

JAVA

www.postparaprogramadores.com

Contenido

Java - Inicio

Java - Descripción general

Java - Configuración del entorno

Java: sintaxis básica

Java - Objeto y clases

Java - Constructores

Java - Tipos de datos básicos

Java - Tipos de variables

Java - Tipos de modificadores

Java - Operadores básicos

Java - Control de bucle

Java - Toma de decisiones

Java - Números

Java - Personajes

Java - Cuerdas

Java: matrices

Java - Fecha y hora

Java - Expresiones regulares

Java - Métodos

Java - Archivos y E / S

Java - Excepciones

Java - Clases internas

Orientado a objetos Java

Java - Herencia

Java: anulación

Java - Polimorfismo

Java - Abstracción

Java - Encapsulación

Java - Interfaces

Java - Paquetes

Java avanzado

Java - Estructuras de datos

Java - Colecciones

Java - Genéricos

Java - Serialización

Java - Redes

Java - Envío de correo electrónico

Java - Multithreading

Java - Conceptos básicos de Applet

Java - Documentación

Java - Descripción general

El lenguaje de programación Java fue desarrollado originalmente por Sun Microsystems, que fue iniciado por James Gosling y lanzado en 1995 como componente principal de la plataforma Java de Sun Microsystems (Java 1.0 [J2SE]).

La última versión de Java Standard Edition es Java SE 8. Con el avance de Java y su gran popularidad, se crearon múltiples configuraciones para adaptarse a varios tipos de plataformas. Por ejemplo: J2EE para aplicaciones empresariales, J2ME para aplicaciones móviles.

Las nuevas versiones de J2 fueron renombradas como Java SE, Java EE y Java ME respectivamente. Se garantiza que Java será **Write Once, Run Anywhere**.

Java es -

- **Orientado a objetos** : en Java, todo es un objeto. Java se puede extender fácilmente ya que se basa en el modelo de objetos.
- **Independiente de la plataforma** : a diferencia de muchos otros lenguajes de programación, incluidos C y C ++, cuando se compila Java, no se compila en una máquina específica de la plataforma, sino en un código de bytes independiente de la plataforma. Este código de bytes se distribuye a través de la web y es interpretado por la máquina virtual (JVM) en cualquier plataforma en la que se ejecute.
- **Simple** : Java está diseñado para ser fácil de aprender. Si comprende el concepto básico de OOP Java, sería fácil de dominar.
- **Seguro** : con la función segura de Java, permite desarrollar sistemas libres de virus y manipulaciones. Las técnicas de autenticación se basan en el cifrado de clave pública.
- **Arquitectura neutral** : el compilador de Java genera un formato de archivo de objeto neutral de arquitectura, que hace que el código compilado sea ejecutable en muchos procesadores, con la presencia del sistema de tiempo de ejecución Java.
- **Portable** : ser neutral en cuanto a la arquitectura y no tener aspectos dependientes de la implementación de la especificación hace que Java sea portátil. El compilador en Java está escrito en ANSI C con un límite de portabilidad limpio, que es un subconjunto POSIX.
- **Robusto** : Java se esfuerza por eliminar las situaciones propensas a errores enfatizando principalmente la verificación de errores en tiempo de compilación y la verificación de tiempo de ejecución.
- **Multiproceso** : con la característica multiproceso de Java, es posible escribir programas que pueden realizar muchas tareas simultáneamente. Esta característica de diseño permite a los desarrolladores construir aplicaciones interactivas que pueden ejecutarse sin problemas.
- **Interpretado** : el código de bytes de Java se traduce sobre la marcha a las instrucciones de la máquina nativa y no se almacena en ningún lugar. El proceso de desarrollo es más rápido y analítico ya que la vinculación es un proceso incremental y liviano.

Descarga más libros de programación GRATIS [click aquí](#)

Download more FREE programming books [click here](#)

- **Alto rendimiento** : con el uso de compiladores Just-In-Time, Java permite un alto rendimiento.
- **Distribuido** : Java está diseñado para el entorno distribuido de Internet.
- **Dinámico** : Java se considera más dinámico que C o C ++, ya que está diseñado para adaptarse a un entorno en evolución. Los programas Java pueden llevar una gran cantidad de información en tiempo de ejecución que se puede utilizar para verificar y resolver los accesos a los objetos en tiempo de ejecución.



Síguenos en Instagram para que estés al tanto de los nuevos libros de programación. [Click aquí](#)

Follow us on Instagram so you are aware of the new programming books [Click here](#)

Historia de Java

James Gosling inició un proyecto de lenguaje Java en junio de 1991 para usarlo en uno de sus muchos proyectos de decodificadores. El lenguaje, inicialmente llamado 'Roble' después de un roble que se encontraba fuera de la oficina de Gosling, también se llamaba 'Verde' y terminó siendo renombrado como Java, de una lista de palabras aleatorias.

Sun lanzó la primera implementación pública como Java 1.0 en 1995. Prometió **Write Once, Run Anywhere** (WORA), proporcionando tiempos de ejecución sin costo en plataformas populares.

El 13 de noviembre de 2006, Sun lanzó gran parte de Java como software libre y de código abierto bajo los términos de la Licencia Pública General de GNU (GPL).

El 8 de mayo de 2007, Sun terminó el proceso, haciendo que todo el código central de Java sea gratuito y de código abierto, aparte de una pequeña porción de código sobre la cual Sun no tenía los derechos de autor.

Herramientas que necesitará

Para realizar los ejemplos discutidos en este tutorial, necesitará una computadora Pentium de 200 MHz con un mínimo de 64 MB de RAM (se recomiendan 128 MB de RAM).

También necesitará los siguientes softwares:

- Sistema operativo Linux 7.1 o Windows xp / 7/8
- Java JDK 8
- Bloc de notas de Microsoft o cualquier otro editor de texto

Este tutorial proporcionará las habilidades necesarias para crear aplicaciones GUI, redes y web utilizando Java.

¿Lo que sigue?

El siguiente capítulo lo guiará a cómo puede obtener Java y su documentación. Finalmente, le indica cómo instalar Java y preparar un entorno para desarrollar aplicaciones Java.

Java - Configuración del entorno

En este capítulo, discutiremos sobre los diferentes aspectos de la configuración de un entorno agradable para Java.

Configuración del entorno local

Si todavía está dispuesto a configurar su entorno para el lenguaje de programación Java, esta sección lo guía sobre cómo descargar y configurar Java en su máquina. Los siguientes son los pasos para configurar el entorno.

Java SE está disponible gratuitamente desde el enlace [Descargar Java](#) . Puede descargar una versión basada en su sistema operativo.

Siga las instrucciones para descargar Java y ejecute **.exe** para instalar Java en su máquina. Una vez que haya instalado Java en su máquina, deberá establecer variables de entorno para que apunten a los directorios de instalación correctos:

Configurando la ruta para Windows

Suponiendo que haya instalado Java en el directorio *c: \ Archivos de programa \ java \ jdk -*

- Haga clic derecho en 'Mi PC' y seleccione 'Propiedades'.
- Haga clic en el botón 'Variables de entorno' debajo de la pestaña 'Avanzado'.
- Ahora, modifique la variable 'Ruta' para que también contenga la ruta al ejecutable de Java. Ejemplo, si la ruta está configurada actualmente en 'C: \ WINDOWS \ SYSTEM32', cambie su ruta para que lea 'C: \ WINDOWS \ SYSTEM32; c: \ Program Files \ java \ jdk \ bin'.

Configurando la ruta para Linux, UNIX, Solaris, FreeBSD

La variable de entorno PATH debe establecerse para que apunte a dónde se han instalado los binarios de Java. Consulte su documentación de shell, si tiene problemas para hacerlo.

Ejemplo, si usa *bash* como su shell, entonces agregaría la siguiente línea al final de su *.bashrc*: `export PATH = / path / to / java: $ PATH`

Editores populares de Java

Para escribir sus programas Java, necesitará un editor de texto. Hay IDEs aún más sofisticados disponibles en el mercado. Pero por ahora, puede considerar uno de los siguientes:

-
-

- **Bloc de notas** : en la máquina con Windows, puede usar cualquier editor de texto simple como el Bloc de notas (recomendado para este tutorial), TextPad.
- **Netbeans** : un IDE de Java que es de código abierto y gratuito que se puede descargar desde <https://www.netbeans.org/index.html> .
- **Eclipse** : un IDE de Java desarrollado por la comunidad de código abierto eclipse y se puede descargar desde <https://www.eclipse.org/> .

¿Lo que sigue?

El próximo capítulo le enseñará cómo escribir y ejecutar su primer programa Java y algunas de las sintaxis básicas importantes en Java necesarias para desarrollar aplicaciones.

Java: sintaxis básica

Cuando consideramos un programa Java, se puede definir como una colección de objetos que se comunican invocando los métodos de cada uno. Veamos ahora brevemente qué significan las clases, objetos, métodos y variables de instancia.

- **Objeto** : los objetos tienen estados y comportamientos. Ejemplo: un perro tiene estados: color, nombre, raza, así como comportamientos como mover la cola, ladrar, comer. Un objeto es una instancia de una clase.
- **Clase** : una clase se puede definir como una plantilla / plano que describe el comportamiento / estado que admite el objeto de su tipo.
- **Métodos** : un método es básicamente un comportamiento. Una clase puede contener muchos métodos. Es en los métodos donde se escriben las lógicas, se manipulan los datos y se ejecutan todas las acciones.
- **Variables de instancia** : cada objeto tiene su conjunto único de variables de instancia. El estado de un objeto es creado por los valores asignados a estas variables de instancia.

Primer programa de Java

Veamos un código simple que imprimirá las palabras **Hello World** .

Ejemplo

```
public class MyFirstJavaProgram {  
  
    /* This is my first java program.  
     * This will print 'Hello World' as the output
```



```
    */  
  
    public static void main(String []args) {  
        System.out.println("Hello World"); // prints Hello  
World  
    }  
}
```

Veamos cómo guardar el archivo, compilar y ejecutar el programa. Por favor siga los siguientes pasos -

- Abra el bloc de notas y agregue el código como se indica arriba.
- Guarde el archivo como: MyFirstJavaProgram.java.
- Abra una ventana del símbolo del sistema y vaya al directorio donde guardó la clase. Supongamos que es C: \.
- Escriba 'javac MyFirstJavaProgram.java' y presione Intro para compilar su código. Si no hay errores en su código, el símbolo del sistema lo llevará a la siguiente línea (Supuesto: la variable de ruta está configurada).
- Ahora, escriba 'java MyFirstJavaProgram' para ejecutar su programa.
- Podrá ver 'Hello World' impreso en la ventana.

Salida

```
C:\> javac MyFirstJavaProgram.java  
C:\> java MyFirstJavaProgram  
Hello World
```

Sintaxis Básica

Acerca de los programas Java, es muy importante tener en cuenta los siguientes puntos.

- **Mayúsculas** y minúsculas: Java distingue entre mayúsculas y minúsculas, lo que significa que los identificadores **Hello** y **hello** tendrían un significado diferente en Java.
- **Nombres de clase** : para todos los nombres de clase, la primera letra debe estar en mayúscula. Si se utilizan varias palabras para formar un nombre de la clase, la primera letra de cada palabra interna debe estar en mayúscula.

Ejemplo: *clase MyFirstJavaClass*

- **Nombres de métodos** : todos los nombres de métodos deben comenzar con una letra minúscula. Si se usan varias palabras para formar el nombre del método, entonces la primera letra de cada palabra interna debe estar en mayúscula.

Ejemplo: *public void myMethodName ()*

- **Nombre del archivo de programa** : el nombre del archivo de programa debe coincidir exactamente con el nombre de la clase.

Al guardar el archivo, debe guardarlo usando el nombre de la clase (Recuerde que Java distingue entre mayúsculas y minúsculas) y agregue '.java' al final del nombre (si el nombre del archivo y el nombre de la clase no coinciden, su programa no se compilará)

Ejemplo: Suponga que 'MyFirstJavaProgram' es el nombre de la clase. Entonces el archivo debe guardarse como '*MyFirstJavaProgram.java*'

- **public static void main (String args [])** : el procesamiento del programa Java comienza desde el método main (), que es una parte obligatoria de cada programa Java.

Identificadores Java

Todos los componentes de Java requieren nombres. Los nombres utilizados para clases, variables y métodos se denominan **identificadores** .

En Java, hay varios puntos para recordar acerca de los identificadores. Son los siguientes:

- Todos los identificadores deben comenzar con una letra (de la A a la Z o de la a a la z), un carácter de moneda (\$) o un guión bajo (_).
- Después del primer carácter, los identificadores pueden tener cualquier combinación de caracteres.
- Una palabra clave no se puede utilizar como identificador.
- Lo más importante, los identificadores distinguen entre mayúsculas y minúsculas.
- Ejemplos de identificadores legales: edad, \$ salario, _valor, __1_valor.
- Ejemplos de identificadores ilegales: 123abc, -salary.

Modificadores Java

Al igual que otros lenguajes, es posible modificar clases, métodos, etc., utilizando modificadores. Hay dos categorías de modificadores:

- **Modificadores de acceso** : predeterminado, público, protegido, privado
- **Modificadores sin acceso** : final, abstracto, estricto fp

Buscaremos más detalles sobre modificadores en la siguiente sección.

Variables Java

Los siguientes son los tipos de variables en Java:

- Variables Locales
- Variables de clase (variables estáticas)
- Variables de instancia (variables no estáticas)

Matrices Java

Las matrices son objetos que almacenan múltiples variables del mismo tipo. Sin embargo, una matriz en sí misma es un objeto en el montón. Veremos cómo declarar, construir e inicializar en los próximos capítulos.

Enums de Java

Las enumeraciones se introdujeron en Java 5.0. Las enumeraciones restringen una variable para tener uno de los pocos valores predefinidos. Los valores en esta lista enumerada se denominan enumeraciones.

Con el uso de enumeraciones es posible reducir la cantidad de errores en su código.

Por ejemplo, si consideramos una aplicación para una tienda de jugos frescos, sería posible restringir el tamaño del vidrio a pequeño, mediano y grande. Esto aseguraría que no permitiría a nadie ordenar cualquier tamaño que no sea pequeño, mediano o grande.

Ejemplo

```
class FreshJuice {
    enum FreshJuiceSize{ SMALL, MEDIUM, LARGE }
    FreshJuiceSize size;
}

public class FreshJuiceTest {

    public static void main(String args[]) {
        FreshJuice juice = new FreshJuice();
        juice.size = FreshJuice.FreshJuiceSize.MEDIUM ;
        System.out.println("Size: " + juice.size);
    }
}
```

El ejemplo anterior producirá el siguiente resultado:

Salida

Size: MEDIUM

Nota : las enumeraciones se pueden declarar como propias o dentro de una clase. Los métodos, las variables y los constructores también se pueden definir dentro de las enumeraciones.

Palabras clave de Java

La siguiente lista muestra las palabras reservadas en Java. Estas palabras reservadas no se pueden usar como constantes o variables ni ningún otro nombre identificador.

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

Comentarios en Java

Java admite comentarios de una o varias líneas muy similares a C y C ++. El compilador de Java ignora todos los caracteres disponibles dentro de cualquier comentario.

Ejemplo

```
public class MyFirstJavaProgram {  
  
    /* This is my first java program.  
     * This will print 'Hello World' as the output  
     * This is an example of multi-line comments.  
     */  
  
    public static void main(String []args) {  
        // This is an example of single line comment  
        /* This is also an example of single line comment. */  
        System.out.println("Hello World");  
    }  
}
```

Salida

Hello World

Usando líneas en blanco

Una línea que contiene solo espacios en blanco, posiblemente con un comentario, se conoce como una línea en blanco, y Java la ignora por completo.

Herencia

En Java, las clases se pueden derivar de las clases. Básicamente, si necesita crear una nueva clase y aquí ya hay una clase que tiene parte del código que necesita, entonces es posible derivar su nueva clase del código ya existente.

Este concepto le permite reutilizar los campos y métodos de la clase existente sin tener que volver a escribir el código en una nueva clase. En este escenario, la clase existente se llama **superclase** y la clase derivada se llama **subclase**.

Interfaces

En lenguaje Java, una interfaz se puede definir como un contrato entre objetos sobre cómo comunicarse entre sí. Las interfaces juegan un papel vital cuando se trata del concepto de herencia.

Una interfaz define los métodos que debe usar una clase derivada (subclase). Pero la implementación de los métodos depende totalmente de la subclase.

¿Lo que sigue?

La siguiente sección explica acerca de los objetos y las clases en la programación Java. Al final de la sesión, podrá obtener una imagen clara de qué son los objetos y qué son las clases en Java.

Java - Objeto y clases

Java es un lenguaje orientado a objetos. Como lenguaje que tiene la función Orientada a Objetos, Java admite los siguientes conceptos fundamentales:

- Polimorfismo
- Herencia
- Encapsulación
- Abstracción
- Clases
- Objetos
- Ejemplo
- Método
- Paso de mensajes

En este capítulo, analizaremos los conceptos: clases y objetos.

- **Objeto** : los objetos tienen estados y comportamientos. Ejemplo: un perro tiene estados (color, nombre, raza y comportamientos), menea la cola, ladra, come. Un objeto es una instancia de una clase.
- **Clase** : una clase se puede definir como una plantilla / plano que describe el comportamiento / estado que admite el objeto de su tipo.

Objetos en Java

Veamos ahora en profundidad qué son los objetos. Si consideramos el mundo real, podemos encontrar muchos objetos a nuestro alrededor, automóviles, perros, humanos, etc. Todos estos objetos tienen un estado y un comportamiento.

Si consideramos un perro, entonces su estado es: nombre, raza, color y el comportamiento es: ladrar, menear la cola, correr.

Si compara el objeto de software con un objeto del mundo real, tienen características muy similares.

Los objetos de software también tienen un estado y un comportamiento. El estado de un objeto de software se almacena en campos y el comportamiento se muestra a través de métodos.

Entonces, en el desarrollo de software, los métodos operan en el estado interno de un objeto y la comunicación de objeto a objeto se realiza a través de métodos.

Clases en Java

Una clase es un plano a partir del cual se crean objetos individuales.

A continuación se muestra una muestra de una clase.

Ejemplo

```
public class Dog {  
    String breed;  
    int age;  
    String color;  
  
    void barking() {  
    }  
  
    void hungry() {  
    }  
  
    void sleeping() {  
    }  
}
```

Una clase puede contener cualquiera de los siguientes tipos de variables.

- **Variables locales** : las variables definidas dentro de los métodos, constructores o bloques se denominan variables locales. La variable se declarará e inicializará dentro del método y la variable se destruirá cuando el método se haya completado.
- **Variables de instancia: las variables de instancia** son variables dentro de una clase pero fuera de cualquier método. Estas variables se inicializan cuando se

instancia la clase. Se puede acceder a las variables de instancia desde cualquier método, constructor o bloque de esa clase en particular.

- **Variables de clase:** las variables de clase son variables declaradas dentro de una clase, fuera de cualquier método, con la palabra clave estática.

Una clase puede tener cualquier número de métodos para acceder al valor de varios tipos de métodos. En el ejemplo anterior, ladrar (), hambriento () y dormir () son métodos.

Los siguientes son algunos de los temas importantes que deben discutirse cuando se analizan las clases del lenguaje Java.

Constructores

Cuando se discute sobre las clases, uno de los subtemas más importantes serían los constructores. Cada clase tiene un constructor. Si no escribimos explícitamente un constructor para una clase, el compilador de Java construye un constructor predeterminado para esa clase.

Cada vez que se crea un nuevo objeto, se invocará al menos un constructor. La regla principal de los constructores es que deben tener el mismo nombre que la clase. Una clase puede tener más de un constructor.

El siguiente es un ejemplo de un constructor:

Ejemplo

```
public class Puppy {
    public Puppy() {
    }

    public Puppy(String name) {
        // This constructor has one parameter, name.
    }
}
```

Java también es compatible con las clases Singleton, donde podrá crear solo una instancia de una clase.

Nota : tenemos dos tipos diferentes de constructores. Vamos a discutir los constructores en detalle en los capítulos siguientes.

Crear un objeto

Como se mencionó anteriormente, una clase proporciona los planos para los objetos. Básicamente, un objeto se crea a partir de una clase. En Java, la nueva palabra clave se usa para crear nuevos objetos.

Hay tres pasos al crear un objeto desde una clase:

- **Declaración :** una declaración de variable con un nombre de variable con un tipo de objeto.
- **Instanciación :** la palabra clave 'nueva' se usa para crear el objeto.

- **Inicialización**: la palabra clave 'nueva' es seguida por una llamada a un constructor. Esta llamada inicializa el nuevo objeto.

El siguiente es un ejemplo de creación de un objeto:

Ejemplo

```
public class Puppy {
    public Puppy(String name) {
        // This constructor has one parameter, name.
        System.out.println("Passed Name is :" + name );
    }

    public static void main(String []args) {
        // Following statement would create an object myPuppy
        Puppy myPuppy = new Puppy( "tommy" );
    }
}
```

Si compilamos y ejecutamos el programa anterior, producirá el siguiente resultado:

Salida

Passed Name is :tommy

Acceso a variables y métodos de instancia

Se accede a las variables y métodos de instancia a través de objetos creados. Para acceder a una variable de instancia, la siguiente es la ruta completa:

```
/* First create an object */
ObjectReference = new Constructor();
```

```
/* Now call a variable as follows */
ObjectReference.variableName;
```

```
/* Now you can call a class method as follows */
ObjectReference.MethodName();
```

Ejemplo

Este ejemplo explica cómo acceder a las variables de instancia y los métodos de una clase.

```
public class Puppy {
    int puppyAge;

    public Puppy(String name) {
```

```

        // This constructor has one parameter, name.
        System.out.println("Name chosen is :" + name );
    }

    public void setAge( int age ) {
        puppyAge = age;
    }

    public int getAge( ) {
        System.out.println("Puppy's age is :" + puppyAge );
        return puppyAge;
    }

    public static void main(String []args) {
        /* Object creation */
        Puppy myPuppy = new Puppy( "tommy" );

        /* Call class method to set puppy's age */
        myPuppy.setAge( 2 );

        /* Call another class method to get puppy's age */
        myPuppy.getAge( );

        /* You can access instance variable as follows as well
        */
        System.out.println("Variable Value :" +
            myPuppy.puppyAge );
    }
}

```

Si compilamos y ejecutamos el programa anterior, producirá el siguiente resultado:

Salida

```

Name chosen is :tommy
Puppy's age is :2
Variable Value :2

```

Reglas de declaración de archivo de origen

Como última parte de esta sección, veamos ahora las reglas de declaración del archivo fuente. Estas reglas son esenciales cuando se declaran clases, declaraciones de *importación* y declaraciones de *paquetes* en un archivo fuente.

- Solo puede haber una clase pública por archivo fuente.
- Un archivo fuente puede tener múltiples clases no públicas.
- El nombre de la clase pública también debe ser el nombre del archivo fuente que debe **agregarse .java** al final. Por ejemplo: el nombre de la clase es *public class Employee {}*, entonces el archivo fuente debe ser como *Employee.java*.

- Si la clase se define dentro de un paquete, entonces la declaración del paquete debe ser la primera declaración en el archivo fuente.
- Si hay declaraciones de importación, entonces deben escribirse entre la declaración del paquete y la declaración de la clase. Si no hay declaraciones de paquete, la declaración de importación debe ser la primera línea del archivo fuente.
- Las declaraciones de importación y paquete implicarán a todas las clases presentes en el archivo fuente. No es posible declarar diferentes declaraciones de importación y / o paquete a diferentes clases en el archivo fuente.

Las clases tienen varios niveles de acceso y hay diferentes tipos de clases; clases abstractas, clases finales, etc. Explicaremos todo esto en el capítulo de modificadores de acceso.

Además de los tipos de clases mencionados anteriormente, Java también tiene algunas clases especiales llamadas clases internas y clases anónimas.

Paquete Java

En palabras simples, es una forma de categorizar las clases y las interfaces. Al desarrollar aplicaciones en Java, se escribirán cientos de clases e interfaces, por lo tanto, categorizar estas clases es imprescindible y hace la vida mucho más fácil.

Declaraciones de Importación

En Java, si se proporciona un nombre completo, que incluye el paquete y el nombre de la clase, el compilador puede localizar fácilmente el código fuente o las clases. La declaración de importación es una forma de proporcionar la ubicación adecuada para que el compilador encuentre esa clase en particular.

Por ejemplo, la siguiente línea le pediría al compilador que cargue todas las clases disponibles en el directorio `java_installation / java / io` -

```
import java.io.*;
```

Un estudio de caso simple

Para nuestro caso de estudio, crearemos dos clases. Son `Employee` y `EmployeeTest`.

Primero abra el bloc de notas y agregue el siguiente código. Recuerde que esta es la clase `Empleado` y la clase es una clase pública. Ahora, guarde este archivo fuente con el nombre `Employee.java`.

La clase `Employee` tiene cuatro variables de instancia: nombre, edad, designación y salario. La clase tiene un constructor definido explícitamente, que toma un parámetro.

Ejemplo

```

import java.io.*;
public class Employee {

    String name;
    int age;
    String designation;
    double salary;

    // This is the constructor of the class Employee
    public Employee(String name) {
        this.name = name;
    }

    // Assign the age of the Employee to the variable age.
    public void empAge(int empAge) {
        age = empAge;
    }

    /* Assign the designation to the variable designation.*/
    public void empDesignation(String empDesig) {
        designation = empDesig;
    }

    /* Assign the salary to the variable salary.*/
    public void empSalary(double empSalary) {
        salary = empSalary;
    }

    /* Print the Employee details */
    public void printEmployee() {
        System.out.println("Name:" + name );
        System.out.println("Age:" + age );
        System.out.println("Designation:" + designation );
        System.out.println("Salary:" + salary);
    }
}

```

Como se mencionó anteriormente en este tutorial, el procesamiento comienza desde el método principal. Por lo tanto, para que podamos ejecutar esta clase Employee, debe haber un método principal y se deben crear objetos. Crearemos una clase separada para estas tareas.

A continuación se muestra la clase *EmployeeTest*, que crea dos instancias de la clase Employee e invoca los métodos para cada objeto para asignar valores para cada variable.

Guarde el siguiente código en el archivo EmployeeTest.java.

```

import java.io.*;
public class EmployeeTest {

    public static void main(String args[]) {
        /* Create two objects using constructor */
        Employee empOne = new Employee("James Smith");
    }
}

```

```
Employee empTwo = new Employee("Mary Anne");

// Invoking methods for each object created
empOne.empAge(26);
empOne.empDesignation("Senior Software Engineer");
empOne.empSalary(1000);
empOne.printEmployee();

empTwo.empAge(21);
empTwo.empDesignation("Software Engineer");
empTwo.empSalary(500);
empTwo.printEmployee();
}
```

Ahora, compile ambas clases y luego ejecute *EmployeeTest* para ver el resultado de la siguiente manera:

Salida

```
C:\> javac Employee.java
C:\> javac EmployeeTest.java
C:\> java EmployeeTest
Name:James Smith
Age:26
Designation:Senior Software Engineer
Salary:1000.0
Name:Mary Anne
Age:21
Designation:Software Engineer
Salary:500.0
```

¿Lo que sigue?

En la próxima sesión, discutiremos los tipos de datos básicos en Java y cómo se pueden usar al desarrollar aplicaciones Java.

Java - Constructores

Un constructor inicializa un objeto cuando se crea. Tiene el mismo nombre que su clase y es sintácticamente similar a un método. Sin embargo, los constructores no tienen un tipo de retorno explícito.

Normalmente, usará un constructor para dar valores iniciales a las variables de instancia definidas por la clase, o para realizar cualquier otro procedimiento de inicio requerido para crear un objeto completamente formado.

Todas las clases tienen constructores, ya sea que defina uno o no, porque Java proporciona automáticamente un constructor predeterminado que inicializa todas las variables miembro a cero. Sin embargo, una vez que define su propio constructor, el constructor predeterminado ya no se usa.

Sintaxis

A continuación se muestra la sintaxis de un constructor:

```
class ClassName {  
    ClassName () {  
    }  
}
```

Java permite dos tipos de constructores, a saber:

- Sin argumento Constructores
- Constructores parametrizados

Sin argumento Constructores

Como el nombre especifica que los constructores sin argumentos de Java no aceptan ningún parámetro, utilizando estos constructores, las variables de instancia de un método se inicializarán con valores fijos para todos los objetos.

Ejemplo

```
Public class MyClass {  
    Int num;  
    MyClass() {  
        num = 100;  
    }  
}
```

Llamaría al constructor para inicializar objetos de la siguiente manera

```
public class ConsDemo {  
    public static void main(String args[]) {  
        MyClass t1 = new MyClass();  
        MyClass t2 = new MyClass();  
        System.out.println(t1.num + " " + t2.num);  
    }  
}
```

Esto produciría el siguiente resultado

100 100

Constructores parametrizados

Muy a menudo, necesitará un constructor que acepte uno o más parámetros. Los parámetros se agregan a un constructor de la misma manera que se agregan a un método, simplemente declare dentro de los paréntesis después del nombre del constructor.

Ejemplo

Aquí hay un ejemplo simple que usa un constructor:

```
// A simple constructor.
class MyClass {
    int x;

    // Following is the constructor
    MyClass(int i ) {
        x = i;
    }
}
```

Llamaría al constructor para inicializar objetos de la siguiente manera:

```
public class ConsDemo {
    public static void main(String args[]) {
        MyClass t1 = new MyClass( 10 );
        MyClass t2 = new MyClass( 20 );
        System.out.println(t1.x + " " + t2.x);
    }
}
```

Esto produciría el siguiente resultado:

```
10 20
```

Java - Tipos de datos básicos

Las variables no son más que ubicaciones de memoria reservadas para almacenar valores. Esto significa que cuando crea una variable, reserva algo de espacio en la memoria.

Según el tipo de datos de una variable, el sistema operativo asigna memoria y decide qué se puede almacenar en la memoria reservada. Por lo tanto, al asignar diferentes tipos de datos a las variables, puede almacenar enteros, decimales o caracteres en estas variables.

Hay dos tipos de datos disponibles en Java:

- Tipos de datos primitivos
- Tipos de datos de referencia / objeto

Tipos de datos primitivos

Hay ocho tipos de datos primitivos compatibles con Java. Los tipos de datos primitivos están predefinidos por el idioma y nombrados por una palabra clave. Veamos ahora en detalle los ocho tipos de datos primitivos.

byte

- El tipo de datos de byte es un entero de dos bytes con signo de 8 bits
- El valor mínimo es -128 (-2^7)
- El valor máximo es 127 (inclusive) ($2^7 - 1$)

- El valor predeterminado es 0
- El tipo de datos de byte se utiliza para ahorrar espacio en matrices grandes, principalmente en lugar de enteros, ya que un byte es cuatro veces más pequeño que un entero.
- Ejemplo: byte a = 100, byte b = -50

corto

- El tipo de datos cortos es un entero de dos bytes con signo de 16 bits
- El valor mínimo es -32,768 (-2^{15})
- El valor máximo es 32,767 (inclusive) ($2^{15} - 1$)
- El tipo de datos cortos también se puede utilizar para guardar memoria como tipo de datos de bytes. Un corto es 2 veces más pequeño que un entero
- El valor predeterminado es 0.
- Ejemplo: corto s = 10000, corto r = -20000

Entero

- El tipo de datos Int es un entero de complemento de dos con signo de 32 bits.
- El valor mínimo es -2,147,483,648 (-2^{31})
- El valor máximo es 2,147,483,647 (inclusive) ($2^{31} - 1$)
- El entero se usa generalmente como el tipo de datos predeterminado para los valores integrales, a menos que exista una preocupación por la memoria.
- El valor predeterminado es 0
- Ejemplo: int a = 100000, int b = -200000

largo

- El tipo de datos largo es un entero de complemento de dos con signo de 64 bits
- El valor mínimo es -9,223,372,036,854,775,808 (-2^{63})
- El valor máximo es 9.223.372.036.854.775.807 (inclusive) ($2^{63} - 1$)
- Este tipo se usa cuando se necesita un rango más amplio que int
- El valor predeterminado es 0L
- Ejemplo: largo a = 100000L, largo b = -200000L

flotador

- El tipo de datos flotantes es un punto flotante IEEE 754 de 32 bits de precisión simple
- Float se utiliza principalmente para ahorrar memoria en grandes conjuntos de números de coma flotante
- El valor predeterminado es 0.0f
- El tipo de datos flotantes nunca se usa para valores precisos como la moneda
- Ejemplo: flotador f1 = 234.5f

doble

- El tipo de datos doble es un punto flotante IEEE 754 de doble precisión de 64 bits
- Este tipo de datos generalmente se usa como el tipo de datos predeterminado para valores decimales, generalmente la opción predeterminada
- El tipo de datos doble nunca debe usarse para valores precisos como la moneda
- El valor predeterminado es 0.0d
- Ejemplo: `doble d1 = 123.4`

booleano

- el tipo de datos booleanos representa un bit de información
- Solo hay dos valores posibles: verdadero y falso
- Este tipo de datos se usa para indicadores simples que rastrean condiciones verdaderas / falsas
- El valor predeterminado es falso
- Ejemplo: `booleano uno = verdadero`

carbonizarse

- El tipo de datos char es un único carácter Unicode de 16 bits
- El valor mínimo es `'\u0000'` (o 0)
- El valor máximo es `'\uffff'` (o 65.535 inclusive)
- El tipo de datos Char se usa para almacenar cualquier personaje
- Ejemplo: `char letterA = 'A'`

Tipos de datos de referencia

- Las variables de referencia se crean utilizando constructores definidos de las clases. Se utilizan para acceder a objetos. Se declara que estas variables son de un tipo específico que no se puede cambiar. Por ejemplo, `Empleado`, `Cachorro`, etc.
- Los objetos de clase y varios tipos de variables de matriz vienen bajo el tipo de datos de referencia.
- El valor predeterminado de cualquier variable de referencia es nulo.
- Se puede usar una variable de referencia para referir cualquier objeto del tipo declarado o cualquier tipo compatible.
- Ejemplo: `Animal animal = nuevo Animal ("jirafa");`

Literales Java

Un literal es una representación del código fuente de un valor fijo. Se representan directamente en el código sin ningún cálculo.

Los literales se pueden asignar a cualquier variable de tipo primitivo. Por ejemplo

```
byte a = 68;  
char a = 'A';
```

byte, int, long y short pueden expresarse también en sistemas de números decimales (base 10), hexadecimales (base 16) u octales (base 8).

El prefijo 0 se usa para indicar octal, y el prefijo 0x indica hexadecimal cuando se usan estos sistemas numéricos para literales. Por ejemplo

```
int decimal = 100;  
int octal = 0144;  
int hexa = 0x64;
```

Los literales de cadena en Java se especifican como en la mayoría de los otros idiomas al encerrar una secuencia de caracteres entre un par de comillas dobles. Ejemplos de literales de cadena son:

Ejemplo

```
"Hello World"  
"two\nlines"  
 "\"This is in quotes\""
```

Los literales de tipo cadena y char pueden contener cualquier carácter Unicode. Por ejemplo

```
char a = '\u0001';  
String a = "\u0001";
```

El lenguaje Java también admite algunas secuencias de escape especiales para los literales String y char. Ellos son

Notación	Personaje representado
\n	Nueva línea (0x0a)
\r	Retorno de carro (0x0d)
\f	Alimentación de formulario (0x0c)
\b	Retroceso (0x08)
\s	Espacio (0x20)
\t	lengüeta

<code>\ "</code>	Cita doble
<code>\ '</code>	Una frase
<code>\\</code>	barra invertida
<code>\ ddd</code>	Carácter octal (ddd)
<code>\ uxxxx</code>	Carácter hexadecimal UNICODE (xxxx)

¿Lo que sigue?

Este capítulo explica los diversos tipos de datos. El siguiente tema explica diferentes tipos de variables y su uso. Esto le dará una buena comprensión de cómo se pueden usar en las clases, interfaces, etc. de Java.

Java - Tipos de variables

Una variable nos proporciona almacenamiento con nombre que nuestros programas pueden manipular. Cada variable en Java tiene un tipo específico, que determina el tamaño y el diseño de la memoria de la variable; el rango de valores que se pueden almacenar dentro de esa memoria; y el conjunto de operaciones que se pueden aplicar a la variable.

Debe declarar todas las variables antes de que puedan usarse. A continuación se presenta la forma básica de una declaración de variable:

```
data type variable [ = value][, variable [ = value] ...] ;
```

Aquí *el tipo de datos* es uno de los tipos de datos de Java y la *variable* es el nombre de la variable. Para declarar más de una variable del tipo especificado, puede usar una lista separada por comas.

Los siguientes son ejemplos válidos de declaración e inicialización de variables en Java:

Ejemplo

```
int a, b, c;           // Declares three ints, a, b, and c.
int a = 10, b = 10;    // Example of initialization
byte B = 22;           // initializes a byte type variable B.
double pi = 3.14159;   // declares and assigns a value of PI.
char a = 'A';          // the char variable a is initialized
with value 'a'
```

Este capítulo explicará varios tipos de variables disponibles en lenguaje Java. Hay tres tipos de variables en Java:

- Variables locales
- Variables de instancia
- Clase / variables estáticas

Variables Locales

- Las variables locales se declaran en métodos, constructores o bloques.
- Las variables locales se crean cuando se ingresa el método, el constructor o el bloque y la variable se destruirá una vez que salga del método, constructor o bloque.
- Los modificadores de acceso no pueden usarse para variables locales.
- Las variables locales solo son visibles dentro del método, constructor o bloque declarado.
- Las variables locales se implementan a nivel de pila internamente.
- No hay un valor predeterminado para las variables locales, por lo que las variables locales deben declararse y debe asignarse un valor inicial antes del primer uso.

