<https://www.sololearn.com/learning/1068/2137/4199/1>

Texto

Descripción generada automáticamente

En Java, cada línea de código que realmente se puede ejecutar debe estar dentro de una clase.

En nuestro ejemplo, llamamos a la clase MyClass. Aprenderá más sobre las clases en los próximos módulos.

En Java, cada aplicación tiene un punto de entrada o punto de partida, que es un método llamado main. Junto con main, las palabras clave public y static también se explicarán más adelante.

Como un resumen:

- Todo programa en Java debe tener una clase.

- Cada programa Java comienza desde el método principal.

**The main Method**

Para ejecutar nuestro programa, el método principal debe ser idéntico a esta firma:

**public static void main(String[ ] args)**

- público: cualquiera puede acceder a él

- estático: el método se puede ejecutar sin crear una instancia de la clase que contiene el método principal

- void: el método no devuelve ningún valor

- main: el nombre del método

Por ejemplo, el siguiente código declara un método llamado prueba, que no devuelve nada y no tiene parámetros:

void test()

Los parámetros del método se declaran dentro de los paréntesis que siguen al nombre del método.

Para main, es una matriz de cadenas llamadas args. Lo usaremos en nuestra próxima lección, así que no se preocupe si no lo entiende todo ahora.

# System.out.println()

El siguiente es el cuerpo del método principal, encerrado entre llaves:

{   
System.out.println("Hello World!");   
}

El método println imprime una línea de texto en la pantalla.

La clase System y su flujo de salida se utilizan para acceder al método println.

En las clases, los métodos y otros códigos de estructuras de control de flujo siempre están encerrados en {}.

# Semicolons in Java

Puede pasar un texto diferente como parámetro al método println para imprimirlo.

Texto

Descripción generada automáticamente

En Java, cada declaración de código debe terminar con un punto y coma.

Recuerde: no use punto y coma después de las declaraciones de métodos y clases que siguen con el cuerpo definido usando las llaves.

# Comments

El propósito de incluir comentarios en su código es explicar qué está haciendo el código.

Java admite comentarios de una sola línea y de varias líneas. El compilador de Java ignora todos los caracteres que aparecen dentro de un comentario.

Un comentario de una sola línea comienza con dos barras diagonales y continúa hasta llegar al final de la línea.

Por ejemplo:

**// this is a single-line comment**  
x = 5; **// a single-line comment after code**

Agregar comentarios a medida que escribe el código es una buena práctica, ya que brindan aclaración y comprensión cuando necesita volver a consultarlo, así como para otras personas que puedan necesitar leerlo.

Comentarios de varias líneas

Java también admite comentarios que abarcan varias líneas.

Este tipo de comentario se inicia con una barra diagonal seguida de un asterisco y se finaliza con un asterisco seguido de una barra diagonal.

Por ejemplo:

/\* This is also a   
comment spanning   
multiple lines \*/

# Documentation Comments

Los comentarios de documentación son comentarios especiales que tienen la apariencia de comentarios de varias líneas, con la diferencia de que generan documentación externa de su código fuente. Estos comienzan con una barra inclinada seguida de dos asteriscos y terminan con un asterisco seguido de una barra inclinada.

Por ejemplo:

/\*\* This is a documentation comment \*/   
  
/\*\* This is also a   
documentation comment \*/

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 tiene la documentación requerida en un formato predefinido.

Cuando un comentario de documentación comienza con más de dos asteriscos, Javadoc asume que desea crear un "cuadro" alrededor del comentario en el código fuente. Simplemente ignora los asteriscos adicionales.

Por ejemplo:

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*   
  
This is the start of a method   
  
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

Esto retendrá solo el texto "Este es el comienzo de un método" para la documentación.

# Variables

Las variables almacenan datos para su procesamiento.

A una variable se le da un nombre (o identificador), como área, edad, altura y similares. El nombre identifica de forma única cada variable, asignando un valor a la variable y recuperando el valor almacenado.

Las variables tienen tipos. Algunos ejemplos:

- int: para enteros (números enteros) como 123 y -456

- double: para números reales o de punto flotante con puntos decimales opcionales y partes fraccionarias en notaciones fijas o científicas, como 3.1416, -55.66.

- **String**: para textos como "Hola" o "¡Buenos días!". Las cadenas de texto se encierran entre comillas dobles.

Puede declarar una variable de un tipo y asignarle un valor.

Ejemplo:

String name = "David";

Esto crea una variable llamada nombre de tipo String y le asigna el valor "David".

Es importante tener en cuenta que una variable está asociada con un tipo y solo puede almacenar valores de ese tipo en particular. Por ejemplo, una variable int puede almacenar valores enteros, como 123; pero no puede almacenar números reales, como 12,34, ni textos, como "Hola".

**Variables**  
Examples of variable declarations:

**Texto

Descripción generada automáticamente**

char significa carácter y tiene un solo carácter.

Otro tipo es el tipo booleano, que tiene solo dos valores posibles: verdadero y falso.

Este tipo de datos se usa para indicadores simples que rastrean condiciones verdaderas o falsas.

Por ejemplo:

**boolean** online = true;

Puede utilizar una lista separada por comas para declarar más de una variable del tipo especificado. Ejemplo: int a = 42, b = 11;

# Getting User Input

Si bien Java proporciona muchos métodos diferentes para obtener información del usuario, el objeto Scanner es el más común y quizás el más fácil de implementar. Importe la clase Scanner para usar el objeto Scanner, como se ve aquí:

**import java.util.Scanner;**

**Para usar la clase Scanner, cree una instancia de la clase usando la siguiente sintaxis:**

**Scanner myVar = new Scanner(System.in);**

**Ahora puede leer diferentes tipos de datos de entrada que ingresa el usuario.**

**Aquí hay algunos métodos que están disponibles a través de la clase Scanner:**

**Leer un byte - nextByte()**

**Leer un corto - nextShort()**

**Leer un int - nextInt()**

**Leer un largo - nextLong()**

**Leer un flotador - nextFloat()**

**Leer un doble - nextDouble()**

**Leer un booleano - nextBoolean()**

**Leer una línea completa - nextLine()**

**Leer una palabra - next()**

**Ejemplo de un programa utilizado para obtener la entrada del usuario:**

**Texto

Descripción generada automáticamente**

# The Math Operators

Java proporciona un rico conjunto de operadores para usar en la manipulación de variables. Un valor usado en cualquier lado de un operador se llama operando.

Por ejemplo, en la siguiente expresión, los números 6 y 3 son operandos del operador más:

int x = 6 + 3;

Java arithmetic operators:   
+ **addition**  
- **subtraction**  
\* **multiplication**  
/ **division**  
% **modulo**

Los operadores aritméticos se usan en expresiones matemáticas de la misma manera que se usan en ecuaciones algebraicas.

# Increment Operators

Un operador de incremento o decremento proporciona una forma más conveniente y compacta de aumentar o disminuir el valor de una variable en uno.

Por ejemplo, la declaración x=x+1; se puede simplificar a ++x;

Ejemplo:

Texto

Descripción generada automáticamente con confianza media

El operador de decremento (--) se utiliza para disminuir el valor de una variable en uno.

