## Clases internas estáticas

Por momentos, querrá crear una clase que esté asociada firmemente (conceptualmente) a una clase, pero donde las reglas del ámbito estén un tanto relajadas, que no requieran una referencia a una instancia de inclusión. Allí es donde las clases internas estáticas entran en juego. Un ejemplo común de esto es implementar un Comparator, que se usa para comparar dos instancias de la misma clase, normalmente con el propósito de ordenar (o clasificar) las clases:

```
public class Manager extends Employee {
. . .
 public static class ManagerComparator implements Comparator<Manager> {
  . . .
 }
}
// Meanwhile, in another method somewhere...
public static void main(String[] args) {
 Manager.ManagerComparator mc = new Manager.ManagerComparator();
 . . .
}
```

En este caso, no necesita una instancia de inclusión. Las clases internas estáticas actúan como sus homólogas clases Java regulares y realmente solo deberían usarse cuando usted necesite asociar firmemente una clase con su definición. Claramente, en el caso de una clase de programa de utilidad como `ManagerComparator`, crear una clase externa es innecesario y abarrota potencialmente su base de códigos. Definir dichas clases como clases internas estáticas es lo que se tiene que hacer.

## Clases internas anónimas

El lenguaje Java le permite declarar clases prácticamente en cualquier lugar, incluso en el medio de un método si es necesario, e incluso sin proporcionar un nombre para la clase. Esto es básicamente un truco de compilador pero hay momentos en que las clases internas anónimas son extremadamente útiles de tener.

El Listado 18 se desarrolla sobre el ejemplo en el Listado 15 y agrega un método predeterminado para el manejo de los tipos de `Employee` que no son `StockOptionEligible`:

## Listado 18. Manejo de los tipos de `Employee` que no son `StockOptionEligible`

```
public static void main(String[] args) {
 Employee employee = new Manager();// perfectly valid
 handleStockOptions(employee);
 employee = new Employee();// not StockOptionEligible
 handleStockOptions(employee);
}
. . .
private static void handleStockOptions(Employee e) {
 if (e instanceof StockOptionEligible) {
  calculateAndAwardStockOptions((StockOptionEligible)e);
 } else {
  calculateAndAwardStockOptions(new StockOptionEligible() {
    @Override
    public void awardStockOptions(int number, BigDecimal price) {
      log.info("Sorry, you're not StockOptionEligible!");
    }
  });
 }
}
. . .
private static void calculateAndAwardStockOptions(StockOptionEligible soe) {
 BigDecimal reallyCheapPrice = BigDecimal.valueOf(0.01);
 int numberOfOptions = 10000;
 soe.processStockOptions(numberOfOptions, reallyCheapPrice);

}
```

En este ejemplo, se proporciona una implementación de la interfaz `StockOptionEligible` al usar una clase interna anónima para instancias de `Employee` que no implementan esa interfaz. Las clases internas anónimas también son útiles para implementar métodos de devolución de llamada y también en el manejo de eventos.

# Expresiones regulares

Una expresión regular es esencialmente un patrón para describir un conjunto de cadenas que comparten ese patrón. Si usted es un programador Perl, debería sentirse como en su hogar con la sintaxis de patrón de expresión regular (regex) en el lenguaje Java. Sin embargo, si no está acostumbrado a la sintaxis de expresiones regulares, puede parecer raro. En esta sección, usted comienza con el uso de expresiones regulares en sus programas Java.

## La API de expresiones regulares

Aquí hay un conjunto de cadenas que tienen algunos aspectos en común:

- Una cadena
- Una cadena más larga
- Una cadena mucho más larga

Observe que cada una de estas cadenas comienza con a y termina con string. La API de expresiones regulares Java (vea Temas relacionados) lo ayuda a sacar estos elementos, ver el patrón entre ellos y realizar cosas importantes con la información que ha averiguado.

La API de expresiones regulares tiene tres clases principales que usará casi todo el tiempo:

- `Pattern` describe un patrón de cadena.
- `Matcher` prueba una cadena para ver si coincide con el patrón.
- `PatternSyntaxException` le dice que algo acerca del patrón que usted intentó definir no fue aceptable.

Comenzará trabajando con un patrón simple de expresiones regulares que usa estas clases pronto. Sin embargo, antes de que haga eso, observará la sintaxis del patrón regex.