Ejemplo

Aquí, la *edad* es una variable local. Esto se define dentro del método *pupAge()* y su alcance se limita solo a este método.

```
public class Test {  
    public void pupAge() {  
        int age = 0;  
        age = age + 7;  
        System.out.println("Puppy age is : " + age);  
    }  
  
    public static void main(String args[]) {  
        Test test = new Test();  
        test.pupAge();  
    }  
}
```

Esto producirá el siguiente resultado:

Salida

Puppy age is: 7

Ejemplo

El siguiente ejemplo usa *age* sin inicializarlo, por lo que daría un error en el momento de la compilación.

```
public class Test {
```

```

public void pupAge() {
    int age;
    age = age + 7;
    System.out.println("Puppy age is : " + age);
}

public static void main(String args[]) {
    Test test = new Test();
    test.pupAge();
}
}

```

Esto producirá el siguiente error al compilarlo:

Salida

```

Test.java:4:variable number might not have been initialized
age = age + 7;
      ^
1 error

```

Variables de instancia

- Las variables de instancia se declaran en una clase, pero fuera de un método, constructor o cualquier bloque.
- Cuando se asigna un espacio para un objeto en el montón, se crea un espacio para cada valor de variable de instancia.
- Las variables de instancia se crean cuando se crea un objeto con el uso de la palabra clave 'nuevo' y se destruye cuando se destruye el objeto.
- Las variables de instancia contienen valores que deben ser referenciados por más de un método, constructor o bloque, o partes esenciales del estado de un objeto que deben estar presentes en toda la clase.
- Las variables de instancia se pueden declarar en el nivel de clase antes o después del uso.
- Se pueden dar modificadores de acceso, por ejemplo, variables.
- Las variables de instancia son visibles para todos los métodos, constructores y bloques de la clase. Normalmente, se recomienda hacer que estas variables sean privadas (nivel de acceso). Sin embargo, se puede dar visibilidad para las subclases para estas variables con el uso de modificadores de acceso.
- Las variables de instancia tienen valores predeterminados. Para los números, el valor predeterminado es 0, para los booleanos es falso y para las referencias a objetos es nulo. Los valores pueden asignarse durante la declaración o dentro del constructor.
- Se puede acceder directamente a las variables de instancia llamando al nombre de la variable dentro de la clase. Sin embargo, dentro de los métodos estáticos (cuando las variables de instancia tienen accesibilidad), deben llamarse con el nombre completo. *ObjectReference.VariableName*.

Ejemplo

```

import java.io.*;
public class Employee {

    // this instance variable is visible for any child class.
    public String name;

    // salary variable is visible in Employee class only.
    private double salary;

    // The name variable is assigned in the constructor.
    public Employee (String empName) {
        name = empName;
    }

    // The salary variable is assigned a value.
    public void setSalary(double empSal) {
        salary = empSal;
    }

    // This method prints the employee details.
    public void printEmp() {
        System.out.println("name : " + name );
        System.out.println("salary :" + salary);
    }

    public static void main(String args[]) {
        Employee empOne = new Employee("Ransika");
        empOne.setSalary(1000);
        empOne.printEmp();
    }
}

```

Esto producirá el siguiente resultado:

Salida

```

name : Ransika
salary :1000.0

```

Clase / Variables Estáticas

- Las variables de clase también conocidas como variables estáticas se declaran con la palabra clave estática en una clase, pero fuera de un método, constructor o bloque.
- Solo habría una copia de cada variable de clase por clase, independientemente de cuántos objetos se creen a partir de ella.
- Raramente se usan variables estáticas, aparte de ser declaradas como constantes. Las constantes son variables que se declaran como públicas / privadas, finales y estáticas. Las variables constantes nunca cambian de su valor inicial.

- Las variables estáticas se almacenan en la memoria estática. Es raro utilizar variables estáticas que no sean declaradas finales y utilizadas como constantes públicas o privadas.
- Las variables estáticas se crean cuando se inicia el programa y se destruyen cuando se detiene.
- La visibilidad es similar a las variables de instancia. Sin embargo, la mayoría de las variables estáticas se declaran públicas, ya que deben estar disponibles para los usuarios de la clase.
- Los valores predeterminados son los mismos que las variables de instancia. Para los números, el valor predeterminado es 0; para los booleanos, es falso; y para referencias de objeto, es nulo. Los valores pueden asignarse durante la declaración o dentro del constructor. Además, los valores se pueden asignar en bloques especiales de inicializador estático.
- Se puede acceder a las variables estáticas llamando al nombre de clase *ClassName.VariableName*.
- Al declarar variables de clase como final estático público, los nombres de las variables (constantes) están en mayúsculas. Si las variables estáticas no son públicas y finales, la sintaxis de denominación es la misma que la instancia y las variables locales.

Ejemplo

```
import java.io.*;
public class Employee {

    // salary variable is a private static variable
    private static double salary;

    // DEPARTMENT is a constant
    public static final String DEPARTMENT = "Development ";

    public static void main(String args[]) {
        salary = 1000;
        System.out.println(DEPARTMENT + "average salary:" +
salary);
    }
}
```

Esto producirá el siguiente resultado:

Salida

Development average salary:1000

Nota : si se accede a las variables desde una clase externa, se debe acceder a la constante como Empleado.

¿Lo que sigue?

Ya ha utilizado modificadores de acceso (públicos y privados) en este capítulo. El próximo capítulo explicará los modificadores de acceso y los modificadores sin acceso en detalle.

Java - Tipos de modificadores

Los modificadores son palabras clave que agrega a esas definiciones para cambiar sus significados. El lenguaje Java tiene una amplia variedad de modificadores, incluidos los siguientes:

- Modificadores de acceso Java
- Modificadores sin acceso

Para usar un modificador, debe incluir su palabra clave en la definición de una clase, método o variable. El modificador precede al resto de la declaración, como en el siguiente ejemplo.

Ejemplo

```
public class className {  
    // ...  
}  
  
private boolean myFlag;  
static final double weeks = 9.5;  
protected static final int BOXWIDTH = 42;  
  
public static void main(String[] arguments) {  
    // body of method  
}
```

Modificadores de control de acceso

Java proporciona una serie de modificadores de acceso para establecer niveles de acceso para clases, variables, métodos y constructores. Los cuatro niveles de acceso son:

- Visible para el paquete, el valor predeterminado. No se necesitan modificadores.
- Visible solo para la clase (privado).
- Visible para el mundo (público).
- Visible para el paquete y todas las subclases (protegido).

Modificadores sin acceso

Java proporciona una serie de modificadores sin acceso para lograr muchas otras funcionalidades.

- El modificador *estático* para crear métodos y variables de clase.
- El modificador *final* para finalizar las implementaciones de clases, métodos y variables.

- El modificador *abstracto* para crear clases y métodos abstractos.
- Los modificadores *sincronizados* y *volátiles* , que se utilizan para subprocesos.

¿Lo que sigue?

En la siguiente sección, discutiremos sobre los operadores básicos utilizados en el lenguaje Java. El capítulo le dará una visión general de cómo se pueden usar estos operadores durante el desarrollo de la aplicación.

Java - Operadores básicos

Java proporciona un amplio conjunto de operadores para manipular variables. Podemos dividir todos los operadores de Java en los siguientes grupos:

- Operadores aritméticos
- Operadores relacionales
- Operadores bit a bit
- Operadores lógicos
- Operadores de Asignación
- Operadores diversos

Los operadores aritméticos

Los operadores aritméticos se usan en expresiones matemáticas de la misma manera que se usan en álgebra. La siguiente tabla enumera los operadores aritméticos:

Suponga que la variable entera A contiene 10 y la variable B contiene 20, entonces -

Operador	Descripción	Ejemplo
+ (Adición)	Agrega valores a ambos lados del operador.	A + B dará 30
- (Resta)	Resta el operando de la derecha del operando de la izquierda.	A - B dará -10
* (Multiplicación)	Multiplica los valores a cada lado del operador.	A * B dará 200
/ (División)	Divide el operando izquierdo por el operando derecho.	B / A

		dará 2
% (Módulo)	Divide el operando de la izquierda por el operando de la derecha y devuelve el resto.	B% A dará 0
++ (Incremento)	Aumenta el valor del operando en 1.	B ++ da 21
- (Decremento)	Disminuye el valor del operando en 1.	B-- da 19

Los operadores relacionales

Existen los siguientes operadores relacionales compatibles con el lenguaje Java.

Suponga que la variable A tiene 10 y la variable B tiene 20, entonces -

Mostrar ejemplos

Operador	Descripción	Ejemplo
== (igual a)	Comprueba si los valores de dos operandos son iguales o no, en caso afirmativo, la condición se vuelve verdadera.	(A == B) no es cierto.
!= (no es igual a)	Comprueba si los valores de dos operandos son iguales o no, si los valores no son iguales, la condición se vuelve verdadera.	(A != B) es cierto.
> (mayor que)	Comprueba si el valor del operando izquierdo es mayor que el valor del operando derecho, en caso afirmativo, la condición se vuelve verdadera.	(A > B) no es cierto.
< (menor que)	Comprueba si el valor del operando izquierdo es menor que el valor del operando derecho, en caso afirmativo, la condición se vuelve verdadera.	(A < B) es cierto.
>= (mayor o igual que)	Comprueba si el valor del operando izquierdo es mayor o igual que el valor del operando derecho, en caso afirmativo, la condición se vuelve verdadera.	(A >= B) no es cierto.

<= (menor o igual que)	Comprueba si el valor del operando izquierdo es menor o igual que el valor del operando derecho, en caso afirmativo, la condición se vuelve verdadera.	(A <= B) es cierto.
------------------------	--	---------------------

Los operadores bit a bit

Java define varios operadores bit a bit, que se pueden aplicar a los tipos enteros, long, int, short, char y byte.

El operador bit a bit trabaja en bits y realiza la operación bit a bit. Suponga que a = 60 y b = 13; ahora en formato binario serán los siguientes:

a = 0011 1100

b = 0000 1101

a & b = 0000 1100

a | b = 0011 1101

a ^ b = 0011 0001

~ a = 1100 0011

La siguiente tabla enumera los operadores bit a bit:

Suponga que la variable entera A tiene 60 y la variable B tiene 13 y luego -

Mostrar ejemplos

Operador	Descripción	Ejemplo
& (bit a bit y)	El operador binario AND copia un poco al resultado si existe en ambos operandos.	(A y B) dará 12, que es 0000 1100
El (bit a bit o)	El operador binario O copia un bit si existe en cualquiera de los operandos.	(A B) dará 61, que es 0011 1101
^ (XOR bit a bit)	El operador binario XOR copia el bit si está establecido en un operando pero no en ambos.	(A ^ B) dará 49, que es 0011 0001
~ (cumplido bit a bit)	El operador de complemento de binarios es unario y tiene el efecto de "voltear" los bits.	(~ A) dará -61, que es 1100 0011 en forma de complemento a 2 debido a un número binario con signo.

<< (desplazamiento a la izquierda)	Operador binario de desplazamiento a la izquierda. El valor de los operandos de la izquierda se mueve hacia la izquierda por la cantidad de bits especificados por el operando de la derecha.	Un << 2 dará 240, que es 1111 0000
>> (desplazamiento a la derecha)	Operador binario de desplazamiento a la derecha. El valor de los operandos de la izquierda se mueve hacia la derecha por la cantidad de bits especificados por el operando de la derecha.	A >> 2 dará 15, que es 1111
>>> (relleno cero desplazamiento a la derecha)	Desplazar a la derecha del operador de relleno cero. El valor de los operandos de la izquierda se mueve hacia la derecha por la cantidad de bits especificados por el operando de la derecha y los valores desplazados se rellenan con ceros.	A >>> 2 dará 15, que es 0000 1111

Los operadores lógicos

La siguiente tabla enumera los operadores lógicos:

Suponga que las variables booleanas A son verdaderas y la variable B es falsa, entonces -

Mostrar ejemplos

Operador	Descripción	Ejemplo
&& (lógico y)	Llamado operador lógico AND. Si ambos operandos son distintos de cero, la condición se vuelve verdadera.	(A y B) es falso
(lógico o)	Llamado Lógico O Operador. Si alguno de los dos operandos no es cero, entonces la condición se vuelve verdadera.	(A B) es cierto
! (lógico no)	Llamado operador lógico NO. Se usa para invertir el estado lógico de su operando. Si una condición es verdadera, entonces el operador lógico NO hará falso.	! (A y B) es cierto

Los operadores de asignación

Los siguientes son los operadores de asignación compatibles con el lenguaje Java:

Mostrar ejemplos

Operador	Descripción	Ejemplo
=	Operador de asignación simple. Asigna valores de operandos del lado derecho al operando del lado izquierdo.	$C = A + B$ asignará el valor de $A + B$ a C
+ =	Añadir operador de asignación AND. Agrega el operando derecho al operando izquierdo y asigna el resultado al operando izquierdo.	$C + = A$ es equivalente a $C = C + A$
- =	Restar operador de asignación AND. Resta el operando derecho del operando izquierdo y asigna el resultado al operando izquierdo.	$C - = A$ es equivalente a $C = C - A$
* =	Operador de multiplicación Y asignación. Multiplica el operando derecho con el operando izquierdo y asigna el resultado al operando izquierdo.	$C * = A$ es equivalente a $C = C * A$
/ =	Dividir Y operador de asignación. Divide el operando izquierdo con el operando derecho y asigna el resultado al operando izquierdo.	$C / = A$ es equivalente a $C = C / A$
% =	Operador de módulo Y asignación. Toma módulo usando dos operandos y asigna el resultado al operando izquierdo.	$C \% = A$ es equivalente a $C = C \% A$
<< =	Desplazamiento a la izquierda Y operador de asignación.	$C << = 2$ es lo mismo que $C = C << 2$
>> =	Desplazamiento a la derecha Y operador de asignación.	$C >> = 2$ es lo mismo que $C = C >> 2$

& =	Operador de asignación Y a nivel de bit.	C & 2 es lo mismo que C & 2
^ =	OR exclusivo bit a bit y operador de asignación.	C ^ 2 es lo mismo que C ^ 2
=	OR inclusivo a nivel de bit y operador de asignación.	C 2 es lo mismo que C 2

Operadores Misceláneos

Hay algunos otros operadores compatibles con Java Language.

Operador condicional (?:)

El operador condicional también se conoce como **operador ternario**. Este operador consta de tres operandos y se utiliza para evaluar expresiones booleanas. El objetivo del operador es decidir qué valor debe asignarse a la variable. El operador se escribe como -

```
variable x = (expression) ? value if true : value if false
```

El siguiente es un ejemplo:

Ejemplo

```
public class Test {

    public static void main(String args[]) {
        int a, b;
        a = 10;
        b = (a == 1) ? 20 : 30;
        System.out.println( "Value of b is : " + b );

        b = (a == 10) ? 20 : 30;
        System.out.println( "Value of b is : " + b );
    }
}
```

Esto producirá el siguiente resultado:

Salida

```
Value of b is : 30
```

Value of b is : 20

instancia del operador

Este operador se usa solo para variables de referencia de objeto. El operador verifica si el objeto es de un tipo particular (tipo de clase o tipo de interfaz). El operador instanceof se escribe como -

```
( Object reference variable ) instanceof (class/interface type)
```

Si el objeto referido por la variable en el lado izquierdo del operador pasa la verificación IS-A para el tipo de clase / interfaz en el lado derecho, entonces el resultado será verdadero. El siguiente es un ejemplo:

Ejemplo

```
public class Test {  
  
    public static void main(String args[]) {  
  
        String name = "James";  
  
        // following will return true since name is type of  
String  
        boolean result = name instanceof String;  
        System.out.println( result );  
    }  
}
```

Esto producirá el siguiente resultado:

Salida

true

Este operador aún devolverá verdadero, si el objeto que se compara es la asignación compatible con el tipo de la derecha. El siguiente es un ejemplo más:

Ejemplo

```
class Vehicle {}  
  
public class Car extends Vehicle {  
  
    public static void main(String args[]) {  
  
        Vehicle a = new Car();  
        boolean result = a instanceof Car;  
        System.out.println( result );  
    }  
}
```

Esto producirá el siguiente resultado:

Salida

```
true
```

Precedencia de operadores Java

La precedencia del operador determina la agrupación de términos en una expresión. Esto afecta cómo se evalúa una expresión. Ciertos operadores tienen mayor prioridad que otros; por ejemplo, el operador de multiplicación tiene mayor prioridad que el operador de suma:

Por ejemplo, $x = 7 + 3 * 2$; aquí x se asigna 13, no 20 porque el operador $*$ tiene mayor prioridad que $+$, por lo que primero se multiplica por $3 * 2$ y luego se suma a 7.

Aquí, los operadores con la precedencia más alta aparecen en la parte superior de la tabla, aquellos con la más baja aparecen en la parte inferior. Dentro de una expresión, los operadores de mayor precedencia se evaluarán primero.

Categoría	Operador	Asociatividad
Sufijo	expresión ++ expresión--	De izquierda a derecha
Unario	++ expresión --expresión + expresión -expresión ~!	De derecha a izquierda
Multiplicativo	* /%	De izquierda a derecha
Aditivo	+ -	De izquierda a derecha
Cambio	<< >> >>>	De izquierda a derecha
Relacional	<> <=> = instancia de	De izquierda a derecha
Igualdad	==! =	De izquierda a derecha
Bitwise Y	Y	De izquierda a derecha
Bitwise XOR	^	De izquierda a derecha

Bitwise O	EI	De izquierda a derecha
Y lógico	&&	De izquierda a derecha
O lógico		De izquierda a derecha
Condicional	?:	De derecha a izquierda
Asignación	= + = - = * = / = % = ^ = = << = >> = >>> =	De derecha a izquierda

¿Lo que sigue?

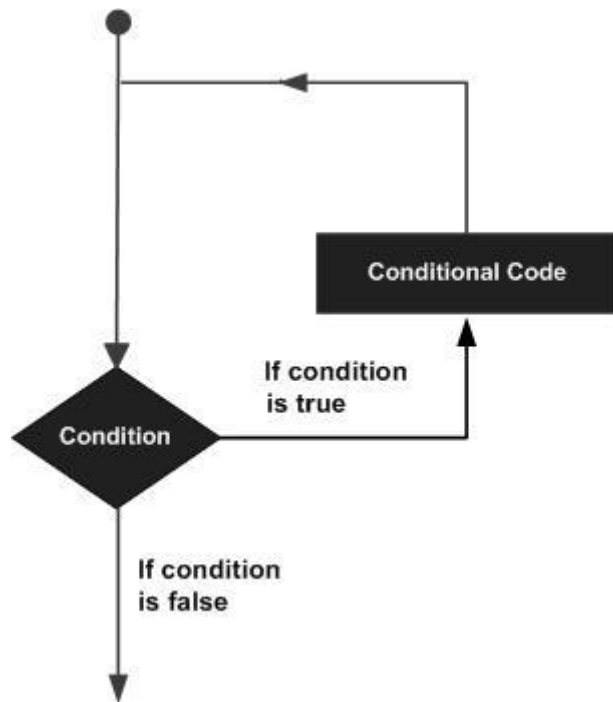
El siguiente capítulo explicará sobre el control de bucle en la programación Java. El capítulo describirá varios tipos de bucles y cómo estos bucles se pueden usar en el desarrollo de programas Java y para qué fines se están utilizando.

Java - Control de bucle

Puede haber una situación en la que necesite ejecutar un bloque de código varias veces. En general, las instrucciones se ejecutan secuencialmente: la primera instrucción de una función se ejecuta primero, seguida de la segunda, y así sucesivamente.

Los lenguajes de programación proporcionan diversas estructuras de control que permiten rutas de ejecución más complicadas.

Una declaración de **bucle** nos permite ejecutar una declaración o grupo de declaraciones varias veces y la siguiente es la forma general de una declaración de bucle en la mayoría de los lenguajes de programación:



El lenguaje de programación Java proporciona los siguientes tipos de bucle para manejar los requisitos de bucle. Haga clic en los siguientes enlaces para verificar sus detalles.

No Señor.	Bucle y descripción
1	<u>mientras bucle</u> Repite una declaración o grupo de declaraciones mientras una condición dada es verdadera. Prueba la condición antes de ejecutar el cuerpo del bucle.
2	<u>en bucle</u> Ejecute una secuencia de declaraciones varias veces y abrevia el código que administra la variable de bucle.
3	<u>hacer ... mientras bucle</u> Como una declaración while, excepto que prueba la condición al final del cuerpo del bucle.

Declaraciones de control de bucle

Las instrucciones de control de bucle cambian la ejecución de su secuencia normal. Cuando la ejecución deja un ámbito, todos los objetos automáticos que se crearon en ese ámbito se destruyen.

Java admite las siguientes declaraciones de control. Haga clic en los siguientes enlaces para verificar sus detalles.

No Señor.	Declaración de control y descripción
1	<u>declaración de ruptura</u> Termina la instrucción loop o switch y transfiere la ejecución a la instrucción que sigue inmediatamente al loop o switch.
2	<u>continuar declaración</u> Hace que el bucle omita el resto de su cuerpo e inmediatamente vuelva a probar su condición antes de reiterar.

Mejorado para loop en Java

A partir de Java 5, se introdujo el bucle for mejorado. Esto se utiliza principalmente para atravesar la colección de elementos, incluidas las matrices.

Sintaxis

A continuación se muestra la sintaxis de bucle mejorado para -

```
for(declaration : expression) {
    // Statements
}
```

- **Declaración** : la variable de bloque recientemente declarada es de un tipo compatible con los elementos de la matriz a la que está accediendo. La variable estará disponible dentro del bloque for y su valor sería el mismo que el elemento de matriz actual.
- **Expresión** : esto evalúa la matriz que necesita recorrer. La expresión puede ser una variable de matriz o una llamada a un método que devuelve una matriz.

Ejemplo

```
public class Test {

    public static void main(String args[]) {
        int [] numbers = {10, 20, 30, 40, 50};

        for(int x : numbers ) {
            System.out.print( x );
            System.out.print(",");
        }
        System.out.print("\n");
        String [] names = {"James", "Larry", "Tom", "Lacy"};

        for( String name : names ) {
```

```
        System.out.print( name );  
        System.out.print(",");  
    }  
}  
}
```

Esto producirá el siguiente resultado:

Salida

```
10, 20, 30, 40, 50,  
James, Larry, Tom, Lacy,
```

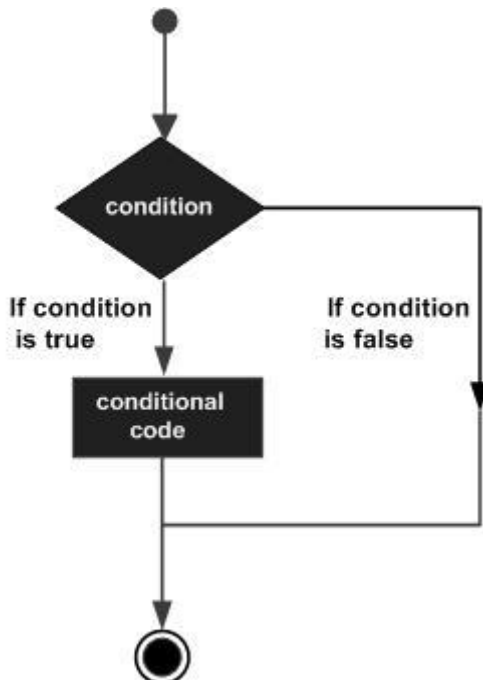
¿Lo que sigue?

En el siguiente capítulo, aprenderemos sobre las declaraciones de toma de decisiones en la programación Java.

Java - Toma de decisiones

Las estructuras de toma de decisiones tienen una o más condiciones para ser evaluadas o probadas por el programa, junto con una declaración o declaraciones que se ejecutarán si se determina que la condición es verdadera y, opcionalmente, otras declaraciones que se ejecutarán si se determina la condición ser falso

A continuación se presenta la forma general de una estructura de toma de decisiones típica que se encuentra en la mayoría de los lenguajes de programación:



El lenguaje de programación Java proporciona los siguientes tipos de declaraciones de toma de decisiones. Haga clic en los siguientes enlaces para verificar sus detalles.

No Señor.	Declaración y descripción
1	<u>si la declaración</u> Una declaración if consiste en una expresión booleana seguida de una o más declaraciones.
2	<u>si ... otra declaración</u> Una instrucción if puede ser seguida por una instrucción else opcional , que se ejecuta cuando la expresión booleana es falsa.
3	<u>anidada si la declaración</u> Puede usar una declaración if o else if dentro de otra declaración if o else if (s).
4 4	<u>declaración de cambio</u> Una declaración de cambio permite que una variable sea probada para la igualdad contra una lista de valores.

Los ? : Operador

Hemos cubierto **operador condicional?** : en el capítulo anterior que se puede usar para reemplazar las declaraciones **if ... else** . Tiene la siguiente forma general:

`Exp1 ? Exp2 : Exp3;`

Donde Exp1, Exp2 y Exp3 son expresiones. Observe el uso y la colocación del colon.

Para determinar el valor de toda la expresión, inicialmente se evalúa exp1.

- Si el valor de exp1 es verdadero, entonces el valor de Exp2 será el valor de toda la expresión.
- Si el valor de exp1 es falso, se evalúa Exp3 y su valor se convierte en el valor de toda la expresión.

¿Lo que sigue?

En el próximo capítulo, discutiremos sobre la clase Number (en el paquete java.lang) y sus subclases en lenguaje Java.

Analizaremos algunas de las situaciones en las que utilizará las instancias de estas clases en lugar de los tipos de datos primitivos, así como clases como el formato, las funciones matemáticas que debe conocer cuando trabaje con Numbers.

Java - Clase de números

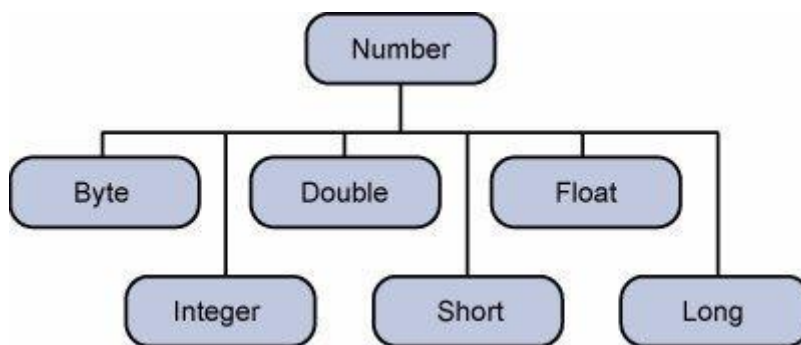
Normalmente, cuando trabajamos con números, usamos tipos de datos primitivos como byte, int, long, double, etc.

Ejemplo

```
int i = 5000;  
float gpa = 13.65;  
double mask = 0xaf;
```

Sin embargo, en el desarrollo, nos encontramos con situaciones en las que necesitamos usar objetos en lugar de tipos de datos primitivos. Para lograr esto, Java proporciona **clases de contenedor**.

Todas las clases de contenedor (Entero, Largo, Byte, Doble, Flotante, Corto) son subclases de la clase abstracta Número.



El objeto de la clase contenedora contiene o ajusta su respectivo tipo de datos primitivos. La conversión de tipos de datos primitivos en objeto se llama **boxeo**, y el compilador se encarga de esto. Por lo tanto, al usar una clase de envoltura, solo necesita pasar el valor del tipo de datos primitivo al constructor de la clase de envoltura.

Y el objeto Wrapper se convertirá de nuevo a un tipo de datos primitivo, y este proceso se llama unboxing. La clase **Number** es parte del paquete java.lang.

El siguiente es un ejemplo de boxeo y unboxing:

Ejemplo

```
public class Test {  
  
    public static void main(String args[]) {  
        Integer x = 5; // boxes int to an Integer object  
        x = x + 10;    // unboxes the Integer to a int  
        System.out.println(x);  
    }  
}
```

Esto producirá el siguiente resultado:

Salida

Cuando a x se le asigna un valor entero, el compilador encajona el número entero porque x es un objeto entero. Más tarde, x se desempaqueta para que se puedan agregar como un entero.

Métodos numéricos

La siguiente es la lista de los métodos de instancia que implementan todas las subclases de la clase Number:

No Señor.	Método y descripción
1	<u>xxxValue ()</u> Convierte el valor de este objeto Number en el tipo de datos xxx y lo devuelve.
2	<u>comparar con()</u> Compara este objeto Número con el argumento.
3	<u>es igual a ()</u> Determina si este objeto numérico es igual al argumento.
4 4	<u>valor de()</u> Devuelve un objeto entero que contiene el valor de la primitiva especificada.
5 5	<u>Encadenar()</u> Devuelve un objeto String que representa el valor de un int o Integer especificado.
6 6	<u>parseInt ()</u> Este método se utiliza para obtener el tipo de datos primitivos de una determinada cadena.
7 7	<u>absominales()</u> Devuelve el valor absoluto del argumento.
8	<u>fortificar techo()</u> Devuelve el entero más pequeño que es mayor o igual que el argumento. Devuelto como un doble.
9 9	<u>piso()</u> Devuelve el entero más grande que es menor o igual que el

	argumento. Devuelto como un doble.
10	<u>rint ()</u> Devuelve el entero que tiene el valor más cercano al argumento. Devuelto como un doble.
11	<u>redondo()</u> Devuelve el largo o int más cercano, como lo indica el tipo de retorno del método al argumento.
12	<u>min ()</u> Devuelve el menor de los dos argumentos.
13	<u>max ()</u> Devuelve el mayor de los dos argumentos.
14	<u>Exp()</u> Devuelve la base de los logaritmos naturales, e, a la potencia del argumento.
15	<u>Iniciar sesión()</u> Devuelve el logaritmo natural del argumento.
dieciséis	<u>pow ()</u> Devuelve el valor del primer argumento elevado a la potencia del segundo argumento.
17	<u>sqrt ()</u> Devuelve la raíz cuadrada del argumento.
18 años	<u>pecado()</u> Devuelve el seno del valor doble especificado.
19	<u>cos ()</u> Devuelve el coseno del valor doble especificado.
20	<u>bronceado()</u> Devuelve la tangente del valor doble especificado.
21	<u>como en()</u> Devuelve el arcoseno del valor doble especificado.
22	<u>acos ()</u>

	Devuelve el arcocoseno del valor doble especificado.
23	<u>un bronceado()</u> Devuelve el arcotangente del valor doble especificado.
24	<u>atan2 ()</u> Convierte coordenadas rectangulares (x, y) en coordenadas polares (r, theta) y devuelve theta.
25	<u>toDegrees ()</u> Convierte el argumento en grados.
26	<u>toRadians ()</u> Convierte el argumento en radianes.
27	<u>aleatorio()</u> Devuelve un número aleatorio.

¿Lo que sigue?

En la siguiente sección, veremos la clase Character en Java. Aprenderá a usar caracteres de objeto y caracteres de tipo de datos primitivos en Java.

Java - Clase de personaje

Normalmente, cuando trabajamos con caracteres, usamos tipos de datos primitivos char.

Ejemplo

```
char ch = 'a';

// Unicode for uppercase Greek omega character
char uniChar = '\u0391';

// an array of chars
char[] charArray = { 'a', 'b', 'c', 'd', 'e' };
```

Sin embargo, en el desarrollo, nos encontramos con situaciones en las que necesitamos usar objetos en lugar de tipos de datos primitivos. Para lograr esto, Java proporciona el **carácter de** clase envoltura para el tipo de datos primitivo char.

La clase de caracteres ofrece varios métodos de clase útiles (es decir, estáticos) para manipular caracteres. Puede crear un objeto Character con el constructor Character:

```
Character ch = new Character('a');
```

El compilador de Java también creará un objeto `Character` en algunas circunstancias. Por ejemplo, si pasa un carácter primitivo a un método que espera un objeto, el compilador convierte automáticamente el carácter en un personaje para usted. Esta característica se llama `autoboxing` o `unboxing`, si la conversión se realiza en sentido contrario.

Ejemplo

```
// Here following primitive char 'a'
// is boxed into the Character object ch
Character ch = 'a';

// Here primitive 'x' is boxed for method test,
// return is unboxed to char 'c'
char c = test('x');
```

Secuencias de escape

Un carácter precedido por una barra invertida (`\`) es una secuencia de escape y tiene un significado especial para el compilador.

El carácter de nueva línea (`\n`) se ha utilizado con frecuencia en este tutorial en las declaraciones `System.out.println()` para avanzar a la siguiente línea después de que se imprime la cadena.

La siguiente tabla muestra las secuencias de escape de Java:

Secuencia de escape	Descripción
<code>\t</code>	Inserta una pestaña en el texto en este punto.
<code>\s</code>	Inserta un retroceso en el texto en este punto.
<code>\n</code>	Inserta una nueva línea en el texto en este punto.
<code>\r</code>	Inserta un retorno de carro en el texto en este punto.
<code>\f</code>	Inserta un feed de formulario en el texto en este punto.
<code>\'</code>	Inserta una comilla simple en el texto en este punto.

\ "	Inserta un carácter de comillas dobles en el texto en este punto.
\\	Inserta un carácter de barra diagonal inversa en el texto en este punto.

Cuando se encuentra una secuencia de escape en una declaración de impresión, el compilador la interpreta en consecuencia.

Ejemplo

Si desea poner comillas entre comillas, debe usar la secuencia de escape, \ ", en las comillas interiores -

```
public class Test {
    public static void main(String args[]) {
        System.out.println("She said \"Hello!\" to me.");
    }
}
```

Esto producirá el siguiente resultado:

Salida

```
She said "Hello!" to me.
```

Métodos de personaje

La siguiente es la lista de los métodos de instancia importantes que implementan todas las subclases de la clase Character:

No Señor.	Método y descripción
1	<u>isLetter ()</u> Determina si el valor de char especificado es una letra.
2	<u>isDigit ()</u> Determina si el valor de char especificado es un dígito.
3	<u>isWhitespace ()</u> Determina si el valor de char especificado es un espacio en blanco.
4 4	<u>isUpperCase ()</u>

	Determina si el valor de char especificado es mayúscula.
5 5	<u>isLowerCase ()</u> Determina si el valor de char especificado es minúscula.
6 6	<u>toUpperCase ()</u> Devuelve la forma en mayúscula del valor de char especificado.
7 7	<u>toLowerCase ()</u> Devuelve la forma en minúscula del valor de char especificado.
8	<u>Encadenar()</u> Devuelve un objeto String que representa el valor de carácter especificado, es decir, una cadena de un carácter.

Para obtener una lista completa de los métodos, consulte la especificación de la API `java.lang.Character`.

¿Lo que sigue?

En la siguiente sección, veremos la clase `String` en Java. Aprenderá cómo declarar y usar cadenas de manera eficiente, así como algunos de los métodos importantes en la clase de cadena.

Java - Clase de cadenas

Las cadenas, que se utilizan ampliamente en la programación Java, son una secuencia de caracteres. En el lenguaje de programación Java, las cadenas se tratan como objetos.

La plataforma Java proporciona la clase `String` para crear y manipular cadenas.

Creando cadenas

La forma más directa de crear una cadena es escribir:

```
String greeting = "Hello world!";
```

Cada vez que encuentra un literal de cadena en su código, el compilador crea un objeto de cadena con su valor en este caso, "¡Hola, mundo!".

Al igual que con cualquier otro objeto, puede crear objetos `String` utilizando la nueva palabra clave y un constructor. La clase `String` tiene 11 constructores que le permiten proporcionar el valor inicial de la cadena utilizando diferentes fuentes, como una matriz de caracteres.

Ejemplo

```
public class StringDemo {  
  
    public static void main(String args[]) {  
        char[] helloArray = { 'h', 'e', 'l', 'l', 'o', '.' };  
        String helloString = new String(helloArray);  
        System.out.println( helloString );  
    }  
}
```

Esto producirá el siguiente resultado:

Salida

hello.

Nota : la clase String es inmutable, por lo que una vez que se crea un objeto String no se puede cambiar. Si es necesario realizar muchas modificaciones en las Cadenas de caracteres, entonces debe usar las Clases de Buffer de Cadena y de Generador de Cadena .

Longitud de la cuerda

Los métodos utilizados para obtener información sobre un objeto se conocen como **métodos de acceso** . Un método de acceso que puede usar con cadenas es el método length (), que devuelve el número de caracteres contenidos en el objeto de cadena.

El siguiente programa es un ejemplo de **longitud ()** , método Clase de cadena.

Ejemplo

```
public class StringDemo {  
  
    public static void main(String args[]) {  
        String palindrome = "Dot saw I was Tod";  
        int len = palindrome.length();  
        System.out.println( "String Length is : " + len );  
    }  
}
```

Esto producirá el siguiente resultado:

Salida

String Length is : 17

Cuerdas Concatenadas

La clase String incluye un método para concatenar dos cadenas:

```
string1.concat(string2);
```

Esto devuelve una nueva cadena que es cadena1 con cadena2 agregada al final. También puede usar el método concat () con literales de cadena, como en -

```
"My name is ".concat("Zara");
```

Las cadenas se concatenan más comúnmente con el operador +, como en -

```
"Hello," + " world" + "!"
```

lo que resulta en -

```
"Hello, world!"
```

Veamos el siguiente ejemplo:

Ejemplo

```
public class StringDemo {  
  
    public static void main(String args[]) {  
        String string1 = "saw I was ";  
        System.out.println("Dot " + string1 + "Tod");  
    }  
}
```

Esto producirá el siguiente resultado:

Salida

```
Dot saw I was Tod
```

Crear cadenas de formato

Tiene los métodos printf () y format () para imprimir la salida con números formateados. La clase String tiene un método de clase equivalente, format (), que devuelve un objeto String en lugar de un objeto PrintStream.