Imagen de la pantalla de un celular de un mensaje en letras negras

Descripción generada automáticamente con confianza baja

# Prefix & Postfix

Se pueden usar dos formas, prefijo y posfijo, con los operadores de incremento y decremento.

Con forma de prefijo, el operador aparece antes del operando, mientras que en forma de sufijo, el operador aparece después del operando. A continuación se muestra una explicación de cómo funcionan las dos formas:

Prefijo: Incrementa el valor de la variable y usa el nuevo valor en la expresión.

Ejemplo:

Imagen de la pantalla de un celular de un mensaje en letras negras

Descripción generada automáticamente con confianza baja

El valor de x se incrementa primero a 35 y luego se asigna a y, por lo que los valores de x e y ahora son 35.

Sufijo: el valor de la variable se usa primero en la expresión y luego se incrementa.

Ejemplo:

Texto

Descripción generada automáticamente con confianza media

x se asigna primero a y, y luego se incrementa en uno. Por lo tanto, x se convierte en 35, mientras que a y se le asigna el valor de 34.

Lo mismo se aplica al operador de decremento.

# Assignment Operators

Ya está familiarizado con el operador de asignación (=), que asigna un valor a una variable.

int value = 5;

Este asignó el valor 5 a una variable llamada valor de tipo int.

Java proporciona una serie de operadores de asignación para facilitar la escritura de código.

Adición y asignación (+=):

Un conjunto de letras blancas en un fondo blanco

Descripción generada automáticamente con confianza media

**Subtraction and assignment (-=):**

Interfaz de usuario gráfica

Descripción generada automáticamente con confianza media

De manera similar, Java admite la multiplicación y la asignación (\*=), la división y la asignación (/=) y el resto y la asignación (%=).

Strings  
Una cadena es un objeto que representa una secuencia de caracteres. Por ejemplo, "Hola" es una cadena de 5 caracteres.

For example:

String s = "SoloLearn";

Se le permite definir una cadena vacía. Por ejemplo, Cadena str = "";

# String Concatenation

El operador + (más) entre cadenas las suma para formar una nueva cadena. Este proceso se llama concatenación.

La cadena resultante es la primera cadena junto con la segunda cadena.

Por ejemplo:

Texto

Descripción generada automáticamente

El tipo de datos char representa un solo carácter.

**Conditionals and Loops**

# Decision Making

Las declaraciones condicionales se utilizan pararealizar diferentes acciones basadas en diferentes condiciones.

**La sentencia if** es una de las sentencias condicionales más utilizadas.

Si la expresión de condición de la declaración if se evalúa como verdadera, se ejecuta el bloque de código dentro de la declaración if. Si se determina que la expresión es falsa, se ejecuta el primer conjunto de código después del final de la instrucción if (después de la llave de cierre).

Sintaxis:

if (condition) {   
//Executes when the condition is true   
}

Cualquiera de los siguientes operadores de comparación puede usarse para formar la condición:

< menos que

> mayor que

!= no igual a

== igual a

<= menor o igual que

>= mayor o igual que

For example:

# Interfaz de usuario gráfica, Texto Descripción generada automáticamente

Recuerde que necesita usar dos signos de igual (==) para probar la igualdad, ya que un solo signo de igual es el operador de asignación.

# if...else Statements

Una declaración if puede ir seguida de una declaración else opcional, que se ejecuta cuando la condición se evalúa como falsa.

Por ejemplo:

Texto

Descripción generada automáticamente

Como la edad es igual a 30, la condición en la instrucción if se evalúa como falsa y se ejecuta la instrucción else.

Nested if Statements (Declaraciones if anidadas**)**  
Puede usar una instrucción if-else dentro de otra instrucción if o else. Por ejemplo:

**Texto

Descripción generada automáticamente**Puede anidar tantas sentencias if-else como desee.

# else if Statements

En lugar de usar sentencias if-else anidadas, puede usar la sentencia else if para verificar varias condiciones.

Por ejemplo:

Texto

Descripción generada automáticamente

El código verificará la condición para evaluar como verdadera y ejecutará las declaraciones dentro de ese bloque.

Puede incluir tantas sentencias if como necesite.

# Logical Operators

Los operadores lógicos se utilizan para combinar varias condiciones.

Digamos que desea que su programa emita "¡Bienvenido!" solo cuando la variable edad es mayor a 18 y la variable dinero es mayor a 500.

Una forma de lograr esto es usar sentencias if anidadas:

Texto

Descripción generada automáticamente

Sin embargo, usar el operador lógico AND (&&) es una mejor manera:

Texto

Descripción generada automáticamente

Si ambos operandos del operador AND son verdaderos, entonces la condición se vuelve verdadera.

# The OR Operator

El operador OR (||) comprueba si alguna de las condiciones es verdadera.

La condición se vuelve verdadera, si cualquiera de los operandos se evalúa como verdadero.

Por ejemplo:

Texto

Descripción generada automáticamente

El código anterior imprimirá "¡Bienvenido!" si la edad es mayor de 18 años o si el dinero es mayor de 500.

El operador lógico NOT !) se utiliza para invertir el estado lógico de su operando. Si una condición es verdadera, el operador lógico NOT la hará falsa.

Ejemplo:

Texto

Descripción generada automáticamente

!(edad > 18) se lee como "si la edad NO es mayor de 18".

# The switch Statement

Una instrucción switch prueba la igualdad de una variable con una lista de valores. Cada valor se llama un caso, y la variable que se activa se comprueba para cada caso. Sintaxis:

**switch (expression) {   
case value1 :   
//Statements   
break; //optional   
case value2 :   
//Statements   
break; //optional   
//You can have any number of case statements.   
default : //Optional   
//Statements   
}**

- Cuando la variable que se activa es igual a un caso, las declaraciones que siguen a ese caso se ejecutarán hasta que se alcance una declaración de ruptura.

- Cuando se llega a una declaración de ruptura, el interruptor finaliza y el flujo de control salta a la siguiente línea después de la declaración de cambio.

- No todos los casos necesitan contener un descanso. Si no aparece ninguna interrupción, el flujo de control pasará a los casos posteriores hasta que se alcance una interrupción.

El siguiente ejemplo prueba el día con un conjunto de valores e imprime el mensaje correspondiente.

Texto

Descripción generada automáticamente

Puede tener cualquier número de declaraciones de casos dentro de un switch. Cada caso va seguido del valor de comparación y dos puntos.

# The default Statement

Una declaración de switch puede tener un caso default opcional.

El caso default (predeterminado) se puede utilizar para realizar una tarea cuando ninguno de los casos coincide.

Por ejemplo:

Texto

Descripción generada automáticamente

No se necesita interrupción en el caso predeterminado, ya que siempre es la última declaración en el cambio.

# The switch Expression

La expresión switch permite múltiples valores separados por comas por caso y devuelve un valor para todo el bloque switch-case.

Por ejemplo:

Texto

Descripción generada automáticamente

La expresión switch hace que el bloque switch-case sea mucho más corto y no usa una instrucción break.

Observe la abreviatura -> después de los casos.