## Sintaxis del patrón regex

Un patrón regex describe la estructura de la cadena que la expresión intentará encontrar en una cadena de entrada. Aquí es donde las expresiones regulares pueden parecer un poco extrañas. De todos modos, una vez que entiende la sintaxis, resulta más fácil de descifrar. La Tabla 2 enumera algunas de las construcciones regex más comunes que usted usará en cadenas de patrones:

## Tabla 2. Construcciones regex comunes

| Construcción regex | ¿Qué califica como una coincidencia? |
|---|---|
| . | Cualquier carácter |
| ? | Cero (0) o uno (1) de lo que vino anteriormente |
| * | Cero (0) o más de lo que vino anteriormente |
| + | Uno (1) o más de lo que vino anteriormente |
| [] | Un rango de caracteres o dígitos |
| ^ | Negación de lo que sea que siga (es decir, "notwhatever") |
| \d | Cualquier dígito (alternativamente, [0-9]) |

| | |
|---|---|
| \D | Cualquiera que no sea un dígito (alternativamente, [^0-9]) |
| \s | Cualquier carácter de espacio en blanco (alternativamente, [\n\t\f\r]) |
| \S | Cualquier carácter sin espacio en blanco (alternativamente, [^\n\t\f\r]) |
| \w | Cualquier carácter alfanumérico (alternativamente, [a-zA-Z_0-9]) |
| \W | Cualquier carácter no alfanumérico (alternativamente, [^\w]) |

Las primeras pocas construcciones se llaman cuantificadoras porque cuantifican lo que está antes que ellas. Las construcciones como \d son clases de caracteres predefinidas. Cualquier carácter que no tenga un significado especial en un patrón es un literal y coincide consigo mismo.

### Coincidencia de patrón

Con la sintaxis de patrón de la Tabla 2, puede ocuparse del simple ejemplo del Listado 19, con el uso de clases en la API de expresiones regulares Java:

## Listado 19. Coincidencia de patrón con regex

```
Pattern pattern = Pattern.compile("a.*string");
Matcher matcher = pattern.matcher("a string");
boolean didMatch = matcher.matches();
Logger.getAnonymousLogger().info (didMatch);
int patternStartIndex = matcher.start();
Logger.getAnonymousLogger().info (patternStartIndex);
int patternEndIndex = matcher.end();
Logger.getAnonymousLogger().info (patternEndIndex);
```

En primer lugar, el Listado 19 crea una clase Pattern al llamar a compile(), que es un método estático en Pattern, con una serie literal que representa el patrón que usted quiere hacer coincidir. Ese literal usa la sintaxis de patrón regex. En este ejemplo, la traducción al español del patrón es la siguiente:

Encuentre una cadena de la forma a seguida por cero caracteres o más, seguidos por la cadena.

### Métodos para la coincidencia

Luego, el Listado 19 llama a matcher() en el Pattern. Esa llamada crea una instancia Matcher. Cuando eso sucede, el Matcher busca la cadena que usted pasó para las coincidencias en comparación con la cadena de patrón que usó cuando creó el Pattern.

Cada cadena del lenguaje Java es una colección indexada de caracteres, que comienza desde 0 y termina con la longitud de la cadena menos uno. El Matcher analiza la cadena, que comienza desde 0, y busca coincidencias con ella. Luego de que se completa ese proceso, el Matcher contiene información acerca de las coincidencias encontradas (o no encontradas) en la cadena de entrada. Puede acceder a esa información a llamar a diversos métodos en el Matcher:

- matches() le dice si toda la secuencia de entrada fue una coincidencia exacta para el patrón.
- start() le dice el valor de índice en la cadena donde comienza la cadena coincidente.
- end() le dice el valor de índice en la cadena donde termina la cadena coincidente, más uno.

El Listado 19 encuentra una sola coincidencia al comenzar en 0 y terminar en 7. De este modo, la llamada a matches() retorna true, la llamada a start() retorna 0 y la llamada a end() retorna 8.

### lookingAt() versus matches()

Si existieran más elementos en su cadena que los caracteres en el patrón que usted buscó, podría usar lookingAt() en lugar de matches(). lookingAt() busca coincidencias de subcadenas para un patrón dado. Por ejemplo, considere la siguiente cadena:

```
Aquí hay una cadena que tiene más que solo el patrón.
```

Podría buscarla para a.*string y obtener una coincidencia si usa lookingAt(). Pero si usa matches(), retornaría false porque la cadena conlleva más que solo lo que está en el patrón.

## Patrones complejos en regex

Las búsquedas simples son fáciles con las clases regex, pero también puede hacer algo altamente sofisticado con la API de expresiones regulares.

Un wiki, como seguramente sabe, es un sistema basado en la web que permite a los usuarios modificar páginas. Los wikis se basan casi por completo en expresiones regulares. Su contenido se basa en las entradas de cadenas de los usuarios, lo cual se analiza y formatea con el uso de expresiones regulares. Cualquier usuario puede crear un enlace a otro tema en un wiki al ingresar una palabra wiki, que normalmente es una serie de palabras concatenadas, cada una de las cuales comienza con una letra mayúscula, como la siguiente:

```
MyWikiWord
```

Al saber eso acerca de los wikis, asuma la siguiente cadena:

```
Aquí hay una WikiWord seguida de AnotherWikiWord y luego YetAnotherWikiWord.
```

Podría buscar palabras wikis en esta cadena con un patrón regex como el siguiente:

```
[A-Z][a-z]*([A-Z][a-z]*)+
```

Y aquí hay un código para buscar palabras wikis:

```
String input = "Here is a WikiWord followed by AnotherWikiWord, then SomeWikiWord.";
Pattern pattern = Pattern.compile("[A-Z][a-z]*([A-Z][a-z]*)+");
Matcher matcher = pattern.matcher(input);
while (matcher.find()) {
 Logger.getAnonymousLogger().info("Found this wiki word: " + matcher.group());
}
```

Ejecute este código y debería ver las tres palabras wikis en su consola.