El uso del método de formato estático () de String le permite crear una cadena con formato que puede reutilizar, en lugar de una declaración de impresión de una sola vez. Por ejemplo, en lugar de -

Ejemplo

```
System.out.printf("The value of the float variable is " +  
                  "%f, while the value of the integer " +  
                  "variable is %d, and the string " +  
                  "is %s", floatVar, intVar, stringVar);
```

Puedes escribir

```
String fs;
```

```
fs = String.format("The value of the float variable is " +
    "%f, while the value of the integer " +
    "variable is %d, and the string " +
    "is %s", floatVar, intVar, stringVar);
System.out.println(fs);
```

Métodos de cuerda

Aquí está la lista de métodos admitidos por la clase String:

No Señor.	Método y descripción
1	<u>char charAt (int int)</u> Devuelve el carácter en el índice especificado.
2	<u>int compareTo (Objeto o)</u> Compara esta cadena con otro objeto.
3	<u>int compareTo (String anotherString)</u> Compara dos cadenas lexicográficamente.
4 4	<u>int compareToIgnoreCase (String str)</u> Compara dos cadenas lexicográficamente, ignorando las diferencias entre mayúsculas y minúsculas.
5 5	<u>String concat (String str)</u> Concatena la cadena especificada al final de esta cadena.
6 6	<u>content booleano Equals (StringBuffer sb)</u> Devuelve verdadero si y solo si esta Cadena representa la misma secuencia de caracteres que el StringBuffer especificado.
7 7	<u>string estático copyValueOf (datos char [])</u> Devuelve una cadena que representa la secuencia de caracteres en la matriz especificada.
8	<u>static String copyValueOf (datos char [], int offset, int count)</u> Devuelve una cadena que representa la secuencia de caracteres en la matriz especificada.
9 9	<u>extremos booleanos con (sufijo de cadena)</u>

	Comprueba si esta cadena termina con el sufijo especificado.
10	<u>boolean esIgual a (Object anObject)</u> Compara esta cadena con el objeto especificado.
11	<u>boolean equalsIgnoreCase (String anotherString)</u> Compara esta Cadena con otra Cadena, ignorando las consideraciones del caso.
12	<u>byte getBytes ()</u> Codifica esta cadena en una secuencia de bytes utilizando el juego de caracteres predeterminado de la plataforma, almacenando el resultado en una nueva matriz de bytes.
13	<u>byte [] getBytes (String charsetName)</u> Codifica esta cadena en una secuencia de bytes usando el juego de caracteres nombrado, almacenando el resultado en una nueva matriz de bytes.
14	<u>void getChars (int srcBegin, int srcEnd, char [] dst, int dstBegin)</u> Copia los caracteres de esta cadena en la matriz de caracteres de destino.
15	<u>int hashCode ()</u> Devuelve un código hash para esta cadena.
dieciséis	<u>int indexOf (int ch)</u> Devuelve el índice dentro de esta cadena de la primera aparición del carácter especificado.
17	<u>int indexOf (int ch, int fromIndex)</u> Devuelve el índice dentro de esta cadena de la primera aparición del carácter especificado, comenzando la búsqueda en el índice especificado.
18 años	<u>int indexOf (String str)</u> Devuelve el índice dentro de esta cadena de la primera aparición de la subcadena especificada.
19	<u>int indexOf (String str, int fromIndex)</u> Devuelve el índice dentro de esta cadena de la primera aparición de la subcadena especificada, comenzando en el índice especificado.
20	<u>Pasante interno ()</u> Devuelve una representación canónica para el objeto de cadena.
21	<u>int lastIndexOf (int ch)</u>

	Devuelve el índice dentro de esta cadena de la última aparición del carácter especificado.
22	<u>int lastIndexOf (int ch, int fromIndex)</u> Devuelve el índice dentro de esta cadena de la última aparición del carácter especificado, buscando hacia atrás comenzando en el índice especificado.
23	<u>int lastIndexOf (String str)</u> Devuelve el índice dentro de esta cadena de la aparición más a la derecha de la subcadena especificada.
24	<u>int lastIndexOf (String str, int fromIndex)</u> Devuelve el índice dentro de esta cadena de la última aparición de la subcadena especificada, buscando hacia atrás comenzando en el índice especificado.
25	<u>int length ()</u> Devuelve la longitud de esta cadena.
26	<u>coincidencias booleanas (expresión regular de cadena)</u> Indica si esta cadena coincide o no con la expresión regular dada.
27	<u>boolean regionMatches (boolean ignoreCase, int toffset, String other, int ooffset, int len)</u> Comprueba si dos regiones de cadena son iguales.
28	<u>boolean regionMatches (int toffset, String other, int ooffset, int len)</u> Comprueba si dos regiones de cadena son iguales.
29	<u>Reemplazo de cadena (char oldChar, char newChar)</u> Devuelve una nueva cadena resultante de reemplazar todas las apariciones de oldChar en esta cadena con newChar.
30	<u>String replaceAll (regex de cadena, reemplazo de cadena)</u> Reemplaza cada subcadena de esta cadena que coincide con la expresión regular dada con el reemplazo dado.
31	<u>String replaceFirst (expresión de cadena, reemplazo de cadena)</u> Reemplaza la primera subcadena de esta cadena que coincide con la expresión regular dada con el reemplazo dado.
32	<u>String [] split (expresión de cadena)</u> Divide esta cadena alrededor de coincidencias de la expresión regular dada.

33	<u>String [] split (String regex, int limit)</u> Divide esta cadena alrededor de coincidencias de la expresión regular dada.
34	<u>boolean comienza con (prefijo de cadena)</u> Comprueba si esta cadena comienza con el prefijo especificado.
35	<u>boolean comienza con (prefijo de cadena, int toffset)</u> Comprueba si esta cadena comienza con el prefijo especificado que comienza un índice específico.
36	<u>Subsecuencia de secuencia de caracteres (int beginIndex, int endIndex)</u> Devuelve una nueva secuencia de caracteres que es una subsecuencia de esta secuencia.
37	<u>Subcadena de cadena (int beginIndex)</u> Devuelve una nueva cadena que es una subcadena de esta cadena.
38	<u>Subcadena de cadena (int beginIndex, int endIndex)</u> Devuelve una nueva cadena que es una subcadena de esta cadena.
39	<u>char [] toCharArray ()</u> Convierte esta cadena en una nueva matriz de caracteres.
40	<u>String toLowerCase ()</u> Convierte todos los caracteres de esta cadena en minúsculas utilizando las reglas de la configuración regional predeterminada.
41	<u>String toLowerCase (Locale locale)</u> Convierte todos los caracteres de esta cadena en minúsculas utilizando las reglas de la configuración regional dada.
42	<u>String toString ()</u> Este objeto (que ya es una cadena) se devuelve.
43	<u>String toUpperCase ()</u> Convierte todos los caracteres de esta cadena a mayúsculas utilizando las reglas de la configuración regional predeterminada.
44	<u>String toUpperCase (configuración regional)</u> Convierte todos los caracteres de esta cadena a mayúsculas utilizando las reglas de la configuración regional dada.
45	<u>Recorte de cuerda ()</u>

	Devuelve una copia de la cadena, con espacios en blanco iniciales y finales omitidos.
46	<u>valor de cadena estático De (tipo de datos primitivo x)</u> Devuelve la representación de cadena del argumento de tipo de datos pasado.

Java: matrices

Java proporciona una estructura de datos, la **matriz**, que almacena una colección secuencial de tamaño fijo de elementos del mismo tipo. Una matriz se usa para almacenar una colección de datos, pero a menudo es más útil pensar en una matriz como una colección de variables del mismo tipo.

En lugar de declarar variables individuales, como número0, número1, ... y número99, declara una variable de matriz como números y usa números [0], números [1] y ..., números [99] para representar variables individuales

Este tutorial presenta cómo declarar variables de matriz, crear matrices y procesar matrices usando variables indexadas.

Declaración de variables de matriz

Para usar una matriz en un programa, debe declarar una variable para hacer referencia a la matriz, y debe especificar el tipo de matriz a la que la variable puede hacer referencia. Aquí está la sintaxis para declarar una variable de matriz:

Sintaxis

```
dataType[] arrayRefVar;    // preferred way.
or
dataType arrayRefVar[];    // works but not preferred way.
```

Nota - Se prefiere el estilo **dataType [] arrayRefVar**. El estilo **dataType arrayRefVar []** proviene del lenguaje C / C ++ y fue adoptado en Java para acomodar a los programadores C / C ++.

Ejemplo

Los siguientes fragmentos de código son ejemplos de esta sintaxis:

```
double[] myList;    // preferred way.
or
double myList[];    // works but not preferred way.
```

Crear matrices

Puede crear una matriz utilizando el nuevo operador con la siguiente sintaxis:

Sintaxis

```
arrayRefVar = new dataType[arraySize];
```

La declaración anterior hace dos cosas:

- Crea una matriz usando el nuevo tipo de datos [arraySize].
- Asigna la referencia de la matriz recién creada a la variable arrayRefVar.

Declarar una variable de matriz, crear una matriz y asignar la referencia de la matriz a la variable se puede combinar en una declaración, como se muestra a continuación:

```
dataType[] arrayRefVar = new dataType[arraySize];
```

Alternativamente, puede crear matrices de la siguiente manera:

```
dataType[] arrayRefVar = {value0, value1, ..., valuek};
```

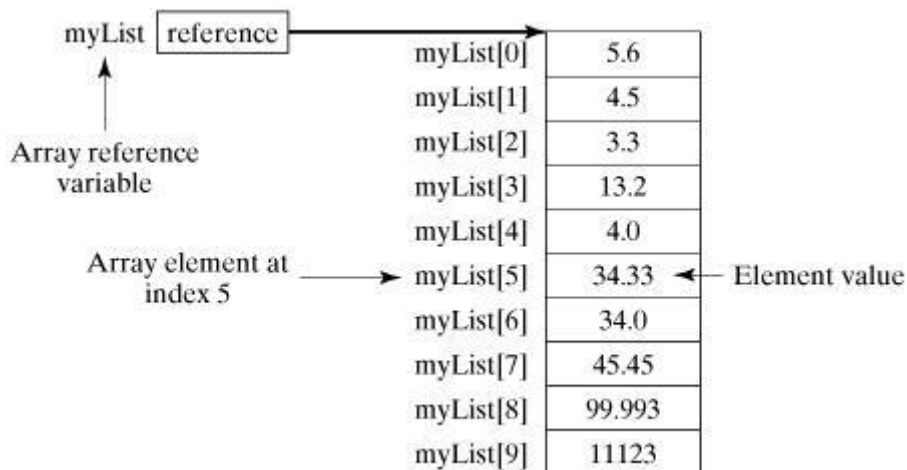
Se accede a los elementos de la matriz a través del **índice**. Los índices de matriz están basados en 0; es decir, comienzan desde 0 hasta **arrayRefVar.length-1**.

Ejemplo

La siguiente declaración declara una variable de matriz, myList, crea una matriz de 10 elementos de doble tipo y asigna su referencia a myList:

```
double[] myList = new double[10];
```

La siguiente imagen representa la matriz myList. Aquí, myList contiene diez valores dobles y los índices son de 0 a 9.



Procesando matrices

Cuando procesamos elementos de matriz, a menudo usamos **for** loop o **foreach** loop porque todos los elementos en una matriz son del mismo tipo y se conoce el tamaño de la matriz.

Ejemplo

Aquí hay un ejemplo completo que muestra cómo crear, inicializar y procesar matrices:

```
public class TestArray {  
  
    public static void main(String[] args) {  
        double[] myList = {1.9, 2.9, 3.4, 3.5};  
  
        // Print all the array elements  
        for (int i = 0; i < myList.length; i++) {  
            System.out.println(myList[i] + " ");  
        }  
  
        // Summing all elements  
        double total = 0;  
        for (int i = 0; i < myList.length; i++) {  
            total += myList[i];  
        }  
        System.out.println("Total is " + total);  
  
        // Finding the largest element  
        double max = myList[0];  
        for (int i = 1; i < myList.length; i++) {  
            if (myList[i] > max) max = myList[i];  
        }  
        System.out.println("Max is " + max);  
    }  
}
```

Esto producirá el siguiente resultado:

Salida

```
1.9  
2.9  
3.4  
3.5  
Total is 11.7  
Max is 3.5
```

Los bucles foreach

JDK 1.5 introdujo un nuevo bucle for conocido como foreach loop o mejorado for loop, que le permite atravesar la matriz completa secuencialmente sin usar una variable de índice.

Ejemplo

El siguiente código muestra todos los elementos en la matriz myList:

```
public class TestArray {  
  
    public static void main(String[] args) {  
        double[] myList = {1.9, 2.9, 3.4, 3.5};  
  
        // Print all the array elements  
        for (double element: myList) {  
            System.out.println(element);  
        }  
    }  
}
```

Esto producirá el siguiente resultado:

Salida

```
1.9  
2.9  
3.4  
3.5
```

Pasar matrices a métodos

Así como puede pasar valores de tipo primitivos a métodos, también puede pasar matrices a métodos. Por ejemplo, el siguiente método muestra los elementos en una matriz **int** :

Ejemplo

```
public static void printArray(int[] array) {  
    for (int i = 0; i < array.length; i++) {  
        System.out.print(array[i] + " ");  
    }  
}
```

Puede invocarlo pasando una matriz. Por ejemplo, la siguiente instrucción invoca el método `printArray` para mostrar 3, 1, 2, 6, 4 y 2:

Ejemplo

```
printArray(new int[]{3, 1, 2, 6, 4, 2});
```

Devolver una matriz de un método

Un método también puede devolver una matriz. Por ejemplo, el siguiente método devuelve una matriz que es la inversión de otra matriz:

Ejemplo

```
public static int[] reverse(int[] list) {
```

```

int[] result = new int[list.length];

for (int i = 0, j = result.length - 1; i < list.length;
i++, j--) {
    result[j] = list[i];
}
return result;
}

```

La clase de matrices

La clase `java.util.Arrays` contiene varios métodos estáticos para ordenar y buscar matrices, comparar matrices y llenar elementos de matriz. Estos métodos están sobrecargados para todos los tipos primitivos.

No Señor.	Método y descripción
1	<p>public static int binarySearch (Object [] a, Object key)</p> <p>Busca en la matriz especificada de Object (Byte, Int, double, etc.) el valor especificado utilizando el algoritmo de búsqueda binaria. La matriz se debe ordenar antes de realizar esta llamada. Esto devuelve el índice de la clave de búsqueda, si está contenido en la lista; de lo contrario, devuelve (- (punto de inserción + 1)).</p>
2	<p>público estático booleano igual (largo [] a, largo [] a2)</p> <p>Devuelve verdadero si las dos matrices de longitudes especificadas son iguales entre sí. Dos matrices se consideran iguales si ambas matrices contienen el mismo número de elementos, y todos los pares de elementos correspondientes en las dos matrices son iguales. Esto devuelve verdadero si las dos matrices son iguales. Todos los demás tipos de datos primitivos podrían usar el mismo método (Byte, short, Int, etc.)</p>
3	<p>relleno de vacío público estático (int [] a, int val)</p> <p>Asigna el valor int especificado a cada elemento de la matriz de entradas especificada. El mismo método podría ser utilizado por todos los demás tipos de datos primitivos (Byte, short, Int, etc.)</p>
4 4	<p>orden público vacío estático (Objeto [] a)</p> <p>Ordena la matriz especificada de objetos en un orden ascendente, de acuerdo con el orden natural de sus elementos. El mismo método podría ser utilizado por todos los demás tipos de datos primitivos (Byte, short, Int, etc.)</p>

Java: fecha y hora

Java proporciona la clase **Date** disponible en el paquete **java.util** , esta clase encapsula la fecha y hora actuales.

La clase Date admite dos constructores como se muestra en la siguiente tabla.

No Señor.	Constructor y Descripción
1	Fecha() Este constructor inicializa el objeto con la fecha y hora actuales.
2	Fecha (milisegundos largos) Este constructor acepta un argumento que equivale al número de milisegundos que han transcurrido desde la medianoche del 1 de enero de 1970.

Los siguientes son los métodos de la clase de fecha.

No Señor.	Método y descripción
1	booleano después (Fecha fecha) Devuelve verdadero si el objeto Fecha de invocación contiene una fecha posterior a la especificada por fecha; de lo contrario, devuelve falso.
2	booleano antes (fecha fecha) Devuelve verdadero si el objeto Fecha de invocación contiene una fecha anterior a la especificada por fecha; de lo contrario, devuelve falso.
3	Objeto clon () Duplica el objeto de invocación de fecha.
4 4	int compareTo (fecha fecha) Compara el valor del objeto que invoca con el de fecha. Devuelve 0 si los valores son iguales. Devuelve un valor negativo si el objeto que invoca es anterior a la fecha. Devuelve un valor positivo si el objeto que invoca es posterior a la fecha.
5 5	int compareTo (Objeto obj) Funciona de manera idéntica para compareTo (Fecha) si obj es de la clase Fecha. De lo contrario, arroja una ClassCastException.
6 6	boolean igual (Fecha del objeto)

	Devuelve verdadero si el objeto Fecha de invocación contiene la misma hora y fecha que la especificada por fecha; de lo contrario, devuelve falso.
7 7	largo getTime () Devuelve el número de milisegundos que han transcurrido desde el 1 de enero de 1970.
8	int hashCode () Devuelve un código hash para el objeto que invoca.
9 9	nulo setTime (mucho tiempo) Establece la hora y la fecha según lo especificado por la hora, que representa un tiempo transcurrido en milisegundos desde la medianoche del 1 de enero de 1970.
10	String toString () Convierte el objeto de invocación Date en una cadena y devuelve el resultado.

Obtener fecha y hora actuales

Este es un método muy fácil para obtener la fecha y hora actual en Java. Puede usar un simple objeto Fecha con el método *toString ()* para imprimir la fecha y hora actuales de la siguiente manera:

Ejemplo

```
import java.util.Date;
public class DateDemo {

    public static void main(String args[]) {
        // Instantiate a Date object
        Date date = new Date();

        // display time and date using toString()
        System.out.println(date.toString());
    }
}
```

Esto producirá el siguiente resultado:

Salida

on May 04 09:51:52 CDT 2009

Comparación de fechas

Las siguientes son las tres formas de comparar dos fechas:

- Puede usar `getTime ()` para obtener el número de milisegundos que han transcurrido desde la medianoche, 1 de enero de 1970, para ambos objetos y luego comparar estos dos valores.
- Puede usar los métodos `before ()`, `after ()` y `equals ()`. Debido a que el 12 del mes viene antes del 18, por ejemplo, la nueva Fecha (99, 2, 12). Antes (nueva Fecha (99, 2, 18)) devuelve verdadero.
- Puede usar el método `compareTo ()`, que se define mediante la interfaz `Comparable` y se implementa por `Date`.

Formato de fecha con SimpleDateFormat

`SimpleDateFormat` es una clase concreta para formatear y analizar fechas de una manera local. `SimpleDateFormat` le permite comenzar eligiendo cualquier patrón definido por el usuario para el formato de fecha y hora.

Ejemplo

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

public class DateDemo {

    public static void main(String args[]) {
        Date dNow = new Date( );
        SimpleDateFormat ft =
            new SimpleDateFormat ("E yyyy.MM.dd 'at' hh:mm:ss a
zzz");

        System.out.println("Current Date: " + ft.format(dNow));
    }
}
```

Esto producirá el siguiente resultado:

Salida

```
Current Date: Sun 2004.07.18 at 04:14:09 PM PDT
```

Códigos de formato simple de formato de fecha

Para especificar el formato de hora, use una cadena de patrón de hora. En este patrón, todas las letras ASCII se reservan como letras de patrón, que se definen de la siguiente manera:

Personaje	Descripción	Ejemplo
sol	Designador de la era	ANUNCIO
y	Año en cuatro dígitos	2001
METRO	Mes en año	Julio o 07
re	Día en mes	10
h	Hora en AM / PM (1 ~ 12)	12
H	Hora del día (0 ~ 23)	22
metro	Minuto en hora	30
s	Segundo en minuto	55
S	Milisegundo	234
mi	Día en semana	martes
re	Día en año	360
F	Día de la semana en el mes	2 (segundo miércoles de julio)
w	Semana en el año	40
W	Semana en mes	1
un	Marcador AM / PM	PM
k	Hora del día (1 ~ 24)	24

K	Hora en AM / PM (0 ~ 11)	10
z	Zona horaria	hora estándar del Este
'	Escapar de texto	Delimitador
"	Una frase	``

Formato de fecha con printf

El formateo de fecha y hora se puede hacer muy fácilmente usando el método **printf**. Utiliza un formato de dos letras, que comienza con **t** y termina en una de las letras de la tabla como se muestra en el siguiente código.

Ejemplo

```
import java.util.Date;
public class DateDemo {

    public static void main(String args[]) {
        // Instantiate a Date object
        Date date = new Date();

        // display time and date
        String str = String.format("Current Date/Time : %tc",
date );

        System.out.printf(str);
    }
}
```

Esto producirá el siguiente resultado:

Salida

```
Current Date/Time : Sat Dec 15 16:37:57 MST 2012
```

Sería un poco tonto si tuviera que proporcionar la fecha varias veces para formatear cada parte. Por esa razón, una cadena de formato puede indicar el índice del argumento que se formateará.

El índice debe seguir inmediatamente al% y debe terminarse con un \$.

Ejemplo

```
import java.util.Date;
public class DateDemo {

    public static void main(String args[]) {
        // Instantiate a Date object
        Date date = new Date();

        // display time and date
        System.out.printf("%1$s %2$tB %2$td, %2$tY", "Due
date:", date);
    }
}
```

Esto producirá el siguiente resultado:

Salida

Due date: February 09, 2004

Alternativamente, puede usar la bandera <. Indica que el mismo argumento que en la especificación de formato anterior debe usarse nuevamente.

Ejemplo

```
import java.util.Date;
public class DateDemo {

    public static void main(String args[]) {
        // Instantiate a Date object
        Date date = new Date();

        // display formatted date
        System.out.printf("%s %tB %<te, %<tY", "Due date:",
date);
    }
}
```

Esto producirá el siguiente resultado:

Salida

Due date: February 09, 2004

Caracteres de conversión de fecha y hora

Personaje	Descripción	Ejemplo
-----------	-------------	---------

C	Fecha y hora completas	Lun 04 de mayo 09:51:52 CDT 2009
F	Fecha ISO 8601	2004-02-09
re	Fecha con formato de EE. UU. (Mes / día / año)	02/09/2004
T	Tiempo de 24 horas	18:05:19
r	Tiempo de 12 horas	06:05:19 pm
R	Tiempo de 24 horas, sin segundos	18:05
Y	Año de cuatro dígitos (con ceros a la izquierda)	2004
y	Últimos dos dígitos del año (con ceros a la izquierda)	04
C	Primeros dos dígitos del año (con ceros a la izquierda)	20
si	Nombre completo del mes	febrero
si	Nombre abreviado del mes	feb
metro	Mes de dos dígitos (con ceros a la izquierda)	02
re	Día de dos dígitos (con ceros a la izquierda)	03
mi	Día de dos dígitos (sin ceros a la izquierda)	9 9
UN	Nombre completo del día de la semana	lunes
un	Nombre abreviado del día de la semana	Lun

j	Día del año de tres dígitos (con ceros a la izquierda)	069
H	Hora de dos dígitos (con ceros a la izquierda), entre 00 y 23	18 años
k	Hora de dos dígitos (sin ceros a la izquierda), entre 0 y 23	18 años
yo	Hora de dos dígitos (con ceros a la izquierda), entre 01 y 12	06
l	Hora de dos dígitos (sin ceros a la izquierda), entre 1 y 12	6 6
METRO	Minutos de dos dígitos (con ceros a la izquierda)	05
S	Segundos de dos dígitos (con ceros a la izquierda)	19
L	Milisegundos de tres dígitos (con ceros a la izquierda)	047
norte	Nanosegundos de nueve dígitos (con ceros a la izquierda)	047000000
PAG	Marcador de mañana o tarde en mayúscula	PM
pag	Marcador de mañana o tarde en minúscula	pm
z	RFC 822 desplazamiento numérico de GMT	-0800
Z	Zona horaria	PST
s	Segundos desde 1970-01-01 00:00:00 GMT	1078884319

Q	Milisegundos desde 1970-01-01 00:00:00 GMT	1078884319047
---	--	---------------

Hay otras clases útiles relacionadas con la fecha y la hora. Para obtener más detalles, puede consultar la documentación de Java Standard.

Analizando cadenas en fechas

La clase `SimpleDateFormat` tiene algunos métodos adicionales, en particular `parse()`, que intenta analizar una cadena de acuerdo con el formato almacenado en el objeto `SimpleDateFormat` dado.

Ejemplo

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

public class DateDemo {

    public static void main(String args[]) {
        SimpleDateFormat ft = new SimpleDateFormat ("yyyy-MM-dd");
        String input = args.length == 0 ? "1818-11-11" :
args[0];

        System.out.print(input + " Parses as ");
        Date t;
        try {
            t = ft.parse(input);
            System.out.println(t);
        } catch (ParseException e) {
            System.out.println("Unparseable using " + ft);
        }
    }
}
```

Una ejecución de muestra del programa anterior produciría el siguiente resultado:

Salida

```
1818-11-11 Parses as Wed Nov 11 00:00:00 EST 1818
```

Durmiendo un rato

Puede dormir durante cualquier período de tiempo, desde un milisegundo hasta la vida útil de su computadora. Por ejemplo, el siguiente programa dormiría durante 3 segundos:

Ejemplo

```
import java.util.*;
public class SleepDemo {

    public static void main(String args[]) {
        try {
            System.out.println(new Date( ) + "\n");
            Thread.sleep(5*60*10);
            System.out.println(new Date( ) + "\n");
        } catch (Exception e) {
            System.out.println("Got an exception!");
        }
    }
}
```

Esto producirá el siguiente resultado:

Salida

```
Sun May 03 18:04:41 GMT 2009
Sun May 03 18:04:51 GMT 2009
```

Medición del tiempo transcurrido

A veces, es posible que necesite medir un punto en el tiempo en milisegundos. Así que volvamos a escribir el ejemplo anterior una vez más:

Ejemplo

```
import java.util.*;
public class DiffDemo {

    public static void main(String args[]) {
        try {
            long start = System.currentTimeMillis( );
            System.out.println(new Date( ) + "\n");

            Thread.sleep(5*60*10);
            System.out.println(new Date( ) + "\n");

            long end = System.currentTimeMillis( );
            long diff = end - start;
            System.out.println("Difference is : " + diff);
        } catch (Exception e) {
            System.out.println("Got an exception!");
        }
    }
}
```

Esto producirá el siguiente resultado:

Salida

```
Sun May 03 18:16:51 GMT 2009
Sun May 03 18:16:57 GMT 2009
Difference is : 5993
```

Clase de calendario gregoriano

GregorianCalendar es una implementación concreta de una clase Calendar que implementa el calendario gregoriano normal con el que está familiarizado. No discutimos la clase Calendar en este tutorial, puede buscar documentación estándar de Java para esto.

El método **getInstance ()** de Calendar devuelve un GregorianCalendar inicializado con la fecha y hora actuales en la ubicación y zona horaria predeterminadas. GregorianCalendar define dos campos: AD y BC. Estos representan las dos eras definidas por el calendario gregoriano.

También hay varios constructores para objetos GregorianCalendar:

No Señor.	Constructor y Descripción
1	Calendario Gregoriano() Construye un calendario gregoriano predeterminado utilizando la hora actual en la zona horaria predeterminada con la configuración regional predeterminada.
2	Calendario gregoriano (int año, int mes, int fecha) Construye un calendario gregoriano con la fecha dada establecida en la zona horaria predeterminada con la configuración regional predeterminada.
3	Calendario gregoriano (int año, int mes, int fecha, int hora, int minuto) Construye un calendario gregoriano con la fecha y hora establecidas para la zona horaria predeterminada con la configuración regional predeterminada.
4 4	Calendario gregoriano (int año, int mes, int fecha, int hora, int minuto, int segundo) Construye un calendario gregoriano con la fecha y hora establecidas para la zona horaria predeterminada con la configuración regional predeterminada.
5 5	GregorianCalendar (Locale aLocale) Construye un calendario gregoriano basado en la hora actual en la zona horaria

	predeterminada con la configuración regional dada.
6 6	Calendario gregoriano (zona horaria) Construye un calendario gregoriano basado en la hora actual en la zona horaria dada con la configuración regional predeterminada.
7 7	GregorianCalendar (zona horaria, Locale aLocale) Construye un calendario gregoriano basado en la hora actual en la zona horaria dada con la configuración regional dada.

Aquí está la lista de algunos métodos de soporte útiles proporcionados por la clase `GregorianCalendar`:

No Señor.	Método y descripción
1	void add (campo int, cantidad int) Agrega la cantidad de tiempo especificada (firmada) al campo de tiempo dado, según las reglas del calendario.
2	vacío protegido computeFields () Convierte UTC como milisegundos en valores de campo de tiempo.
3	vacío protegido computeTime () Anula el calendario Convierte los valores del campo de tiempo a UTC en milisegundos.
4 4	boolean igual (objeto obj) Compara este <code>GregorianCalendar</code> con una referencia de objeto.
5 5	int get (campo int) Obtiene el valor para un campo de tiempo dado.
6 6	int getActualMaximum (campo int) Devuelve el valor máximo que podría tener este campo, dada la fecha actual.
7 7	int getActualMinimum (campo int) Devuelve el valor mínimo que podría tener este campo, dada la fecha actual.

8	int getGreatestMinimum (campo int) Devuelve el valor mínimo más alto para el campo dado si varía.
9 9	Fecha getGregorianChange () Obtiene la fecha de cambio del calendario gregoriano.
10	int getLeastMaximum (campo int) Devuelve el valor máximo más bajo para el campo dado si varía.
11	int getMaximum (campo int) Devuelve el valor máximo para el campo dado.
12	Fecha getTime () Obtiene la hora actual de este calendario.
13	largo getTimeInMillis () Obtiene el tiempo actual de este calendario como largo.
14	TimeZone getTimeZone () Obtiene la zona horaria.
15	int getMinimum (campo int) Devuelve el valor mínimo para el campo dado.
dieciséis	int hashCode () Invalida el código hash.
17	boolean isLeapYear (int año) Determina si el año dado es un año bisiesto.
18 años	rollo vacío (campo int, booleano arriba) Agrega o resta (arriba / abajo) una sola unidad de tiempo en el campo de tiempo dado sin cambiar los campos más grandes.
19	conjunto vacío (campo int, valor int)

	Establece el campo de tiempo con el valor dado.
20	conjunto nulo (int año, int mes, int fecha) Establece los valores para los campos año, mes y fecha.
21	conjunto nulo (int año, int mes, int fecha, int hora, int minuto) Establece los valores para los campos año, mes, fecha, hora y minuto.
22	conjunto nulo (int año, int mes, int fecha, int hora, int minuto, int segundo) Establece los valores para los campos año, mes, fecha, hora, minuto y segundo.
23	void setGregorianChange (Fecha fecha) Establece la fecha de cambio de GregorianCalendar.
24	nulo setTime (fecha fecha) Establece la hora actual de este calendario con la fecha dada.
25	nulo setTimeInMillis (long millis) Establece la hora actual de este calendario a partir del valor largo dado.
26	nulo setTimeZone (valor de TimeZone) Establece la zona horaria con el valor de zona horaria dado.
27	String toString () Devuelve una representación de cadena de este calendario.

Ejemplo

```
import java.util.*;
public class GregorianCalendarDemo {

    public static void main(String args[]) {
        String months[] = {"Jan", "Feb", "Mar", "Apr", "May",
            "Jun", "Jul", "Aug", "Sep",
            "Oct", "Nov", "Dec"};

        int year;
        // Create a Gregorian calendar initialized
```

```

// with the current date and time in the
// default locale and timezone.

GregorianCalendar gcalendar = new GregorianCalendar();

// Display current time and date information.
System.out.print("Date: ");

System.out.print(months[gcalendar.get(Calendar.MONTH)]);
System.out.print(" " + gcalendar.get(Calendar.DATE) + "
");
System.out.println(year =
gcalendar.get(Calendar.YEAR));
System.out.print("Time: ");
System.out.print(gcalendar.get(Calendar.HOUR) + ":");
System.out.print(gcalendar.get(Calendar.MINUTE) + ":");
System.out.println(gcalendar.get(Calendar.SECOND));

// Test if the current year is a leap year
if(gcalendar.isLeapYear(year)) {
    System.out.println("The current year is a leap
year");
} else {
    System.out.println("The current year is not a leap
year");
}
}
}

```

Esto producirá el siguiente resultado:

Salida

```

Date: Apr 22 2009
Time: 11:25:27
The current year is not a leap year

```

Para obtener una lista completa de las constantes disponibles en la clase Calendario, puede consultar la documentación estándar de Java.

Java - Expresiones regulares

Java proporciona el paquete `java.util.regex` para la coincidencia de patrones con expresiones regulares. Las expresiones regulares de Java son muy similares al lenguaje de programación Perl y muy fáciles de aprender.

Una expresión regular es una secuencia especial de caracteres que le ayuda a unir o encontrar otras cadenas o conjuntos de cadenas, utilizando una sintaxis especializada contenida en un patrón. Se pueden usar para buscar, editar o manipular texto y datos.

El paquete `java.util.regex` consta principalmente de las siguientes tres clases:

- **Clase de patrón** : un objeto de patrón es una representación compilada de una expresión regular. La clase `Pattern` no proporciona constructores públicos. Para

crear un patrón, primero debe invocar uno de sus métodos públicos de **compilación** estática (**()**), que luego devolverá un objeto Patrón. Estos métodos aceptan una expresión regular como primer argumento.

- **Clase Matcher** : un objeto Matcher es el motor que interpreta el patrón y realiza operaciones de coincidencia con una cadena de entrada. Al igual que la clase Pattern, Matcher no define constructores públicos. **Obtiene** un objeto Matcher invocando el método **matcher ()** en un objeto Pattern.
- **PatternSyntaxException** : un objeto PatternSyntaxException es una excepción no verificada que indica un error de sintaxis en un patrón de expresión regular.

Capturando grupos

Los grupos de captura son una forma de tratar a varios personajes como una sola unidad. Se crean colocando los caracteres que se agruparán dentro de un conjunto de paréntesis. Por ejemplo, la expresión regular (perro) crea un solo grupo que contiene las letras "d", "o" y "g".

Los grupos de captura se numeran contando sus paréntesis de apertura de izquierda a derecha. En la expresión ((A) (B (C))), por ejemplo, hay cuatro de estos grupos:

- ((A B C)))
- (UN)
- (ANTES DE CRISTO))
- (C)

Para averiguar cuántos grupos están presentes en la expresión, llame al método `groupCount` en un objeto de coincidencia. El método `groupCount` devuelve un **int** que muestra el número de grupos de captura presentes en el patrón del emparejador.

También hay un grupo especial, el grupo 0, que siempre representa la expresión completa. Este grupo no está incluido en el total informado por `groupCount`.

Ejemplo

El siguiente ejemplo ilustra cómo encontrar una cadena de dígitos de la cadena alfanumérica dada:

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegexMatches {

    public static void main( String args[] ) {
        // String to be scanned to find the pattern.
        String line = "This order was placed for QT3000! OK?";
        String pattern = "(.*) (\\d+) (.*)";

        // Create a Pattern object
        Pattern r = Pattern.compile(pattern);
```

```

// Now create matcher object.
Matcher m = r.matcher(line);
if (m.find( )) {
    System.out.println("Found value: " + m.group(0) );
    System.out.println("Found value: " + m.group(1) );
    System.out.println("Found value: " + m.group(2) );
}else {
    System.out.println("NO MATCH");
}
}
}

```

Esto producirá el siguiente resultado:

Salida

```

Found value: This order was placed for QT3000! OK?
Found value: This order was placed for QT300
Found value: 0

```

Sintaxis de expresiones regulares

Aquí está la tabla que enumera todas las sintaxis de metacaracteres de expresiones regulares disponibles en Java:

Subexpresión	Partidos
^	Coincide con el comienzo de la línea.
\$	Coincide con el final de la línea.
.	Coincide con cualquier carácter, excepto la nueva línea. Usar la opción m también le permite coincidir con la nueva línea.
[...]	Coincide con cualquier carácter entre paréntesis.
[^ ...]	Coincide con cualquier carácter que no esté entre paréntesis
\u	Comienzo de toda la cuerda.
\z	Fin de toda la cadena.

<code>\Z</code>	Fin de toda la cadena excepto el terminador de línea final permitido.
<code>re*</code>	Coincide con 0 o más ocurrencias de la expresión anterior.
<code>re +</code>	Coincide con 1 o más de lo anterior.
<code>¿re?</code>	Coincide con 0 o 1 aparición de la expresión anterior.
<code>re {n}</code>	Coincide exactamente n número de ocurrencias de la expresión anterior.
<code>re {n,}</code>	Coincide con n o más ocurrencias de la expresión anterior.
<code>re {n, m}</code>	Coincide al menos n y como máximo m ocurrencias de la expresión anterior.
<code>a si</code>	Coincide con a o b.
<code>(re)</code>	Agrupar expresiones regulares y recordar el texto coincidente.
<code>(?: re)</code>	Agrupar expresiones regulares sin recordar el texto coincidente.
<code>(?> re)</code>	Coincide con el patrón independiente sin retroceder.
<code>\w</code>	Coincide con los caracteres de la palabra.
<code>\W</code>	Coincide con los caracteres que no son palabras.
<code>\s</code>	Coincide con el espacio en blanco. Equivalente a <code>[\t\n\r\f]</code> .
<code>\S</code>	Coincide con el espacio en blanco.
<code>\d</code>	Coincide con los dígitos. Equivalente a <code>[0-9]</code> .
<code>\D</code>	Coincide con los no dígitos.

\UN	Coincide con el comienzo de la cadena.
\Z	Coincide con el final de la cadena. Si existe una nueva línea, coincide justo antes de la nueva línea.
\z	Coincide con el final de la cadena.
\SOL	Coincide con el punto donde terminó el último partido.
\norte	Referencia posterior al número de grupo de captura "n".
\si	Coincide con los límites de la palabra cuando está fuera de los corchetes. Coincide con el retroceso (0x08) cuando está dentro de los corchetes.
\SI	Coincide con los límites sin palabras.
\n, \t, etc.	Coincide con nuevas líneas, retornos de carro, pestañas, etc.
\Q	Escapar (citar) todos los caracteres hasta \E.
\M	Finaliza la cita comenzada con \Q.

Métodos de la clase Matcher

Aquí hay una lista de métodos de instancia útiles:

Métodos de índice

Los métodos de índice proporcionan valores de índice útiles que muestran con precisión dónde se encontró la coincidencia en la cadena de entrada:

No Señor.	Método y descripción
1	public int start () Devuelve el índice de inicio de la coincidencia anterior.

2	public int start (grupo int) Devuelve el índice de inicio de la subsecuencia capturada por el grupo dado durante la operación de coincidencia anterior.
3	public int end () Devuelve el desplazamiento después de que coincida el último carácter.
4 4	public int end (int group) Devuelve el desplazamiento después del último carácter de la subsecuencia capturado por el grupo dado durante la operación de coincidencia anterior.

Métodos de estudio

Los métodos de estudio revisan la cadena de entrada y devuelven un valor booleano que indica si se encuentra o no el patrón:

No Señor.	Método y descripción
1	Aspecto público booleano () Intenta hacer coincidir la secuencia de entrada, comenzando por el comienzo de la región, con el patrón.
2	public boolean find () Intenta encontrar la siguiente subsecuencia de la secuencia de entrada que coincida con el patrón.
3	public boolean find (int start) Restablece este emparejador y luego intenta encontrar la siguiente subsecuencia de la secuencia de entrada que coincida con el patrón, comenzando en el índice especificado.
4 4	partidos booleanos públicos () Intenta hacer coincidir toda la región con el patrón.

Métodos de reemplazo

Los métodos de reemplazo son métodos útiles para reemplazar texto en una cadena de entrada:

No Señor.	Método y descripción
1	public Matcher appendReplacement (StringBuffer sb, reemplazo de cadena) Implementa un paso de agregar y reemplazar no terminal.
2	public StringBuffer appendTail (StringBuffer sb) Implementa un paso de agregar y reemplazar terminal.
3	Cadena pública replaceAll (Reemplazo de cadena) Reemplaza cada subsecuencia de la secuencia de entrada que coincide con el patrón con la cadena de reemplazo dada.
4 4	Cadena pública replaceFirst (Reemplazo de cadena) Reemplaza la primera subsecuencia de la secuencia de entrada que coincide con el patrón con la cadena de reemplazo dada.
5 5	Presupuesto de cadena estática pública Reemplazo (cadena) Devuelve una Cadena de reemplazo literal para la Cadena especificada. Este método produce una cadena que funcionará como un reemplazo literal s en el método appendReplacement de la clase Matcher.

Los métodos de inicio y fin

El siguiente es el ejemplo que cuenta la cantidad de veces que aparece la palabra "cat" en la cadena de entrada:

Ejemplo

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegexMatches {

    private static final String REGEX = "\\bcat\\b";
    private static final String INPUT = "cat cat cat cattie
cat";

    public static void main( String args[] ) {
        Pattern p = Pattern.compile(REGEX);
        Matcher m = p.matcher(INPUT);    // get a matcher object
        int count = 0;

        while(m.find()) {
```

```

        count++;
        System.out.println("Match number "+count);
        System.out.println("start(): "+m.start());
        System.out.println("end(): "+m.end());
    }
}

```

Esto producirá el siguiente resultado:

Salida

```

Match number 1
start(): 0
end(): 3
Match number 2
start(): 4
end(): 7
Match number 3
start(): 8
end(): 11
Match number 4
start(): 19
end(): 22

```

Puede ver que este ejemplo utiliza límites de palabras para garantizar que las letras "c" "a" "t" no sean simplemente una subcadena en una palabra más larga. También proporciona información útil sobre dónde se produjo la coincidencia en la cadena de entrada.