# while Loops

Una declaración de bucle (loop) permite ejecutar repetidamente una declaración o grupo de declaraciones.

Una declaración de bucle while ejecuta repetidamente una declaración de destino siempre que una condición dada sea verdadera.

Ejemplo:

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

Los bucles while verifican la condición x > 0. Si se evalúa como verdadero, ejecuta las declaraciones dentro de su cuerpo. Luego verifica la declaración nuevamente y repite.

Observe la afirmación x--.

Esto decrementa x cada vez que se ejecuta el ciclo y hace que el ciclo se detenga cuando x llega a 0. Sin la declaración, el bucle se ejecutaría para siempre.

# while Loops

Cuando se prueba la expresión y el resultado es falso, se omite el cuerpo del ciclo y se ejecuta la primera declaración después del ciclo while.

Ejemplo:

Texto

Descripción generada automáticamente

Tenga en cuenta que el último método de impresión está fuera del alcance de while.

# for Loops

Otra estructura de bucle es el bucle for. Un ciclo for le permite escribir eficientemente un ciclo que necesita ejecutarse un número específico de veces.

Sintaxis:

**for (initialization; condition; increment/decrement) {   
statement(s)   
}**

Inicialización: la expresión se ejecuta solo una vez durante el comienzo del ciclo

Condición: Se evalúa cada vez que itera el ciclo. El ciclo ejecuta la declaración repetidamente, hasta que esta condición devuelve falso.

Incremento/Decremento: se ejecuta después de cada iteración del bucle.

Texto

Descripción generada automáticamente

Esto inicializa x al valor 1, e imprime repetidamente el valor de x, hasta que la condición x<=5 se vuelve falsa. En cada iteración, se ejecuta la sentencia x++, incrementando x en uno.

Observe el punto y coma (;) después de la inicialización y la condición en la sintaxis.

# for Loops

Puede tener cualquier tipo de condición y cualquier tipo de instrucción de incremento en el ciclo for.

El siguiente ejemplo imprime solo los valores pares entre 0 y 10:

# Texto Descripción generada automáticamente

Un bucle for es mejor cuando se conocen los números inicial y final.

# do...while Loops

# Un ciclo do...while es similar a un ciclo while, excepto que se garantiza que un ciclo do...while se ejecutará al menos una vez.

# Ejemplo:

# Texto Descripción generada automáticamente

# Tenga en cuenta que la condición aparece al final del ciclo, por lo que las declaraciones en el ciclo se ejecutan una vez antes de que se pruebe. Incluso con una condición falsa, el código se ejecutará una vez.

# Ejemplo:

# Interfaz de usuario gráfica, Texto Descripción generada automáticamente con confianza media

# Tenga en cuenta que en los bucles do...while, el while es solo la condición y no tiene un cuerpo en sí mismo.

# Loop Control Statements

# Las sentencias break y continue cambian el flujo de ejecución del ciclo.

# La sentencia break termina el bucle y transfiere la ejecución a la sentencia que sigue inmediatamente al bucle.

# Ejemplo:

# Texto Descripción generada automáticamente La declaración de continuación hace que el bucle omita el resto de su cuerpo y luego

# vuelva a probar inmediatamente su condición antes de reiterar. En otras palabras, hace que el ciclo salte a su próxima iteración. Ejemplo:

# Interfaz de usuario gráfica, Aplicación Descripción generada automáticamente Como puede ver, el código anterior omite el valor de 30, como lo indica la declaración de continuar.

**Arrays**Una matriz es una colección de variables del mismo tipo. Cuando necesite almacenar una lista de valores, como números, puede almacenarlos en una matriz, en lugar de declarar variables separadas para cada número. Para declarar una matriz, debe definir el tipo de los elementos entre corchetes. Por ejemplo, para declarar una matriz de enteros:

# int[ ] arr;

El nombre de la matriz es arr. El tipo de elementos que contendrá es int.

Ahora, debe definir la capacidad de la matriz o la cantidad de elementos que contendrá. Para lograr esto, use la palabra clave **new**

# int[ ] arr = new int[5];

El código anterior declara una matriz de 5 enteros.

En una matriz, los elementos están ordenados y cada uno tiene una posición específica y constante, que se denomina **index** (índice).

Para hacer referencia a los elementos de una matriz, escriba el nombre de la matriz seguido de la posición del índice dentro de un par de corchetes.

Ejemplo:

**arr[2] = 42;**

Esto asigna un valor de 42 al elemento con 2 como su índice.

Tenga en cuenta que los elementos de la matriz se identifican con números de índice de base cero, lo que significa que el índice del primer elemento es 0 en lugar de uno. Entonces, el índice máximo de la matriz int[5] es 4.

# Initializing Arrays

Java proporciona un atajo para crear instancias de arrays de strings y tipos primitivos.

Si ya sabe qué valores insertar en la array, puede usar un array literal (literal de matriz).

Ejemplo de un literal de matriz:

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

Coloque los valores en una lista separada por comas, entre llaves.

El código anterior inicializa automáticamente una matriz que contiene 4 elementos y almacena los valores proporcionados.

A veces, es posible que vea los corchetes colocados después del nombre de la matriz, lo que también funciona, pero la forma preferida es colocar los corchetes después del tipo de datos de la matriz.

# Array Length

Puede acceder a la longitud de una array (la cantidad de elementos que almacena) a través de su propiedad de **length**.

Ejemplo:

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

No olvide que en las matrices, los índices comienzan desde 0. Entonces, en el ejemplo anterior, el último índice es 4.

Ahora que sabemos cómo configurar y obtener elementos de matriz, podemos calcular la suma de todos los elementos de una matriz mediante el uso de bucles.

El ciclo for es el ciclo más utilizado cuando se trabaja con arreglos, ya que podemos usar la longitud del arreglo para determinar cuántas veces ejecutar el ciclo.

Texto

Descripción generada automáticamente

En el código anterior, declaramos una suma variable para almacenar el resultado y le asignamos 0.

Luego usamos un bucle for para iterar a través de la matriz y agregamos el valor de cada elemento a la variable.

La condición del bucle for es x<myArr.length, ya que el índice del último elemento es myArr.length-1.

# Enhanced for Loop

El bucle for mejorado (a veces llamado bucle "for each") se utiliza para atravesar elementos en matrices.

Las ventajas son que elimina la posibilidad de errores y hace que el código sea más fácil de leer.

Ejemplo:

# Texto Descripción generada automáticamente

El bucle for mejorado declara una variable de un tipo compatible con los elementos de la matriz a la que se accede. La variable estará disponible dentro del bloque for y su valor será el mismo que el del elemento de matriz actual.

Entonces, en cada iteración del ciclo, la variable t será igual al elemento correspondiente en la matriz.

Observe los dos puntos después de la variable en la sintaxis.

# Multidimensional Arrays

Los arreglos multidimensionales son arreglos que contienen otros arreglos. La matriz bidimensional es la matriz multidimensional más básica.

Para crear matrices multidimensionales, coloque cada matriz dentro de su propio conjunto de corchetes.

Ejemplo de una matriz bidimensional:

**int[ ][ ] sample = { {1, 2, 3}, {4, 5, 6} };**

Esto declara una matriz con dos matrices como sus elementos.