### Sustitución de cadenas

Buscar coincidencias es útil pero también puede manipular cadenas una vez que encuentra una coincidencia para ellas. Puede hace eso al sustituir cadenas coincidentes con algo más, del mismo modo en que podría buscar algún texto en un programa procesador de textos y sustituirlo con otro texto. Matcher tiene un par de métodos para sustituir elementos de cadenas:

- replaceAll() sustituye todas las coincidencias con una cadena especificada.

- replaceFirst() sustituye solo la primera coincidencia con una cadena especificada.

Usar métodos replace de Matcher es sencillo:

```
String input = "Here is a WikiWord followed by AnotherWikiWord, then SomeWikiWord.";
Pattern pattern = Pattern.compile("[A-Z][a-z]*([A-Z][a-z]*)+");
Matcher matcher = pattern.matcher(input);
Logger.getAnonymousLogger().info("Before: " + input);
String result = matcher.replaceAll("replacement");
Logger.getAnonymousLogger().info("After: " + result);
```

Este código encuentra palabras wikis, como antes. Cuando el Matcher encuentra una coincidencia, sustituye el texto de la palabra wiki con su sustitución. Cuando usted ejecuta este código, debería ver lo siguiente en su consola:

```
Antes: Aquí hay una WikiWord seguida de AnotherWikiWord y luego SomeWikiWord.
Después: Aquí hay una sustitución seguida de sustitución y luego sustitución.
```

Si usted hubiera usado replaceFirst(), habría visto lo siguiente:

```
Antes: Aquí hay una PalabraWiki seguida de AnotherWikiWord y luego SomeWikiWord.
Después: Aquí hay una sustitución seguida de AnotherWikiWord y luego SomeWikiWord.
```

## Coincidencia y manipulación de grupos

Cuando busca coincidencias con un patrón regex, puede obtener información sobre lo que encontró. Ha visto algo de eso con los métodos start() y end() en el Matcher. Pero también es posible hacer referencia a coincidencias al capturar grupos.

En cada patrón, usted normalmente crea grupos al encerrar partes del patrón en paréntesis. Los grupos se enumeran de izquierda a derecha, comenzando desde 1 (el grupo 0 representa toda la coincidencia). El código en el Listado 20 sustituye cada palabra wiki con una cadena que "envuelve" la palabra:

## Listado 20. Coincidencia de grupos

```
String input = "Here is a WikiWord followed by AnotherWikiWord, then SomeWikiWord.";
Pattern pattern = Pattern.compile("[A-Z][a-z]*([A-Z][a-z]*)+");
Matcher matcher = pattern.matcher(input);
Logger.getAnonymousLogger().info("Before: " + input);
String result = matcher.replaceAll("blah$0blah");
Logger.getAnonymousLogger().info("After: " + result);
```

Ejecute este código y debería obtener la siguiente salida de la consola:

```
Antes: Aquí hay una PalabraWiki seguida de AnotherWikiWord y luego SomeWikiWord.
Después: Aquí hay una blahWikiWordblah seguida de blahAnotherWikiWordblah y luego blahSomeWikiWordblah.
```

### Otro abordaje a la coincidencia de grupos

El Listado 20 hace referencia a toda la coincidencia al incluir $0 en la cadena de sustitución. Cualquier parte de una cadena de sustitución de la forma $int se refiere al grupo identificado por el entero (por lo tanto, $1 se refiere al grupo 1 y así sucesivamente). En otras palabras, $0 es equivalente a matcher.group(0);.

Usted podría cumplir el mismo objetivo de sustitución al usar algunos otros métodos. En lugar de llamar a replaceAll(), podría hacer lo siguiente:

```
StringBuffer buffer = new StringBuffer();
while (matcher.find()) {
 matcher.appendReplacement(buffer, "blah$0blah");
}
matcher.appendTail(buffer);
Logger.getAnonymousLogger().info("After: " + buffer.toString());
```

Y obtendría el mismo resultado:

```
Antes: Aquí hay una PalabraWiki seguida de AnotherWikiWord y luego SomeWikiWord.
Después: Aquí hay una blahWikiWordblah seguida de blahAnotherWikiWordblah y luego blahSomeWikiWordblah.
```

# Genéricos

La introducción de los genéricos en JDK 5 marcó un gran avance para el lenguaje Java. Si usted ha usado plantillas C++, descubrirá que los genéricos en el lenguaje Java son similares, pero no exactamente iguales. Si no ha usado plantillas C++, entonces no se preocupe: Esta sección ofrece una introducción de alto nivel a los genéricos en el lenguaje Java.

## ¿Qué son los genéricos?

Con el release de JDK 5, el lenguaje Java de repente germinó una nueva sintaxis extraña y emocionante. Básicamente, algunas clases JDK familiares se sustituyeron por sus equivalentes genéricos.