El método de inicio devuelve el índice de inicio de la subsecuencia capturada por el grupo dado durante la operación de coincidencia anterior, y el final devuelve el índice del último carácter coincidente, más uno.

Los partidos y la mirada En los métodos

Los métodos de coincidencias y `lookingAt` intentan hacer coincidir una secuencia de entrada con un patrón. La diferencia, sin embargo, es que las coincidencias requieren que toda la secuencia de entrada coincida, mientras que `lookingAt` no.

Ambos métodos siempre comienzan al principio de la cadena de entrada. Aquí está el ejemplo que explica la funcionalidad:

Ejemplo

```

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegexMatches {

    private static final String REGEX = "foo";
    private static final String INPUT = "fooooooooooooooooo";
    private static Pattern pattern;

```

```

private static Matcher matcher;

public static void main( String args[] ) {
    pattern = Pattern.compile( REGEX );
    matcher = pattern.matcher( INPUT );

    System.out.println( "Current REGEX is: "+REGEX );
    System.out.println( "Current INPUT is: "+INPUT );

    System.out.println( "lookingAt():
"+matcher.lookingAt() );
    System.out.println( "matches(): "+matcher.matches() );
}
}

```

Esto producirá el siguiente resultado:

Salida

```

Current REGEX is: foo
Current INPUT is: fooooooooooooooooooooo
lookingAt(): true
matches(): false

```

Los métodos replaceFirst y replaceAll

Los métodos replaceFirst y replaceAll reemplazan el texto que coincide con una expresión regular dada. Como indican sus nombres, replaceFirst reemplaza la primera aparición, y replaceAll reemplaza todas las ocurrencias.

Aquí está el ejemplo que explica la funcionalidad:

Ejemplo

```

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegexMatches {

    private static String REGEX = "dog";
    private static String INPUT = "The dog says meow. " + "All
dogs say meow.";
    private static String REPLACE = "cat";

    public static void main(String[] args) {
        Pattern p = Pattern.compile( REGEX );

        // get a matcher object
        Matcher m = p.matcher( INPUT );
        INPUT = m.replaceAll( REPLACE );
        System.out.println( INPUT );
    }
}

```

Esto producirá el siguiente resultado:

Salida

The cat says meow. All cats say meow.

Los métodos appendReplacement y appendTail

La clase `Matcher` también proporciona métodos `appendReplacement` y `appendTail` para el reemplazo de texto.

Aquí está el ejemplo que explica la funcionalidad:

Ejemplo

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegexMatches {

    private static String REGEX = "a*b";
    private static String INPUT = "aabfooaabfooabfoob";
    private static String REPLACE = "-";
    public static void main(String[] args) {

        Pattern p = Pattern.compile(REGEX);

        // get a matcher object
        Matcher m = p.matcher(INPUT);
        StringBuffer sb = new StringBuffer();
        while(m.find()) {
            m.appendReplacement(sb, REPLACE);
        }
        m.appendTail(sb);
        System.out.println(sb.toString());
    }
}
```

Esto producirá el siguiente resultado:

Salida

-foo-foo-foo-

PatternSyntaxException Clase Métodos

`PatternSyntaxException` es una excepción no verificada que indica un error de sintaxis en un patrón de expresión regular. La clase `PatternSyntaxException` proporciona los siguientes métodos para ayudarlo a determinar qué salió mal:

No Señor.	Método y descripción
-----------	----------------------

1	public String getDescription () Recupera la descripción del error.
2	public int getIndex () Recupera el índice de error.
3	public String getPattern () Recupera el patrón de expresión regular erróneo.
4 4	public String getMessage () Devuelve una cadena de varias líneas que contiene la descripción del error de sintaxis y su índice, el patrón de expresión regular erróneo y una indicación visual del índice de error dentro del patrón.

Java - Métodos

Un método Java es una colección de declaraciones que se agrupan para realizar una operación. Cuando llamas a System.out. **El método println ()** , por ejemplo, el sistema realmente ejecuta varias declaraciones para mostrar un mensaje en la consola.

Ahora aprenderá cómo crear sus propios métodos con o sin valores de retorno, invocar un método con o sin parámetros y aplicar abstracción de métodos en el diseño del programa.

Método de creación

Considerando el siguiente ejemplo para explicar la sintaxis de un método:

Sintaxis

```
public static int methodName(int a, int b) {
    // body
}
```

Aquí,

- **public static** - modificador
- **int** - tipo de retorno
- **methodName** - nombre del método
- **a, b** - parámetros formales
- **int a, int b** - lista de parámetros

La definición del método consiste en un encabezado y un cuerpo del método. Lo mismo se muestra en la siguiente sintaxis:

Sintaxis


```
modifier returnType nameOfMethod (Parameter List) {  
    // method body  
}
```

La sintaxis que se muestra arriba incluye:

- **modificador** : define el tipo de acceso del método y es opcional de usar.
- **returnType** : el método puede devolver un valor.
- **nameOfMethod** : este es el nombre del método. La firma del método consiste en el nombre del método y la lista de parámetros.
- **Lista de parámetros** : la lista de parámetros, es el tipo, el orden y el número de parámetros de un método. Estos son opcionales, el método puede contener cero parámetros.
- **cuerpo del método: el cuerpo del** método define lo que hace el método con las declaraciones.

Ejemplo

Aquí está el código fuente del método definido anteriormente llamado **min()**. Este método toma dos parámetros num1 y num2 y devuelve el máximo entre los dos:

```
/** the snippet returns the minimum between two numbers */  
  
public static int minFunction(int n1, int n2) {  
    int min;  
    if (n1 > n2)  
        min = n2;  
    else  
        min = n1;  
  
    return min;  
}
```

Método de llamada

Para usar un método, debería llamarse. Hay dos formas en que se llama a un método, es decir, el método devuelve un valor o no devuelve nada (sin valor de retorno).

El proceso de llamada al método es simple. Cuando un programa invoca un método, el control del programa se transfiere al método llamado. Este método llamado luego devuelve el control a la persona que llama en dos condiciones, cuando:

- Se ejecuta la declaración de devolución.
- llega al método que termina cerrando la llave.

Los métodos que devuelven nulo se consideran como una llamada a una declaración. Consideremos un ejemplo:

```
System.out.println("This is postparaprogramadores.com!");
```

El método que devuelve el valor se puede entender con el siguiente ejemplo:

```
int result = sum(6, 9);
```

El siguiente es el ejemplo para demostrar cómo definir un método y cómo llamarlo:

Ejemplo

```
public class ExampleMinNumber {

    public static void main(String[] args) {
        int a = 11;
        int b = 6;
        int c = minFunction(a, b);
        System.out.println("Minimum Value = " + c);
    }

    /** returns the minimum of two numbers */
    public static int minFunction(int n1, int n2) {
        int min;
        if (n1 > n2)
            min = n2;
        else
            min = n1;

        return min;
    }
}
```

Esto producirá el siguiente resultado:

Salida

Minimum value = 6

La palabra clave vacía

La palabra clave nula nos permite crear métodos que no devuelven un valor. Aquí, en el siguiente ejemplo, estamos considerando un método de método *vacíoRankPoints*. Este método es un método nulo, que no devuelve ningún valor. La llamada a un método nulo debe ser una declaración, es decir, *methodRankPoints (255.7);*. Es una declaración de Java que termina con un punto y coma como se muestra en el siguiente ejemplo.

Ejemplo

```
public class ExampleVoid {

    public static void main(String[] args) {
        methodRankPoints(255.7);
    }
}
```

```
public static void methodRankPoints(double points) {
    if (points >= 202.5) {
        System.out.println("Rank:A1");
    } else if (points >= 122.4) {
        System.out.println("Rank:A2");
    } else {
        System.out.println("Rank:A3");
    }
}
```

Esto producirá el siguiente resultado:

Salida

Rank:A1

Parámetros de paso por valor

Mientras se trabaja en el proceso de llamada, se deben pasar los argumentos. Estos deben estar en el mismo orden que sus respectivos parámetros en la especificación del método. Los parámetros se pueden pasar por valor o por referencia.

Pasar parámetros por valor significa llamar a un método con un parámetro. A través de esto, el valor del argumento se pasa al parámetro.

Ejemplo

El siguiente programa muestra un ejemplo de paso de parámetro por valor. Los valores de los argumentos siguen siendo los mismos incluso después de la invocación del método.

```
public class swappingExample {

    public static void main(String[] args) {
        int a = 30;
        int b = 45;
        System.out.println("Before swapping, a = " + a + " and b = " + b);

        // Invoke the swap method
        swapFunction(a, b);
        System.out.println("\n**Now, Before and After swapping values will be same here**");
        System.out.println("After swapping, a = " + a + " and b is " + b);
    }

    public static void swapFunction(int a, int b) {
        System.out.println("Before swapping(Inside), a = " + a + " b = " + b);
    }
}
```

```

        // Swap n1 with n2
        int c = a;
        a = b;
        b = c;
        System.out.println("After swapping(Inside), a = " + a +
" b = " + b);
    }
}

```

Esto producirá el siguiente resultado:

Salida

```

Before swapping, a = 30 and b = 45
Before swapping(Inside), a = 30 b = 45
After swapping(Inside), a = 45 b = 30

```

****Now, Before and After swapping values will be same here**:**
After swapping, a = 30 and b is 45

Método de sobrecarga

Cuando una clase tiene dos o más métodos con el mismo nombre pero con parámetros diferentes, se conoce como sobrecarga de métodos. Es diferente de anular. Al anular, un método tiene el mismo nombre, tipo, número de parámetros, etc.

Consideremos el ejemplo discutido anteriormente para encontrar números mínimos de tipo entero. Si, digamos que queremos encontrar el número mínimo de tipos dobles. Luego, se introducirá el concepto de sobrecarga para crear dos o más métodos con el mismo nombre pero con parámetros diferentes.

El siguiente ejemplo explica lo mismo:

Ejemplo

```

public class ExampleOverloading {

    public static void main(String[] args) {
        int a = 11;
        int b = 6;
        double c = 7.3;
        double d = 9.4;
        int result1 = minFunction(a, b);

        // same function name with different parameters
        double result2 = minFunction(c, d);
        System.out.println("Minimum Value = " + result1);
        System.out.println("Minimum Value = " + result2);
    }

    // for integer

```

```

public static int minFunction(int n1, int n2) {
    int min;
    if (n1 > n2)
        min = n2;
    else
        min = n1;

    return min;
}

// for double
public static double minFunction(double n1, double n2) {
    double min;
    if (n1 > n2)
        min = n2;
    else
        min = n1;

    return min;
}
}

```

Esto producirá el siguiente resultado:

Salida

```

Minimum Value = 6
Minimum Value = 7.3

```

Los métodos de sobrecarga hacen que el programa sea legible. Aquí, dos métodos se dan con el mismo nombre pero con diferentes parámetros. El número mínimo de tipos enteros y dobles es el resultado.

Uso de argumentos de línea de comandos

Algunas veces querrás pasar cierta información a un programa cuando lo ejecutes. Esto se logra pasando los argumentos de la línea de comandos a main ().

Un argumento de línea de comando es la información que sigue directamente el nombre del programa en la línea de comando cuando se ejecuta. Acceder a los argumentos de la línea de comandos dentro de un programa Java es bastante fácil. Se almacenan como cadenas en el conjunto de cadenas pasado a main ().

Ejemplo

El siguiente programa muestra todos los argumentos de la línea de comandos con los que se llama:

```

public class CommandLine {

    public static void main(String args[]) {
        for(int i = 0; i<args.length; i++) {
            System.out.println("args[" + i + "]: " + args[i]);
        }
    }
}

```

```
}  
}  
}
```

Intente ejecutar este programa como se muestra aquí:

```
$java CommandLine this is a command line 200 -100
```

Esto producirá el siguiente resultado:

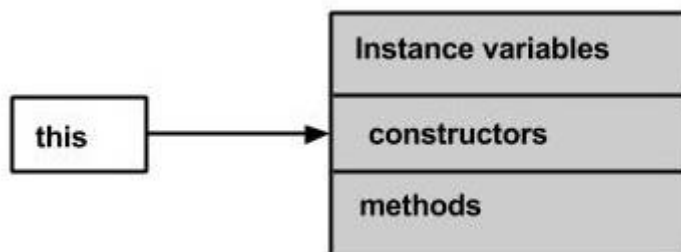
Salida

```
args[0]: this  
args[1]: is  
args[2]: a  
args[3]: command  
args[4]: line  
args[5]: 200  
args[6]: -100
```

La esta palabra clave

Esta es una palabra clave en Java que se utiliza como referencia para el objeto de la clase actual, en un método de instancia o un constructor. Con *esto*, puede hacer referencia a los miembros de una clase, como constructores, variables y métodos.

Nota: la palabra clave *this* se usa solo dentro de métodos de instancia o constructores



En general, la palabra clave *de este* se utiliza para -

- Diferenciar las variables de instancia de las variables locales si tienen los mismos nombres, dentro de un constructor o un método.

```
class Student {  
    int age;  
    Student(int age) {  
        this.age = age;  
    }  
}
```

- Llame a un tipo de constructor (constructor parametrizado o predeterminado) de otro en una clase. Se conoce como invocación explícita de constructor.

```
class Student {  
    int age
```

```
Student() {  
    this(20);  
}  
  
Student(int age) {  
    this.age = age;  
}  
}
```

Ejemplo

Aquí hay un ejemplo que usa *esta* palabra clave para acceder a los miembros de una clase. Copie y pegue el siguiente programa en un archivo con el nombre **This_Example.java** .

```
public class This_Example {  
    // Instance variable num  
    int num = 10;  
  
    This_Example() {  
        System.out.println("This is an example program on  
keyword this");  
    }  
  
    This_Example(int num) {  
        // Invoking the default constructor  
        this();  
  
        // Assigning the local variable num to the instance  
variable num  
        this.num = num;  
    }  
  
    public void greet() {  
        System.out.println("Hi Welcome to  
Postparaprogramadores");  
    }  
  
    public void print() {  
        // Local variable num  
        int num = 20;  
  
        // Printing the local variable  
        System.out.println("value of local variable num is :  
"+num);  
  
        // Printing the instance variable  
        System.out.println("value of instance variable num is :  
"+this.num);  
  
        // Invoking the greet method of a class  
        this.greet();  
    }  
}
```

```

public static void main(String[] args) {
    // Instantiating the class
    This_Example obj1 = new This_Example();

    // Invoking the print method
    obj1.print();

    // Passing a new value to the num variable through
    Parameterized constructor
    This_Example obj2 = new This_Example(30);

    // Invoking the print method again
    obj2.print();
}
}

```

Esto producirá el siguiente resultado:

Salida

```

This is an example program on keyword this
value of local variable num is : 20
value of instance variable num is : 10
Hi Welcome to Postparaprogramadores
This is an example program on keyword this
value of local variable num is : 20
value of instance variable num is : 30
Hi Welcome to Postparaprogramadores

```

Argumentos variables (var-args)

JDK 1.5 le permite pasar un número variable de argumentos del mismo tipo a un método. El parámetro en el método se declara de la siguiente manera:

```
typeName... parameterName
```

En la declaración del método, especifica el tipo seguido de puntos suspensivos (...). Solo se puede especificar un parámetro de longitud variable en un método, y este parámetro debe ser el último parámetro. Cualquier parámetro regular debe precederlo.

Ejemplo

```

public class VarargsDemo {

    public static void main(String args[]) {
        // Call method with variable args
        printMax(34, 3, 3, 2, 56.5);
        printMax(new double[]{1, 2, 3});
    }

    public static void printMax( double... numbers) {
        if (numbers.length == 0) {

```



```

        System.out.println("No argument passed");
        return;
    }

    double result = numbers[0];

    for (int i = 1; i < numbers.length; i++)
        if (numbers[i] > result)
            result = numbers[i];
    System.out.println("The max value is " + result);
}
}

```

Esto producirá el siguiente resultado:

Salida

```

The max value is 56.5
The max value is 3.0

```

El método finalize ()

Es posible definir un método que será llamado justo antes de la destrucción final de un objeto por el recolector de basura. Este método se llama **finalize ()**, y se puede usar para asegurar que un objeto termine limpiamente.

Por ejemplo, puede usar `finalize ()` para asegurarse de que un archivo abierto propiedad de ese objeto esté cerrado.

Para agregar un finalizador a una clase, simplemente defina el método `finalize ()`. El tiempo de ejecución de Java llama a ese método cada vez que está a punto de reciclar un objeto de esa clase.

Dentro del método `finalize ()`, especificará aquellas acciones que deben realizarse antes de que un objeto sea destruido.

El método `finalize ()` tiene esta forma general:

```

protected void finalize( ) {
    // finalization code here
}

```

Aquí, la palabra clave protegida es un especificador que impide el acceso a `finalize ()` por código definido fuera de su clase.

Esto significa que no puede saber cuándo o incluso si se ejecutará `finalize ()`. Por ejemplo, si su programa finaliza antes de que se produzca la recolección de basura, `finalize ()` no se ejecutará.

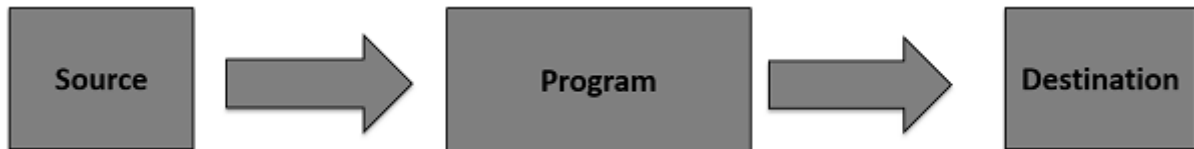
Java - Archivos y E / S

El paquete `java.io` contiene casi todas las clases que pueda necesitar para realizar entradas y salidas (E / S) en Java. Todos estos flujos representan una fuente de entrada y un destino de salida. La secuencia en el paquete `java.io` admite muchos datos, como primitivas, objetos, caracteres localizados, etc.

Corriente

Una secuencia se puede definir como una secuencia de datos. Hay dos tipos de transmisiones:

- **InPutStream** : InputStream se utiliza para leer datos de una fuente.
- **OutPutStream** : OutputStream se utiliza para escribir datos en un destino.



Java proporciona un soporte sólido pero flexible para E / S relacionadas con archivos y redes, pero este tutorial cubre funcionalidades muy básicas relacionadas con transmisiones y E / S. Veremos los ejemplos más utilizados uno por uno:

Byte Streams

Las secuencias de bytes de Java se utilizan para realizar entradas y salidas de bytes de 8 bits. Aunque hay muchas clases relacionadas con las secuencias de bytes, las clases más utilizadas son **FileInputStream** y **FileOutputStream**. El siguiente es un ejemplo que utiliza estas dos clases para copiar un archivo de entrada en un archivo de salida:

Ejemplo

```
import java.io.*;
public class CopyFile {

    public static void main(String args[]) throws IOException
    {
        FileInputStream in = null;
        FileOutputStream out = null;

        try {
            in = new FileInputStream("input.txt");
            out = new FileOutputStream("output.txt");

            int c;
            while ((c = in.read()) != -1) {
                out.write(c);
            }
        } finally {
            if (in != null) {
                in.close();
            }
            if (out != null) {
                out.close();
            }
        }
    }
}
```

```
}  
}
```

Ahora tengamos un archivo **input.txt** con el siguiente contenido:

This is test for copy file.

Como siguiente paso, compile el programa anterior y ejecútelo, lo que dará como resultado la creación del archivo output.txt con el mismo contenido que tenemos en input.txt. Pongamos el código anterior en el archivo CopyFile.java y hagamos lo siguiente:

```
$javac CopyFile.java  
$java CopyFile
```

Corrientes de personajes

Las secuencias de **bytes de** Java se utilizan para realizar entradas y salidas de bytes de 8 bits, mientras que las secuencias de **caracteres de** Java se utilizan para realizar entradas y salidas para unicode de 16 bits. Aunque hay muchas clases relacionadas con las secuencias de caracteres, las clases más utilizadas son **FileReader** y **FileWriter**. Aunque internamente FileReader usa FileInputStream y FileWriter usa FileOutputStream, pero aquí la diferencia principal es que FileReader lee dos bytes a la vez y FileWriter escribe dos bytes a la vez.

Podemos reescribir el ejemplo anterior, que utiliza estas dos clases para copiar un archivo de entrada (que tiene caracteres unicode) en un archivo de salida:

Ejemplo

```
import java.io.*;  
public class CopyFile {  
  
    public static void main(String args[]) throws IOException  
    {  
        FileReader in = null;  
        FileWriter out = null;  
  
        try {  
            in = new FileReader("input.txt");  
            out = new FileWriter("output.txt");  
  
            int c;  
            while ((c = in.read()) != -1) {  
                out.write(c);  
            }  
        } finally {  
            if (in != null) {  
                in.close();  
            }  
            if (out != null) {  
                out.close();  
            }  
        }  
    }  
}
```

```
}  
}  
}
```

Ahora tengamos un archivo **input.txt** con el siguiente contenido:

```
This is test for copy file.
```

Como siguiente paso, compile el programa anterior y ejecútelo, lo que dará como resultado la creación del archivo **output.txt** con el mismo contenido que tenemos en **input.txt**. Pongamos el código anterior en el archivo **CopyFile.java** y hagamos lo siguiente:

```
$javac CopyFile.java  
$java CopyFile
```

Streams estándar

Todos los lenguajes de programación brindan soporte para E / S estándar donde el programa del usuario puede recibir información de un teclado y luego producir una salida en la pantalla de la computadora. Si conoce los lenguajes de programación C o C ++, debe conocer los tres dispositivos estándar STDIN, STDOUT y STDERR. Del mismo modo, Java proporciona las siguientes tres secuencias estándar:

- **Entrada estándar** : se usa para alimentar los datos al programa del usuario y generalmente se usa un teclado como flujo de entrada estándar y se representa como **System.in** .
- **Salida estándar** : se utiliza para generar los datos producidos por el programa del usuario y, por lo general, se utiliza una pantalla de computadora para el flujo de salida estándar y se representa como **System.out** .
- **Error estándar** : se utiliza para generar los datos de error producidos por el programa del usuario y, por lo general, se utiliza una pantalla de computadora para el flujo de error estándar y se representa como **System.err** .

El siguiente es un programa simple, que crea **InputStreamReader** para leer el flujo de entrada estándar hasta que el usuario escriba una "q":

Ejemplo

```
import java.io.*;  
public class ReadConsole {  
  
    public static void main(String args[]) throws IOException  
    {  
        InputStreamReader cin = null;  
  
        try {  
            cin = new InputStreamReader(System.in);  
            System.out.println("Enter characters, 'q' to  
quit.");  
            char c;  
            do {
```

```
        c = (char) cin.read();
        System.out.print(c);
    } while (c != 'q');
} finally {
    if (cin != null) {
        cin.close();
    }
}
}
```

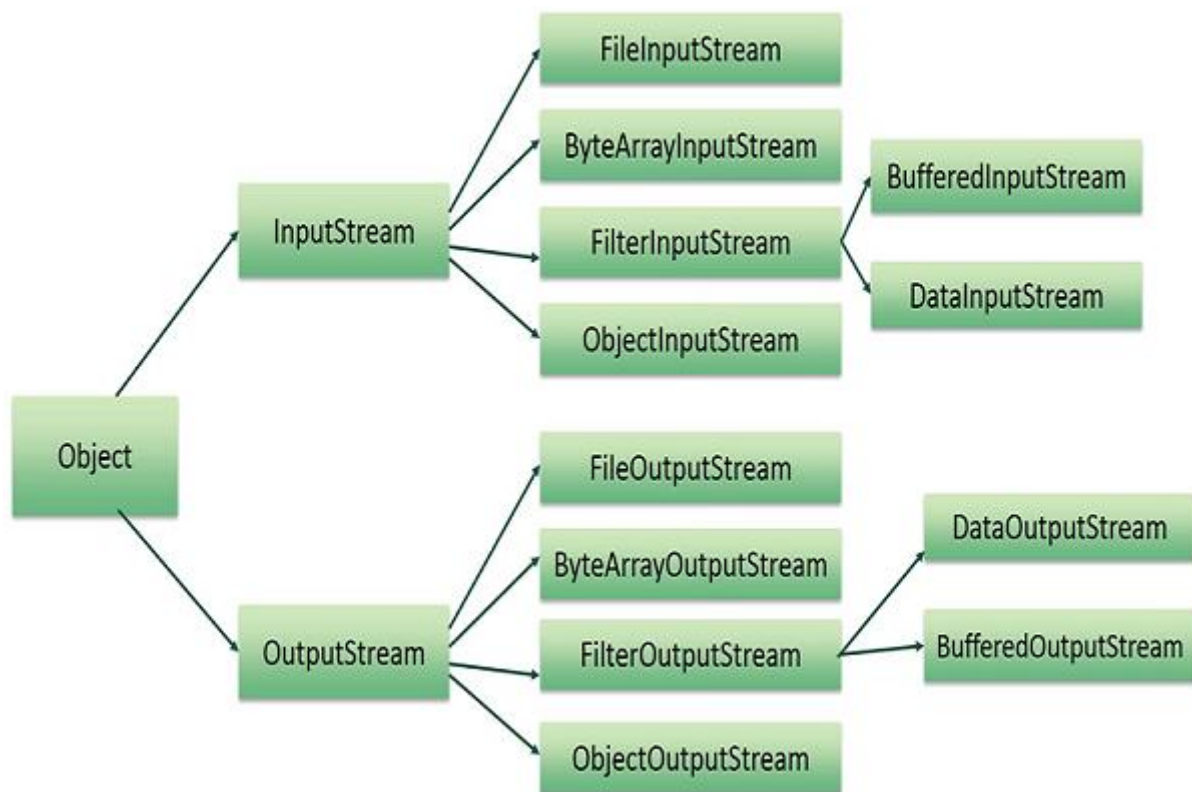
Conservemos el código anterior en el archivo ReadConsole.java e intentemos compilarlo y ejecutarlo como se muestra en el siguiente programa. Este programa continúa leyendo y mostrando el mismo carácter hasta que presionamos 'q' -

```
$javac ReadConsole.java
$java ReadConsole
Enter characters, 'q' to quit.
1
1
e
e
q
q
```

Leer y escribir archivos

Como se describió anteriormente, una secuencia se puede definir como una secuencia de datos. El **InputStream** se utiliza para leer los datos de una fuente y el **OutputStream** se utiliza para escribir datos en un destino.

Aquí hay una jerarquía de clases para tratar con flujos de entrada y salida.



Las dos transmisiones importantes son **FileInputStream** y **FileOutputStream** , que se analizarán en este tutorial.

FileInputStream

Esta secuencia se utiliza para leer datos de los archivos. Los objetos se pueden crear usando la palabra clave **new** y hay varios tipos de constructores disponibles.

El siguiente constructor toma un nombre de archivo como una cadena para crear un objeto de flujo de entrada para leer el archivo:

```
InputStream f = new FileInputStream("C:/java/hello");
```

El siguiente constructor toma un objeto de archivo para crear un objeto de flujo de entrada para leer el archivo. Primero creamos un objeto de archivo usando el método `File ()` de la siguiente manera:

```
File f = new File("C:/java/hello");
InputStream f = new FileInputStream(f);
```

Una vez que tenga el objeto *InputStream* en la mano, hay una lista de métodos auxiliares que se pueden usar para leer para transmitir o para realizar otras operaciones en la transmisión.

No Señor.	Método y descripción
-----------	----------------------

1	public void close () arroja IOException {} Este método cierra la secuencia de salida del archivo. Libera cualquier recurso del sistema asociado con el archivo. Lanza una IOException.
2	El vacío protegido finalize () arroja IOException {} Este método limpia la conexión al archivo. Asegura que se llama al método de cierre de esta secuencia de salida de archivo cuando no hay más referencias a esta secuencia. Lanza una IOException.
3	public int read (int r) arroja IOException {} Este método lee el byte de datos especificado de InputStream. Devuelve un int. Devuelve el siguiente byte de datos y -1 se devolverá si es el final del archivo.
4 4	public int read (byte [] r) arroja IOException {} Este método lee r.length bytes de la secuencia de entrada en una matriz. Devuelve el número total de bytes leídos. Si es el final del archivo, se devolverá -1.
5 5	public int available () arroja IOException {} Da el número de bytes que se pueden leer desde esta secuencia de entrada de archivo. Devuelve un int.

Hay otros flujos de entrada importantes disponibles, para más detalles puede consultar los siguientes enlaces:

- [ByteArrayInputStream](#)
- [DataInputStream](#)

FileOutputStream

FileOutputStream se utiliza para crear un archivo y escribir datos en él. La secuencia crearía un archivo, si aún no existe, antes de abrirlo para la salida.

Aquí hay dos constructores que pueden usarse para crear un objeto FileOutputStream.

El siguiente constructor toma un nombre de archivo como una cadena para crear un objeto de flujo de entrada para escribir el archivo:

```
OutputStream f = new FileOutputStream("C:/java/hello")
```

El siguiente constructor toma un objeto de archivo para crear un objeto de flujo de salida para escribir el archivo. Primero, creamos un objeto de archivo usando el método File () de la siguiente manera:

```
File f = new File("C:/java/hello");
OutputStream f = new FileOutputStream(f);
```

Una vez que tenga el objeto *OutputStream* en la mano, hay una lista de métodos auxiliares, que se pueden usar para escribir en la transmisión o para realizar otras operaciones en la transmisión.

No Señor.	Método y descripción
1	public void close () arroja IOException {} Este método cierra la secuencia de salida del archivo. Libera cualquier recurso del sistema asociado con el archivo. Lanza una IOException.
2	El vacío protegido finalize () arroja IOException {} Este método limpia la conexión al archivo. Asegura que se llama al método de cierre de esta secuencia de salida de archivo cuando no hay más referencias a esta secuencia. Lanza una IOException.
3	public void write (int w) lanza IOException {} Este método escribe el byte especificado en la secuencia de salida.
4 4	escritura pública vacía (byte [] w) Escribe w.length bytes de la matriz de bytes mencionada en OutputStream.

Hay otros flujos de salida importantes disponibles, para más detalles puede consultar los siguientes enlaces:

- [ByteArrayOutputStream](#)
- [DataOutputStream](#)

Ejemplo

El siguiente es el ejemplo para demostrar InputStream y OutputStream:

```
import java.io.*;
public class FileStreamTest {

    public static void main(String args[]) {

        try {
            byte bWrite [] = {11,21,3,40,5};
            OutputStream os = new FileOutputStream("test.txt");
            for(int x = 0; x < bWrite.length ; x++) {
                os.write( bWrite[x] );    // writes the bytes
            }
            os.close();

            InputStream is = new FileInputStream("test.txt");
            int size = is.available();
```



```

        for(int i = 0; i < size; i++) {
            System.out.print((char)is.read() + " ");
        }
        is.close();
    } catch (IOException e) {
        System.out.print("Exception");
    }
}
}

```

El código anterior crearía el archivo test.txt y escribiría números dados en formato binario. Lo mismo sería la salida en la pantalla estándar.

Navegación de archivos y E / S

Hay varias otras clases por las que estaríamos pasando para conocer los conceptos básicos de File Navigation y I / O.

- [Clase de archivo](#)
- [Clase FileReader](#)
- [Clase FileWriter](#)

Directorios en Java

Un directorio es un archivo que puede contener una lista de otros archivos y directorios. Utiliza el objeto **Archivo** para crear directorios, para enumerar los archivos disponibles en un directorio. Para obtener detalles completos, consulte una lista de todos los métodos a los que puede llamar en el objeto Archivo y lo que está relacionado con los directorios.

Crear directorios

Hay dos métodos útiles de utilidad de **archivos** , que se pueden utilizar para crear directorios:

- El método **mkdir ()** crea un directorio, devolviendo verdadero en caso de éxito y falso en caso de error. La falla indica que la ruta especificada en el objeto File ya existe o que el directorio no se puede crear porque toda la ruta aún no existe.
- El método **mkdirs ()** crea un directorio y todos los padres del directorio.

El siguiente ejemplo crea el directorio "/ tmp / user / java / bin" -

Ejemplo

```

import java.io.File;
public class CreateDir {

    public static void main(String args[]) {
        String dirname = "/tmp/user/java/bin";
        File d = new File(dirname);

        // Create directory now.
    }
}

```

```
        d.mkdirs();  
    }  
}
```

Compile y ejecute el código anterior para crear "/ tmp / user / java / bin".

Nota : Java se ocupa automáticamente de los separadores de ruta en UNIX y Windows según las convenciones. Si usa una barra diagonal (/) en una versión de Windows de Java, la ruta aún se resolverá correctamente.

Listado de directorios

Puede usar el método **list ()** proporcionado por el objeto **File** para enumerar todos los archivos y directorios disponibles en un directorio de la siguiente manera:

Ejemplo

```
import java.io.File;  
public class ReadDir {  
  
    public static void main(String[] args) {  
        File file = null;  
        String[] paths;  
  
        try {  
            // create new file object  
            file = new File("/tmp");  
  
            // array of files and directory  
            paths = file.list();  
  
            // for each name in the path array  
            for(String path:paths) {  
                // prints filename and directory name  
                System.out.println(path);  
            }  
        } catch (Exception e) {  
            // if any error occurs  
            e.printStackTrace();  
        }  
    }  
}
```

Esto producirá el siguiente resultado basado en los directorios y archivos disponibles en su directorio **/ tmp** :

Salida

```
test1.txt  
test2.txt  
ReadDir.java  
ReadDir.class
```

Java - Excepciones

Una excepción (o evento excepcional) es un problema que surge durante la ejecución de un programa. Cuando ocurre una **excepción**, el flujo normal del programa se interrumpe y el programa / aplicación finaliza de manera anormal, lo que no se recomienda, por lo tanto, estas excepciones deben ser manejadas.

Una excepción puede ocurrir por muchas razones diferentes. Los siguientes son algunos escenarios donde ocurre una excepción.

- Un usuario ha ingresado datos no válidos.
- No se puede encontrar un archivo que deba abrirse.
- Se perdió una conexión de red en medio de las comunicaciones o la JVM se quedó sin memoria.

Algunas de estas excepciones son causadas por un error del usuario, otras por un error del programador y otras por recursos físicos que han fallado de alguna manera.

En base a esto, tenemos tres categorías de excepciones. Debe comprenderlos para saber cómo funciona el manejo de excepciones en Java.

- **Excepciones comprobadas** : una excepción comprobada es una excepción que el compilador verifica (notifica) en el momento de la compilación, también se denominan excepciones de tiempo de compilación. Estas excepciones no pueden simplemente ignorarse, el programador debe encargarse de (manejar) estas excepciones.

Por ejemplo, si usa la clase **FileReader** en su programa para leer datos de un archivo, si el archivo especificado en su constructor no existe, se produce una *excepción* `FileNotFoundException` y el compilador solicita al programador que maneje la excepción.

Ejemplo

```
import java.io.File;
import java.io.FileReader;

public class FileNotFoundException_Demo {

    public static void main(String args[]) {
        File file = new File("E://file.txt");
        FileReader fr = new FileReader(file);
    }
}
```

Si intenta compilar el programa anterior, obtendrá las siguientes excepciones.

Salida

```
C:\>javac FileNotFoundException_Demo.java
FileNotFoundException_Demo.java:8: error: unreported exception
FileNotFoundException; must be caught or declared to be
thrown
```

```
FileReader fr = new FileReader(file);  
                ^
```

1 error

Nota : Dado que los métodos **read ()** y **close ()** de la clase `FileReader` arrojan `IOException`, puede observar que el compilador notifica que maneja `IOException`, junto con `FileNotFoundException`.

- **Excepciones no verificadas:** una excepción no verificada es una excepción que ocurre en el momento de la ejecución. También se denominan **excepciones de tiempo de ejecución**. Estos incluyen errores de programación, como errores lógicos o uso incorrecto de una API. Las excepciones de tiempo de ejecución se ignoran en el momento de la compilación.

Por ejemplo, si se ha declarado una matriz de tamaño 5 en su programa, y tratando de llamar a la 6ª elemento de la matriz a continuación, un `ArrayIndexOutOfBoundsException` se produce.

Ejemplo

```
public class Unchecked_Demo {  
  
    public static void main(String args[]) {  
        int num[] = {1, 2, 3, 4};  
        System.out.println(num[5]);  
    }  
}
```

Si compila y ejecuta el programa anterior, obtendrá la siguiente excepción.

Salida

```
Exception in thread "main"  
java.lang.ArrayIndexOutOfBoundsException: 5  
    at  
Exceptions.Unchecked_Demo.main(Unchecked_Demo.java:8)
```

- **Errores :** no se trata de excepciones, sino de problemas que surgen fuera del control del usuario o del programador. Los errores generalmente se ignoran en su código porque rara vez puede hacer algo al respecto. Por ejemplo, si se produce un desbordamiento de la pila, se producirá un error. También se ignoran en el momento de la compilación.

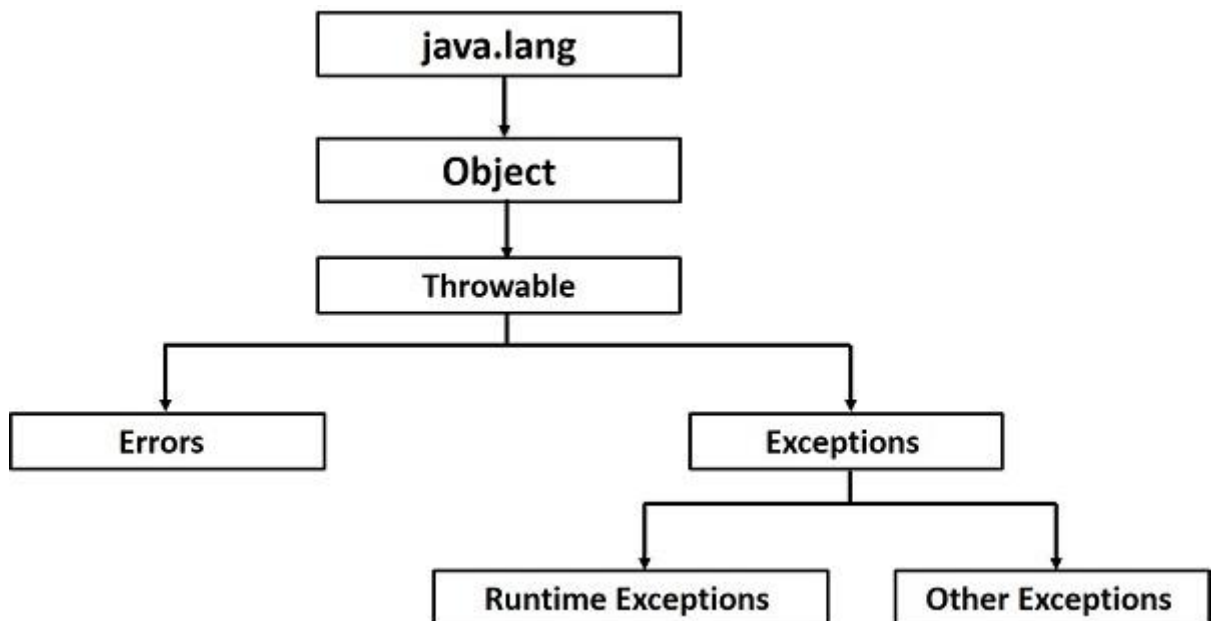
Jerarquía de excepciones

Todas las clases de excepción son subtipos de la clase `java.lang.Exception`. La clase de excepción es una subclase de la clase `Throwable`. Además de la clase de excepción, hay otra subclase llamada `Error` que se deriva de la clase `Throwable`.

Los errores son condiciones anormales que ocurren en caso de fallas graves, estos no son manejados por los programas Java. Los errores se generan para indicar errores generados por el entorno de tiempo de ejecución. Ejemplo:

JVM no tiene memoria. Normalmente, los programas no pueden recuperarse de los errores.

La clase Exception tiene dos subclases principales: la clase IOException y la clase RuntimeException.



A continuación se incluye una lista de las excepciones incorporadas de Java más comunes marcadas y sin marcar .

Métodos de excepciones

La siguiente es la lista de métodos importantes disponibles en la clase Throwable.

No Señor.	Método y descripción
1	public String getMessage () Devuelve un mensaje detallado sobre la excepción que ha ocurrido. Este mensaje se inicializa en el constructor Throwable.
2	public Throwable getCause () Devuelve la causa de la excepción representada por un objeto Throwable.
3	public String toString () Devuelve el nombre de la clase concatenada con el resultado de getMessage ().
4 4	public void printStackTrace ()

	Imprime el resultado de toString () junto con el seguimiento de la pila en System.err, la secuencia de salida de error.
5 5	public StackTraceElement [] getStackTrace () Devuelve una matriz que contiene cada elemento en el seguimiento de la pila. El elemento en el índice 0 representa la parte superior de la pila de llamadas, y el último elemento en la matriz representa el método en la parte inferior de la pila de llamadas.
6 6	public Throwable fillInStackTrace () Llena el seguimiento de la pila de este objeto Throwable con el seguimiento de la pila actual, agregando a cualquier información previa en el seguimiento de la pila.

Capturando excepciones

Un método detecta una excepción utilizando una combinación de las palabras clave **try** y **catch** . Se coloca un bloque try / catch alrededor del código que podría generar una excepción. El código dentro de un bloque try / catch se conoce como código protegido, y la sintaxis para usar try / catch es similar a la siguiente:

Sintaxis

```
try {
    // Protected code
} catch (ExceptionName e1) {
    // Catch block
}
```

El código que es propenso a excepciones se coloca en el bloque try. Cuando se produce una excepción, esa excepción se maneja mediante el bloque catch asociado a ella. Cada bloque de prueba debe ser seguido inmediatamente por un bloque de captura o finalmente un bloque.

Una declaración catch implica declarar el tipo de excepción que está intentando detectar. Si se produce una excepción en el código protegido, se comprueba el bloque (o bloques) de captura que sigue al intento. Si el tipo de excepción que se produjo aparece en un bloque catch, la excepción se pasa al bloque catch de la misma forma que se pasa un argumento a un parámetro de método.

Ejemplo

La siguiente es una matriz declarada con 2 elementos. A continuación, el código intenta acceder a la 3^{ra} elemento de la matriz, que se produce una excepción.