Para acceder a un elemento en la matriz bidimensional, proporcione dos índices, uno para la matriz y otro para el elemento dentro de esa matriz.

El siguiente ejemplo accede al primer elemento en la segunda matriz de muestra.

# Texto Descripción generada automáticamente

Los dos índices de la matriz se denominan índice de fila e índice de columna.

Puede obtener y establecer los elementos de una matriz multidimensional utilizando el mismo par de corchetes.

Ejemplo:

Texto

Descripción generada automáticamente

La matriz bidimensional anterior contiene tres matrices. La primera matriz tiene tres elementos, la segunda tiene un solo elemento y la última de estas tiene tres elementos.

En Java, no está limitado solo a matrices bidimensionales. Los arreglos se pueden anidar dentro de arreglos en tantos niveles como necesite su programa. Todo lo que necesita para declarar una matriz con más de dos dimensiones es agregar tantos conjuntos de corchetes vacíos como necesite. Sin embargo, estos son más difíciles de mantener.

Recuerde que todos los miembros de la matriz deben ser del mismo tipo.

Problem

Invertir una cadena

Escriba un programa para tomar una cadena como entrada y generar su inversa.

El código dado toma una cadena como entrada y la convierte en una matriz de caracteres, que contiene letras de la cadena como elementos.

Entrada de muestra:

Hola

Salida de muestra:

ereht olleh

Puede recorrer la matriz de caracteres, comenzando desde el final, usando arr.length para obtener el tamaño de la matriz.

import java.util.Scanner;

public class Program

{

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

String text = scanner.nextLine();

char[] arr = text.toCharArray();

//your code goes here

String reverse = "";

for(int i = arr.length - 1; i >= 0; i--){

reverse = reverse + arr[i];

}

System.out.println(reverse);

}

}

**Classes and Objects**

# Object-Orientation

Java utiliza la Programación Orientada a Objetos (POO), un estilo de programación que tiene como objetivo hacer que pensar en la programación se acerque más a pensar en el mundo real.

En OOP, cada objeto es una unidad independiente con una identidad única, tal como lo son los objetos en el mundo real.

Una manzana es un objeto; también lo es una taza. Cada uno tiene su identidad única. Es posible tener dos tazas que parezcan idénticas, pero aún así son objetos separados y únicos.

Los objetos también tienen características, que se utilizan para describirlos.

Por ejemplo, un automóvil puede ser rojo o azul, una taza puede estar llena o vacía, etc. Estas características también se denominan atributos. Un atributo describe el estado actual de un objeto.

En el mundo real, cada objeto se comporta a su manera. El coche se mueve, suena el teléfono, etc.

Lo mismo se aplica a los objetos: el comportamiento es específico del tipo de objeto.

En resumen, en la programación orientada a objetos, cada objeto tiene tres dimensiones: identidad, atributos y comportamiento.

Los atributos describen el estado actual del objeto y lo que el objeto es capaz de hacer se demuestra a través del comportamiento del objeto.

# Classes

Una clase describe lo que será el objeto, pero está separada del objeto mismo.

En otras palabras, las clases se pueden describir como planos, descripciones o definiciones para un objeto. Puede usar la misma clase como modelo para crear varios objetos. El primer paso es definir la clase, que luego se convierte en un modelo para la creación de objetos.

Cada clase tiene un nombre y cada una se usa para definir atributos y comportamiento.

Algunos ejemplos de atributos y comportamiento:

Texto

Descripción generada automáticamente con confianza baja

En otras palabras, un objeto es una instancia de una clase.

# Methods

Los métodos definen el comportamiento. Un método es una colección de instrucciones que se agrupan para realizar una operación. System.out.println() es un ejemplo de un método.

Puede definir sus propios métodos para realizar las tareas deseadas.

Consideremos el siguiente código:

Texto

Descripción generada automáticamente

El código anterior declara un método llamado "sayHello", que imprime un texto y luego se llama en main.

Para llamar a un método, escriba su nombre y luego siga el nombre con un conjunto de paréntesis.

# Calling Methods

Puede llamar a un método tantas veces como sea necesario.

Cuando se ejecuta un método, el código salta a donde está definido el método, ejecuta el código dentro de él, luego regresa y pasa a la siguiente línea.

Ejemplo:

Texto

Descripción generada automáticamente

En casos como el anterior, donde se repite lo mismo una y otra vez, puedes lograr el mismo resultado usando bucles (while o for).

# Method Parameters

También puede crear un método que tome algunos datos, llamados parámetros, junto con él cuando lo llame. Escriba los parámetros entre paréntesis del método.

Por ejemplo, podemos modificar nuestro método sayHello() para tomar y generar un parámetro de cadena.

Texto

Descripción generada automáticamente

El método anterior toma una cadena llamada nombre como parámetro, que se usa en el cuerpo del método. Luego, al llamar al método, pasamos el valor del parámetro entre paréntesis.

Los métodos pueden tomar varios parámetros separados por comas.

Las ventajas de usar métodos en lugar de declaraciones simples incluyen lo siguiente:

- reutilización de código: puede escribir un método una vez y usarlo varias veces, sin tener que volver a escribir el código cada vez.

- parámetros: según los parámetros pasados, los métodos pueden realizar varias acciones.

# The Return Type

La palabra clave return se puede utilizar en métodos para devolver un valor.

Por ejemplo, podríamos definir un método llamado sum que devuelva la suma de sus dos parámetros.

**static int sum(int val1, int val2) {   
return val1 + val2;**   
}

Observe que en la definición del método, definimos el tipo de retorno antes de definir el nombre del método. Para nuestro método de suma, es int, ya que toma dos parámetros del tipo int y devuelve su suma, que también es un int.

La palabra clave estática se discutirá en una lección futura.

Ahora, podemos usar el método en nuestra principal.

Texto

Descripción generada automáticamente

Como el método devuelve un valor, podemos asignarlo a una variable.

Cuando no necesite devolver ningún valor de su método, use la palabra clave void.

Observe la palabra clave void en la definición del método principal; esto significa que main no devuelve nada.

Eche un vistazo al mismo código de nuestra lección anterior con comentarios explicativos, para que pueda comprender mejor cómo funciona:

**// returns an int value 5  
static int returnFive() {   
return 5;   
}   
  
// has a parameter  
static void sayHelloTo(String name) {   
System.out.println("Hello " + name);   
}   
  
// simply prints"Hello World!"  
static void sayHello() {   
System.out.println("Hello World!");   
}**

Habiendo adquirido conocimiento de los tipos y parámetros de devolución de métodos, echemos otro vistazo a la definición del método principal.

**public static void main(String[ ] args)**  
Esta definición indica que el método main toma una array de cadenas como sus parámetros y no devuelve ningún valor.

Vamos a crear un método que tome dos parámetros de tipo int y devuelva el mayor, luego llámelo en main:

**Texto

Descripción generada automáticamente**

Un método puede tener un tipo de parámetro (o parámetros) y devolver otro tipo diferente. Por ejemplo, puede tomar dos dobles y devolver un int.