Generics es un mecanismo compilador por el cual usted puede crear (y usar) tipos de elementos (tales como clases e interfaces) de un modo genérico al cosechar el código común y parametrizar (o hacer plantillas) el resto.

### Genéricos en funcionamiento

Para ver la diferencia que hacen los genéricos, considere el ejemplo de una clase que ha estado en el JDK por mucho tiempo: java.util.ArrayList, que es una Lista de objetos que está respaldada por una matriz.

El Listado 12 muestra cómo java.util.ArrayList se crea como instancia:

## Listado 21. Instanciación de una ArrayList

```
ArrayList arrayList = new ArrayList();
arrayList.add("A String");
arrayList.add(new Integer(10));
arrayList.add("Another String");
// So far, so good.
```

Como puede ver, la ArrayList es heterogénea: contiene dos tipos de String y un tipo de Integer. Antes de JDK 5, no había nada en el lenguaje Java para restringir este comportamiento, lo cual causó muchos errores de codificación. En el Listado 21, por ejemplo, todo parece estar bien hasta el momento. Pero, ¿qué tal si se accede a los elementos de la ArrayList, lo que el Listado 22 intenta hacer?

## Listado 22. Un intento para acceder a los elementos en la ArrayList

```
ArrayList arrayList = new ArrayList();
arrayList.add("A String");
arrayList.add(new Integer(10));
arrayList.add("Another String");
// So far, so good.
*processArrayList(arrayList);
*// In some later part of the code...
private void processArrayList(ArrayList theList) {
   for (int aa = 0; aa < theList.size(); aa++) {
     // At some point, this will fail...
     String s = (String)theList.get(aa);
   }
}
```

Sin conocimiento previo de lo que está en la `ArrayList`, se debe verificar el elemento al que quiere acceder para ver si puede manejar su tipo o enfrentar una posible `ClassCastException`.

Con los genéricos, usted puede especificar el tipo de elemento que se puso en la `ArrayList`. El Listado 23 muestra cómo:

## Listado 23. Un segundo intento, con el uso de genéricos

```
ArrayList<String> arrayList = new ArrayList<String>();
arrayList.add("A String");
arrayList.add(new Integer(10));// compiler error!
arrayList.add("Another String");
// So far, so good.
*processArrayList(arrayList);
*// In some later part of the code...
private void processArrayList(ArrayList<String> theList) {
   for (int aa = 0; aa < theList.size(); aa++) {
     // No cast necessary...
     String s = theList.get(aa);
   }
}
```

### Iteración con genéricos

Los genéricos mejoran el lenguaje Java con una sintaxis especial para lidiar con entidades como `List`s que usted comúnmente quiere seguir paso a paso, elemento por elemento. Si quiere iterar por la `ArrayList`, por ejemplo, podría reescribir el código del Listado 23 del siguiente modo:

```
private void processArrayList(ArrayList<String> theList) {
 for (String s : theList) {
   String s = theList.get(aa);
 }
}
```

Esta sintaxis funciona para cualquier tipo de objeto que sea `Iterable` (es decir, implementa la interfaz `Iterable`).

## Clases parametrizadas

Las clases parametrizadas realmente brillan cuando se trata de colecciones, por lo tanto así es como usted las observará. Considere la interfaz de `List` (real). Representa una colección ordenada de objetos. En el caso de

uso más común, se agregan elementos a la List y luego se accede a aquellos elementos ya sea por índice o por iterar por la List.

Si está pensando en parametrizar una clase, considere si aplican los siguientes criterios:

- Una clase principal se encuentra en el centro de algún tipo de derivador: es decir, la "cosa" en el centro de la clase puede aplicarse ampliamente y las funciones (por ejemplo, los atributos) que la rodean son idénticas.
- Comportamiento común: usted realiza prácticamente las mismas operaciones sin tener en cuenta la "cosa" en el centro de la clase.

Al aplicar estos dos criterios, es bastante evidente que una colección concuerde con la cuenta:

- La "cosa" es la clase de la cual consta la colección.
- Las operaciones (tales como add, remove, size, y clear) son prácticamente las mismas sin importar el objeto del que consta la colección.

### Una List parametrizada

En la sintaxis de genéricos, el código para crear una List se parece a este:

```
List<E> listReference = new concreteListClass<E>();
```

La E, que significa Elemento, es la "cosa" que mencioné anteriormente. La concreteListClass es la clase del JDK que usted está creando como instancia. El JDK incluye varias implementaciones List<E> pero usted usará ArrayList<E>. Otra forma en que podría ver que se discute una clase genérica es Class<T>, donde T significa Tipo. Cuando ve E en el código Java, normalmente se está refiriendo a una colección de algún tipo. Y cuando ve T, está denotando una clase parametrizada.

Por lo tanto, para crear una ArrayList de, digamos, java.lang.Integer, usted haría lo siguiente:

```
List<Integer> listOfIntegers = new ArrayList<Integer>();
```

### SimpleList: Una clase parametrizada

Ahora suponga que quiere crear su propia clase parametrizada llamada SimpleList, con tres métodos:

- add() agrega un elemento al final de la SimpleList.
- size() retorna la cantidad actual de elementos en la SimpleList.
- clear() borra por completo los contenidos de la SimpleList.