```
// File Name : ExceptTest.java
import java.io.*;

public class ExceptTest {

    public static void main(String args[]) {
        try {
            int a[] = new int[2];
            System.out.println("Access element three :" + a[3]);
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Exception thrown :" + e);
        }
        System.out.println("Out of the block");
    }
}
```

Esto producirá el siguiente resultado:

Salida

```
Exception thrown :java.lang.ArrayIndexOutOfBoundsException:
3
Out of the block
```

Múltiples bloques de captura

Un bloque de prueba puede ser seguido por múltiples bloques de captura. La sintaxis para múltiples bloques catch es similar a la siguiente:

Sintaxis

```
try {
    // Protected code
} catch (ExceptionType1 e1) {
    // Catch block
} catch (ExceptionType2 e2) {
    // Catch block
} catch (ExceptionType3 e3) {
    // Catch block
}
```

Las declaraciones anteriores demuestran tres bloques catch, pero puede tener cualquier número de ellos después de un solo intento. Si se produce una excepción en el código protegido, la excepción se lanza al primer bloque catch de la lista. Si el tipo de datos de la excepción lanzada coincide con `ExceptionType1`, queda atrapado allí. Si no, la excepción pasa a la segunda instrucción catch. Esto continúa hasta que la excepción se detecta o cae en todas las capturas, en cuyo caso el método actual detiene la ejecución y la excepción se arroja al método anterior en la pila de llamadas.

Ejemplo

Aquí hay un segmento de código que muestra cómo usar múltiples declaraciones try / catch.

```
try {
    file = new FileInputStream(fileName);
    x = (byte) file.read();
} catch (IOException i) {
    i.printStackTrace();
    return -1;
} catch (FileNotFoundException f) // Not valid! {
    f.printStackTrace();
    return -1;
}
```

Capturar múltiples tipos de excepciones

Desde Java 7, puede manejar más de una excepción usando un solo bloque catch, esta característica simplifica el código. Así es como lo harías:

```
catch (IOException|FileNotFoundException ex) {
    logger.log(ex);
    throw ex;
}
```

El lanzamiento / Palabras clave del lanzamiento

Si un método no maneja una excepción marcada, el método debe declararlo usando la palabra clave **throws** . La palabra clave throws aparece al final de la firma de un método.

Puede lanzar una excepción, ya sea una instancia nueva o una excepción que acaba de detectar, utilizando la palabra clave **throw** .

Intente comprender la diferencia entre throws y throw keywords, *throws* se usa para posponer el manejo de una excepción marcada y *throw* se usa para invocar una excepción explícitamente.

El siguiente método declara que arroja una RemoteException:

Ejemplo

```
import java.io.*;
public class className {

    public void deposit(double amount) throws RemoteException
    {
        // Method implementation
        throw new RemoteException();
    }
    // Remainder of class definition
}
```

Un método puede declarar que arroja más de una excepción, en cuyo caso las excepciones se declaran en una lista separada por comas. Por ejemplo, el

siguiente método declara que arroja una `RemoteException` y una `InsufficientFundsException`:

Ejemplo

```
import java.io.*;
public class className {

    public void withdraw(double amount) throws
RemoteException,
    InsufficientFundsException {
        // Method implementation
    }
    // Remainder of class definition
}
```

El finalmente bloque

El bloque finalmente sigue a un bloque de prueba o un bloque de captura. Un bloque de código finalmente siempre se ejecuta, independientemente de la aparición de una excepción.

El uso de un último bloque le permite ejecutar cualquier tipo de instrucción de limpieza que desee ejecutar, sin importar lo que suceda en el código protegido.

Finalmente aparece un bloque al final de los bloques catch y tiene la siguiente sintaxis:

Sintaxis

```
try {
    // Protected code
} catch (ExceptionType1 e1) {
    // Catch block
} catch (ExceptionType2 e2) {
    // Catch block
} catch (ExceptionType3 e3) {
    // Catch block
}finally {
    // The finally block always executes.
}
```

Ejemplo

```
public class Exceptest {

    public static void main(String args[]) {
        int a[] = new int[2];
        try {
```

```

        System.out.println("Access element three :" + a[3]);
    } catch (ArrayIndexOutOfBoundsException e) {
        System.out.println("Exception thrown  :" + e);
    } finally {
        a[0] = 6;
        System.out.println("First element value: " + a[0]);
        System.out.println("The finally statement is
executed");
    }
}
}

```

Esto producirá el siguiente resultado:

Salida

```

Exception thrown  :java.lang.ArrayIndexOutOfBoundsException:
3
First element value: 6
The finally statement is executed

```

Tenga en cuenta lo siguiente:

- Una cláusula catch no puede existir sin una declaración try.
- No es obligatorio tener cláusulas finales siempre que haya un bloque try / catch.
- El bloque try no puede estar presente sin la cláusula catch o la cláusula finalmente.
- Cualquier código no puede estar presente entre los bloques try, catch y finalmente.

La prueba con recursos

En general, cuando usamos recursos como flujos, conexiones, etc., tenemos que cerrarlos explícitamente usando finalmente block. En el siguiente programa, estamos leyendo datos de un archivo usando **FileReader** y los estamos cerrando usando finalmente block.

Ejemplo

```

import java.io.File;
import java.io.FileReader;
import java.io.IOException;

public class ReadData_Demo {

    public static void main(String args[]) {
        FileReader fr = null;
        try {
            File file = new File("file.txt");
            fr = new FileReader(file); char [] a = new char[50];
            fr.read(a);    // reads the content to the array
            for(char c : a)

```

```

        System.out.print(c);    // prints the characters one
by one
    } catch (IOException e) {
        e.printStackTrace();
    } finally {
        try {
            fr.close();
        } catch (IOException ex) {
            ex.printStackTrace();
        }
    }
}
}
}

```

try-with-resources, también conocido como **gestión automática de recursos**, es un nuevo mecanismo de manejo de excepciones que se introdujo en Java 7, que cierra automáticamente los recursos utilizados dentro del bloque try catch.

Para usar esta declaración, simplemente necesita declarar los recursos requeridos dentro del paréntesis, y el recurso creado se cerrará automáticamente al final del bloque. La siguiente es la sintaxis de la declaración try-with-resources.

Sintaxis

```

try(FileReader fr = new FileReader("file path")) {
    // use the resource
} catch () {
    // body of catch
}
}

```

El siguiente es el programa que lee los datos en un archivo usando la declaración try-with-resources.

Ejemplo

```

import java.io.FileReader;
import java.io.IOException;

public class Try_withDemo {

    public static void main(String args[]) {
        try(FileReader fr = new FileReader("E://file.txt")) {
            char [] a = new char[50];
            fr.read(a);    // reads the content to the array
            for(char c : a)
                System.out.print(c);    // prints the characters one
by one
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}

```

```
}  
}
```

Los siguientes puntos deben tenerse en cuenta al trabajar con la declaración de prueba con recursos.

- Para usar una clase con la declaración try-with-resources, debe implementar la interfaz **AutoCloseable** y el método **close ()** se invoca automáticamente en tiempo de ejecución.
- Puede declarar más de una clase en la declaración de prueba con recursos.
- Mientras declara varias clases en el bloque try de la declaración try-with-resources, estas clases se cierran en orden inverso.
- Excepto la declaración de recursos dentro del paréntesis, todo es lo mismo que el bloque try / catch normal de un bloque try.
- El recurso declarado en try se instancia justo antes del inicio del try-block.
- El recurso declarado en el bloque try se declara implícitamente como final.

Excepciones definidas por el usuario

Puede crear sus propias excepciones en Java. Tenga en cuenta los siguientes puntos al escribir sus propias clases de excepción:

- Todas las excepciones deben ser hijo de Throwable.
- Si desea escribir una excepción marcada que se aplica automáticamente mediante la Regla de manejo o declaración, debe extender la clase Excepción.
- Si desea escribir una excepción de tiempo de ejecución, debe extender la clase RuntimeException.

Podemos definir nuestra propia clase de Excepción de la siguiente manera:

```
class MyException extends Exception {  
}
```

Solo necesita extender la clase de **excepción** predefinida para crear su propia excepción. Estas se consideran excepciones comprobadas. La siguiente clase **InsufficientFundsException** es una excepción definida por el usuario que extiende la clase Exception, convirtiéndola en una excepción comprobada. Una clase de excepción es como cualquier otra clase, que contiene campos y métodos útiles.

Ejemplo

```
// File Name InsufficientFundsException.java  
import java.io.*;  
  
public class InsufficientFundsException extends Exception {  
    private double amount;  
  
    public InsufficientFundsException(double amount) {  
        this.amount = amount;  
    }  
}
```

```
    public double getAmount() {  
        return amount;  
    }  
}
```

Para demostrar el uso de nuestra excepción definida por el usuario, la siguiente clase `CheckingAccount` contiene un método retiro () que arroja una excepción `InsufficientFundsException`.

```
// File Name CheckingAccount.java  
import java.io.*;  
  
public class CheckingAccount {  
    private double balance;  
    private int number;  
  
    public CheckingAccount(int number) {  
        this.number = number;  
    }  
  
    public void deposit(double amount) {  
        balance += amount;  
    }  
  
    public void withdraw(double amount) throws  
InsufficientFundsException {  
        if(amount <= balance) {  
            balance -= amount;  
        }else {  
            double needs = amount - balance;  
            throw new InsufficientFundsException(needs);  
        }  
    }  
  
    public double getBalance() {  
        return balance;  
    }  
  
    public int getNumber() {  
        return number;  
    }  
}
```

El siguiente programa de `BankDemo` demuestra cómo invocar los métodos `deposit ()` y `retiro ()` de `CheckingAccount`.

```
// File Name BankDemo.java  
public class BankDemo {  
  
    public static void main(String [] args) {  
        CheckingAccount c = new CheckingAccount(101);  
        System.out.println("Depositing $500...");  
        c.deposit(500.00);  
    }  
}
```

```

try {
    System.out.println("\nWithdrawing $100...");
    c.withdraw(100.00);
    System.out.println("\nWithdrawing $600...");
    c.withdraw(600.00);
} catch (InsufficientFundsException e) {
    System.out.println("Sorry, but you are short $" +
e.getAmount());
    e.printStackTrace();
}
}
}

```

Compile los tres archivos anteriores y ejecute BankDemo. Esto producirá el siguiente resultado:

Salida

Depositing \$500...

Withdrawing \$100...

Withdrawing \$600...

Sorry, but you are short \$200.0

InsufficientFundsException

at CheckingAccount.withdraw(CheckingAccount.java:25)

at BankDemo.main(BankDemo.java:13)

Excepciones Comunes

En Java, es posible definir dos categorías de Excepciones y Errores.

- **Excepciones de JVM**: estas son excepciones / errores que **JVM genera** exclusiva o lógicamente. Ejemplos: `NullPointerException`, `ArrayIndexOutOfBoundsException`, `ClassCastException`.
- **Excepciones programáticas**: estas excepciones son lanzadas explícitamente por la aplicación o los programadores de API. Ejemplos: `IllegalArgumentException`, `IllegalStateException`.

Java - Clases internas

En este capítulo, discutiremos las clases internas de Java.

Clases anidadas

En Java, al igual que los métodos, las variables de una clase también pueden tener otra clase como miembro. Escribir una clase dentro de otra está permitido en Java. La clase escrita dentro se llama **clase anidada**, y la clase que contiene la clase interna se llama **clase externa**.

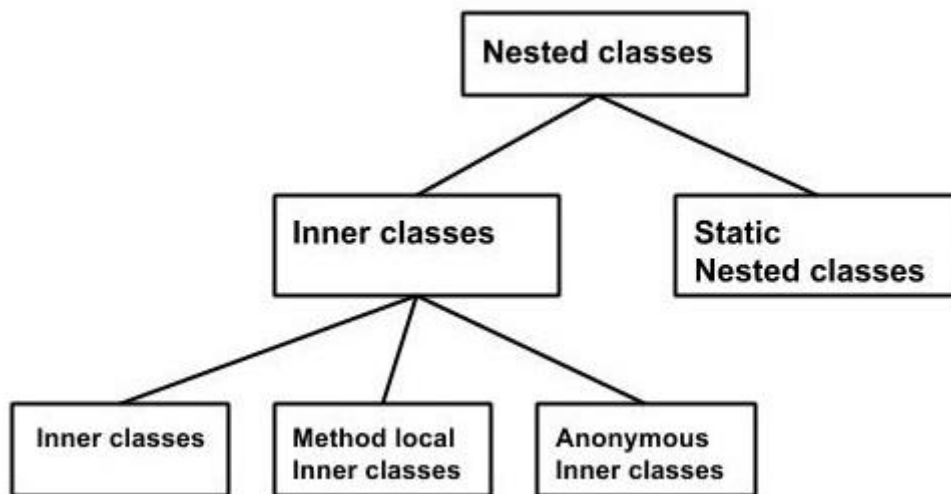
Sintaxis

La siguiente es la sintaxis para escribir una clase anidada. Aquí, la clase **Outer_Demo** es la clase externa y la clase **Inner_Demo** es la clase anidada.

```
class Outer_Demo {  
    class Inner_Demo {  
    }  
}
```

Las clases anidadas se dividen en dos tipos:

- **Clases anidadas no estáticas** : son los miembros no estáticos de una clase.
- **Clases anidadas estáticas** : son los miembros estáticos de una clase.



Clases internas (clases anidadas no estáticas)

Las clases internas son un mecanismo de seguridad en Java. Sabemos que una clase no puede asociarse con el modificador de acceso **privado**, pero si tenemos la clase como miembro de otra clase, entonces la clase interna puede hacerse privada. Y esto también se utiliza para acceder a los miembros privados de una clase.

Las clases internas son de tres tipos dependiendo de cómo y dónde las defina. Ellos son

- Clase interior
- Método de clase interna local
- Clase interna anónima

Clase interior

Crear una clase interna es bastante simple. Solo necesita escribir una clase dentro de una clase. A diferencia de una clase, una clase interna puede ser privada y una vez que declaras que una clase interna es privada, no se puede acceder desde un objeto fuera de la clase.

El siguiente es el programa para crear una clase interna y acceder a ella. En el ejemplo dado, hacemos que la clase interna sea privada y accedemos a la clase a través de un método.

Ejemplo

```
class Outer_Demo {
    int num;

    // inner class
    private class Inner_Demo {
        public void print() {
            System.out.println("This is an inner class");
        }
    }

    // Accessing the inner class from the method within
    void display_Inner() {
        Inner_Demo inner = new Inner_Demo();
        inner.print();
    }
}

public class My_class {

    public static void main(String args[]) {
        // Instantiating the outer class
        Outer_Demo outer = new Outer_Demo();

        // Accessing the display_Inner() method.
        outer.display_Inner();
    }
}
```

Aquí puede observar que **Outer_Demo** es la clase externa, **Inner_Demo** es la clase interna, **display_Inner ()** es el método dentro del cual estamos instanciando la clase interna, y este método se invoca desde el método **principal** .

Si compila y ejecuta el programa anterior, obtendrá el siguiente resultado:

Salida

This is an inner class.

Accediendo a los miembros privados

Como se mencionó anteriormente, las clases internas también se utilizan para acceder a los miembros privados de una clase. Supongamos que una clase tiene miembros privados para acceder a ellos. Escriba una clase interna en él, devuelva los miembros privados de un método dentro de la clase interna, digamos, **getValue ()** , y finalmente de otra clase (desde la que desea acceder a los miembros privados) llame al método **getValue ()** del interno clase.

Para crear una instancia de la clase interna, inicialmente debe crear una instancia de la clase externa. Posteriormente, utilizando el objeto de la clase externa, a continuación se muestra la forma en que puede crear una instancia de la clase interna.

```
Outer_Demo outer = new Outer_Demo();
Outer_Demo.Inner_Demo inner = outer.new Inner_Demo();
```

El siguiente programa muestra cómo acceder a los miembros privados de una clase usando la clase interna.

Ejemplo

```
class Outer_Demo {
    // private variable of the outer class
    private int num = 175;

    // inner class
    public class Inner_Demo {
        public int getNum() {
            System.out.println("This is the getnum method of the
inner class");
            return num;
        }
    }
}

public class My_class2 {

    public static void main(String args[]) {
        // Instantiating the outer class
        Outer_Demo outer = new Outer_Demo();

        // Instantiating the inner class
        Outer_Demo.Inner_Demo inner = outer.new Inner_Demo();
        System.out.println(inner.getNum());
    }
}
```

Si compila y ejecuta el programa anterior, obtendrá el siguiente resultado:

Salida

```
This is the getnum method of the inner class: 175
```

Método de clase interna local

En Java, podemos escribir una clase dentro de un método y este será un tipo local. Al igual que las variables locales, el alcance de la clase interna está restringido dentro del método.

Una clase interna de método local solo puede instanciarse dentro del método donde se define la clase interna. El siguiente programa muestra cómo usar una clase interna de método local.

Ejemplo

```
public class Outerclass {
    // instance method of the outer class
    void my_Method() {
        int num = 23;

        // method-local inner class
        class MethodInner_Demo {
            public void print() {
                System.out.println("This is method inner class
"+num);
            }
        } // end of inner class

        // Accessing the inner class
        MethodInner_Demo inner = new MethodInner_Demo();
        inner.print();
    }

    public static void main(String args[]) {
        Outerclass outer = new Outerclass();
        outer.my_Method();
    }
}
```

Si compila y ejecuta el programa anterior, obtendrá el siguiente resultado:

Salida

```
This is method inner class 23
```

Clase interna anónima

Una clase interna declarada sin un nombre de clase se conoce como una **clase interna anónima**. En caso de clases internas anónimas, las declaramos y las instanciamos al mismo tiempo. En general, se usan siempre que necesite anular el método de una clase o una interfaz. La sintaxis de una clase interna anónima es la siguiente:

Sintaxis

```
AnonymousInner an_inner = new AnonymousInner() {
    public void my_method() {
        .....
        .....
    }
};
```

El siguiente programa muestra cómo anular el método de una clase usando una clase interna anónima.

Ejemplo

```

abstract class AnonymousInner {
    public abstract void mymethod();
}

public class Outer_class {

    public static void main(String args[]) {
        AnonymousInner inner = new AnonymousInner() {
            public void mymethod() {
                System.out.println("This is an example of
anonymous inner class");
            }
        };
        inner.mymethod();
    }
}

```

Si compila y ejecuta el programa anterior, obtendrá el siguiente resultado:

Salida

```
This is an example of anonymous inner class
```

Del mismo modo, puede anular los métodos de la clase concreta, así como la interfaz, utilizando una clase interna anónima.

Clase interna anónima como argumento

En general, si un método acepta un objeto de una interfaz, una clase abstracta o una clase concreta, entonces podemos implementar la interfaz, extender la clase abstracta y pasar el objeto al método. Si es una clase, entonces podemos pasarla directamente al método.

Pero en los tres casos, puede pasar una clase interna anónima al método. Aquí está la sintaxis de pasar una clase interna anónima como argumento de método:

```

obj.my_Method(new My_Class() {
    public void Do() {
        .....
        .....
    }
});

```

El siguiente programa muestra cómo pasar una clase interna anónima como argumento de método.

Ejemplo

```

// interface
interface Message {
    String greet();
}

```

```

public class My_class {
    // method which accepts the object of interface Message
    public void displayMessage(Message m) {
        System.out.println(m.greet() +
            ", This is an example of anonymous inner class as an
argument");
    }

    public static void main(String args[]) {
        // Instantiating the class
        My_class obj = new My_class();

        // Passing an anonymous inner class as an argument
        obj.displayMessage(new Message() {
            public String greet() {
                return "Hello";
            }
        });
    }
}

```

Si compila y ejecuta el programa anterior, le dará el siguiente resultado:

Salida

```

Hello, This is an example of anonymous inner class as an
argument

```

Clase anidada estática

Una clase interna estática es una clase anidada que es un miembro estático de la clase externa. Se puede acceder sin crear instancias de la clase externa, utilizando otros miembros estáticos. Al igual que los miembros estáticos, una clase anidada estática no tiene acceso a las variables de instancia y los métodos de la clase externa. La sintaxis de la clase anidada estática es la siguiente:

Sintaxis

```

class MyOuter {
    static class Nested_Demo {
    }
}

```

Crear instancias de una clase anidada estática es un poco diferente de crear instancias de una clase interna. El siguiente programa muestra cómo usar una clase anidada estática.

Ejemplo

```

public class Outer {
    static class Nested_Demo {
        public void my_method() {

```

```

        System.out.println("This is my nested class");
    }
}

public static void main(String args[]) {
    Outer.Nested_Demo nested = new Outer.Nested_Demo();
    nested.my_method();
}
}

```

Si compila y ejecuta el programa anterior, obtendrá el siguiente resultado:

Salida

```
This is my nested class
```

Java - Herencia

La herencia se puede definir como el proceso donde una clase adquiere las propiedades (métodos y campos) de otra. Con el uso de la herencia, la información se hace manejable en un orden jerárquico.

La clase que hereda las propiedades de otros se conoce como subclase (clase derivada, clase secundaria) y la clase cuyas propiedades se heredan se conoce como superclase (clase base, clase primaria).

extiende la palabra clave

extend es la palabra clave utilizada para heredar las propiedades de una clase. La siguiente es la sintaxis de la palabra clave extend.

Sintaxis

```

class Super {
    .....
    .....
}
class Sub extends Super {
    .....
    .....
}

```

Código de muestra

El siguiente es un ejemplo que demuestra la herencia de Java. En este ejemplo, puede observar dos clases, a saber, Cálculo y Mi cálculo.

Usando la palabra clave extend, My_Calculation hereda los métodos suma () y Resta () de la clase Cálculo.

Copie y pegue el siguiente programa en un archivo con el nombre My_Calculation.java

Ejemplo

```

class Calculation {
    int z;

    public void addition(int x, int y) {
        z = x + y;
        System.out.println("The sum of the given numbers:"+z);
    }

    public void Subtraction(int x, int y) {
        z = x - y;
        System.out.println("The difference between the given
numbers:"+z);
    }
}

public class My_Calculation extends Calculation {
    public void multiplication(int x, int y) {
        z = x * y;
        System.out.println("The product of the given
numbers:"+z);
    }

    public static void main(String args[]) {
        int a = 20, b = 10;
        My_Calculation demo = new My_Calculation();
        demo.addition(a, b);
        demo.Subtraction(a, b);
        demo.multiplication(a, b);
    }
}

```

Compile y ejecute el código anterior como se muestra a continuación.

```

javac My_Calculation.java
java My_Calculation

```

Después de ejecutar el programa, producirá el siguiente resultado:

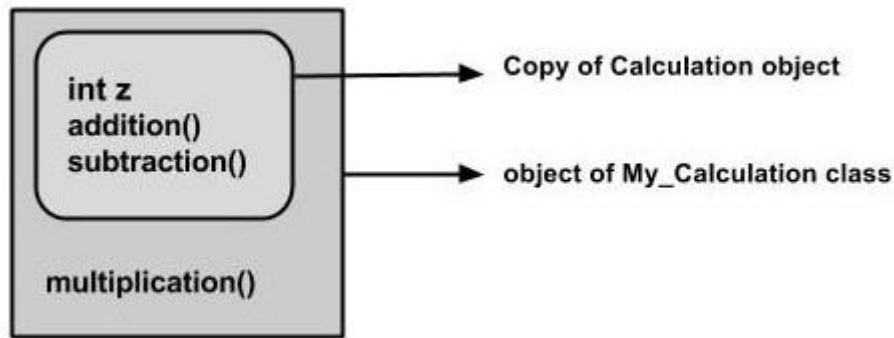
Salida

```

The sum of the given numbers:30
The difference between the given numbers:10
The product of the given numbers:200

```

En el programa dado, cuando se crea un objeto para la clase **My_Calculation**, se crea una copia de los contenidos de la superclase. Es por eso que, utilizando el objeto de la subclase, puede acceder a los miembros de una superclase.



La variable de referencia Superclass puede contener el objeto de subclase, pero al usar esa variable solo puede acceder a los miembros de la superclase, por lo que para acceder a los miembros de ambas clases se recomienda crear siempre una variable de referencia para la subclase.

Si considera el programa anterior, puede crear una instancia de la clase como se indica a continuación. Pero usando la variable de referencia de la superclase (**cal** en este caso) no se puede llamar al método **multiplication()** , que pertenece a la subclase **My_Calculation**.

```
Calculation demo = new My_Calculation();  
demo.addition(a, b);  
demo.Subtraction(a, b);
```

Nota : una subclase hereda todos los miembros (campos, métodos y clases anidadas) de su superclase. Los constructores no son miembros, por lo que no son heredados por las subclases, pero el constructor de la superclase se puede invocar desde la subclase.

La súper palabra clave

La palabra clave **super** es similar a **esta** palabra clave. Los siguientes son los escenarios donde se usa la palabra clave **super**.

- Se utiliza para **diferenciar los miembros** de la superclase de los miembros de la subclase, si tienen los mismos nombres.
- Se utiliza para **invocar al** constructor de **la superclase** desde la subclase.

Diferenciando a los miembros

Si una clase hereda las propiedades de otra clase. Y si los miembros de la superclase tienen los mismos nombres que la subclase, para diferenciar estas variables usamos la palabra clave **super** como se muestra a continuación.

```
super.variable  
super.method();
```

Código de muestra

Esta sección le proporciona un programa que demuestra el uso de la palabra clave **super** .

En el programa dado, tiene dos clases, a saber, *Sub_class* y *Super_class*, ambas tienen un método llamado `display ()` con diferentes implementaciones y una variable llamada `num` con diferentes valores. Invocamos el método `display ()` de ambas clases e imprimimos el valor de la variable `num` de ambas clases. Aquí puede observar que hemos utilizado la palabra clave `super` para diferenciar los miembros de la superclase de la subclase.

Copie y pegue el programa en un archivo con el nombre `Sub_class.java`.

Ejemplo

```
class Super_class {
    int num = 20;

    // display method of superclass
    public void display() {
        System.out.println("This is the display method of
superclass");
    }
}

public class Sub_class extends Super_class {
    int num = 10;

    // display method of sub class
    public void display() {
        System.out.println("This is the display method of
subclass");
    }

    public void my_method() {
        // Instantiating subclass
        Sub_class sub = new Sub_class();

        // Invoking the display() method of sub class
        sub.display();

        // Invoking the display() method of superclass
        super.display();

        // printing the value of variable num of subclass
        System.out.println("value of the variable named num in
sub class:"+ sub.num);

        // printing the value of variable num of superclass
        System.out.println("value of the variable named num in
super class:"+ super.num);
    }

    public static void main(String args[]) {
        Sub_class obj = new Sub_class();
        obj.my_method();
    }
}
```



```
}
```

Compile y ejecute el código anterior utilizando la siguiente sintaxis.

```
javac Super_Demo  
java Super
```

Al ejecutar el programa, obtendrá el siguiente resultado:

Salida

```
This is the display method of subclass  
This is the display method of superclass  
value of the variable named num in sub class:10  
value of the variable named num in super class:20
```

Invocación de constructor de superclase

Si una clase hereda las propiedades de otra clase, la subclase adquiere automáticamente el constructor predeterminado de la superclase. Pero si desea llamar a un constructor parametrizado de la superclase, debe usar la palabra clave `super` como se muestra a continuación.

```
super(values);
```

Código de muestra

El programa dado en esta sección muestra cómo usar la palabra clave `super` para invocar al constructor parametrizado de la superclase. Este programa contiene una superclase y una subclase, donde la superclase contiene un constructor parametrizado que acepta un valor entero, y usamos la palabra clave `super` para invocar al constructor parametrizado de la superclase.

Copie y pegue el siguiente programa en un archivo con el nombre `Subclass.java`

Ejemplo

```
class Superclass {  
    int age;  
  
    Superclass(int age) {  
        this.age = age;  
    }  
  
    public void getAge() {  
        System.out.println("The value of the variable named age  
in super class is: " +age);  
    }  
}  
  
public class Subclass extends Superclass {  
    Subclass(int age) {  
        super(age);  
    }  
}
```

```
}

public static void main(String argd[]) {
    Subclass s = new Subclass(24);
    s.getAge();
}
}
```

Compile y ejecute el código anterior utilizando la siguiente sintaxis.

```
javac Subclass
java Subclass
```

Al ejecutar el programa, obtendrá el siguiente resultado:

Salida

```
The value of the variable named age in super class is: 24
```

Relación IS-A

IS-A es una forma de decir: este objeto es un tipo de ese objeto. Veamos cómo la **extiende** palabra clave se utiliza para lograr herencia.

```
clase pública animal { }

clase pública Mammal extiende Animal { }

Reptil de clase pública extiende Animal { }

clase pública Perro extiende Mamífero { }
```

Ahora, según el ejemplo anterior, en términos orientados a objetos, lo siguiente es cierto:

- Animal es la superclase de la clase Mamífero.
- Animal es la superclase de la clase Reptile.
- Mamífero y Reptil son subclases de la clase Animal.
- El perro es la subclase de las clases de mamíferos y animales.

Ahora, si consideramos la relación IS-A, podemos decir:

- Mamífero IS-A Animal
- Reptil IS-A Animal
- Perro IS-A Mamífero
- Por lo tanto: el perro es un animal también

Con el uso de la palabra clave extend, las subclases podrán heredar todas las propiedades de la superclase, excepto las propiedades privadas de la superclase.

Podemos asegurar que Mammal es realmente un Animal con el uso del operador de instancia.

Ejemplo

```
class Animal {  
}  
  
class Mammal extends Animal {  
}  
  
class Reptile extends Animal {  
}  
  
public class Dog extends Mammal {  
  
    public static void main(String args[]) {  
        Animal a = new Animal();  
        Mammal m = new Mammal();  
        Dog d = new Dog();  
  
        System.out.println(m instanceof Animal);  
        System.out.println(d instanceof Mammal);  
        System.out.println(d instanceof Animal);  
    }  
}
```

Esto producirá el siguiente resultado:

Salida

```
true  
true  
true
```

Como conocemos bien la palabra clave **extendidos** , veamos cómo se usa la palabra clave **implements** para obtener la relación IS-A.

En general, la palabra clave **implements** se usa con clases para heredar las propiedades de una interfaz. Las interfaces nunca pueden ser extendidas por una clase.

Ejemplo

```
public interface Animal {  
}  
  
public class Mammal implements Animal {  
}  
  
public class Dog extends Mammal {  
}
```

La instancia de palabra clave

Usemos el operador **instanceof** para verificar si Mammal es realmente un Animal y si el perro es realmente un Animal.

Ejemplo

```
interface Animal{}
class Mammal implements Animal{}

public class Dog extends Mammal {

    public static void main(String args[]) {
        Mammal m = new Mammal();
        Dog d = new Dog();

        System.out.println(m instanceof Animal);
        System.out.println(d instanceof Mammal);
        System.out.println(d instanceof Animal);
    }
}
```

Esto producirá el siguiente resultado:

Salida

```
true
true
true
```

Tiene una relación

Estas relaciones se basan principalmente en el uso. Esto determina si una determinada clase tiene **una** determinada cosa. Esta relación ayuda a reducir la duplicación de código y los errores.

Veamos un ejemplo:

Ejemplo

```
public class Vehicle{}
public class Speed{}

public class Van extends Vehicle {
    private Speed sp;
}
```

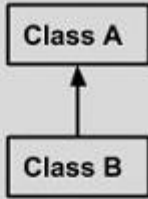
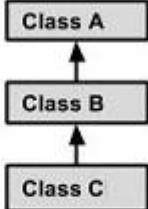
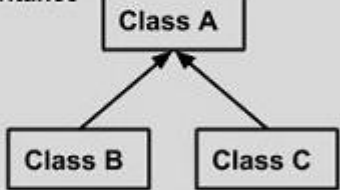
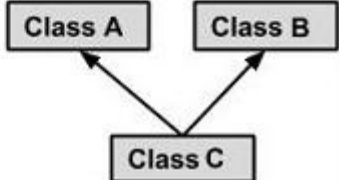
Esto muestra esa clase Van HAS-A Speed. Al tener una clase separada para Speed, no tenemos que poner todo el código que pertenece a speed dentro de la clase Van, lo que hace posible reutilizar la clase Speed en múltiples aplicaciones.

En la función Orientada a objetos, los usuarios no necesitan preocuparse sobre qué objeto está haciendo el trabajo real. Para lograr esto, la clase Van oculta los detalles de implementación de los usuarios de la clase Van. Entonces, básicamente, lo que sucede es que los usuarios pedirían a la

clase Van que realice una determinada acción y la clase Van hará el trabajo por sí misma o le pedirá a otra clase que realice la acción.

Tipos de herencia

Hay varios tipos de herencia como se demuestra a continuación.

Single Inheritance  <pre> graph BT B[Class B] --> A[Class A] </pre>	<pre> public class A { } public class B extends A { } </pre>
Multi Level Inheritance  <pre> graph BT C[Class C] --> B[Class B] B --> A[Class A] </pre>	<pre> public class A {} public class B extends A {.....} public class C extends B {.....} </pre>
Hierarchical Inheritance  <pre> graph BT B[Class B] --> A[Class A] C[Class C] --> A </pre>	<pre> public class A {} public class B extends A {.....} public class C extends A {.....} </pre>
Multiple Inheritance  <pre> graph BT A[Class A] --> C[Class C] B[Class B] --> C </pre>	<pre> public class A {} public class B {.....} public class C extends A,B { } // Java does not support mutiple Inheritance </pre>

Un hecho muy importante para recordar es que Java no admite la herencia múltiple. Esto significa que una clase no puede extender más de una clase. Por lo tanto, seguir es ilegal:

Ejemplo

```
public class extends Animal, Mammal{}
```

Sin embargo, una clase puede implementar una o más interfaces, lo que ha ayudado a Java a deshacerse de la imposibilidad de herencia múltiple.

Java: anulación

En el capítulo anterior, hablamos sobre superclases y subclases. Si una clase hereda un método de su superclase, existe la posibilidad de anular el método siempre que no esté marcado como final.

El beneficio de la anulación es: la capacidad de definir un comportamiento específico para el tipo de subclase, lo que significa que una subclase puede implementar un método de clase padre en función de sus requisitos.

En términos orientados a objetos, anular significa anular la funcionalidad de un método existente.

Ejemplo

Veamos un ejemplo.