# Creating Classes

Para crear sus propios objetos personalizados, primero debe crear las clases correspondientes. Esto se logra haciendo clic derecho en la carpeta src en Eclipse y seleccionando Create->New->Class. Asigne un nombre a su clase y haga clic en Finalizar para agregar la nueva clase a su proyecto:

Como puede ver, Eclipse ya agregó el código inicial para la clase.

Ahora vamos a crear un método simple en nuestra nueva clase.

Animal.java

**public class Animal {   
void bark() {   
System.out.println("Woof-Woof");   
}   
}**

Declaramos un método bark() en nuestra clase Animal. Ahora, para usar la clase y sus métodos, necesitamos declarar un objeto de esa clase.

# Creating Objects

Vayamos a nuestro main y creemos un nuevo objeto de nuestra clase.

MiClase.java

Texto

Descripción generada automáticamente

Ahora bien, dog es un objeto de tipo Animal. Así podemos llamar a su método bark(), usando el nombre del objeto y un punto.

La notación de puntos se utiliza para acceder a los atributos y métodos del objeto.

¡Acabas de crear tu primer objeto!

## Drag and drop from the options below to create an object of the A class in the B class and call its "test" method.

public class A {

public void test() {

System.out.println("Hi");

}

}

class B {

public static void main(String args[ ]) {

A obj = new A();

obj.test();

}

}

# Defining Attributes

Una clase tiene atributos y métodos. Los atributos son básicamente variables dentro de una clase.

Vamos a crear una clase llamada Vehicle, con sus correspondientes atributos y métodos.

**public class Vehicle {   
int maxSpeed;   
int wheels;   
String color;   
double fuelCapacity;   
  
void horn() {   
System.out.println("Beep!");   
}   
}**

maxSpeed, wheels, color y fuelCapacity son los atributos de nuestra clase Vehicle, y horn() es el único método.

Puede definir tantos atributos y métodos como sea necesario.

A continuación, podemos crear múltiples objetos de nuestra clase Vehicle y usar la sintaxis de puntos para acceder a sus atributos y métodos.

Texto

Descripción generada automáticamente

**Access Modifiers**

Ahora analicemos la palabra clave **public** frente al método principal.

**public static void main(String[ ] args)**

public es un **access modifier**, lo que significa que se utiliza para establecer el nivel de acceso. Puede utilizar modificadores de acceso para clases, atributos y métodos.

Para las clases, los modificadores disponibles son public or default

(en blanco), como se describe a continuación:

**public:** La clase es accesible por cualquier otra clase.

**default:** solo las clases del mismo paquete pueden acceder a la clase.

Las siguientes opciones están disponibles para atributos y métodos:

**default:** una variable o método declarado sin modificador de control de acceso está disponible para cualquier otra clase en el mismo paquete.

**public:** Accesible desde cualquier otra clase.

**protected:** proporciona el mismo acceso que el modificador de acceso predeterminado, con la adición de que las subclases pueden acceder a métodos y variables protegidos de la superclase (las subclases y las superclases se tratan en las próximas lecciones).

**private**: Accesible solo dentro de la propia clase declarada.

Ejemplo:

**public class Vehicle {   
private int maxSpeed;   
private int wheels;   
private String color;   
private double fuelCapacity;   
  
public void horn() {   
System.out.println("Beep!");   
}   
}**

Es una buena práctica mantener privadas las variables dentro de una clase. Las variables son accesibles y modificadas usando Getters y Setters.

# Getters & Setters

Getters y Setters se utilizan para proteger eficazmente sus datos, especialmente al crear clases. Para cada variable, el método get devuelve su valor, mientras que el método set establece el valor.

Los Getters comienzan con get, seguido del nombre de la variable, con la primera letra del nombre de la variable en mayúscula.

Los setters comienzan con set, seguido del nombre de la variable, con la primera letra del nombre de la variable en mayúscula.

Ejemplo:

**public class Vehicle {   
private String color;   
  
// Getter   
public String getColor() {   
return color;   
}   
  
// Setter   
public void setColor(String c) {   
this.color = c;   
}   
}**

El método getter devuelve el valor del atributo.

El método setter toma un parámetro y lo asigna al atributo.

La palabra clave **this** se utiliza para referirse al objeto actual. Básicamente, this.color es el atributo de color del objeto actual.

## Drag and drop from the options below to define the set and get methods.

class A {

private int x;

public int getX() {

return x;

}

public void setX(int x) {

this.x = x;

}

}

Una vez que nuestro getter y setter han sido definidos, podemos usarlo en nuestro main

Texto

Descripción generada automáticamente

Getters y setters nos permiten tener control sobre los valores. Puede, por ejemplo, validar el valor dado en el setter antes de establecer el valor.

Getters y setters son bloques de construcción fundamentales para la encapsulación, que se tratarán en el próximo módulo.

# Constructors

Los constructores son métodos especiales que se invocan cuando se crea un objeto y se utilizan para inicializarlos.

Se puede usar un constructor para proporcionar valores iniciales para los atributos del objeto.

- El nombre de un constructor debe ser el mismo que su nombre de clase.

- Un constructor no debe tener un tipo de retorno explícito.

Ejemplo de un constructor:

**public class Vehicle {   
private String color;   
Vehicle() {   
color = "Red";   
}   
}**

El método Vehicle() es el constructor de nuestra clase, por lo que cada vez que se crea un objeto de esa clase, el atributo de color se establecerá en "Rojo".

Un constructor también puede tomar parámetros para inicializar atributos.

**public class Vehicle {   
private String color;   
Vehicle(String c) {   
color = c;   
}   
}**

Puede pensar en los constructores como métodos que configurarán su clase de forma predeterminada, por lo que no necesita repetir el mismo código cada vez.

# Using Constructors

Se llama al constructor cuando crea un objeto usando la palabra clave **new**.

Ejemplo:

**public class MyClass {   
public static void main(String[ ] args) {   
Vehicle v = new Vehicle("Blue");   
}   
}**

Esto llamará al constructor, que establecerá el atributo de color en "Blue".

Una sola clase puede tener múltiples constructores con diferentes números de parámetros.

Los métodos setter dentro de los constructores se pueden usar para establecer los valores de los atributos.

**public class Vehicle {   
private String color;   
  
Vehicle() {   
this.setColor("Red");   
}   
Vehicle(String c) {   
this.setColor(c);   
}   
  
// Setter   
public void setColor(String c) {   
this.color = c;   
}   
}**

La clase anterior tiene dos constructores, uno sin ningún parámetro que establezca el atributo de color en un valor predeterminado de "Red" y otro constructor que acepta un parámetro y lo asigna al atributo.

Ahora, podemos usar los constructores para crear objetos de nuestra clase.

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente con confianza media

Java proporciona automáticamente un constructor predeterminado, por lo que todas las clases tienen un constructor, ya sea que esté específicamente definido o no.

**Value & Reference Types**

# Value Types

Los tipos de valor son los tipos básicos e incluyen byte, short, int, long, float, double, boolean y char.

Estos tipos de datos almacenan los valores que se les asignan en las ubicaciones de memoria correspondientes.

Entonces, cuando los pasa a un método, básicamente opera en el valor de la variable, en lugar de en la variable misma.