El Listado 24 muestra la sintaxis para parametrizar la SimpleList:

## Listado 24. Parametrización de la SimpleList

```
package com.makotogroup.intro;
import java.util.ArrayList;
import java.util.List;
public class SimpleList<E> {
 private List<E> backingStore;
 public SimpleList() {
   backingStore = new ArrayList<E>();
```

```
 }
 public E add(E e) {
   if (backingStore.add(e))
     return e;
   else
     return null;
 }
 public int size() {
   return backingStore.size();
 }
 public void clear() {
   backingStore.clear();
 }
}
```

SimpleList puede parametrizarse con cualquier subclase de Object. Para crear y usar una SimpleList de, digamos, objetos java.math.BigDecimal, usted haría lo siguiente:

```
public static void main(String[] args) {
 SimpleList<BigDecimal> sl = new SimpleList<BigDecimal>();
 sl.add(BigDecimal.ONE);
 log.info("SimpleList size is : " + sl.size());
 sl.add(BigDecimal.ZERO);
 log.info("SimpleList size is : " + sl.size());
 sl.clear();
 log.info("SimpleList size is : " + sl.size());
}
```

Y obtendría esta salida:

```
May 5, 2010 6:28:58 PM com.makotogroup.intro.Application main
INFO: El tamaño de la SimpleList es: 1
May 5, 2010 6:28:58 PM com.makotogroup.intro.Application main
INFO: El tamaño de la SimpleList es: 2
May 5, 2010 6:28:58 PM com.makotogroup.intro.Application main
INFO: El tamaño de la SimpleList es: 0
```

## Tipos de enumeración

En JDK 5, un nuevo tipo de datos se agregó al lenguaje Java, denominado enum. No se lo debe confundir con java.util.Enumeration, enum representa un conjunto de objetos constantes que están todos relacionados con un concepto particular, cada uno de los cuales representa un valor constante diferente en ese conjunto. Antes de que se introdujera enum al lenguaje Java, usted habría definido un conjunto de valores constantes para un concepto (por ejemplo, gender) del siguiente modo:

```
public class Person {
 public static final String MALE = "male";
 public static final String FEMALE = "female";
}
```

Cualquier código que se necesite para hacer referencia a ese valor constante se habría escrito más o menos del siguiente modo:

```
public void myMethod() {
 //. . .
 String genderMale = Person.MALE;
 //. . .
}
```

### Definición de constantes con enum

Usar el tipo enum hace que definir constantes sea mucho más formal y también más poderoso. Aquí está la definición de enum para Gender:

```
public enum Gender {
 MALE,
 FEMALE
}
```

Eso es solo el comienzo de lo que puede hacer con enums. De hecho, las enums son muy parecidas a las clases, por lo tanto pueden tener constructores, atributos y métodos:

```
package com.makotogroup.intro;

public enum Gender {
 MALE("male"),
 FEMALE("female");

 private String displayName;
 private Gender(String displayName) {
   this.displayName = displayName;
 }

 public String getDisplayName() {
   return this.displayName;
 }
}
```

Una diferencia entre una clase y una enum es que el constructor de una enum debe declararse private y no puede extender (o heredar de) otras enums. Sin embargo, una enum puede implementar una interfaz.

### Una enum implementa una interfaz.

Suponga que usted define una interfaz, Displayable:

```
package com.makotogroup.intro;
public interface Displayable {
 public String getDisplayName();
}
```

Su enum Gender podría implementar esta interfaz (así como también cualquier otra enum que se necesite para producir un nombre de pantalla amigable), como del siguiente modo:

```
package com.makotogroup.intro;

public enum Gender implements Displayable {
 MALE("male"),
 FEMALE("female");

 private String displayName;
 private Gender(String displayName) {
   this.displayName = displayName;
 }
 @Override
 public String getDisplayName() {
   return this.displayName;
 }
}
```

Vea Temas relacionados para aprender más acerca de genéricos).

# E/S

Esta sección es una visión general del paquete java.io. Aprenderá a usar algunas de sus herramientas para recolectar y manipular datos de una variedad de orígenes.

## Trabajo con datos externos

En mucha ocasiones, los datos que usted usa en sus programas Java vendrán de un origen de datos externo, tales como una base de datos, una transferencia directa de bytes por un socket o un almacenamiento de archivos. El lenguaje Java le da muchas herramientas para conseguir información de estos orígenes y la mayoría de ellas están ubicadas en el paquete java.io.