```
class Animal {
    public void move() {
        System.out.println("Animals can move");
    }
}

class Dog extends Animal {
    public void move() {
        System.out.println("Dogs can walk and run");
    }
}

public class TestDog {

    public static void main(String args[]) {
        Animal a = new Animal();    // Animal reference and
        object
        Animal b = new Dog();       // Animal reference but Dog
        object

        a.move();    // runs the method in Animal class
        b.move();    // runs the method in Dog class
    }
}
```

Esto producirá el siguiente resultado:

Salida

```
Animals can move
Dogs can walk and run
```

En el ejemplo anterior, puede ver que aunque **b** es un tipo de animal, ejecuta el método de movimiento en la clase Perro. La razón de esto es: en tiempo de compilación, la verificación se realiza en el tipo de referencia. Sin embargo, en el tiempo de ejecución, JVM calcula el tipo de objeto y ejecuta el método que pertenece a ese objeto en particular.

Por lo tanto, en el ejemplo anterior, el programa se compilará correctamente ya que la clase Animal tiene el método move. Luego, en el tiempo de ejecución, ejecuta el método específico para ese objeto.

Considere el siguiente ejemplo:

Ejemplo

```
class Animal {
    public void move() {
        System.out.println("Animals can move");
    }
}

class Dog extends Animal {
    public void move() {
        System.out.println("Dogs can walk and run");
    }
    public void bark() {
        System.out.println("Dogs can bark");
    }
}

public class TestDog {

    public static void main(String args[]) {
        Animal a = new Animal();    // Animal reference and
object
        Animal b = new Dog();       // Animal reference but Dog
object

        a.move();    // runs the method in Animal class
        b.move();    // runs the method in Dog class
        b.bark();

    }
}
```

Esto producirá el siguiente resultado:

Salida

```
TestDog.java:26: error: cannot find symbol
    b.bark();
    ^
    symbol:   method bark()
    location: variable b of type Animal
1 error
```

Este programa arrojará un error de tiempo de compilación ya que el tipo de referencia de b Animal no tiene un método con el nombre de ladrar.

Reglas para anulación de métodos

- La lista de argumentos debe ser exactamente la misma que la del método anulado.

- El tipo de retorno debe ser el mismo o un subtipo del tipo de retorno declarado en el método anulado original en la superclase.
- El nivel de acceso no puede ser más restrictivo que el nivel de acceso del método anulado. Por ejemplo: si el método de la superclase se declara público, el método de anulación en la subclase no puede ser privado ni estar protegido.
- Los métodos de instancia solo pueden anularse si son heredados por la subclase.
- Un método declarado final no se puede anular.
- Un método declarado estático no se puede anular, pero se puede volver a declarar.
- Si un método no se puede heredar, no se puede anular.
- Una subclase dentro del mismo paquete que la superclase de la instancia puede anular cualquier método de superclase que no se declare privado o final.
- Una subclase en un paquete diferente solo puede anular los métodos no finales declarados públicos o protegidos.
- Un método de anulación puede arrojar cualquier excepción desmarcada, independientemente de si el método anulado arroja excepciones o no. Sin embargo, el método de anulación no debe arrojar excepciones marcadas que sean nuevas o más amplias que las declaradas por el método anulado. El método de anulación puede generar excepciones más estrechas o menos que el método anulado.
- Los constructores no pueden ser anulados.

Usando la súper palabra clave

Cuando se invoca una versión de superclase de un método anulado, se utiliza la palabra clave **super** .

Ejemplo

```
class Animal {
    public void move() {
        System.out.println("Animals can move");
    }
}

class Dog extends Animal {
    public void move() {
        super.move();    // invokes the super class method
        System.out.println("Dogs can walk and run");
    }
}

public class TestDog {

    public static void main(String args[]) {
        Animal b = new Dog();    // Animal reference but Dog
object
```



```
b.move();    // runs the method in Dog class
    }
}
```

Esto producirá el siguiente resultado:

Salida

```
Animals can move
Dogs can walk and run
```

Java - Polimorfismo

El polimorfismo es la capacidad de un objeto de adoptar muchas formas. El uso más común del polimorfismo en OOP ocurre cuando se usa una referencia de clase primaria para referirse a un objeto de clase secundaria.

Cualquier objeto Java que pueda pasar más de una prueba IS-A se considera polimórfico. En Java, todos los objetos de Java son polimórficos ya que cualquier objeto pasará la prueba IS-A para su propio tipo y para la clase Object.

Es importante saber que la única forma posible de acceder a un objeto es a través de una variable de referencia. Una variable de referencia puede ser de un solo tipo. Una vez declarado, el tipo de una variable de referencia no se puede cambiar.

La variable de referencia se puede reasignar a otros objetos siempre que no se declare final. El tipo de la variable de referencia determinaría los métodos que puede invocar en el objeto.

Una variable de referencia puede hacer referencia a cualquier objeto de su tipo declarado o cualquier subtipo de su tipo declarado. Una variable de referencia puede declararse como una clase o tipo de interfaz.

Ejemplo

Veamos un ejemplo.

```
public interface Vegetarian{}
public class Animal{}
public class Deer extends Animal implements Vegetarian{}
```

Ahora, se considera que la clase Deer es polimórfica, ya que tiene herencia múltiple. Los siguientes son verdaderos para los ejemplos anteriores:

- Un ciervo es un animal
- Un ciervo es un vegetariano
- Un ciervo es un ciervo
- Un ciervo es un objeto

Cuando aplicamos los hechos variables de referencia a una referencia de objeto Deer, las siguientes declaraciones son legales:

Ejemplo

```
Deer d = new Deer();  
Animal a = d;  
Vegetarian v = d;  
Object o = d;
```

Todas las variables de referencia d, a, v, o se refieren al mismo objeto Deer en el montón.

Métodos virtuales

En esta sección, le mostraré cómo el comportamiento de los métodos anulados en Java le permite aprovechar el polimorfismo al diseñar sus clases.

Ya hemos discutido la anulación de métodos, donde una clase secundaria puede anular un método en su padre. Un método anulado está esencialmente oculto en la clase primaria y no se invoca a menos que la clase secundaria use la palabra clave super dentro del método de anulación.

Ejemplo

```
/* File name : Employee.java */  
public class Employee {  
    private String name;  
    private String address;  
    private int number;  
  
    public Employee(String name, String address, int number) {  
        System.out.println("Constructing an Employee");  
        this.name = name;  
        this.address = address;  
        this.number = number;  
    }  
  
    public void mailCheck() {  
        System.out.println("Mailing a check to " + this.name +  
" " + this.address);  
    }  
  
    public String toString() {  
        return name + " " + address + " " + number;  
    }  
  
    public String getName() {  
        return name;  
    }  
  
    public String getAddress() {  
        return address;  
    }  
}
```

```

    public void setAddress(String newAddress) {
        address = newAddress;
    }

    public int getNumber() {
        return number;
    }
}

```

Ahora supongamos que ampliamos la clase de empleado de la siguiente manera:

```

/* File name : Salary.java */
public class Salary extends Employee {
    private double salary; // Annual salary

    public Salary(String name, String address, int number,
double salary) {
        super(name, address, number);
        setSalary(salary);
    }

    public void mailCheck() {
        System.out.println("Within mailCheck of Salary class
");
        System.out.println("Mailing check to " + getName()
+ " with salary " + salary);
    }

    public double getSalary() {
        return salary;
    }

    public void setSalary(double newSalary) {
        if(newSalary >= 0.0) {
            salary = newSalary;
        }
    }

    public double computePay() {
        System.out.println("Computing salary pay for " +
getName());
        return salary/52;
    }
}

```

Ahora, estudie cuidadosamente el siguiente programa e intente determinar su salida:

```

/* File name : VirtualDemo.java */
public class VirtualDemo {

    public static void main(String [] args) {

```

```

        Salary s = new Salary("Mohd Mohtashim", "Ambehta, UP",
3, 3600.00);
        Employee e = new Salary("John Adams", "Boston, MA", 2,
2400.00);
        System.out.println("Call mailCheck using Salary
reference --");
        s.mailCheck();
        System.out.println("\n Call mailCheck using Employee
reference--");
        e.mailCheck();
    }
}

```

Esto producirá el siguiente resultado:

Salida

Constructing an Employee
Constructing an Employee

Call mailCheck using Salary reference --
Within mailCheck of Salary class
Mailing check to Mohd Mohtashim with salary 3600.0

Call mailCheck using Employee reference--
Within mailCheck of Salary class
Mailing check to John Adams with salary 2400.0

Aquí, instanciamos dos objetos Salario. Uno usando una referencia de Salario **s** , y el otro usando una referencia de Empleado **e** .

Al invocar *s.mailCheck()* , el compilador ve *mailCheck()* en la clase Salario en tiempo de compilación, y la JVM invoca *mailCheck()* en la clase Salario en tiempo de ejecución.

mailCheck() en **e** es bastante diferente porque **e** es una referencia de Empleado. Cuando el compilador ve *e.mailCheck()* , ve el método *mailCheck()* en la clase Employee.

Aquí, en el momento de la compilación, el compilador usó *mailCheck()* en Employee para validar esta declaración. Sin embargo, en tiempo de ejecución, la JVM invoca *mailCheck()* en la clase Salario.

Este comportamiento se denomina invocación de método virtual, y estos métodos se denominan métodos virtuales. Se invoca un método anulado en tiempo de ejecución, sin importar qué tipo de datos sea la referencia que se utilizó en el código fuente en tiempo de compilación.

Java - Abstracción

Según el diccionario, la **abstracción** es la calidad de tratar con ideas en lugar de eventos. Por ejemplo, cuando considera el caso del correo electrónico, los detalles complejos como lo que sucede tan pronto como envía un correo electrónico, el protocolo que utiliza su servidor de correo electrónico quedan

ocultos para el usuario. Por lo tanto, para enviar un correo electrónico solo necesita escribir el contenido, mencionar la dirección del receptor y hacer clic en enviar.

Del mismo modo, en la programación orientada a objetos, la abstracción es un proceso de ocultar los detalles de implementación del usuario, solo la funcionalidad se proporcionará al usuario. En otras palabras, el usuario tendrá la información sobre lo que hace el objeto en lugar de cómo lo hace.

En Java, la abstracción se logra usando clases abstractas e interfaces.

Clase abstracta

Una clase que contiene la palabra clave **abstracta** en su declaración se conoce como clase abstracta.

- Las clases abstractas pueden o no contener *métodos abstractos*, es decir, métodos sin cuerpo (public void get ());
- Pero, si una clase tiene al menos un método abstracto, entonces la clase **debe** declararse abstracta.
- Si una clase se declara abstracta, no se puede instanciar.
- Para usar una clase abstracta, debe heredarla de otra clase, proporcionar implementaciones a los métodos abstractos en ella.
- Si hereda una clase abstracta, debe proporcionar implementaciones a todos los métodos abstractos que contiene.

Ejemplo

Esta sección le proporciona un ejemplo de la clase abstracta. Para crear una clase abstracta, solo use la palabra clave **abstract** antes de la palabra clave **class**, en la declaración de clase.

```
/* File name : Employee.java */
public abstract class Employee {
    private String name;
    private String address;
    private int number;

    public Employee(String name, String address, int number) {
        System.out.println("Constructing an Employee");
        this.name = name;
        this.address = address;
        this.number = number;
    }

    public double computePay() {
        System.out.println("Inside Employee computePay");
        return 0.0;
    }

    public void mailCheck() {
```

```

        System.out.println("Mailing a check to " + this.name +
" " + this.address);
    }

    public String toString() {
        return name + " " + address + " " + number;
    }

    public String getName() {
        return name;
    }

    public String getAddress() {
        return address;
    }

    public void setAddress(String newAddress) {
        address = newAddress;
    }

    public int getNumber() {
        return number;
    }
}

```

Puede observar que, excepto los métodos abstractos, la clase Employee es la misma que la clase normal en Java. La clase ahora es abstracta, pero todavía tiene tres campos, siete métodos y un constructor.

Ahora puede intentar crear una instancia de la clase Empleado de la siguiente manera:

```

/* File name : AbstractDemo.java */
public class AbstractDemo {

    public static void main(String [] args) {
        /* Following is not allowed and would raise error */
        Employee e = new Employee("George W.", "Houston, TX",
43);
        System.out.println("\n Call mailCheck using Employee
reference--");
        e.mailCheck();
    }
}

```

Cuando compila la clase anterior, le da el siguiente error:

```

Employee.java:46: Employee is abstract; cannot be
instantiated
        Employee e = new Employee("George W.", "Houston, TX",
43);
                        ^
1 error

```

Heredar la clase abstracta

Podemos heredar las propiedades de la clase Employee al igual que la clase concreta de la siguiente manera:

Ejemplo

```
/* File name : Salary.java */
public class Salary extends Employee {
    private double salary;    // Annual salary

    public Salary(String name, String address, int number,
double salary) {
        super(name, address, number);
        setSalary(salary);
    }

    public void mailCheck() {
        System.out.println("Within mailCheck of Salary class
");
        System.out.println("Mailing check to " + getName() + "
with salary " + salary);
    }

    public double getSalary() {
        return salary;
    }

    public void setSalary(double newSalary) {
        if(newSalary >= 0.0) {
            salary = newSalary;
        }
    }

    public double computePay() {
        System.out.println("Computing salary pay for " +
getName());
        return salary/52;
    }
}
```

Aquí, no puede crear una instancia de la clase Empleado, pero puede crear una instancia de la Clase Salario, y utilizando esta instancia puede acceder a los tres campos y siete métodos de la clase Empleado como se muestra a continuación.

```
/* File name : AbstractDemo.java */
public class AbstractDemo {

    public static void main(String [] args) {
        Salary s = new Salary("Mohd Mohtashim", "Ambehta, UP",
3, 3600.00);
    }
}
```

```

        Employee e = new Salary("John Adams", "Boston, MA", 2,
2400.00);
        System.out.println("Call mailCheck using Salary
reference --");
        s.mailCheck();
        System.out.println("\n Call mailCheck using Employee
reference--");
        e.mailCheck();
    }
}

```

Esto produce el siguiente resultado:

Salida

```

Constructing an Employee
Constructing an Employee
Call mailCheck using Salary reference --
Within mailCheck of Salary class
Mailing check to Mohd Mohtashim with salary 3600.0

```

```

    Call mailCheck using Employee reference--
Within mailCheck of Salary class
Mailing check to John Adams with salary 2400.0

```

Métodos abstractos

Si desea que una clase contenga un método particular pero desea que la implementación real de ese método sea determinada por las clases secundarias, puede declarar el método en la clase primaria como un resumen.

- La palabra clave **abstract** se usa para declarar el método como abstract.
- Debe colocar la palabra clave **abstracta** antes del nombre del método en la declaración del método.
- Un método abstracto contiene una firma de método, pero ningún cuerpo de método.
- En lugar de llaves, un método abstracto tendrá un semoi colon (;) al final.

El siguiente es un ejemplo del método abstracto.

Ejemplo

```

public abstract class Employee {
    private String name;
    private String address;
    private int number;

    public abstract double computePay();
    // Remainder of class definition
}

```


Declarar un método como abstracto tiene dos consecuencias:

- La clase que lo contiene debe declararse como abstracta.
- Cualquier clase que herede la clase actual debe anular el método abstracto o declararse como abstracto.

Nota - Eventualmente, una clase descendiente tiene que implementar el método abstracto; de lo contrario, tendría una jerarquía de clases abstractas que no se pueden instanciar.

Supongamos que la clase Salario hereda la clase Empleado, entonces debería implementar el método **computePay ()** como se muestra a continuación:

```
/* File name : Salary.java */
public class Salary extends Employee {
    private double salary;    // Annual salary

    public double computePay() {
        System.out.println("Computing salary pay for " +
getName());
        return salary/52;
    }
    // Remainder of class definition
}
```

Java - Encapsulación

La encapsulación es uno de los cuatro conceptos fundamentales de OOP. Los otros tres son herencia, polimorfismo y abstracción.

La encapsulación en Java es un mecanismo para envolver los datos (variables) y el código que actúa sobre los datos (métodos) juntos como una sola unidad. En la encapsulación, las variables de una clase se ocultarán de otras clases y solo se podrá acceder a ellas a través de los métodos de su clase actual. Por lo tanto, también se conoce como **ocultación de datos** .

Para lograr la encapsulación en Java:

- Declarar las variables de una clase como privadas.
- Proporcione métodos setter y getter públicos para modificar y ver los valores de las variables.

Ejemplo

El siguiente es un ejemplo que demuestra cómo lograr la Encapsulación en Java:

```
/* File name : EncapTest.java */
public class EncapTest {
    private String name;
    private String idNum;
    private int age;

    public int getAge() {
        return age;
    }
}
```

```

    }

    public String getName() {
        return name;
    }

    public String getIdNum() {
        return idNum;
    }

    public void setAge( int newAge) {
        age = newAge;
    }

    public void setName(String newName) {
        name = newName;
    }

    public void setIdNum( String newId) {
        idNum = newId;
    }
}

```

Los métodos public setXXX () y getXXX () son los puntos de acceso de las variables de instancia de la clase EncapTest. Normalmente, estos métodos se denominan getters y setters. Por lo tanto, cualquier clase que quiera acceder a las variables debe acceder a ellas a través de estos captadores y definidores.

Se puede acceder a las variables de la clase EncapTest utilizando el siguiente programa:

```

/* File name : RunEncap.java */
public class RunEncap {

    public static void main(String args[]) {
        EncapTest encap = new EncapTest();
        encap.setName("James");
        encap.setAge(20);
        encap.setIdNum("12343ms");

        System.out.print("Name : " + encap.getName() + " Age : " + encap.getAge());
    }
}

```

Esto producirá el siguiente resultado:

Salida

Name : James Age : 20

Beneficios de la encapsulación

- Los campos de una clase se pueden hacer de solo lectura o solo de escritura.
- Una clase puede tener control total sobre lo que está almacenado en sus campos.

Java - Interfaces

Una interfaz es un tipo de referencia en Java. Es similar a la clase. Es una colección de métodos abstractos. Una clase implementa una interfaz, heredando así los métodos abstractos de la interfaz.

Junto con los métodos abstractos, una interfaz también puede contener constantes, métodos predeterminados, métodos estáticos y tipos anidados. Los cuerpos de método existen solo para métodos predeterminados y métodos estáticos.

Escribir una interfaz es similar a escribir una clase. Pero una clase describe los atributos y comportamientos de un objeto. Y una interfaz contiene comportamientos que implementa una clase.

A menos que la clase que implementa la interfaz sea abstracta, todos los métodos de la interfaz deben definirse en la clase.

Una interfaz es similar a una clase de las siguientes maneras:

- Una interfaz puede contener cualquier cantidad de métodos.
- Una interfaz se escribe en un archivo con una extensión **.java** , con el nombre de la interfaz que coincide con el nombre del archivo.
- El código de byte de una interfaz aparece en un archivo **.class** .
- Las interfaces aparecen en paquetes, y su archivo de bytecode correspondiente debe estar en una estructura de directorio que coincida con el nombre del paquete.

Sin embargo, una interfaz es diferente de una clase en varias formas, incluyendo:

- No puede crear una instancia de una interfaz.
- Una interfaz no contiene ningún constructor.
- Todos los métodos en una interfaz son abstractos.
- Una interfaz no puede contener campos de instancia. Los únicos campos que pueden aparecer en una interfaz deben declararse tanto estáticos como finales.
- Una clase no extiende una interfaz; Es implementado por una clase.
- Una interfaz puede extender múltiples interfaces.

Declarando interfaces

La palabra clave de **interfaz** se utiliza para declarar una interfaz. Aquí hay un ejemplo simple para declarar una interfaz:

Ejemplo

El siguiente es un ejemplo de una interfaz:

```

/* File name : NameOfInterface.java */
import java.lang.*;
// Any number of import statements

public interface NameOfInterface {
    // Any number of final, static fields
    // Any number of abstract method declarations\
}

```

Las interfaces tienen las siguientes propiedades:

- Una interfaz es implícitamente abstracta. No necesita usar la palabra clave **abstracta** al declarar una interfaz.
- Cada método en una interfaz también es implícitamente abstracto, por lo que no se necesita la palabra clave abstracta.
- Los métodos en una interfaz son implícitamente públicos.

Ejemplo

```

/* File name : Animal.java */
interface Animal {
    public void eat();
    public void travel();
}

```

Implementación de interfaces

Cuando una clase implementa una interfaz, puede pensar en la clase como firmando un contrato, acordando realizar los comportamientos específicos de la interfaz. Si una clase no realiza todos los comportamientos de la interfaz, la clase debe declararse como abstracta.

Una clase utiliza los **instrumentos** de palabras clave para implementar una interfaz. La palabra clave `implements` aparece en la declaración de clase después de la porción extendida de la declaración.

Ejemplo

```

/* File name : MammalInt.java */
public class MammalInt implements Animal {

    public void eat() {
        System.out.println("Mammal eats");
    }

    public void travel() {
        System.out.println("Mammal travels");
    }

    public int noOfLegs() {
        return 0;
    }
}

```

```

    }

    public static void main(String args[]) {
        MammalInt m = new MammalInt();
        m.eat();
        m.travel();
    }
}

```

Esto producirá el siguiente resultado:

Salida

```

Mammal eats
Mammal travels

```

Al anular los métodos definidos en las interfaces, hay varias reglas a seguir:

- Las excepciones marcadas no deben declararse en métodos de implementación que no sean los declarados por el método de interfaz o las subclases de los declarados por el método de interfaz.
- La firma del método de interfaz y el mismo tipo o subtipo de retorno deben mantenerse al anular los métodos.
- Una clase de implementación en sí misma puede ser abstracta y, de ser así, no es necesario implementar métodos de interfaz.

Cuando las interfaces de implementación, hay varias reglas:

- Una clase puede implementar más de una interfaz a la vez.
- Una clase puede extender solo una clase, pero implementar muchas interfaces.
- Una interfaz puede extender otra interfaz, de manera similar a como una clase puede extender otra clase.

Interfaces extensibles

Una interfaz puede extender otra interfaz de la misma manera que una clase puede extender otra clase. La palabra clave **extend** se utiliza para extender una interfaz, y la interfaz secundaria hereda los métodos de la interfaz principal.

La siguiente interfaz de deportes se amplía con las interfaces de hockey y fútbol.

Ejemplo

```

// Filename: Sports.java
public interface Sports {
    public void setHomeTeam(String name);
    public void setVisitingTeam(String name);
}

// Filename: Football.java

```

```

public interface Football extends Sports {
    public void homeTeamScored(int points);
    public void visitingTeamScored(int points);
    public void endOfQuarter(int quarter);
}

// Filename: Hockey.java
public interface Hockey extends Sports {
    public void homeGoalScored();
    public void visitingGoalScored();
    public void endOfPeriod(int period);
    public void overtimePeriod(int ot);
}

```

La interfaz de Hockey tiene cuatro métodos, pero hereda dos de Deportes; por lo tanto, una clase que implementa Hockey necesita implementar los seis métodos. Del mismo modo, una clase que implementa Fútbol debe definir los tres métodos de Fútbol y los dos métodos de Deportes.

Extendiendo múltiples interfaces

Una clase Java solo puede extender una clase padre. La herencia múltiple no está permitida. Sin embargo, las interfaces no son clases, y una interfaz puede extender más de una interfaz principal.

La palabra clave extendidos se usa una vez y las interfaces principales se declaran en una lista separada por comas.

Por ejemplo, si la interfaz de Hockey extendiera tanto Deportes como Evento, se declararía como:

Ejemplo

```

public interface Hockey extends Sports, Event

```

Etiquetado de interfaces

El uso más común de las interfaces de extensión se produce cuando la interfaz principal no contiene ningún método. Por ejemplo, la interfaz `MouseListener` en el paquete `java.awt.event` extendió `java.util.EventListener`, que se define como -

Ejemplo

```

package java.util;
public interface EventListener
{
}

```

Una interfaz sin métodos se denomina interfaz de **etiquetado**. Hay dos propósitos de diseño básicos para etiquetar interfaces:

Crea un padre común : al igual que con la interfaz `EventListener`, que se extiende por docenas de otras interfaces en la API de Java, puede usar una interfaz de etiquetado para crear un padre común entre un grupo de interfaces. Por ejemplo, cuando una interfaz extiende `EventListener`, la JVM sabe que esta interfaz en particular se utilizará en un escenario de delegación de eventos.

Agrega un tipo de datos a una clase : esta situación es de donde proviene el término etiquetado. Una clase que implementa una interfaz de etiquetado no necesita definir ningún método (ya que la interfaz no tiene ninguno), pero la clase se convierte en un tipo de interfaz a través del polimorfismo.

Java - Paquetes

Los paquetes se utilizan en Java para evitar conflictos de nombres, controlar el acceso, facilitar la búsqueda / localización y el uso de clases, interfaces, enumeraciones y anotaciones, etc.

Un **paquete** se puede definir como una agrupación de tipos relacionados (clases, interfaces, enumeraciones y anotaciones) que proporcionan protección de acceso y gestión de espacios de nombres.

Algunos de los paquetes existentes en Java son:

- **java.lang** - agrupa las clases fundamentales
- **java.io** : las clases para las funciones de entrada y salida se incluyen en este paquete

Los programadores pueden definir sus propios paquetes para agrupar grupos de clases / interfaces, etc. Es una buena práctica agrupar clases relacionadas implementadas por usted para que un programador pueda determinar fácilmente que las clases, interfaces, enumeraciones y anotaciones están relacionadas.

Como el paquete crea un nuevo espacio de nombres, no habrá conflictos de nombres con nombres en otros paquetes. Usando paquetes, es más fácil proporcionar control de acceso y también es más fácil localizar las clases relacionadas.

Crear un paquete

Al crear un paquete, debe elegir un nombre para el paquete e incluir una declaración de **paquete** junto con ese nombre en la parte superior de cada archivo fuente que contenga las clases, interfaces, enumeraciones y tipos de anotaciones que desea incluir en el paquete.

La declaración del paquete debe ser la primera línea del archivo fuente. Solo puede haber una declaración de paquete en cada archivo fuente, y se aplica a todos los tipos en el archivo.

Si no se usa una declaración de paquete, la clase, las interfaces, las enumeraciones y los tipos de anotaciones se colocarán en el paquete predeterminado actual.

Para compilar los programas Java con instrucciones de paquete, debe usar la opción `-d` como se muestra a continuación.

```
javac -d Destination_folder file_name.java
```

Luego, se crea una carpeta con el nombre de paquete dado en el destino especificado, y los archivos de clase compilados se colocarán en esa carpeta.

Ejemplo

Veamos un ejemplo que crea un paquete llamado **animales**. Es una buena práctica usar nombres de paquetes con letras minúsculas para evitar conflictos con los nombres de clases e interfaces.

El siguiente ejemplo de paquete contiene una interfaz llamada *animales*:

```
/* File name : Animal.java */
package animals;

interface Animal {
    public void eat();
    public void travel();
}
```

Ahora, implementemos la interfaz anterior en el mismo paquete de *animales*:

```
package animals;
/* File name : MammalInt.java */

public class MammalInt implements Animal {

    public void eat() {
        System.out.println("Mammal eats");
    }

    public void travel() {
        System.out.println("Mammal travels");
    }

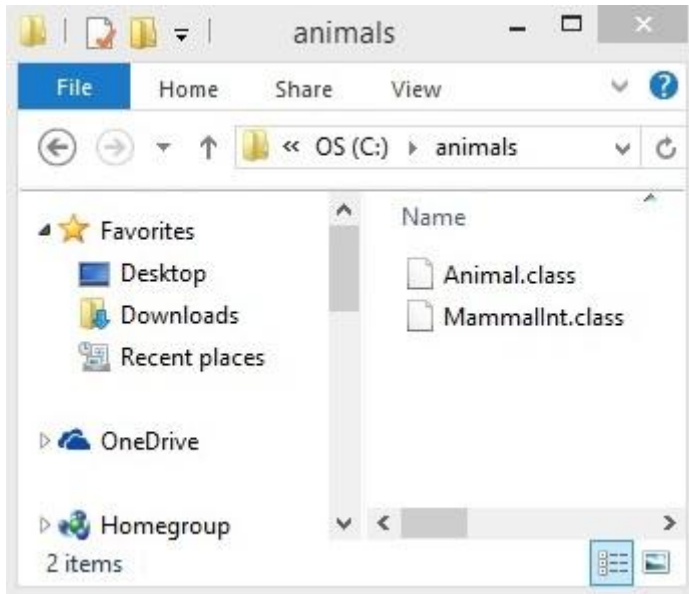
    public int noOfLegs() {
        return 0;
    }

    public static void main(String args[]) {
        MammalInt m = new MammalInt();
        m.eat();
        m.travel();
    }
}
```

Ahora compile los archivos java como se muestra a continuación:

```
$ javac -d . Animal.java
$ javac -d . MammalInt.java
```


Ahora se creará un paquete / carpeta con el nombre **animales** en el directorio actual y estos archivos de clase se colocarán en él como se muestra a continuación.



Puede ejecutar el archivo de clase dentro del paquete y obtener el resultado como se muestra a continuación.

```
Mammal eats  
Mammal travels
```

La palabra clave de importación

Si una clase quiere usar otra clase en el mismo paquete, el nombre del paquete no necesita ser usado. Las clases en el mismo paquete se encuentran sin ninguna sintaxis especial.

Ejemplo

Aquí, se agrega una clase llamada Jefe al paquete de nómina que ya contiene Empleado. El Jefe puede referirse a la clase Empleado sin usar el prefijo de nómina, como lo demuestra la siguiente clase Jefe.

```
package payroll;  
public class Boss {  
    public void payEmployee(Employee e) {  
        e.mailCheck();  
    }  
}
```

¿Qué sucede si la clase Empleado no está en el paquete de nómina? La clase Boss debe usar una de las siguientes técnicas para referirse a una clase en un paquete diferente.

- Se puede usar el nombre completo de la clase. Por ejemplo

```
payroll.Employee
```

- El paquete se puede importar utilizando la palabra clave import y el comodín (*). Por ejemplo

```
import payroll.*;
```

- La clase en sí se puede importar utilizando la palabra clave import. Por ejemplo

```
import payroll.Employee;
```

Nota : Un archivo de clase puede contener cualquier número de declaraciones de importación. Las declaraciones de importación deben aparecer después de la declaración del paquete y antes de la declaración de la clase.

La estructura de directorios de los paquetes

Dos resultados principales ocurren cuando una clase se coloca en un paquete:

- El nombre del paquete se convierte en una parte del nombre de la clase, como acabamos de comentar en la sección anterior.
- El nombre del paquete debe coincidir con la estructura del directorio donde reside el bytecode correspondiente.

Aquí hay una manera simple de administrar sus archivos en Java:

Ponga el código fuente de una clase, interfaz, enumeración o tipo de anotación en un archivo de texto cuyo nombre es el nombre simple del tipo y cuya extensión es **.java** .

Por ejemplo

```
// File Name : Car.java
package vehicle;

public class Car {
    // Class implementation.
}
```

Ahora, coloque el archivo fuente en un directorio cuyo nombre refleje el nombre del paquete al que pertenece la clase:

....\vehicle\Car.java

Ahora, el nombre de clase calificado y el nombre de ruta serían los siguientes:

- Nombre de clase → vehículo.
- Nombre de ruta → vehículo \ Car.java (en windows)

En general, una empresa utiliza su nombre de dominio de Internet invertido para los nombres de sus paquetes.

Ejemplo : el nombre de dominio de Internet de una empresa es apple.com, luego todos los nombres de sus paquetes comenzarían con com.apple. Cada componente del nombre del paquete corresponde a un subdirectorio.

Ejemplo : la compañía tenía un paquete com.apple.computers que contenía un archivo fuente Dell.java, que estaría contenido en una serie de subdirectorios como este:

```
....\com\apple\computers\Dell.java
```

En el momento de la compilación, el compilador crea un archivo de salida diferente para cada clase, interfaz y enumeración definida en él. El nombre base del archivo de salida es el nombre del tipo y su extensión es **.class**.

Por ejemplo

```
// File Name: Dell.java
package com.apple.computers;

public class Dell {
}

class Ups {
}
```

Ahora, compile este archivo de la siguiente manera usando la opción -d -

```
$javac -d . Dell.java
```

Los archivos se compilarán de la siguiente manera:

```
.\com\apple\computers\Dell.class
.\com\apple\computers\Ups.class
```

Puede importar todas las clases o interfaces definidas en `\ com \ apple \ computers` de la siguiente manera:

```
import com.apple.computers.*;
```

Al igual que los archivos fuente .java, los archivos compilados .class deben estar en una serie de directorios que reflejen el nombre del paquete. Sin embargo, la ruta a los archivos .class no tiene que ser la misma que la ruta a los archivos fuente .java. Puede organizar sus directorios fuente y de clase por separado, como:

```
<path-one>\sources\com\apple\computers\Dell.java
```

```
<path-two>\classes\com\apple\computers\Dell.class
```

Al hacer esto, es posible dar acceso al directorio de clases a otros programadores sin revelar sus fuentes. También debe administrar los archivos fuente y de clase de esta manera para que el compilador y la máquina virtual Java (JVM) puedan encontrar todos los tipos que utiliza su programa.

La ruta completa al directorio de clases, `<path-two> \ classes`, se denomina ruta de clase y se establece con la variable de sistema CLASSPATH. Tanto el compilador como la JVM construyen la ruta a sus archivos .class agregando el nombre del paquete a la ruta de la clase.

Digamos que `<path-two> \ classes` es la ruta de clase, y el nombre del paquete es `com.apple.computers`, entonces el compilador y JVM buscarán archivos .class en `<path-two> \ classes \ com \ apple \ computers`.

Una ruta de clase puede incluir varias rutas. Las rutas múltiples deben estar separadas por punto y coma (Windows) o dos puntos (Unix). Por defecto, el compilador y la JVM buscan el directorio actual y el archivo JAR que contiene

las clases de la plataforma Java para que estos directorios estén automáticamente en la ruta de clase.

Establecer variable de sistema CLASSPATH

Para mostrar la variable CLASSPATH actual, use los siguientes comandos en Windows y UNIX (shell Bourne):

- En Windows → C: \> establecer CLASSPATH
- En UNIX → % echo \$ CLASSPATH

Para eliminar el contenido actual de la variable CLASSPATH, use -

- En Windows → C: \> set CLASSPATH =
- En UNIX → % unset CLASSPATH; exportar CLASSPATH

Para establecer la variable CLASSPATH:

- En Windows → establezca CLASSPATH = C: \ users \ jack \ java \ classes
- En UNIX → % CLASSPATH = / home / jack / java / classes; exportar CLASSPATH

Java - Estructuras de datos

Las estructuras de datos proporcionadas por el paquete de utilidades Java son muy potentes y realizan una amplia gama de funciones. Estas estructuras de datos consisten en la siguiente interfaz y clases:

- Enumeración
- BitSet
- Vector
- Apilar
- Diccionario
- Tabla de picadillo
- Propiedades

Todas estas clases ahora son heredadas y Java-2 ha introducido un nuevo marco llamado Marco de colecciones, que se analiza en el próximo capítulo. -

La enumeración

La interfaz de enumeración no es en sí misma una estructura de datos, pero es muy importante dentro del contexto de otras estructuras de datos. La interfaz de enumeración define un medio para recuperar elementos sucesivos de una estructura de datos.

Por ejemplo, Enumeration define un método llamado nextElement que se usa para obtener el siguiente elemento en una estructura de datos que contiene múltiples elementos.

Para tener más detalles sobre esta interfaz, consulte La enumeración .

El conjunto de bits

La clase BitSet implementa un grupo de bits o marcas que se pueden establecer y borrar individualmente.

Esta clase es muy útil en casos en los que necesita mantenerse al día con un conjunto de valores booleanos; solo asigna un bit a cada valor y lo establece o borra según corresponda.

Para obtener más detalles sobre esta clase, consulte [The BitSet](#) .

El vector

La clase Vector es similar a una matriz Java tradicional, excepto que puede crecer según sea necesario para acomodar nuevos elementos.

Al igual que una matriz, se puede acceder a los elementos de un objeto Vector a través de un índice en el vector.

Lo bueno de usar la clase Vector es que no tiene que preocuparse por establecerla en un tamaño específico después de la creación; se encoge y crece automáticamente cuando es necesario.

Para obtener más detalles sobre esta clase, consulte [The Vector](#) .

La pila

La clase Stack implementa una pila de elementos de último en entrar, primero en salir (LIFO).

Puedes pensar en una pila literalmente como una pila vertical de objetos; cuando agrega un nuevo elemento, se apila sobre los demás.

Cuando sacas un elemento de la pila, sale de la parte superior. En otras palabras, el último elemento que agregaste a la pila es el primero en volver.

Para obtener más detalles sobre esta clase, consulte [The Stack](#) .

El diccionario

La clase Dictionary es una clase abstracta que define una estructura de datos para asignar claves a valores.

Esto es útil en los casos en que desea poder acceder a los datos a través de una clave particular en lugar de un índice entero.

Dado que la clase Dictionary es abstracta, proporciona solo el marco para una estructura de datos mapeada de claves en lugar de una implementación específica.

Para más detalles sobre esta clase, consulte [The Dictionary](#) .

La tabla hash

La clase Hashtable proporciona un medio para organizar datos basados en alguna estructura clave definida por el usuario.

Por ejemplo, en una tabla hash de la lista de direcciones, puede almacenar y ordenar datos según una clave como el código postal en lugar del nombre de una persona.

El significado específico de las claves con respecto a las tablas hash depende totalmente del uso de la tabla hash y de los datos que contiene.

Para obtener más detalles sobre esta clase, consulte [The Hashtable](#) .

Las propiedades

Properties es una subclase de Hashtable. Se utiliza para mantener listas de valores en los que la clave es una Cadena y el valor también es una Cadena.

La clase Propiedades es utilizada por muchas otras clases Java. Por ejemplo, es el tipo de objeto devuelto por `System.getProperties ()` al obtener valores ambientales.

Para obtener más detalles sobre esta clase, consulte [Las propiedades](#) .

Java - Marco de colecciones

Antes de Java 2, Java proporcionaba clases ad hoc como **Diccionario**, **Vector**, **Pila** y **Propiedades** para almacenar y manipular grupos de objetos. Aunque estas clases fueron bastante útiles, carecían de un tema central y unificador. Por lo tanto, la forma en que usó Vector era diferente de la forma en que usó Propiedades.

El marco de colecciones fue diseñado para cumplir varios objetivos, tales como:

- El marco tenía que ser de alto rendimiento. Las implementaciones para las colecciones fundamentales (matrices dinámicas, listas vinculadas, árboles y tablas hash) serían altamente eficientes.
- El marco tenía que permitir que diferentes tipos de colecciones funcionaran de manera similar y con un alto grado de interoperabilidad.
- El marco tuvo que ampliar y / o adaptar una colección fácilmente.

Con este fin, todo el marco de colecciones está diseñado en torno a un conjunto de interfaces estándar. Se proporcionan varias implementaciones estándar, como **LinkedList**, **HashSet** y **TreeSet** , de estas interfaces que puede utilizar tal cual y también puede implementar su propia colección, si lo desea.

Un marco de colecciones es una arquitectura unificada para representar y manipular colecciones. Todos los marcos de colecciones contienen lo siguiente:

- **Interfaces** : son tipos de datos abstractos que representan colecciones. Las interfaces permiten manipular las colecciones independientemente de los detalles de su representación. En lenguajes orientados a objetos, las interfaces generalmente forman una jerarquía.

- **Implementaciones, es decir, clases** : estas son las implementaciones concretas de las interfaces de recopilación. En esencia, son estructuras de datos reutilizables.
- **Algoritmos** : estos son los métodos que realizan cálculos útiles, como buscar y ordenar, en objetos que implementan interfaces de recopilación. Se dice que los algoritmos son polimórficos: es decir, se puede usar el mismo método en muchas implementaciones diferentes de la interfaz de recopilación adecuada.

Además de las colecciones, el marco define varias interfaces y clases de mapas. Los mapas almacenan pares clave / valor. Aunque los mapas no son *colecciones* en el uso adecuado del término, están totalmente integrados con las colecciones.

Las interfaces de colección

El marco de colecciones define varias interfaces. Esta sección proporciona una descripción general de cada interfaz:

No Señor.	Interfaz y descripción
1	La interfaz de la colección Esto le permite trabajar con grupos de objetos; está en la parte superior de la jerarquía de colecciones.
2	La interfaz de la lista Esto extiende la Colección y una instancia de Lista almacena una colección ordenada de elementos.
3	El conjunto Esto extiende la Colección para manejar conjuntos, que deben contener elementos únicos.
4 4	El conjunto ordenado Esto extiende Set para manejar conjuntos ordenados.
5 5	El mapa Esto asigna claves únicas a valores.
6 6	The Map.Entry Esto describe un elemento (un par clave / valor) en un mapa. Esta es una clase interna de Mapa.
7 7	El mapa ordenado Esto extiende Map para que las claves se mantengan en orden ascendente.

8	<p>La enumeración</p> <p>Esta interfaz heredada define los métodos por los cuales puede enumerar (obtener uno a la vez) los elementos en una colección de objetos. Esta interfaz heredada ha sido reemplazada por Iterator.</p>
---	---

Las clases de colección

Java proporciona un conjunto de clases de recopilación estándar que implementan interfaces de recopilación. Algunas de las clases proporcionan implementaciones completas que se pueden usar tal cual y otras son de clase abstracta, proporcionando implementaciones esqueléticas que se utilizan como puntos de partida para crear colecciones concretas.

Las clases de colección estándar se resumen en la siguiente tabla:

No Señor.	Clase y descripción
1	<p>ResumenColección</p> <p>Implementa la mayor parte de la interfaz de la Colección.</p>
2	<p>AbstractList</p> <p>Extiende AbstractCollection e implementa la mayor parte de la interfaz List.</p>
3	<p>AbstractSequentialList</p> <p>Extiende AbstractList para que lo use una colección que usa acceso secuencial en lugar de aleatorio de sus elementos.</p>
4 4	<p>Lista enlazada</p> <p>Implementa una lista vinculada al extender AbstractSequentialList.</p>
5 5	<p>Lista de arreglo</p> <p>Implementa una matriz dinámica extendiendo AbstractList.</p>
6 6	<p>AbstractSet</p> <p>Extiende AbstractCollection e implementa la mayor parte de la interfaz Set.</p>
7 7	<p>HashSet</p> <p>Extiende AbstractSet para usar con una tabla hash.</p>
8	<p>LinkedHashSet</p>

	Extiende HashSet para permitir iteraciones de orden de inserción.
9 9	TreeSet Implementa un conjunto almacenado en un árbol. Extiende AbstractSet.
10	ResumenMapa Implementa la mayor parte de la interfaz del mapa.
11	HashMap Extiende AbstractMap para usar una tabla hash.
12	TreeMap Extiende AbstractMap para usar un árbol.
13	WeakHashMap Extiende AbstractMap para usar una tabla hash con claves débiles.
14	LinkedHashMap Extiende HashMap para permitir iteraciones de orden de inserción.
15	IdentityHashMap Extiende AbstractMap y utiliza la igualdad de referencia al comparar documentos.

Las clases *AbstractCollection*, *AbstractSet*, *AbstractList*, *AbstractSequentialList* y *AbstractMap* proporcionan implementaciones esqueléticas de las interfaces de la colección principal, para minimizar el esfuerzo requerido para implementarlas.

Las siguientes clases heredadas definidas por java.util se analizaron en el capítulo anterior:

No Señor.	Clase y descripción
1	Vector Esto implementa una matriz dinámica. Es similar a ArrayList, pero con algunas diferencias.
2	Apilar Stack es una subclase de Vector que implementa una pila estándar de último en entrar, primero en salir.

3	<p>Diccionario</p> <p>Dictionary es una clase abstracta que representa un repositorio de almacenamiento de clave / valor y funciona de manera muy similar a Map.</p>
4 4	<p>Tabla de picadillo</p> <p>Hashtable era parte del java.util original y es una implementación concreta de un Diccionario.</p>
5 5	<p>Propiedades</p> <p>Properties es una subclase de Hashtable. Se utiliza para mantener listas de valores en los que la clave es una Cadena y el valor también es una Cadena.</p>
6 6	<p>BitSet</p> <p>Una clase BitSet crea un tipo especial de matriz que contiene valores de bit. Esta matriz puede aumentar de tamaño según sea necesario.</p>

Los algoritmos de colección

El marco de colecciones define varios algoritmos que se pueden aplicar a colecciones y mapas. Estos algoritmos se definen como métodos estáticos dentro de la clase Colecciones.

Varios de los métodos pueden arrojar una **ClassCastException**, que ocurre cuando se intenta comparar tipos incompatibles, o una **UnsupportedOperationException**, que ocurre cuando se intenta modificar una colección no modificable.

Las colecciones definen tres variables estáticas: EMPTY_SET, EMPTY_LIST y EMPTY_MAP. Todos son inmutables.

No Señor.	Algoritmo y Descripción
1	<p>Los algoritmos de colección</p> <p>Aquí hay una lista de toda la implementación del algoritmo.</p>

¿Cómo usar un iterador?

A menudo, querrá recorrer los elementos de una colección. Por ejemplo, es posible que desee mostrar cada elemento.

La manera más fácil de hacer esto es emplear un iterador, que es un objeto que implementa la interfaz Iterator o ListIterator.

Iterator le permite recorrer una colección, obtener o eliminar elementos. ListIterator extiende Iterator para permitir el recorrido bidireccional de una lista y la modificación de elementos.

No Señor.	Método y descripción del iterador
1	Usando Java Iterator Aquí hay una lista de todos los métodos con ejemplos proporcionados por las interfaces Iterator y ListIterator.

¿Cómo usar un comparador?

Tanto TreeSet como TreeMap almacenan elementos en un orden ordenado. Sin embargo, es el comparador el que define con precisión lo que significa el *orden ordenado*.

Esta interfaz nos permite ordenar una colección dada de diferentes maneras. Además, esta interfaz se puede utilizar para ordenar cualquier instancia de cualquier clase (incluso las clases que no podemos modificar).

No Señor.	Método y descripción del iterador
1	Usando Java Comparator Aquí hay una lista de todos los métodos con ejemplos proporcionados por Comparator Interface.

Resumen

El marco de colecciones de Java le da al programador acceso a estructuras de datos preempaquetadas, así como a algoritmos para manipularlas.

Una colección es un objeto que puede contener referencias a otros objetos. Las interfaces de recopilación declaran las operaciones que se pueden realizar en cada tipo de recopilación.

Las clases e interfaces del marco de colecciones están en el paquete java.util.

Java - Genéricos

Sería bueno si pudiéramos escribir un único método de ordenación que pudiera ordenar los elementos en una matriz Integer, una matriz String o una matriz de cualquier tipo que admita el orden.

Los métodos **genéricos** de Java y las clases genéricas permiten a los programadores especificar, con una sola declaración de método, un conjunto de métodos relacionados, o con una sola declaración de clase, un conjunto de tipos relacionados, respectivamente.

Los genéricos también proporcionan seguridad de tipo en tiempo de compilación que permite a los programadores capturar tipos no válidos en tiempo de compilación.

Usando el concepto genérico de Java, podríamos escribir un método genérico para ordenar una matriz de objetos, luego invocar el método genérico con matrices enteras, matrices dobles, matrices de cadenas, etc., para clasificar los elementos de la matriz.

Métodos genéricos

Puede escribir una única declaración de método genérico que se pueda invocar con argumentos de diferentes tipos. Según los tipos de argumentos pasados al método genérico, el compilador maneja cada llamada al método de manera apropiada. Las siguientes son las reglas para definir los métodos genéricos:

- Todas las declaraciones de métodos genéricos tienen una sección de parámetros de tipo delimitada por corchetes angulares (<y>) que precede al tipo de retorno del método (<E> en el siguiente ejemplo).
- Cada sección de parámetros de tipo contiene uno o más parámetros de tipo separados por comas. Un parámetro de tipo, también conocido como variable de tipo, es un identificador que especifica un nombre de tipo genérico.
- Los parámetros de tipo se pueden usar para declarar el tipo de retorno y actuar como marcadores de posición para los tipos de argumentos pasados al método genérico, que se conocen como argumentos de tipo real.
- El cuerpo de un método genérico se declara como el de cualquier otro método. Tenga en cuenta que los parámetros de tipo solo pueden representar tipos de referencia, no tipos primitivos (como int, double y char).

Ejemplo

El siguiente ejemplo ilustra cómo podemos imprimir una matriz de diferentes tipos usando un único método genérico:

```
public class GenericMethodTest {
    // generic method printArray
    public static < E > void printArray( E[] inputArray ) {
        // Display array elements
        for(E element : inputArray) {
            System.out.printf("%s ", element);
        }
        System.out.println();
    }

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

```

// Create arrays of Integer, Double and Character
Integer[] intArray = { 1, 2, 3, 4, 5 };
Double[] doubleArray = { 1.1, 2.2, 3.3, 4.4 };
Character[] charArray = { 'H', 'E', 'L', 'L', 'O' };

System.out.println("Array integerArray contains:");
printArray(intArray);    // pass an Integer array

System.out.println("\nArray doubleArray contains:");
printArray(doubleArray); // pass a Double array

System.out.println("\nArray characterArray contains:");
printArray(charArray);   // pass a Character array
    }
}

```

Esto producirá el siguiente resultado:

Salida

```

Array integerArray contains:
1 2 3 4 5

```

```

Array doubleArray contains:
1.1 2.2 3.3 4.4

```

```

Array characterArray contains:
H E L L O

```

Parámetros de tipo acotado

Puede haber ocasiones en las que desee restringir los tipos de tipos que se pueden pasar a un parámetro de tipo. Por ejemplo, un método que funciona con números solo puede aceptar instancias de Number o sus subclases. Para eso están los parámetros de tipo acotado.

Para declarar un parámetro de tipo acotado, enumere el nombre del parámetro de tipo, seguido de la palabra clave `extends`, seguido de su límite superior.

Ejemplo

El siguiente ejemplo ilustra cómo las extensiones se usan en un sentido general para significar "extensiones" (como en las clases) o "implementos" (como en las interfaces). Este ejemplo es un método genérico para devolver el mayor de tres objetos comparables:

```

public class MaximumTest {
    // determines the largest of three Comparable objects

    public static <T extends Comparable<T>> T maximum(T x, T
y, T z) {

```

```

    T max = x;    // assume x is initially the largest

    if(y.compareTo(max) > 0) {
        max = y;    // y is the largest so far
    }

    if(z.compareTo(max) > 0) {
        max = z;    // z is the largest now
    }
    return max;    // returns the largest object
}

public static void main(String args[]) {
    System.out.printf("Max of %d, %d and %d is %d\n\n",
        3, 4, 5, maximum( 3, 4, 5 ));

    System.out.printf("Max of %.1f,%.1f and %.1f is
%.1f\n\n",
        6.6, 8.8, 7.7, maximum( 6.6, 8.8, 7.7 ));

    System.out.printf("Max of %s, %s and %s is
%s\n", "pear",
        "apple", "orange", maximum("pear", "apple",
"orange"));
}
}

```

Esto producirá el siguiente resultado:

Salida

Max of 3, 4 and 5 is 5

Max of 6.6,8.8 and 7.7 is 8.8

Max of pear, apple and orange is pear

Clases Genéricas

Una declaración de clase genérica se parece a una declaración de clase no genérica, excepto que el nombre de la clase es seguido por una sección de parámetro de tipo.

Al igual que con los métodos genéricos, la sección de parámetros de tipo de una clase genérica puede tener uno o más parámetros de tipo separados por comas. Estas clases se conocen como clases parametrizadas o tipos parametrizados porque aceptan uno o más parámetros.

Ejemplo

El siguiente ejemplo ilustra cómo podemos definir una clase genérica:

```

public class Box<T> {
    private T t;

    public void add(T t) {
        this.t = t;
    }

    public T get() {
        return t;
    }

    public static void main(String[] args) {
        Box<Integer> integerBox = new Box<Integer>();
        Box<String> stringBox = new Box<String>();

        integerBox.add(new Integer(10));
        stringBox.add(new String("Hello World"));

        System.out.printf("Integer Value :%d\n\n",
integerBox.get());
        System.out.printf("String Value :%s\n",
stringBox.get());
    }
}

```

Esto producirá el siguiente resultado:

Salida

```

Integer Value :10
String Value :Hello World

```

Java - Serialización

Java proporciona un mecanismo, llamado serialización de objetos, donde un objeto puede representarse como una secuencia de bytes que incluye los datos del objeto, así como información sobre el tipo de objeto y los tipos de datos almacenados en el objeto.

Después de que un objeto serializado se haya escrito en un archivo, se puede leer del archivo y deserializarlo, es decir, la información de tipo y los bytes que representan el objeto y sus datos se pueden usar para recrear el objeto en la memoria.

Lo más impresionante es que todo el proceso es independiente de JVM, lo que significa que un objeto se puede serializar en una plataforma y deserializar en una plataforma completamente diferente.

Las clases **ObjectInputStream** y **ObjectOutputStream** son **flujos de** alto nivel que contienen los métodos para serializar y deserializar un objeto.

La clase **ObjectOutputStream** contiene muchos métodos de escritura para escribir varios tipos de datos, pero un método en particular se destaca:

```
public final void writeObject(Object x) throws IOException
```

El método anterior serializa un objeto y lo envía a la secuencia de salida. Del mismo modo, la clase `ObjectInputStream` contiene el siguiente método para deserializar un objeto:

```
public final Object readObject() throws IOException,  
ClassNotFoundException
```

Este método recupera el siguiente objeto fuera de la secuencia y lo deserializa. El valor de retorno es `Object`, por lo que deberá convertirlo a su tipo de datos apropiado.

Para demostrar cómo funciona la serialización en Java, voy a usar la clase `Employee` que discutimos al principio del libro. Supongamos que tenemos la siguiente clase `Employee`, que implementa la interfaz `Serializable`:

Ejemplo

```
public class Employee implements java.io.Serializable {  
    public String name;  
    public String address;  
    public transient int SSN;  
    public int number;  
  
    public void mailCheck() {  
        System.out.println("Mailing a check to " + name + " " +  
address);  
    }  
}
```

Tenga en cuenta que para que una clase se serialice correctamente, se deben cumplir dos condiciones:

- La clase debe implementar la interfaz `java.io.Serializable`.
- Todos los campos de la clase deben ser serializables. Si un campo no es serializable, debe marcarse como **transitorio**.

Si tiene curiosidad por saber si una clase estándar de Java es serializable o no, consulte la documentación de la clase. La prueba es simple: si la clase implementa `java.io.Serializable`, entonces es serializable; de lo contrario, no lo es.

Serializar un objeto

La clase `ObjectOutputStream` se usa para serializar un objeto. El siguiente programa `SerializeDemo` crea una instancia de un objeto `Employee` y lo serializa en un archivo.

Cuando el programa termina de ejecutarse, se crea un archivo llamado `employee.ser`. El programa no genera ningún resultado, pero estudia el código e intenta determinar qué está haciendo el programa.

Nota : Al serializar un objeto en un archivo, la convención estándar en Java es darle al archivo una extensión **.ser** .

Ejemplo

```
import java.io.*;
public class SerializeDemo {

    public static void main(String [] args) {
        Employee e = new Employee();
        e.name = "Reyan Ali";
        e.address = "Phokka Kuan, Ambehta Peer";
        e.SSN = 11122333;
        e.number = 101;

        try {
            FileOutputStream fileOut =
                new FileOutputStream("/tmp/employee.ser");
            ObjectOutputStream out = new
ObjectOutputStream(fileOut);
            out.writeObject(e);
            out.close();
            fileOut.close();
            System.out.printf("Serialized data is saved in
/tmp/employee.ser");
        } catch (IOException i) {
            i.printStackTrace();
        }
    }
}
```

Deserializar un objeto

El siguiente programa DeserializeDemo deserializa el objeto Empleado creado en el programa SerializeDemo. Estudie el programa e intente determinar su salida.

Ejemplo

```
import java.io.*;
public class DeserializeDemo {

    public static void main(String [] args) {
        Employee e = null;
        try {
            FileInputStream fileIn = new
FileInputStream("/tmp/employee.ser");
            ObjectInputStream in = new
ObjectInputStream(fileIn);
            e = (Employee) in.readObject();
            in.close();
        }
```

```

        fileIn.close();
    } catch (IOException i) {
        i.printStackTrace();
        return;
    } catch (ClassNotFoundException c) {
        System.out.println("Employee class not found");
        c.printStackTrace();
        return;
    }

    System.out.println("Deserialized Employee...");
    System.out.println("Name: " + e.name);
    System.out.println("Address: " + e.address);
    System.out.println("SSN: " + e.SSN);
    System.out.println("Number: " + e.number);
}
}

```

Esto producirá el siguiente resultado:

Salida

```

Deserialized Employee...
Name: Reyan Ali
Address:Phokka Kuan, Ambehta Peer
SSN: 0
Number:101

```

Aquí están los siguientes puntos importantes a tener en cuenta:

- El bloque try / catch intenta atrapar una `ClassNotFoundException`, que es declarada por el método `readObject ()`. Para que una JVM pueda deserializar un objeto, debe poder encontrar el código de bytes para la clase. Si la JVM no puede encontrar una clase durante la deserialización de un objeto, arroja una `ClassNotFoundException`.
- Observe que el valor de retorno de `readObject ()` se convierte en una referencia de empleado.
- El valor del campo SSN era 11122333 cuando se serializó el objeto, pero debido a que el campo es transitorio, este valor no se envió a la secuencia de salida. El campo SSN del objeto empleado deserializado es 0.

Java - Redes

El término *programación de red* se refiere a escribir programas que se ejecutan a través de múltiples dispositivos (computadoras), en los cuales todos los dispositivos están conectados entre sí mediante una red.

El paquete `java.net` de las API J2SE contiene una colección de clases e interfaces que proporcionan los detalles de comunicación de bajo nivel, lo que le permite escribir programas que se centran en resolver el problema en cuestión.

El paquete `java.net` proporciona soporte para los dos protocolos de red comunes:

- **TCP** : TCP significa Protocolo de control de transmisión, que permite una comunicación confiable entre dos aplicaciones. TCP generalmente se usa sobre el Protocolo de Internet, que se conoce como TCP / IP.
- **UDP** : UDP significa User Datagram Protocol, un protocolo sin conexión que permite la transmisión de paquetes de datos entre aplicaciones.

Este capítulo ofrece una buena comprensión de los siguientes dos temas:

- **Programación de sockets** : este es el concepto más utilizado en redes y se ha explicado con mucho detalle.
- **Procesamiento de URL** : esto se cubriría por separado. Haga clic aquí para obtener información sobre el procesamiento de URL en lenguaje Java.

Programación de socket

Los sockets proporcionan el mecanismo de comunicación entre dos computadoras que utilizan TCP. Un programa cliente crea un socket en su extremo de la comunicación e intenta conectar ese socket a un servidor.

Cuando se realiza la conexión, el servidor crea un objeto de socket en su extremo de la comunicación. El cliente y el servidor ahora pueden comunicarse escribiendo y leyendo desde el socket.

La clase `java.net.Socket` representa un socket y la clase `java.net.ServerSocket` proporciona un mecanismo para que el programa del servidor escuche a los clientes y establezca conexiones con ellos.

Los siguientes pasos ocurren cuando se establece una conexión TCP entre dos computadoras usando sockets:

- El servidor crea una instancia de un objeto `ServerSocket`, que indica en qué comunicación de número de puerto se producirá.
- El servidor invoca el método `accept ()` de la clase `ServerSocket`. Este método espera hasta que un cliente se conecta al servidor en el puerto dado.
- Después de que el servidor está esperando, un cliente crea una instancia de un objeto `Socket`, especificando el nombre del servidor y el número de puerto para conectarse.
- El constructor de la clase `Socket` intenta conectar al cliente con el servidor especificado y el número de puerto. Si se establece la comunicación, el cliente ahora tiene un objeto `Socket` capaz de comunicarse con el servidor.
- En el lado del servidor, el método `accept ()` devuelve una referencia a un nuevo socket en el servidor que está conectado al socket del cliente.

Una vez que se establecen las conexiones, la comunicación puede ocurrir usando flujos de E / S. Cada socket tiene un `OutputStream` y un `InputStream`. `OutputStream` del cliente está conectado a `InputStream` del servidor, y `InputStream` del cliente está conectado a `OutputStream` del servidor.

TCP es un protocolo de comunicación bidireccional, por lo tanto, los datos pueden enviarse a través de ambos flujos al mismo tiempo. Las siguientes son las clases útiles que proporcionan un conjunto completo de métodos para implementar sockets.

Métodos de clase ServerSocket

Las aplicaciones del servidor utilizan la clase **java.net.ServerSocket** para obtener un puerto y escuchar las solicitudes de los clientes.

La clase ServerSocket tiene cuatro constructores:

No Señor.	Método y descripción
1	ServerSocket público (puerto int) arroja IOException Intenta crear un socket de servidor vinculado al puerto especificado. Se produce una excepción si el puerto ya está vinculado por otra aplicación.
2	Public ServerSocket (int port, int backlog) lanza IOException De manera similar al constructor anterior, el parámetro de trabajo atrasado especifica cuántos clientes entrantes almacenar en una cola de espera.
3	ServerSocket público (int port, int backlog, InetAddress address) arroja IOException Similar al constructor anterior, el parámetro InetAddress especifica la dirección IP local a la que se debe vincular. InetAddress se usa para servidores que pueden tener múltiples direcciones IP, lo que permite al servidor especificar en qué direcciones IP aceptar solicitudes de clientes.
4 4	public ServerSocket () arroja IOException Crea un socket de servidor independiente. Cuando use este constructor, use el método bind () cuando esté listo para enlazar el socket del servidor.

Si el constructor ServerSocket no produce una excepción, significa que su aplicación se ha vinculado correctamente al puerto especificado y está lista para las solicitudes del cliente.

Los siguientes son algunos de los métodos comunes de la clase ServerSocket:

No Señor.	Método y descripción
1	public int getLocalPort () Devuelve el puerto en el que escucha el socket del servidor. Este método es útil si pasó 0 como número de puerto en un constructor y deja que el servidor encuentre un puerto para usted.

2	Socket público accept () arroja IOException Espera a un cliente entrante. Este método se bloquea hasta que un cliente se conecta al servidor en el puerto especificado o se agota el tiempo de espera del socket, suponiendo que el valor de tiempo de espera se haya establecido utilizando el método <code>setSoTimeout ()</code> . De lo contrario, este método se bloquea indefinidamente.
3	public void setSoTimeout (int timeout) Establece el valor de tiempo de espera para el tiempo que el socket del servidor espera a un cliente durante el <code>accept ()</code> .
4 4	enlace público vacío (SocketAddress host, int backlog) Vincula el socket al servidor y puerto especificados en el objeto <code>SocketAddress</code> . Use este método si ha instanciado el <code>ServerSocket</code> usando el constructor sin argumentos.

Cuando `ServerSocket` invoca `accept ()`, el método no regresa hasta que se conecta un cliente. Después de que un cliente se conecta, `ServerSocket` crea un nuevo `Socket` en un puerto no especificado y devuelve una referencia a este nuevo `Socket`. Ahora existe una conexión TCP entre el cliente y el servidor, y puede comenzar la comunicación.

Métodos de clase de socket

La clase **java.net.Socket** representa el socket que tanto el cliente como el servidor usan para comunicarse entre sí. El cliente obtiene un objeto `Socket` instanciando uno, mientras que el servidor obtiene un objeto `Socket` a partir del valor de retorno del método `accept ()`.

La clase `Socket` tiene cinco constructores que un cliente usa para conectarse a un servidor:

No Señor.	Método y descripción
1	Public Socket (String host, int port) lanza UnknownHostException, IOException. Este método intenta conectarse al servidor especificado en el puerto especificado. Si este constructor no produce una excepción, la conexión es exitosa y el cliente está conectado al servidor.
2	Public Socket (host InetAddress, puerto int) lanza IOException Este método es idéntico al constructor anterior, excepto que el host se denota por un objeto <code>InetAddress</code> .

3	Socket público (String host, int port, InetAddress localAddress, int localPort) genera IOException. Se conecta al host y puerto especificados, creando un socket en el host local en la dirección y puerto especificados.
4 4	Public Socket (host InetAddress, puerto int, InetAddress localAddress, int localPort) arroja IOException. Este método es idéntico al constructor anterior, excepto que el host se denota mediante un objeto InetAddress en lugar de una Cadena.
5 5	Enchufe público () Crea un zócalo desconectado. Use el método connect () para conectar este socket a un servidor.

Cuando el constructor Socket regresa, no simplemente crea una instancia de un objeto Socket sino que en realidad intenta conectarse al servidor y puerto especificados.

Aquí se enumeran algunos métodos de interés en la clase Socket. Tenga en cuenta que tanto el cliente como el servidor tienen un objeto Socket, por lo que el cliente y el servidor pueden invocar estos métodos.

No Señor.	Método y descripción
1	public void connect (host SocketAddress, int timeout) arroja IOException Este método conecta el zócalo al host especificado. Este método solo es necesario cuando crea una instancia del Socket utilizando el constructor sin argumentos.
2	public InetAddress getInetAddress () Este método devuelve la dirección de la otra computadora a la que está conectado este socket.
3	public int getPort () Devuelve el puerto al que está vinculado el socket en la máquina remota.
4 4	public int getLocalPort () Devuelve el puerto al que está vinculado el socket en la máquina local.
5 5	public SocketAddress getRemoteSocketAddress ()

	Devuelve la dirección del zócalo remoto.
6 6	público InputStream getInputStream () lanza IOException Devuelve el flujo de entrada del socket. El flujo de entrada está conectado al flujo de salida del zócalo remoto.
7 7	public OutputStream getOutputStream () arroja IOException Devuelve el flujo de salida del socket. La secuencia de salida está conectada a la secuencia de entrada del zócalo remoto.
8	public void close () arroja IOException Cierra el socket, lo que hace que este objeto Socket ya no sea capaz de conectarse nuevamente a ningún servidor.

Métodos de clase InetAddress

Esta clase representa una dirección de Protocolo de Internet (IP). Estos son los siguientes métodos útiles que necesitaría al hacer la programación de sockets:

No Señor.	Método y descripción
1	InetAddress estática getByAddress (byte [] addr) Devuelve un objeto InetAddress dada la dirección IP sin formato.
2	InetAddress estática getByAddress (host de cadena, byte [] addr) Crea una dirección Inet basada en el nombre de host y la dirección IP proporcionados.
3	InetAddress estática getName (host de cadena) Determina la dirección IP de un host, dado el nombre del host.
4 4	String getHostAddress () Devuelve la cadena de la dirección IP en una presentación textual.
5 5	String getHostName () Obtiene el nombre de host para esta dirección IP.

6 6	InetAddress estática InetAddress getLocalHost () Devuelve el host local.
7 7	String toString () Convierte esta dirección IP en una cadena.

Ejemplo de Socket Client

El siguiente GreetingClient es un programa cliente que se conecta a un servidor mediante un socket y envía un saludo, y luego espera una respuesta.

Ejemplo

```
// File Name GreetingClient.java
import java.net.*;
import java.io.*;

public class GreetingClient {

    public static void main(String [] args) {
        String serverName = args[0];
        int port = Integer.parseInt(args[1]);
        try {
            System.out.println("Connecting to " + serverName + "
on port " + port);
            Socket client = new Socket(serverName, port);

            System.out.println("Just connected to " +
client.getRemoteSocketAddress());
            OutputStream outToServer = client.getOutputStream();
            DataOutputStream out = new
DataOutputStream(outToServer);

            out.writeUTF("Hello from " +
client.getLocalSocketAddress());
            InputStream inFromServer = client.getInputStream();
            DataInputStream in = new
DataInputStream(inFromServer);

            System.out.println("Server says " + in.readUTF());
            client.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```


Ejemplo de servidor de sockets

El siguiente programa GreetingServer es un ejemplo de una aplicación de servidor que usa la clase Socket para escuchar a los clientes en un número de puerto especificado por un argumento de línea de comandos:

Ejemplo

```
// File Name GreetingServer.java
import java.net.*;
import java.io.*;

public class GreetingServer extends Thread {
    private ServerSocket serverSocket;

    public GreetingServer(int port) throws IOException {
        serverSocket = new ServerSocket(port);
        serverSocket.setSoTimeout(10000);
    }

    public void run() {
        while(true) {
            try {
                System.out.println("Waiting for client on port "
+
                serverSocket.getLocalPort() + "...");
                Socket server = serverSocket.accept();

                System.out.println("Just connected to " +
server.getRemoteSocketAddress());
                DataInputStream in = new
DataInputStream(server.getInputStream());

                System.out.println(in.readUTF());
                DataOutputStream out = new
DataOutputStream(server.getOutputStream());
                out.writeUTF("Thank you for connecting to " +
server.getLocalSocketAddress()
+ "\nGoodbye!");
                server.close();

            } catch (SocketTimeoutException s) {
                System.out.println("Socket timed out!");
                break;
            } catch (IOException e) {
                e.printStackTrace();
                break;
            }
        }
    }

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

```
int port = Integer.parseInt(args[0]);
try {
    Thread t = new GreetingServer(port);
    t.start();
} catch (IOException e) {
    e.printStackTrace();
}
}
```

Compile el cliente y el servidor y luego inicie el servidor de la siguiente manera:

```
$ java GreetingServer 6066
Waiting for client on port 6066...
```

Verifique el programa cliente de la siguiente manera:

Salida

```
$ java GreetingClient localhost 6066
Connecting to localhost on port 6066
Just connected to localhost/127.0.0.1:6066
Server says Thank you for connecting to /127.0.0.1:6066
Goodbye!
```

Java - Envío de correo electrónico

Enviar un correo electrónico usando su aplicación Java es bastante simple, pero para comenzar debe tener **JavaMail API** y **Java Activation Framework (JAF)** instalados en su máquina.

- Puede descargar la última versión de JavaMail (Versión 1.2) del sitio web estándar de Java.
- Puede descargar la última versión de JAF (Versión 1.1.1) del sitio web estándar de Java.

Descargue y descomprima estos archivos, en los directorios de nivel superior recién creados encontrará una cantidad de archivos jar para ambas aplicaciones. Es necesario añadir **mail.jar** y **activation.jar** archivos en su CLASSPATH.

Enviar un correo electrónico simple

Aquí hay un ejemplo para enviar un correo electrónico simple desde su máquina. Se supone que su **localhost** está conectado a Internet y es lo suficientemente capaz de enviar un correo electrónico.

Ejemplo

```
// File Name SendEmail.java

import java.util.*;
```

```

import javax.mail.*;
import javax.mail.internet.*;
import javax.activation.*;

public class SendEmail {

    public static void main(String [] args) {
        // Recipient's email ID needs to be mentioned.
        String to = "abcd@gmail.com";

        // Sender's email ID needs to be mentioned
        String from = "web@gmail.com";

        // Assuming you are sending email from localhost
        String host = "localhost";

        // Get system properties
        Properties properties = System.getProperties();

        // Setup mail server
        properties.setProperty("mail.smtp.host", host);

        // Get the default Session object.
        Session session =
        Session.getDefaultInstance(properties);

        try {
            // Create a default MimeMessage object.
            MimeMessage message = new MimeMessage(session);

            // Set From: header field of the header.
            message.setFrom(new InternetAddress(from));

            // Set To: header field of the header.
            message.addRecipient(Message.RecipientType.TO, new
            InternetAddress(to));

            // Set Subject: header field
            message.setSubject("This is the Subject Line!");

            // Now set the actual message
            message.setText("This is actual message");

            // Send message
            Transport.send(message);
            System.out.println("Sent message successfully....");
        } catch (MessagingException mex) {
            mex.printStackTrace();
        }
    }
}

```

Compile y ejecute este programa para enviar un correo electrónico simple:

Salida

```
$ java SendEmail  
Sent message successfully....
```

Si desea enviar un correo electrónico a múltiples destinatarios, los siguientes métodos se utilizarían para especificar múltiples ID de correo electrónico:

```
void addRecipients(Message.RecipientType type, Address[]  
addresses)  
    throws MessagingException
```

Aquí está la descripción de los parámetros:

- **tipo** : se establecería en TO, CC o BCC. Aquí CC representa Carbon Copy y BCC representa Black Carbon Copy. Ejemplo: *Message.RecipientType.TO*
- **direcciones** : esta es una matriz de ID de correo electrónico. Debería usar el método `InternetAddress ()` al especificar ID de correo electrónico.

Enviar un correo electrónico HTML

Aquí hay un ejemplo para enviar un correo electrónico HTML desde su máquina. Aquí se supone que su **host local** está conectado a Internet y es lo suficientemente capaz de enviar un correo electrónico.

Este ejemplo es muy similar al anterior, excepto que aquí estamos usando el método `setContent ()` para establecer contenido cuyo segundo argumento es "text / html" para especificar que el contenido HTML está incluido en el mensaje.

Con este ejemplo, puede enviar contenido tan grande como HTML que desee.

Ejemplo

```
// File Name SendHTMLEmail.java  
  
import java.util.*;  
import javax.mail.*;  
import javax.mail.internet.*;  
import javax.activation.*;  
  
public class SendHTMLEmail {  
  
    public static void main(String [] args) {  
        // Recipient's email ID needs to be mentioned.  
        String to = "abcd@gmail.com";  
  
        // Sender's email ID needs to be mentioned  
        String from = "web@gmail.com";  
  
        // Assuming you are sending email from localhost  
        String host = "localhost";  
  
        // Get system properties
```

```

        Properties properties = System.getProperties();

        // Setup mail server
        properties.setProperty("mail.smtp.host", host);

        // Get the default Session object.
        Session session =
        Session.getDefaultInstance(properties);

        try {
            // Create a default MimeMessage object.
            MimeMessage message = new MimeMessage(session);

            // Set From: header field of the header.
            message.setFrom(new InternetAddress(from));

            // Set To: header field of the header.
            message.addRecipient(Message.RecipientType.TO, new
            InternetAddress(to));

            // Set Subject: header field
            message.setSubject("This is the Subject Line!");

            // Send the actual HTML message, as big as you like
            message.setContent("<h1>This is actual
            message</h1>", "text/html");

            // Send message
            Transport.send(message);
            System.out.println("Sent message successfully....");
        } catch (MessagingException mex) {
            mex.printStackTrace();
        }
    }
}

```

Compile y ejecute este programa para enviar un correo electrónico HTML:

Salida

```

$ java SendHTMLEmail
Sent message successfully....

```

Enviar adjunto en correo electrónico

Aquí hay un ejemplo para enviar un correo electrónico con datos adjuntos desde su máquina. Aquí se supone que su **host local** está conectado a Internet y es capaz de enviar un correo electrónico.

Ejemplo

```

// File Name SendFileEmail.java

```

```
import java.util.*;
import javax.mail.*;
import javax.mail.internet.*;
import javax.activation.*;

public class SendFileEmail {

    public static void main(String [] args) {
        // Recipient's email ID needs to be mentioned.
        String to = "abcd@gmail.com";

        // Sender's email ID needs to be mentioned
        String from = "web@gmail.com";

        // Assuming you are sending email from localhost
        String host = "localhost";

        // Get system properties
        Properties properties = System.getProperties();

        // Setup mail server
        properties.setProperty("mail.smtp.host", host);

        // Get the default Session object.
        Session session =
        Session.getDefaultInstance(properties);

        try {
            // Create a default MimeMessage object.
            MimeMessage message = new MimeMessage(session);

            // Set From: header field of the header.
            message.setFrom(new InternetAddress(from));

            // Set To: header field of the header.
            message.addRecipient(Message.RecipientType.TO, new
            InternetAddress(to));

            // Set Subject: header field
            message.setSubject("This is the Subject Line!");

            // Create the message part
            BodyPart messageBodyPart = new MimeBodyPart();

            // Fill the message
            messageBodyPart.setText("This is message body");

            // Create a multipart message
            Multipart multipart = new MimeMultipart();

            // Set text message part
            multipart.addBodyPart(messageBodyPart);
```

```

        // Part two is attachment
        messageBodyPart = new MimeBodyPart();
        String filename = "file.txt";
        DataSource source = new FileDataSource(filename);
        messageBodyPart.setDataHandler(new
DataHandler(source));
        messageBodyPart.setFileName(filename);
        multipart.addBodyPart(messageBodyPart);

        // Send the complete message parts
        message.setContent(multipart );

        // Send message
        Transport.send(message);
        System.out.println("Sent message successfully....");
    } catch (MessagingException mex) {
        mex.printStackTrace();
    }
}
}

```

Compile y ejecute este programa para enviar un correo electrónico HTML:

Salida

```

$ java SendFileEmail
Sent message successfully....