Ejemplo:

Texto

Descripción generada automáticamente

El método del ejemplo anterior toma el valor de su parámetro, por lo que la variable original no se ve afectada y su valor sigue siendo 5.

# Reference Types

Un tipo de referencia almacena una referencia (o dirección) a la ubicación de la memoria donde se almacenan los datos correspondientes.

Cuando crea un objeto usando el constructor, crea una variable de referencia.

Por ejemplo, considere tener una clase Person definida:

Texto

Descripción generada automáticamente

El método celebrateBirthday toma un objeto Persona como su parámetro e incrementa su atributo.

Debido a que j es un tipo de referencia, el método afecta al objeto mismo y puede cambiar el valor real de su atributo.

Arrays y Strings también son tipos de datos de referencia.

# The Math Class

El JDK define una serie de clases útiles, una de ellas es la clase Math, que proporciona métodos predefinidos para operaciones matemáticas.

No necesita crear un objeto de la clase Math para usarlo. Para acceder a él, simplemente escriba Math. y el método correspondiente.

Math.abs() devuelve el valor absoluto de su parámetro.

Texto

Descripción generada automáticamente

Math.ceil() redondea un valor de punto flotante al valor entero más cercano. El valor redondeado se devuelve como un double.

Texto

Descripción generada automáticamente

De manera similar, Math.floor() redondea un valor de punto flotante al valor entero más cercano.



Math.max() devuelve el mayor de sus parámetros.

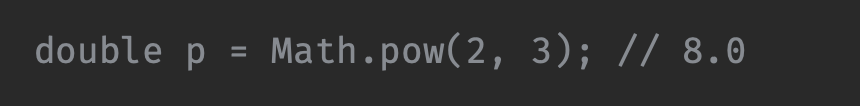
Imagen que contiene Texto

Descripción generada automáticamente

Por el contrario, Math.min() devuelve el parámetro más pequeño.



Math.pow() toma dos parámetros y devuelve el primer parámetro elevado a la potencia del segundo parámetro.



Hay una serie de otros métodos disponibles en la clase de Math, que incluyen:

sqrt() para raíz cuadrada, sin() para seno, cos() para coseno y otros.

# Static

Cuando declara una variable o un método como estático, pertenece a la clase, en lugar de a una instancia específica. Esto significa que solo existe una instancia de un miembro estático, incluso si crea varios objetos de la clase o si no crea ninguno. Será compartido por todos los objetos.

Ejemplo:

**public class Counter {   
public static int COUNT=0;   
Counter() {   
COUNT++;   
}   
}**

La variable COUNT será compartida por todos los objetos de esa clase.

Ahora, podemos crear objetos de nuestra clase Counter en main y acceder a la variable estática.

Texto

Descripción generada automáticamente

El resultado es 2, porque la variable COUNT es estática y se incrementa en uno cada vez que se crea un nuevo objeto de la clase Counter. En el código anterior, creamos 2 objetos.

También puede acceder a la variable estática utilizando cualquier objeto de esa clase, como c1.COUNT.

Es una práctica común usar mayúsculas al nombrar una variable estática, aunque no es obligatorio.

El mismo concepto se aplica a los métodos estáticos.

**public class Vehicle {   
public static void horn() {   
System.out.println("Beep");   
}   
}**

Ahora, el método horn se puede llamar sin crear un objeto:

Texto

Descripción generada automáticamente

Otro ejemplo de métodos estáticos son los de la clase Math, por lo que puedes llamarlos sin crear un objeto Math.

Además, el método principal siempre debe ser estático.

# final

Use la palabra clave final para marcar una variable constante, de modo que se pueda asignar solo una vez.

Ejemplo:

Texto

Descripción generada automáticamente

PI es ahora una constante. Cualquier intento de asignarle un valor provocará un error.

Los métodos y las clases también se pueden marcar como finales. Esto sirve para restringir los métodos para que no puedan anularse y las clases para que no puedan subclasificarse.

Estos conceptos se tratarán en el siguiente módulo.

# Packages

Los paquetes se utilizan para evitar conflictos de nombres y para controlar el acceso a las clases.

Un paquete se puede definir como un grupo formado por tipos similares de clases, junto con subpaquetes.

Crear un paquete en Java es bastante fácil. Simplemente haga clic con el botón derecho en su directorio src y haga clic en Nuevo->Paquete. Asigne un nombre a su paquete y haga clic en Finalizar.

Notará que el nuevo paquete aparece en el directorio del proyecto. Ahora puede mover y crear clases dentro de ese paquete. Hemos trasladado nuestras clases de vehículos, contadores y animales a las muestras de paquetes.

Cuando mueve/crea una clase en su paquete, el siguiente código aparecerá en la parte superior de la lista de archivos.

**package** samples;

Esto indica el paquete al que pertenece la clase.

Ahora, necesitamos importar las clases que están dentro de un paquete en nuestro principal para poder usarlas.

El siguiente ejemplo muestra cómo utilizar la clase Vehicle del paquete de ejemplos.

import samples.Vehicle;  
  
**class MyClass {   
public static void main(String[ ] args) {   
Vehicle v1 = new Vehicle();   
v1.horn();   
}   
}**

Dos resultados principales ocurren cuando una clase se coloca en un paquete. Primero, el nombre del paquete se convierte en parte del nombre de la clase. En segundo lugar, el nombre del paquete debe coincidir con la estructura de directorios donde reside el archivo de clase correspondiente.

Use un comodín para importar todas las clases en un paquete.

Por ejemplo, import samples.\* importará todas las clases en el paquete de samples.

**Encapsulation**

Hay 4 conceptos básicos en OOP: encapsulación, herencia, polimorfismo y abstracción.

La idea detrás de la encapsulación es garantizar que los detalles de implementación no sean visibles para los usuarios. Las variables de una clase estarán ocultas de las otras clases, accesibles solo a través de los métodos de la clase actual. Esto se llama ocultación de datos.

Para lograr la encapsulación en Java, declare las variables de la clase como privadas y proporcione métodos públicos setter y getter para modificar y ver los valores de las variables.

**For example:**

class BankAccount {   
private double balance=0;   
public void deposit(double x) {   
if(x > 0) {   
balance += x;   
}   
}   
}

Esta implementación oculta la variable balance, permitiendo el acceso a la misma únicamente a través del método de depósito, que valida el monto a depositar antes de modificar la variable.

En resumen, la encapsulación proporciona los siguientes beneficios:

- Control de la forma en que se accede a los datos o se modifican

- Código más flexible y fácil de cambiar

- Posibilidad de cambiar una parte del código sin afectar otras partes

# Inheritance

La herencia es el proceso que permite que una clase adquiera las propiedades (métodos y variables) de otra. Con la herencia, la información se coloca en un orden jerárquico más manejable.

La clase que hereda las propiedades de otra es la subclase (también llamada clase derivada o clase secundaria); la clase cuyas propiedades se heredan es la superclase (clase base o clase padre).

Para heredar de una clase, use la palabra clave extends.

Este ejemplo muestra cómo heredar la clase Perro de la clase Animal.

class Dog extends Animal {   
// some code   
}

Aquí, Dog es la subclase y Animal es la superclase.