## Archivos

De todos los orígenes de datos disponibles para sus aplicaciones Java, los archivos son los más comunes y, con frecuencia, los más convenientes. Si usted quiere leer un archivo en su aplicación Java, debe usar streams que analicen sus bytes entrantes en los tipos de lenguaje Java.

java.io.File es una clase que define un recurso en su sistema de archivos y representa ese recurso en un modo abstracto. Crear un objeto File es fácil:

```
File f = new File("temp.txt");
File f2 = new File("/home/steve/testFile.txt");
```

El constructor File toma el nombre del archivo que creará. La primera llamada crea un archivo llamado temp.txt en el directorio dado. La segunda llamada crea un archivo en una ubicación específica en mi sistema Linux. Usted puede pasar cualquier String al constructor del File, siempre y cuando sea un nombre de archivo válido para su OS, ya sea que el archivo al que hace referencia incluso exista o no.

Este código le pregunta al objeto File recientemente creado si el archivo existe:

```
File f2 = new File("/home/steve/testFile.txt");
if (f2.exists()) {
 // File exists. Procesarlo...
} else {
 // File doesn't exist. Crearlo...
 f2.createNewFile();
}
```

java.io.File tiene otros métodos útiles que puede usar para suprimir archivos, crear directorios (al pasar el nombre de un directorio como el argumento al constructor del File), determinar si un recurso es un archivo, directorio o enlace simbólico, y más.

La acción real de E/S de Java está en la escritura a y lectura de los orígenes de datos, que es donde entran las secuencias.

## Uso de secuencias en E/S de Java

Puede acceder a archivos en el sistema de archivos con el uso de secuencias. En el nivel más bajo, las secuencias le permiten a un programa recibir bytes de un origen o enviar salidas a un destino. Algunas

secuencias manejan todo tipo de caracteres de 16 bits (tipos de Reader y Writer). Otras manejan solo bytes de 8 bits (tipos de InputStream y OutputStream). Dentro de estas jerarquías hay varios sabores de secuencias, que se encuentran todos en el paquete java.io. En el nivel más alto de abstracción están las secuencias de caracteres y secuencias de bytes.

Las secuencias de bytes leen (InputStream y subclases) y escriben (OutputStream y subclases) bytes de 8 bits. En otras palabras, una secuencia de bytes puede considerarse un tipo más crudo de secuencia. Aquí hay un resumen de dos secuencias de bytes comunes y su uso:

- **FileInputStream**/**FileOutputStream**: Lee bytes desde un archivo, escribe bytes para un archivo.
- **ByteArrayInputStream**/**ByteArrayOutputStream**: Lee bytes desde una matriz en memoria, escribe bytes para una matriz en memoria.

### Secuencias de caracteres

Las secuencias de caracteres leen (Reader y sus subclases) y escriben (Writer y sus subclases) caracteres de 16 bits. Aquí hay un listado seleccionado de secuencias de caracteres y sus usos:

- **StringReader**/**StringWriter**: Lee y escribe caracteres para y desde Strings en memoria.
- **InputStreamReader**/**InputStreamWriter** (y subclases **FileReader**/**FileWriter**): Forma un puente entre secuencias de bytes y secuencias de caracteres. Los sabores del Reader leen bytes desde una secuencia de bytes y los convierte en caracteres. Los sabores del Writer convierten los caracteres en bytes para ponerlos en secuencias de bytes.
- **BufferedReader**/**BufferedWriter**: Almacenan datos mientras leen o escriben otra secuencia, lo cual hace que las operaciones de leer y escribir sean más eficientes.

En lugar de intentar cubrir secuencias en su totalidad, me centraré en las secuencias recomendadas para leer y escribir archivos. En la mayoría de los casos, estas son secuencias de caracteres.

### Lectura desde un File

Hay varios modos de leer desde un File. Podría decirse que el abordaje más simple es:

1. Crear un InputStreamReader en el File del cual usted quiere leer.
2. Llamar a read() para leer un carácter por vez hasta que llegue al final del archivo.

El Listado 25 es un ejemplo de la lectura desde un File:

## Listado 25. Lectura desde un File

```
Logger log = Logger.getAnonymousLogger();
StringBuilder sb = new StringBuilder();
try {
 InputStream inputStream = new FileInputStream(new File("input.txt"));
 InputStreamReader reader = new InputStreamReader(inputStream);
 try {
   int c = reader.read();
   while (c != -1) {
     sb.append(c);
   }
 } finally {
 reader.close();
 }
} catch (IOException e) {
 log.info("Caught exception while processing file: " + e.getMessage());
}
```

### Escritura para un File

Como con la lectura desde un File, hay varios modos para escribir para un File. Una vez más, elegiré el abordaje más simple:

1. Crear un FileOutputStream en el File para el cual usted quiere escribir.
2. Llamar a write() para escribir la secuencia de caracteres.