```

Parte de autenticación de usuario

Si es necesario proporcionar una ID de usuario y una contraseña al servidor de correo electrónico para fines de autenticación, puede establecer estas propiedades de la siguiente manera:

```

props.setProperty("mail.user", "myuser");
props.setProperty("mail.password", "mypwd");

```

El resto del mecanismo de envío de correo electrónico permanecería como se explicó anteriormente.

Java - Multithreading

Java es un *lenguaje de programación multiproceso*, lo que significa que podemos desarrollar un programa multiproceso utilizando Java. Un programa multiproceso contiene dos o más partes que pueden ejecutarse simultáneamente y cada parte puede manejar una tarea diferente al mismo tiempo haciendo un uso óptimo de los recursos disponibles, especialmente cuando su computadora tiene múltiples CPU.

Por definición, la multitarea es cuando varios procesos comparten recursos de procesamiento comunes, como una CPU. Multi-threading amplía la idea de la multitarea en aplicaciones donde puede subdividir operaciones específicas

dentro de una sola aplicación en subprocesos individuales. Cada uno de los hilos puede ejecutarse en paralelo. El sistema operativo divide el tiempo de procesamiento no solo entre diferentes aplicaciones, sino también entre cada subproceso dentro de una aplicación.

Multi-threading le permite escribir de una manera en la que múltiples actividades pueden realizarse simultáneamente en el mismo programa.

Ciclo de vida de un hilo

Un hilo pasa por varias etapas en su ciclo de vida. Por ejemplo, un hilo nace, comienza, se ejecuta y luego muere. El siguiente diagrama muestra el ciclo de vida completo de un hilo.

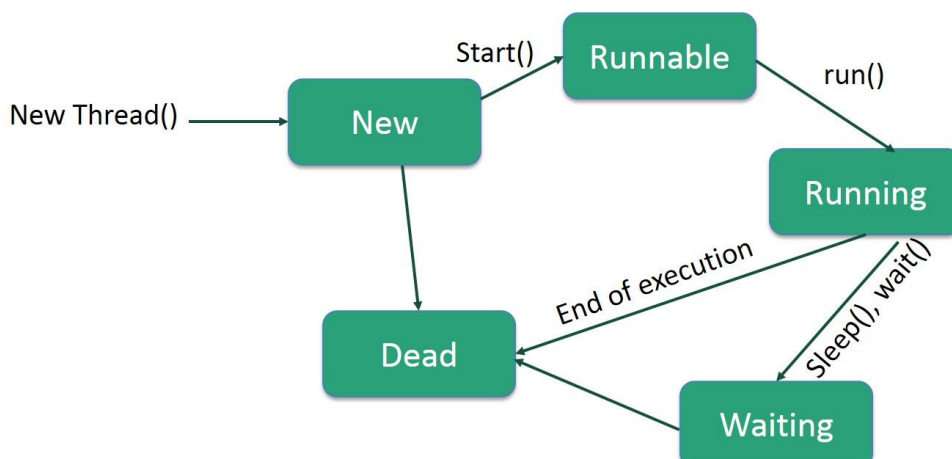
Las siguientes son las etapas del ciclo de vida:

- **Nuevo** : un nuevo hilo comienza su ciclo de vida en el nuevo estado. Permanece en este estado hasta que el programa inicia el hilo. También se conoce como un **hilo nacido** .
- **Ejecutable** : después de que se inicia un subproceso recién nacido, el subproceso se vuelve ejecutable. Se considera que un subproceso en este estado está ejecutando su tarea.
- **En espera** : a veces, un subproceso pasa al estado de espera mientras el subproceso espera a que otro subproceso realice una tarea. Un subproceso vuelve al estado ejecutable solo cuando otro subproceso le indica al subproceso en espera que continúe ejecutándose.
- **Espera programada** : un subproceso ejecutable puede ingresar al estado de espera programada durante un intervalo de tiempo especificado. Un subproceso en este estado vuelve al estado ejecutable cuando expira ese intervalo de tiempo o cuando ocurre el evento que está esperando.
- **Terminado (muerto)** : un subproceso ejecutable ingresa al estado terminado cuando completa su tarea o termina de otra manera.

Prioridades del hilo

Cada subproceso Java tiene una prioridad que ayuda al sistema operativo a determinar el orden en que se programan los subprocesos.

Las prioridades de subprocesos de Java están en el rango entre MIN_PRIORITY (una constante de 1) y MAX_PRIORITY (una constante de 10). Por defecto, cada hilo tiene prioridad NORM_PRIORITY (una constante de 5).



Los subprocesos con mayor prioridad son más importantes para un programa y se les debe asignar tiempo de procesador antes que los subprocesos de menor prioridad. Sin embargo, las prioridades de subprocesos no pueden garantizar el orden en que se ejecutan los subprocesos y dependen mucho de la plataforma.

Crear un hilo implementando una interfaz ejecutable

Si su clase está destinada a ejecutarse como un subproceso, puede lograr esto implementando una interfaz **Runnable**. Deberá seguir tres pasos básicos:

Paso 1

Como primer paso, debe implementar un método `run()` proporcionado por una interfaz **Runnable**. Este método proporciona un punto de entrada para el hilo y colocará su lógica comercial completa dentro de este método. La siguiente es una sintaxis simple del método `run()`:

```
public void run( )
```

Paso 2

Como segundo paso, creará una instancia de un objeto **Thread** utilizando el siguiente constructor:

```
Thread(Runnable threadObj, String threadName);
```

Cuando, *threadObj* es una instancia de una clase que implementa la **Ejecutable** interfaz y **threadName** es el nombre dado a la nueva hilo.

Paso 3

Una vez que se crea un objeto `Thread`, puede iniciarlo llamando al método **start()**, que ejecuta un método `call to run()`. Lo siguiente es una sintaxis simple del método `start()`:

```
void start();
```

Ejemplo

Aquí hay un ejemplo que crea un nuevo hilo y comienza a ejecutarlo:

```

class RunnableDemo implements Runnable {
    private Thread t;
    private String threadName;

    RunnableDemo( String name) {
        threadName = name;
        System.out.println("Creating " + threadName );
    }

    public void run() {
        System.out.println("Running " + threadName );
        try {
            for(int i = 4; i > 0; i--) {
                System.out.println("Thread: " + threadName + ", "
+ i);

                // Let the thread sleep for a while.
                Thread.sleep(50);
            }
        } catch (InterruptedException e) {
            System.out.println("Thread " + threadName + "
interrupted.");
        }
        System.out.println("Thread " + threadName + "
exiting.");
    }

    public void start () {
        System.out.println("Starting " + threadName );
        if (t == null) {
            t = new Thread (this, threadName);
            t.start ();
        }
    }
}

public class TestThread {

    public static void main(String args[]) {
        RunnableDemo R1 = new RunnableDemo( "Thread-1");
        R1.start();

        RunnableDemo R2 = new RunnableDemo( "Thread-2");
        R2.start();
    }
}

```

Esto producirá el siguiente resultado:

Salida

```

Creating Thread-1
Starting Thread-1

```

```
Creating Thread-2
Starting Thread-2
Running Thread-1
Thread: Thread-1, 4
Running Thread-2
Thread: Thread-2, 4
Thread: Thread-1, 3
Thread: Thread-2, 3
Thread: Thread-1, 2
Thread: Thread-2, 2
Thread: Thread-1, 1
Thread: Thread-2, 1
Thread Thread-1 exiting.
Thread Thread-2 exiting.
```

Crear un hilo ampliando una clase de hilo

La segunda forma de crear un subproceso es crear una nueva clase que amplíe la clase de **subproceso** utilizando los siguientes dos pasos simples. Este enfoque proporciona más flexibilidad en el manejo de múltiples hilos creados usando los métodos disponibles en la clase Thread.

Paso 1

Deberá anular el método **run ()** disponible en la clase Thread. Este método proporciona un punto de entrada para el hilo y colocará su lógica comercial completa dentro de este método. Lo siguiente es una sintaxis simple del método run ():

```
public void run( )
```

Paso 2

Una vez que se crea el objeto Thread, puede iniciarlo llamando al método **start ()**, que ejecuta una llamada al método run (). Lo siguiente es una sintaxis simple del método start ():

```
void start( );
```

Ejemplo

Aquí está el programa anterior reescrito para extender el hilo -

```
class ThreadDemo extends Thread {
    private Thread t;
    private String threadName;

    ThreadDemo( String name) {
        threadName = name;
        System.out.println("Creating " + threadName );
    }
}
```

```

    public void run() {
        System.out.println("Running " + threadName );
        try {
            for(int i = 4; i > 0; i--) {
                System.out.println("Thread: " + threadName + ", "
+ i);

                // Let the thread sleep for a while.
                Thread.sleep(50);
            }
        } catch (InterruptedException e) {
            System.out.println("Thread " + threadName + "
interrupted.");
        }
        System.out.println("Thread " + threadName + "
exiting.");
    }

    public void start () {
        System.out.println("Starting " + threadName );
        if (t == null) {
            t = new Thread (this, threadName);
            t.start ();
        }
    }
}

public class TestThread {

    public static void main(String args[]) {
        ThreadDemo T1 = new ThreadDemo( "Thread-1");
        T1.start();

        ThreadDemo T2 = new ThreadDemo( "Thread-2");
        T2.start();
    }
}

```

Esto producirá el siguiente resultado:

Salida

```

Creating Thread-1
Starting Thread-1
Creating Thread-2
Starting Thread-2
Running Thread-1
Thread: Thread-1, 4
Running Thread-2
Thread: Thread-2, 4
Thread: Thread-1, 3
Thread: Thread-2, 3
Thread: Thread-1, 2

```

```
Thread: Thread-2, 2
Thread: Thread-1, 1
Thread: Thread-2, 1
Thread Thread-1 exiting.
Thread Thread-2 exiting.
```

Métodos de hilo

La siguiente es la lista de métodos importantes disponibles en la clase Thread.

No Señor.	Método y descripción
1	inicio nulo público () Inicia el hilo en una ruta de ejecución separada, luego invoca el método run () en este objeto Thread.
2	vacío público run () Si este objeto Thread fue instanciado usando un objetivo Runnable separado, el método run () se invoca en ese objeto Runnable.
3	public final void setName (nombre de cadena) Cambia el nombre del objeto Thread. También hay un método getName () para recuperar el nombre.
4 4	public final void setPriority (int prioridad) Establece la prioridad de este objeto Thread. Los valores posibles están entre 1 y 10.
5 5	public final void setDaemon (booleano encendido) Un parámetro de verdadero denota este Hilo como un hilo de demonio.
6 6	unión nula pública final (milisegundos largos) El subproceso actual invoca este método en un segundo subproceso, lo que hace que el subproceso actual se bloquee hasta que el segundo subproceso finalice o pase el número especificado de milisegundos.
7 7	interrupción pública nula () Interrumpe este hilo, haciendo que continúe la ejecución si fue bloqueado por alguna razón.

8	public boolean final isAlive () Devuelve verdadero si el hilo está vivo, que es en cualquier momento después de que se ha iniciado el hilo pero antes de que se ejecute hasta su finalización.
---	--

Los métodos anteriores se invocan en un objeto Thread particular. Los siguientes métodos en la clase Thread son estáticos. Invocar uno de los métodos estáticos realiza la operación en el hilo actualmente en ejecución.

No Señor.	Método y descripción
1	rendimiento vacío público estático () Hace que el subproceso actualmente en ejecución ceda el paso a cualquier otro subproceso de la misma prioridad que esté esperando ser programado.
2	sueño vacío público estático (milisegundos largos) Hace que el subproceso actualmente en ejecución se bloquee durante al menos el número especificado de milisegundos.
3	booleano público estático HoldLock (Object x) Devuelve verdadero si el hilo actual mantiene el bloqueo en el objeto dado.
4 4	Hilo público estático currentThread () Devuelve una referencia al hilo que se está ejecutando actualmente, que es el hilo que invoca este método.
5 5	public static void dumpStack () Imprime el seguimiento de la pila para el subproceso que se está ejecutando actualmente, lo cual es útil al depurar una aplicación multiproceso.

Ejemplo

El siguiente programa ThreadClassDemo muestra algunos de estos métodos de la clase Thread. Considere una clase **DisplayMessage** que implementa **Runnable** -

```
// File Name : DisplayMessage.java
// Create a thread to implement Runnable

public class DisplayMessage implements Runnable {
    private String message;

    public DisplayMessage(String message) {
```

```

        this.message = message;
    }

    public void run() {
        while(true) {
            System.out.println(message);
        }
    }
}

```

La siguiente es otra clase que extiende la clase Thread:

```

// File Name : GuessANumber.java
// Create a thread to extentd Thread

public class GuessANumber extends Thread {
    private int number;
    public GuessANumber(int number) {
        this.number = number;
    }

    public void run() {
        int counter = 0;
        int guess = 0;
        do {
            guess = (int) (Math.random() * 100 + 1);
            System.out.println(this.getName() + " guesses " +
guess);
            counter++;
        } while(guess != number);
        System.out.println("** Correct!" + this.getName() +
"in" + counter + "guesses.**");
    }
}

```

El siguiente es el programa principal, que hace uso de las clases definidas anteriormente:

```

// File Name : ThreadClassDemo.java
public class ThreadClassDemo {

    public static void main(String [] args) {
        Runnable hello = new DisplayMessage("Hello");
        Thread thread1 = new Thread(hello);
        thread1.setDaemon(true);
        thread1.setName("hello");
        System.out.println("Starting hello thread...");
        thread1.start();

        Runnable bye = new DisplayMessage("Goodbye");
        Thread thread2 = new Thread(bye);
        thread2.setPriority(Thread.MIN_PRIORITY);
        thread2.setDaemon(true);
        System.out.println("Starting goodbye thread...");
    }
}

```

```

        thread2.start();

        System.out.println("Starting thread3...");
        Thread thread3 = new GuessANumber(27);
        thread3.start();
        try {
            thread3.join();
        } catch (InterruptedException e) {
            System.out.println("Thread interrupted.");
        }
        System.out.println("Starting thread4...");
        Thread thread4 = new GuessANumber(75);

        thread4.start();
        System.out.println("main() is ending...");
    }
}

```

Esto producirá el siguiente resultado. Puede probar este ejemplo una y otra vez y obtendrá un resultado diferente cada vez.

Salida

```

Starting hello thread...
Starting goodbye thread...
Hello
Hello
Hello
Hello
Hello
Hello
Goodbye
Goodbye
Goodbye
Goodbye
Goodbye
.....

```

Conceptos principales de subprocesos múltiples de Java

Mientras realiza la programación de subprocesos múltiples en Java, necesitaría tener los siguientes conceptos muy útiles:

- ¿Qué es la sincronización de subprocesos?
- Manejo de la comunicación entre hilos
- Manejo de bloqueo de hilo
- Operaciones principales de hilo

Java - Conceptos básicos de Applet

Un **applet** es un programa Java que se ejecuta en un navegador web. Un applet puede ser una aplicación Java completamente funcional porque tiene toda la API Java a su disposición.

Existen algunas diferencias importantes entre un applet y una aplicación Java independiente, que incluyen las siguientes:

- Un applet es una clase Java que extiende la clase `java.applet.Applet`.
- No se invoca un método `main ()` en un applet, y una clase de applet no definirá `main ()`.
- Los applets están diseñados para integrarse en una página HTML.
- Cuando un usuario ve una página HTML que contiene un applet, el código del applet se descarga en la máquina del usuario.
- Se requiere una JVM para ver un applet. La JVM puede ser un complemento del navegador web o un entorno de tiempo de ejecución separado.
- La JVM en la máquina del usuario crea una instancia de la clase de applet e invoca varios métodos durante la vida útil del applet.
- Los applets tienen reglas de seguridad estrictas que son aplicadas por el navegador web. La seguridad de un applet a menudo se denomina seguridad sandbox, comparando el applet con un niño que juega en un sandbox con varias reglas que deben seguirse.
- Otras clases que necesita el applet se pueden descargar en un solo archivo Java Archive (JAR).

Ciclo de vida de un applet

Cuatro métodos en la clase `Applet` le brindan el marco en el que construye cualquier applet serio:

- **init** : este método está destinado a cualquier inicialización necesaria para su applet. Se llama después de que se hayan procesado las etiquetas param dentro de la etiqueta del applet.
- **inicio** : este método se llama automáticamente después de que el navegador llama al método `init`. También se llama cada vez que el usuario vuelve a la página que contiene el applet después de haber salido a otras páginas.
- **stop** : este método se llama automáticamente cuando el usuario se mueve fuera de la página en la que se encuentra el applet. Por lo tanto, se puede llamar repetidamente en el mismo applet.
- **Destruir** : este método solo se llama cuando el navegador se apaga normalmente. Debido a que los applets están destinados a vivir en una página HTML, normalmente no debe dejar recursos después de que un usuario abandone la página que contiene el applet.
- **paint** : se invoca inmediatamente después del método `start ()` y también cada vez que el applet necesita volver a pintarse en el navegador. El método `paint ()` en realidad se hereda de `java.awt`.

Un Applet "Hola Mundo"

El siguiente es un applet simple llamado `HelloWorldApplet.java`:

```
import java.applet.*;
import java.awt.*;

public class HelloWorldApplet extends Applet {
    public void paint (Graphics g) {
        g.drawString ("Hello World", 25, 50);
    }
}
```

Estas declaraciones de importación llevan las clases al alcance de nuestra clase de applet:

- java.applet.Applet
- java.awt.Graphics

Sin esas declaraciones de importación, el compilador de Java no reconocería las clases Applet y Graphics, a las que se refiere la clase applet.

La clase de applet

Cada applet es una extensión de la *clase java.applet.Applet*. La clase Applet básica proporciona métodos que una clase Applet derivada puede llamar para obtener información y servicios del contexto del navegador.

Estos incluyen métodos que hacen lo siguiente:

- Obtener parámetros de applet
- Obtenga la ubicación de red del archivo HTML que contiene el applet
- Obtenga la ubicación de red del directorio de clase de applet
- Imprime un mensaje de estado en el navegador
- Obtener una imagen
- Obtener un clip de audio
- Reproduce un clip de audio
- Cambiar el tamaño del applet

Además, la clase Applet proporciona una interfaz mediante la cual el espectador o el navegador obtienen información sobre el applet y controlan la ejecución del applet. El espectador puede ...

- Solicitar información sobre el autor, la versión y los derechos de autor del applet
- Solicitar una descripción de los parámetros que reconoce el applet
- Inicializar el applet
- Destruye el applet
- Comienza la ejecución del applet
- Detener la ejecución del applet

La clase Applet proporciona implementaciones predeterminadas de cada uno de estos métodos. Esas implementaciones pueden anularse según sea necesario.

El applet "Hola, Mundo" está completo tal como está. El único método anulado es el método de pintura.

Invocar un applet

Se puede invocar un applet incrustando directivas en un archivo HTML y visualizando el archivo a través de un visor de applet o un navegador habilitado para Java.

La etiqueta <applet> es la base para incrustar un applet en un archivo HTML. El siguiente es un ejemplo que invoca el applet "Hello, World":

```
<html>
  <title>The Hello, World Applet</title>
  <hr>
  <applet code = "HelloWorldApplet.class" width = "320"
height = "120">
    If your browser was Java-enabled, a "Hello, World"
    message would appear here.
  </applet>
  <hr>
</html>
```

Nota : puede consultar la etiqueta HTML Applet para obtener más información sobre cómo llamar a applet desde HTML.

Se requiere el atributo de código de la etiqueta <applet>. Especifica la clase de Applet para ejecutar. El ancho y la altura también son necesarios para especificar el tamaño inicial del panel en el que se ejecuta un applet. La directiva de applet debe cerrarse con una etiqueta </applet>.

Si un applet toma parámetros, se pueden pasar valores para los parámetros agregando etiquetas <param> entre <applet> y </applet>. El navegador ignora el texto y otras etiquetas entre las etiquetas de applet.

Los navegadores no habilitados para Java no procesan <applet> y </applet>. Por lo tanto, cualquier cosa que aparezca entre las etiquetas, no relacionada con el applet, es visible en los navegadores no habilitados para Java.

El visor o navegador busca el código Java compilado en la ubicación del documento. Para especificar lo contrario, use el atributo codebase de la etiqueta <applet> como se muestra:

```
<applet codebase = "https://amrood.com/applets" code =
"HelloWorldApplet.class"
  width = "320" height = "120">
```

Si un applet reside en un paquete distinto al predeterminado, el paquete de retención debe especificarse en el atributo de código utilizando el carácter de punto (.) Para separar los componentes del paquete / clase. Por ejemplo

```
<applet  = "mypackage.subpackage.TestApplet.class"
  width = "320" height = "120">
```

Obteniendo Parámetros de Applet

El siguiente ejemplo muestra cómo hacer que un applet responda a los parámetros de configuración especificados en el documento. Este applet muestra un patrón de tablero de ajedrez de color negro y un segundo color.

El segundo color y el tamaño de cada cuadrado pueden especificarse como parámetros para el applet dentro del documento.

CheckerApplet obtiene sus parámetros en el método `init ()`. También puede obtener sus parámetros en el método `paint ()`. Sin embargo, obtener los valores y guardar la configuración una vez al comienzo del applet, en lugar de cada actualización, es conveniente y eficiente.

El visor o navegador de applets llama al método `init ()` de cada applet que ejecuta. El espectador llama a `init ()` una vez, inmediatamente después de cargar el applet. (`Applet.init ()` se implementa para no hacer nada). Anule la implementación predeterminada para insertar un código de inicialización personalizado.

El método `Applet.getParameter ()` obtiene un parámetro dado el nombre del parámetro (el valor de un parámetro siempre es una cadena). Si el valor es numérico u otros datos sin caracteres, la cadena debe analizarse.

El siguiente es un esqueleto de `CheckerApplet.java`:

```
import java.applet.*;
import java.awt.*;

public class CheckerApplet extends Applet {
    int squareSize = 50;    // initialized to default size
    public void init() {}
    private void parseSquareSize (String param) {}
    private Color parseColor (String param) {}
    public void paint (Graphics g) {}
}
```

Estos son los métodos `init ()` y `parseSquareSize ()` privados de `CheckerApplet`:

```
public void init () {
    String squareSizeParam = getParameter ("squareSize");
    parseSquareSize (squareSizeParam);

    String colorParam = getParameter ("color");
    Color fg = parseColor (colorParam);

    setBackground (Color.black);
    setForeground (fg);
}

private void parseSquareSize (String param) {
    if (param == null) return;
    try {
        squareSize = Integer.parseInt (param);
    } catch (Exception e) {
```

```
        // Let default value remain
    }
}
```

El applet llama a `parseSquareSize ()` para analizar el parámetro `squareSize`. `parseSquareSize ()` llama al método de biblioteca `Integer.parseInt ()`, que analiza una cadena y devuelve un entero. `Integer.parseInt ()` lanza una excepción cada vez que su argumento no es válido.

Por lo tanto, `parseSquareSize ()` captura excepciones, en lugar de permitir que el applet falle en una entrada incorrecta.

El applet llama a `parseColor ()` para analizar el parámetro de color en un valor de `Color`. `parseColor ()` realiza una serie de comparaciones de cadenas para hacer coincidir el valor del parámetro con el nombre de un color predefinido. Debe implementar estos métodos para que este applet funcione.

Especificación de parámetros de applet

El siguiente es un ejemplo de un archivo HTML con un `CheckerApplet` incrustado en él. El archivo HTML especifica ambos parámetros para el applet mediante la etiqueta `<param>`.

```
<html>
  <title>Checkerboard Applet</title>
  <hr>
  <applet code = "CheckerApplet.class" width = "480" height
= "320">
    <param name = "color" value = "blue">
    <param name = "squaresize" value = "30">
  </applet>
  <hr>
</html>
```

Nota - Los nombres de los parámetros no distinguen entre mayúsculas y minúsculas.

Conversión de aplicaciones a applets

Es fácil convertir una aplicación gráfica de Java (es decir, una aplicación que utiliza AWT y que puede comenzar con el iniciador de programas Java) en un applet que puede incrustar en una página web.

Los siguientes son los pasos específicos para convertir una aplicación en un applet.

- Cree una página HTML con la etiqueta adecuada para cargar el código del applet.
- Proporcione una subclase de la clase `JApplet`. Haz pública esta clase. De lo contrario, el applet no se puede cargar.
- Eliminar el método principal en la aplicación. No construya una ventana de marco para la aplicación. Su aplicación se mostrará dentro del navegador.

- Mueva cualquier código de inicialización del constructor de la ventana de marco al método `init` del applet. No necesita construir explícitamente el objeto applet. El navegador lo crea una instancia y llama al método `init`.
- Eliminar la llamada a `setSize`; para los applets, el dimensionamiento se realiza con los parámetros ancho y alto en el archivo HTML.
- Elimine la llamada a `setDefaultCloseOperation`. Un applet no puede cerrarse; termina cuando el navegador se cierra.
- Si la aplicación llama a `setTitle`, elimine la llamada al método. Los applets no pueden tener barras de título. (Por supuesto, puede titular la página web en sí, utilizando la etiqueta de título HTML).
- No llame a `setVisible` (verdadero). El applet se muestra automáticamente.

Manejo de eventos

Los applets heredan un grupo de métodos de manejo de eventos de la clase `Container`. La clase `Container` define varios métodos, como `processKeyEvent` y `processMouseEvent`, para manejar tipos particulares de eventos, y luego un método general llamado `processEvent`.

Para reaccionar a un evento, un applet debe anular el método apropiado específico del evento.

```
import java.awt.event.MouseListener;
import java.awt.event.MouseEvent;
import java.applet.Applet;
import java.awt.Graphics;

public class ExampleEventHandling extends Applet implements
MouseListener {
    StringBuffer strBuffer;

    public void init() {
        addMouseListener(this);
        strBuffer = new StringBuffer();
        addItem("initializing the apple ");
    }

    public void start() {
        addItem("starting the applet ");
    }

    public void stop() {
        addItem("stopping the applet ");
    }

    public void destroy() {
        addItem("unloading the applet");
    }

    void addItem(String word) {
        System.out.println(word);
    }
}
```

```

        strBuffer.append(word);
        repaint();
    }

    public void paint(Graphics g) {
        // Draw a Rectangle around the applet's display area.
        g.drawRect(0, 0,
            getWidth() - 1,
            getHeight() - 1);

        // display the string inside the rectangle.
        g.drawString(strBuffer.toString(), 10, 20);
    }

    public void mouseEntered(MouseEvent event) {
    }
    public void mouseExited(MouseEvent event) {
    }
    public void mousePressed(MouseEvent event) {
    }
    public void mouseReleased(MouseEvent event) {
    }
    public void mouseClicked(MouseEvent event) {
        addItem("mouse clicked! ");
    }
}

```

Ahora, llamemos a este applet de la siguiente manera:

```

<html>
  <title>Event Handling</title>
  <hr>
  <applet code = "ExampleEventHandling.class"
    width = "300" height = "300">
  </applet>
  <hr>
</html>

```

Inicialmente, el applet mostrará "inicializando el applet. Iniciando el applet". Luego, una vez que haga clic dentro del rectángulo, también se mostrará "clic del mouse".

Mostrar imágenes

Un applet puede mostrar imágenes en formato GIF, JPEG, BMP y otros. Para mostrar una imagen dentro del applet, utilice el método `drawImage()` que se encuentra en la clase `java.awt.Graphics`.

El siguiente es un ejemplo que ilustra todos los pasos para mostrar imágenes:

```

import java.applet.*;
import java.awt.*;

```

```

import java.net.*;

public class ImageDemo extends Applet {
    private Image image;
    private AppletContext context;

    public void init() {
        context = this.getAppletContext();
        String imageURL = this.getParameter("image");
        if(imageURL == null) {
            imageURL = "java.jpg";
        }
        try {
            URL url = new URL(this.getDocumentBase(), imageURL);
            image = context.getImage(url);
        } catch (MalformedURLException e) {
            e.printStackTrace();
            // Display in browser status bar
            context.showStatus("Could not load image!");
        }
    }

    public void paint(Graphics g) {
        context.showStatus("Displaying image");
        g.drawImage(image, 0, 0, 200, 84, null);
        g.drawString("www.javalicense.com", 35, 100);
    }
}

```

Ahora, llamemos a este applet de la siguiente manera:

```

<html>
  <title>The ImageDemo applet</title>
  <hr>
  <applet code = "ImageDemo.class" width = "300" height =
"200">
    <param name = "image" value = "java.jpg">
  </applet>
  <hr>
</html>

```

Reproducción de audio

Un applet puede reproducir un archivo de audio representado por la interfaz AudioClip en el paquete java.applet. La interfaz de AudioClip tiene tres métodos, que incluyen:

- **public void play ()** : reproduce el clip de audio una vez, desde el principio.
- **public void loop ()** : hace que el clip de audio se reproduzca continuamente.
- **public void stop ()** : deja de reproducir el clip de audio.

Para obtener un objeto AudioClip, debe invocar el método `getAudioClip()` de la clase `Applet`. El método `getAudioClip()` regresa inmediatamente, ya sea que la URL se resuelva o no en un archivo de audio real. El archivo de audio no se descarga hasta que se intente reproducir el clip de audio.

El siguiente es un ejemplo que ilustra todos los pasos para reproducir un audio:

```
import java.applet.*;
import java.awt.*;
import java.net.*;

public class AudioDemo extends Applet {
    private AudioClip clip;
    private AppletContext context;

    public void init() {
        context = this.getAppletContext();
        String audioURL = this.getParameter("audio");
        if(audioURL == null) {
            audioURL = "default.au";
        }
        try {
            URL url = new URL(this.getDocumentBase(), audioURL);
            clip = context.getAudioClip(url);
        } catch (MalformedURLException e) {
            e.printStackTrace();
            context.showStatus("Could not load audio file!");
        }
    }

    public void start() {
        if(clip != null) {
            clip.loop();
        }
    }

    public void stop() {
        if(clip != null) {
            clip.stop();
        }
    }
}
```

Ahora, llamemos a este applet de la siguiente manera:

```
<html>
  <title>The ImageDemo applet</title>
  <hr>
  <applet code = "ImageDemo.class" width = "0" height = "0">
    <param name = "audio" value = "test.wav">
  </applet>
  <hr>
</html>
```

Puede usar test.wav en su PC para probar el ejemplo anterior.

Java - Comentarios de documentación

El lenguaje Java admite tres tipos de comentarios:

No Señor.	Comentario y descripción
1	<code>/* texto */</code> El compilador ignora todo, desde <code>/*</code> hasta <code>*/</code> .
2	<code>//texto</code> El compilador ignora todo, desde <code>//</code> hasta el final de la línea.
3	<code>/** documentación */</code> Este es un comentario de documentación y, en general, se llama comentario de documentación . La herramienta JDK javadoc utiliza <i>comentarios de documentos</i> al preparar documentación generada automáticamente.

Este capítulo trata sobre la explicación de Javadoc. Veremos cómo podemos hacer uso de Javadoc para generar documentación útil para el código Java.

¿Qué es Javadoc?

Javadoc es una herramienta que viene con JDK y se usa para generar documentación de código Java en formato HTML a partir del código fuente de Java, que requiere documentación en un formato predefinido.

El siguiente es un ejemplo simple donde las líneas dentro de `/*...*/` son comentarios de múltiples líneas de Java. Del mismo modo, la línea que precede `//` es un comentario de línea única de Java.

Ejemplo

```
/**
 * The HelloWorld program implements an application that
 * simply displays "Hello World!" to the standard output.
 *
 * @author  Zara Ali
 * @version 1.0
 * @since   2014-03-31
 */
public class HelloWorld {

    public static void main(String[] args) {
        // Prints Hello, World! on standard output.
    }
}
```

```
        System.out.println("Hello World!");
    }
}
```

Puede incluir las etiquetas HTML requeridas dentro de la parte de descripción. Por ejemplo, el siguiente ejemplo utiliza `<h1> </h1>` para el encabezado y `<p>` se ha utilizado para crear un salto de párrafo:

Ejemplo

```
/**
 * <h1>Hello, World!</h1>
 * The HelloWorld program implements an application that
 * simply displays "Hello World!" to the standard output.
 * <p>
 * Giving proper comments in your program makes it more
 * user friendly and it is assumed as a high quality code.
 *
 *
 * @author  Zara Ali
 * @version 1.0
 * @since   2014-03-31
 */
public class HelloWorld {

    public static void main(String[] args) {
        // Prints Hello, World! on standard output.
        System.out.println("Hello World!");
    }
}
```

Las etiquetas javadoc

La herramienta javadoc reconoce las siguientes etiquetas:

Etiqueta	Descripción	Sintaxis
@autor	Agrega el autor de una clase.	@autor nombre-texto
{@código}	Muestra el texto en fuente de código sin interpretar el texto como marcado HTML o etiquetas javadoc anidadas.	{@code text}
{@docRoot}	Representa la ruta relativa al directorio raíz del documento generado desde cualquier página generada.	{@docRoot}

@obsoleto	Agrega un comentario que indica que esta API ya no debe usarse.	@deprecated deprecatedtext
@excepción	Agrega un subtítulo Throws a la documentación generada, con el nombre de la clase y el texto descriptivo.	Descripción de nombre de clase @exception
{@inheritDoc}	Hereda un comentario de la clase heredable o interfaz implementable más cercana .	Hereda un comentario de la clase superior inmediata.
{@enlace}	Inserta un enlace en línea con la etiqueta de texto visible que apunta a la documentación para el paquete, clase o nombre de miembro especificado de una clase referenciada.	{@link package.class # member label}
{@linkplain}	Idéntico a {@link}, excepto que la etiqueta del enlace se muestra en texto sin formato que la fuente del código.	{@linkplain package.class # etiqueta de miembro}
@param	Agrega un parámetro con el nombre del parámetro especificado seguido de la descripción especificada en la sección "Parámetros".	Descripción del nombre del parámetro @param
@regreso	Agrega una sección de "Devoluciones" con el texto descriptivo.	Descripción de @return
@ver	Agrega un encabezado "Ver también" con un enlace o entrada de texto que apunta a la referencia.	@ver referencia
@de serie	Se usa en el comentario del documento para un campo serializable predeterminado.	@serial field- description incluir excluir
@serialData	Documenta los datos escritos por los métodos writeObject () o writeExternal ().	Descripción de datos de @serialData
@serialField	Documenta un componente ObjectOutputStreamField.	@serialField nombre- campo tipo-campo descripción-campo

@ya que	Agrega un encabezado "Since" con el texto since especificado a la documentación generada.	@desde el lanzamiento
@throws	Las etiquetas @throws y @exception son sinónimos.	Descripción de nombre de clase de @throws
{@valor}	Cuando se utiliza {@value} en el comentario del documento de un campo estático, muestra el valor de esa constante.	{@value package.class # field}
@versión	Agrega un subtítulo "Versión" con el texto de versión especificado a los documentos generados cuando se usa la opción -version.	@version version-text

Ejemplo

El siguiente programa utiliza algunas de las etiquetas importantes disponibles para comentarios de documentación. Puede utilizar otras etiquetas según sus requisitos.

La documentación sobre la clase AddNum se producirá en el archivo HTML AddNum.html, pero al mismo tiempo también se creará un archivo maestro con un nombre index.html.

```
import java.io.*;

/**
 * <h1>Add Two Numbers!</h1>
 * The AddNum program implements an application that
 * simply adds two given integer numbers and Prints
 * the output on the screen.
 * <p>
 * <b>Note:</b> Giving proper comments in your program makes
 * it more
 * user friendly and it is assumed as a high quality code.
 *
 * @author  Zara Ali
 * @version 1.0
 * @since   2014-03-31
 */
public class AddNum {
    /**
     * This method is used to add two integers. This is
     * a the simplest form of a class method, just to
     * show the usage of various javadoc Tags.
     * @param numA This is the first paramter to addNum method
     * @param numB This is the second parameter to addNum
     method

```

```

    * @return int This returns sum of numA and numB.
    */
    public int addNum(int numA, int numB) {
        return numA + numB;
    }

    /**
     * This is the main method which makes use of addNum
method.
     * @param args Unused.
     * @return Nothing.
     * @exception IOException On input error.
     * @see IOException
     */

    public static void main(String args[]) throws IOException
{
    AddNum obj = new AddNum();
    int sum = obj.addNum(10, 20);

    System.out.println("Sum of 10 and 20 is :" + sum);
}
}

```

Ahora, procese el archivo AddNum.java anterior utilizando la utilidad javadoc de la siguiente manera:

```

$ javadoc AddNum.java
Loading source file AddNum.java...
Constructing Javadoc information...
Standard Doclet version 1.7.0_51
Building tree for all the packages and classes...
Generating /AddNum.html...
AddNum.java:36: warning - @return tag cannot be used in
method with void return type.
Generating /package-frame.html...
Generating /package-summary.html...
Generating /package-tree.html...
Generating /constant-values.html...
Building index for all the packages and classes...
Generating /overview-tree.html...
Generating /index-all.html...
Generating /deprecated-list.html...
Building index for all classes...
Generating /allclasses-frame.html...
Generating /allclasses-noframe.html...
Generating /index.html...
Generating /help-doc.html...
1 warning
$

```

Puede consultar toda la documentación generada aquí: AddNum. Si está utilizando JDK 1.7, entonces javadoc no genera un gran **stylesheet.css**, por

lo que le sugerimos que descargue y use la hoja de estilo estándar de <https://docs.oracle.com/javase/7/docs/api/stylesheet.css>



Síguenos en Instagram para que estés al tanto de los nuevos libros de programación. [Click aqui](#)

Follow us on Instagram so you are aware of the new programming books [Click here](#)

Descarga más libros de programación GRATIS [click aquí](#)

Download more FREE programming books [click here](#)

Descarga más libros de programación GRATIS [click aquí](#)

Download more FREE programming books [click here](#)