Cuando una clase se hereda de otra clase, hereda todas las variables y métodos no privados de la superclase.

Ejemplo:

class Animal {   
protected int legs;   
public void eat() {   
System.out.println("Animal eats");   
}   
}   
  
class Dog extends Animal {   
Dog() {   
legs = 4;   
}   
}

Como puede ver, la clase Dog hereda la variable legs de la clase Animal.

Ahora podemos declarar un objeto Dog y llamar al método eat de su superclase:

Texto

Descripción generada automáticamente

Recuerde el modificador de acceso protected, que hace que los miembros sean visibles solo para las subclases.

Los constructores no son métodos miembro, por lo que las subclases no los heredan.

Sin embargo, se llama al constructor de la superclase cuando se crea una instancia de la subclase.

Ejemplo:

Texto

Descripción generada automáticamente

Puede acceder a la superclase desde la subclase utilizando la palabra clave super.

Por ejemplo, super.var accede al miembro var de la superclase.

Polymorphism

El polimorfismo, que hace referencia a la idea de "tener muchas formas", se da cuando existe una jerarquía de clases relacionadas entre sí por herencia.

Una llamada a un método miembro hará que se ejecute una implementación diferente, dependiendo del tipo de objeto que invoca el método.

He aquí un ejemplo: Dog y Cat son clases que heredan de la clase Animal. Cada clase tiene su propia implementación del método makeSound().

class Animal {   
public void makeSound() {   
System.out.println("Grr...");   
}   
}   
class Cat extends Animal {   
public void makeSound() {   
System.out.println("Meow");   
}   
}   
class Dog extends Animal {   
public void makeSound() {   
System.out.println("Woof");   
}   
}

Como todos los objetos Cat y Dog son objetos Animal, podemos hacer lo siguiente en main:

public static void main(String[ ] args) {   
Animal a = new Dog();   
Animal b = new Cat();   
}

Hemos creado dos variables de referencia de tipo Animal y las hemos apuntado a los objetos Cat y Dog.

Ahora, podemos llamar a los métodos makeSound().

Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente

Como la variable de referencia a contiene un objeto Dog, se llamará al método makeSound() de la clase Dog.

Lo mismo se aplica a la variable b.

Esto demuestra que puede usar la variable Animal sin saber realmente que contiene un objeto de la subclase.

Esto es muy útil cuando tiene varias subclases de la superclase.

Method Overriding

Como vimos en la lección anterior, una subclase puede definir un comportamiento que es específico del tipo de subclase, lo que significa que una subclase puede implementar un método de clase principal en función de sus requisitos.

Esta función se conoce como anulación de métodos.

Ejemplo:

Texto

Descripción generada automáticamente

En el código de arriba, la clase Cat anula el método makeSound() de su superclase Animal.

Reglas para la anulación de métodos:

- Debe tener el mismo tipo de retorno y argumentos

- El nivel de acceso no puede ser más restrictivo que el nivel de acceso del método anulado (Ejemplo: si el método de superclase se declara público, el método anulado en la subclase no puede ser ni privado ni protegido)

- Un método declarado final o estático no se puede anular

- Si un método no se puede heredar, no se puede anular

- Los constructores no pueden ser anulados

La anulación de métodos también se conoce como polimorfismo en tiempo de ejecución.

# Method Overloading

Cuando los métodos tienen el mismo nombre, pero diferentes parámetros, se conoce como sobrecarga de métodos.

Esto puede ser muy útil cuando necesita la misma funcionalidad de método para diferentes tipos de parámetros.

El siguiente ejemplo ilustra un método que devuelve el máximo de sus dos parámetros.

# int max(int a, int b) { if(a > b) { return a; } else { return b; } }

El método que se muestra arriba solo funcionará para parámetros de tipo entero.

Sin embargo, es posible que también queramos usarlo para dobles. Para eso, necesitas sobrecargar el método max:

Texto

Descripción generada automáticamente

Ahora, nuestro método max también funcionará con dobles.

Un método sobrecargado debe tener una lista de argumentos diferente; los parámetros deben diferir en su tipo, número o ambos.

Otro nombre para la sobrecarga de métodos es polimorfismo en tiempo de compilación.

# Abstraction

La abstracción de datos proporciona al mundo exterior solo información esencial, en un proceso de representación de características esenciales sin incluir detalles de implementación.

Un buen ejemplo del mundo real es un libro. Cuando escuchas el término libro, no conoces los detalles exactos, como el número de páginas, el color o el tamaño, pero entiendes la idea o abstracción de un libro.

El concepto de abstracción es que nos enfocamos en las cualidades esenciales, en lugar de las características específicas de un ejemplo en particular.

En Java, la abstracción se logra utilizando clases e interfaces **abstract**.

Una clase abstracta se define utilizando la palabra clave **abstract**.

- Si una clase se declara abstracta, no se puede instanciar (no se pueden crear objetos de ese tipo).

- Para usar una clase abstracta, debe heredarla de otra clase.

- Cualquier clase que contenga un método abstracto debe definirse como abstracta.

Un método abstracto es un método que se declara sin una implementación (sin llaves y seguido de un punto y coma): abstract void walk();

**Abstract Class**

Por ejemplo, podemos definir nuestra clase Animal como abstracta:

**abstract class Animal {   
int legs = 0;   
abstract void makeSound();   
}**

El método makeSound también es abstracto, ya que no tiene implementación en la superclase.

Podemos heredar de la clase Animal y definir el método makeSound() para la subclase:

Texto

Descripción generada automáticamente

Cada Animal hace un sonido, pero cada uno tiene una forma diferente de hacerlo. Es por eso que definimos una clase abstracta Animal y dejamos la implementación de cómo hacen los sonidos a las subclases.

Esto se usa cuando no hay una definición significativa para el método en la superclase.

# Interfaces

Una interfaz es una clase completamente abstracta que contiene solo métodos abstractos.

Algunas especificaciones para las interfaces:

- Definido usando la palabra clave de **interface**.

- Puede contener solo variables finales estáticas.

- No puede contener un constructor porque no se pueden crear instancias de las interfaces.

- Las interfaces pueden extender otras interfaces.

- Una clase puede implementar cualquier número de interfaces.

Un ejemplo de una interfaz simple:

# interface Animal { public void eat(); public void makeSound(); }

Las interfaces tienen las siguientes propiedades:

- Una interfaz es implícitamente abstracta. No necesita usar la palabra clave abstract al declarar una interfaz.

- Cada método en una interfaz también es implícitamente abstracto, por lo que no se necesita la palabra clave abstract.

- Los métodos en una interfaz son implícitamente públicos.

Una clase puede heredar de una sola superclase, ¡pero puede implementar múltiples interfaces!

Use la palabra clave **implements** para usar una interfaz con su clase.

Texto

Descripción generada automáticamente

Cuando implementa una interfaz, debe anular todos sus métodos.

# Type Casting

Asignar un valor de un tipo a una variable de otro tipo se conoce como Type Casting.

Para convertir un valor a un tipo específico, coloque el tipo entre paréntesis y colóquelo delante del valor.