El Listado 26 es un ejemplo de la escritura para un File:

## Listado 26. Escritura para un File

```
Logger log = Logger.getAnonymousLogger();
StringBuilder sb = getStringToWriteSomehow();
try {
 OutputStream outputStream = new FileOutputStream(new File("output.txt"));
 OutputStreamWriter writer = new OutputStreamWriter(outputStream);
 try {
   writer.write(sb.toString());
 } finally {
   writer.close();
 }
} catch (IOException e) {
 log.info("Caught exception while processing file: " + e.getMessage());
}
```

### Almacenamiento intermedio de secuencias

Leer y escribir secuencias de caracteres de un carácter por vez no es exactamente eficiente, por lo tanto, en la mayoría de los casos, usted probablemente quiera, en cambio, usar E/S almacenado. Para leer desde un archivo usando E/S almacenado, el código se parece exactamente como el Listado 25, con la excepción de que se envuelve el InputStreamReader en un BufferedReader, como se muestra en el Listado 27:

### Listado 27. Leer desde un File con E/S almacenado

```
Logger log = Logger.getAnonymousLogger();
StringBuilder sb = new StringBuilder();
try {
 InputStream inputStream = new FileInputStream(new File("input.txt"));
 BufferedReader reader = new BufferedReader(new InputStreamReader(inputStream));
 try {
   String line = reader.readLine();
   while (line != null) {
     sb.append(line);
     line = reader.readLine();
   }
 } finally {
 reader.close();
 }
} catch (IOException e) {
 log.info("Caught exception while processing file: " + e.getMessage());
}
```

Escribir para un archivo usando el E/S almacenado es lo mismo: solo se envuelve el OutputStreamWriter en un BufferedWriter, como se muestra en el Listado 28.

### Listado 28. Escribir para un File con E/S almacenado

```
Logger log = Logger.getAnonymousLogger();
StringBuilder sb = getStringToWriteSomehow();
try {
 OutputStream outputStream = new FileOutputStream(new File("output.txt"));
 BufferedWriter writer = new BufferedWriter(new OutputStreamWriter(outputStream));
 try {
   writer.write(sb.toString());
 } finally {
   writer.close();
 }
} catch (IOException e) {
 log.info("Caught exception while processing file: " + e.getMessage());
}
```

Apenas he rozado la superficie de lo que es posible con esta biblioteca Java esencial. Por su cuenta, intente aplicar lo que ha aprendido sobre los archivos a otros orígenes de datos.

## Serialización Java

La serialización Java es otra de las bibliotecas esenciales de la plataforma Java. La serialización se usa principalmente para la persistencia de objetos y objetos remotos, dos casos de uso donde necesita poder tomar una instantánea del estado de un objeto y luego reconstituirlo posteriormente. Esta sección le da una idea de la API de serialización Java y le muestra cómo usarlo en sus programas.

### ¿Qué es la serialización de objetos?

La serialización es un proceso en el que el estado de un objeto y sus metadatos (como el nombre de clase del objeto y los nombres de sus atributos) se almacenan en un formato binario especial. Poner el objeto en este formato,— serializándolo,— preserva toda la información necesaria para reconstituir (o deserializar) el objeto cuando usted necesite hacerlo.

Hay dos casos de usos primarios para la serialización de objetos:

- La persistencia de objetos significa almacenar el estado del objeto en un mecanismo de persistencia permanente, como por ejemplo una base de datos.
- Los objetos remotos implican el envío del objeto a otra computadora o sistema.

## java.io.Serializable

El primer paso para hacer que la serialización funcione es permitir que sus objetos usen el mecanismo. Cada objeto que usted quiera que sea serializable debe implementar una interfaz llamada java.io.Serializable:

```
import java.io.Serializable;
public class Person implements Serializable {
// etc...
}
```

La interfaz Serializable marca los objetos de la clase Person al tiempo de ejecución como serializable. Cada subclase de Person también se marcará como serializable.

Cualquier atributo de un objeto que no sea serializable causará que el tiempo de ejecución Java arroje una NotSerializableException si intenta serializar su objeto. Puede gestionar esto al usar la palabra clave transient para comunicarle al tiempo de ejecución que no intente serializar ciertos atributos. En ese caso, usted es responsable de asegurarse de que los atributos se restauren para que su objeto funcione adecuadamente.

### Serializar un objeto

Ahora probará un ejemplo que combina lo que acaba de aprender acerca de E/S de Java con lo que está aprendiendo ahora acerca de la serialización.

Suponga que usted crea y llena un objeto Manager (recuerde que Manager está en el gráfico de herencias de Person, que es serializable) y luego quiere serializar ese objeto para una OutputStream, es este caso para un archivo. Ese proceso se muestra en el Listado 29:

## Listado 29. Serializar un objeto

```
Manager m = new Manager();
m.setEmployeeNumber("0001");

m.setGender(Gender.FEMALE);
m.setAge(29);
m.setHeight(170);
m.setName("Mary D. Boss");
m.setTaxpayerIdentificationNumber("123-45-6789");
log.info("About to write object using serialization... object looks like:");
m.printAudit(log);
try {
 String filename = "Manager-" + m.hashCode() + ".ser";
 ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream(filename));
 oos.writeObject(m);
 log.info("Wrote object...");
} catch (Exception e) {
 log.log(Level.SEVERE, "Caught Exception processing object", e);
}
```

El primer paso es crear el objeto y establecer algunos valores de atributos. Luego, se crea una OutputStream, en este caso una FileOutputStream y luego se llama a writeObject() en esa secuencia. writeObject() es un método que usa serialización Java para serializar un objeto para la secuencia.

En este ejemplo, usted está almacenando el objeto en un archivo pero se usa esta misma técnica para cualquier tipo de serialización.

### Deserialización de un objeto

Todo el punto de la serialización de un objeto es poder reconstituirlo, o deserializarlo. El Listado 30 lee el archivo que usted acaba de serializar y deserializa sus contenidos, restaurando de este modo el estado del objeto `Manager`:

## Listado 30. Deserialización de un objeto

```
Manager m = new Manager();
m.setEmployeeNumber("0001");
m.setGender(Gender.FEMALE);
m.setAge(29);
m.setHeight(170);
m.setName("Mary D. Boss");
m.setTaxpayerIdentificationNumber("123-45-6789");
log.info("About to write object using serialization... object looks like:");
m.printAudit(log);
try {
 String filename = "Manager-" + m.hashCode() + ".ser";
 ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream(filename));
 oos.writeObject(m);
 log.info("Wrote object...");

 ObjectInputStream ois = new ObjectInputStream(new FileInputStream(filename));
 m = (Manager)ois.readObject();
 log.info("Read object using serialization... object looks like:");
 m.printAudit(log);
} catch (Exception e) {
 log.log(Level.SEVERE, "Caught Exception processing object", e);
}
```

Para la mayoría de los propósitos de aplicaciones, marcar sus objetos como `serializables` es de todo lo que tendrá que preocupar siempre cuando se trate de la serialización. En casos donde usted sí necesita serializar y deserializar sus objetos explícitamente, puede usar la técnica que se muestra en los Listados 29 y 30. Pero mientras los objetos de sus aplicaciones de desarrollan, y usted les agrega y elimina atributos, la serialización toma una nueva capa de complejidad.

## serialVersionUID

En los primeros días del middleware y la comunicación de objetos remotos, los desarrolladores eran en gran medida responsables de controlar el "formato de conexión física" de sus objetos, lo cual causó un sinfín de dolores de cabeza mientras la tecnología comenzaba a evolucionar.

Suponga que usted agregó un atributo a un objeto, lo recompiló y redistribuyó el código a cada máquina en un clúster de aplicaciones. El objeto se almacenaría en una máquina con una versión del código de serialización pero accederían otras máquinas que podrían tener una versión diferente del código. Cuando esas máquinas intentaran deserializar el objeto, a menudo sucederían cosas malas.

Los metadatos de la serialización Java, — la información incluida en el formato de serialización binaria, — son sofisticados y resuelven muchos de los problemas que asediaban a los primeros desarrolladores de middleware. Pero no pueden resolver todos los problemas.

La serialización Java usa una propiedad denominada `serialVersionUID` para ayudarlo a lidiar con diferentes versiones de objetos en un escenario de serialización. No necesita declarar esta propiedad en sus objetos; de forma predeterminada, la plataforma Java usa un algoritmo que computa un valor para ella en base a los atributos de su clase, su nombre de clase y posición en el clúster galáctico local. La mayoría de las veces, eso funciona bien. Pero si agrega o elimina un atributo, ese valor dinámicamente generado cambiará y el tiempo de ejecución Java arrojará una `InvalidClassException`.

Para evitar esto, usted debería acostumbrase a declarar explícitamente una `serialVersionUID`:

```
import java.io.Serializable;
public class Person implements Serializable {
 private static final long serialVersionUID = 20100515;
// etc...
}
```

Recomiendo usar algún tipo de plan para su número de versión de `serialVersionUID` (he usado la fecha actual en el ejemplo anterior) y debería declararlo `private static final` y de tipo `long`.

Tal vez se esté preguntando cuándo cambiar esta propiedad. La respuesta breve es que debería cambiarla cuando usted hace un cambio incompatible a la clase, lo que normalmente significa que ha eliminado un atributo. Si usted tiene una versión del objeto en una máquina en la que se ha eliminado el atributo, y el objeto se hace remoto en una máquina con una versión del objeto donde se espera el atributo, entonces las cosas se pueden tornar extrañas.

Como regla general, siempre que agregue o elimine funciones (es decir, atributos o métodos) de una clase, debería cambiar su `serialVersionUID`. Es mejor obtener una `InvalidClassException` en el otro extremo de la conexión que un error de aplicación que se debe a un cambio de clase incompatible.

## Conclusión para la Parte 2

El tutorial "Introducción a la programación Java" ha cubierto una parte importante del lenguaje Java, pero el lenguaje es enorme. Un solo tutorial no puede abarcarlo todo.

Mientras continúa aprendiendo acerca del lenguaje y la plataforma Java, probablemente quiera estudiar en más detalle temas como expresiones regulares, genéricos y serialización Java. Con el tiempo, quizás también quiera explorar temas que no se cubrieron en este tutorial introductorio, tales como la concurrencia y la persistencia. Otro tema que merece explorarse es Java 7, que traerá muchos cambios potencialmente innovadores a la plataforma Java. Vea los Recursos para algunos buenos puntos de partida para aprender más acerca de los conceptos de programación Java, incluidos aquellos que son muy avanzados para que se exploren en este formato introductorio.

# 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.