Ejemplo:

**Interfaz de usuario gráfica, Texto

Descripción generada automáticamente**

El código anterior convierte el valor 3.14 en un número entero, con 3 como valor resultante.

Otro ejemplo:

**Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente**

Java admite la conversión automática de tipos de enteros a puntos flotantes, ya que no hay pérdida de precisión.

Por otro lado, la conversión de tipos es obligatoria cuando se asignan valores de coma flotante a variables enteras.

Para las clases, hay dos tipos de casting.

Upcasting

Puede convertir una instancia de una subclase en su superclase.

Considere el siguiente ejemplo, asumiendo que Cat es una subclase de Animal.

Animal a = new Cat();

Java automáticamente convirtió la variable de tipo Cat al tipo Animal.

## Downcasting

La conversión de un objeto de una superclase a su subclase se denomina conversión descendente.

Ejemplo:

# Animal a = new Cat(); ((Cat)a).makeSound();

Esto intentará convertir la variable **a** al tipo Cat y llamar a su método makeSound().

¿Por qué el upcasting es automático y el downcasting manual? Bueno, upcasting nunca puede fallar. Pero si tienes un grupo de diferentes animales y quieres convertirlos a todos en un gato, entonces existe la posibilidad de que algunos de estos animales sean en realidad perros, por lo que el proceso falla.

# Anonymous Classes

Las clases anónimas son una forma de ampliar las clases existentes sobre la marcha.

Por ejemplo, considere tener una Machine de clase:

**class Machine {   
public void start() {   
System.out.println("Starting...");   
}   
}**

Al crear el objeto Machine, podemos cambiar el método de inicio sobre la marcha.

Texto

Descripción generada automáticamente

Después de la llamada al constructor, abrimos las llaves y anulamos la implementación del método de inicio sobre la marcha.

La anotación @Override se usa para que su código sea más fácil de entender, porque hace que sea más obvio cuando se anulan los métodos.

La modificación es aplicable solo al objeto actual y no a la clase en sí. Entonces, si creamos otro objeto de esa clase, la implementación del método de inicio será la definida en la clase.

**Texto

Descripción generada automáticamente**

# Inner Classes

Java admite clases anidadas; una clase puede ser miembro de otra clase.

Crear una clase interna es bastante simple. Solo escribe una clase dentro de una clase. A diferencia de una clase, una clase interna puede ser privada. Una vez que declara privada una clase interna, no se puede acceder a ella desde un objeto fuera de la clase.

Ejemplo:

**Texto

Descripción generada automáticamente**

La clase Robot tiene un Brain de clase interno. La clase interna puede acceder a todas las variables miembro y métodos de su clase externa, pero no se puede acceder desde ninguna clase externa.

# Comparing Objects

Recuerde que cuando crea objetos, las variables almacenan referencias a los objetos.

Entonces, cuando compara objetos usando el operador de prueba de igualdad (==), en realidad compara las referencias y no los valores de los objetos.

Ejemplo:

**Texto

Descripción generada automáticamente**

A pesar de tener dos objetos con el mismo nombre, la prueba de igualdad devuelve falso, porque tenemos dos objetos diferentes (dos referencias o ubicaciones de memoria diferentes).

# equals()

Cada objeto tiene un método equals() predefinido que se utiliza para las pruebas de igualdad semántica.

Pero, para que funcione para nuestras clases, debemos anularlo y verificar las condiciones que necesitamos.

Existe una forma sencilla y rápida de generar el método equals(), además de escribirlo manualmente.

Simplemente haga clic derecho en su clase, vaya a Source->Generate hashCode() and equals()...

Esto creará automáticamente los métodos necesarios.

# class Animal { String name; Animal(String n) { name = n; } @Override public int hashCode() { final int prime = 31; int result = 1; result = prime \* result + ((name == null) ? 0 : name.hashCode()); return result; } @Override public boolean equals(Object obj) { if (this == obj) return true; if (obj == null) return false; if (getClass() != obj.getClass()) return false; Animal other = (Animal) obj; if (name == null) { if (other.name != null) return false; } else if (!name.equals(other.name)) return false; return true; } }

El método hashCode() generado automáticamente se utiliza para determinar dónde almacenar el objeto internamente. Siempre que implemente equals, también DEBE implementar hashCode.

Podemos ejecutar la prueba nuevamente, usando el método de igualdad:

**Texto

Descripción generada automáticamente**

Puede usar el mismo menú para generar otros métodos útiles, como getters y setters para sus atributos de clase**.**

# Enums

Un Enum es un tipo especial usado para definir colecciones de constantes.

Aquí hay un ejemplo simple de Enum:

# enum Rank { SOLDIER, SERGEANT, CAPTAIN }

Tenga en cuenta que los valores están separados por comas.

Puede hacer referencia a las constantes en la enumeración anterior con la sintaxis de puntos.

**Rank a = Rank**.SOLDIER**;**

Básicamente, las enums definen variables que representan miembros de un conjunto fijo.

Después de declarar un Enum, podemos verificar los valores correspondientes con, por ejemplo, una declaración de switch.

Texto

Descripción generada automáticamente

Siempre debe usar Enums cuando una variable (especialmente un parámetro de método) solo puede tomar uno de un pequeño conjunto de valores posibles.

Si usa Enums en lugar de números enteros (o códigos de cadena), aumenta la verificación en tiempo de compilación y evita que los errores pasen constantes no válidas, y documenta qué valores son legales para usar.

Algunos ejemplos de usos de Enum incluyen nombres de meses, días de la semana, baraja de cartas, etc.

# Java API

La API de Java es una colección de clases e interfaces que se han escrito para su uso.

La documentación de la API de Java con todas las API disponibles se puede encontrar en el sitio web de Oracle en

http://docs.oracle.com/javase/7/docs/api/

Una vez que localice el paquete que desea usar, debe importarlo a su código.

El paquete se puede importar utilizando la palabra clave de import.

Por ejemplo:

import java.awt.\*;

El paquete awt contiene todas las clases para crear interfaces de usuario y pintar gráficos e imágenes.

El carácter comodín (\*) se utiliza para importar todas las clases del paquete.

# Shapes You are working on a graphical app, which includes multiple different shapes. The given code declares a base Shape class with an abstract area() method and a width attribute. You need to create two Shape subclasses, Square and Circle, which initialize the width attribute using their constructor, and define their area() methods. The area() for the Square class should output the area of the square (the square of the width), while for the Circle, it should output the area of the given circle (PI\*width\*width). The code in main creates two objects with the given user input and calls the area() methods. Sample Input: 5 2 Sample Output: 25 12.566370614359172 The area of the square is 5\*5=25, while the area of the circle is PI\*2\*2=12.566370614359172

Use the **Math.PI**constant for the area calculation of the circle.

import java.util.Scanner;

abstract class Shape {

int width;

abstract void area();

}

//your code goes here

class Square extends Shape {

Square(int x) {

this.width = x;

}

void area() {

//return x \* x;

System.out.println(width \* width);

}

}

class Circle extends Shape {

Circle(int y){

this.width = y;

}

public void area(){

System.out.println(Math.PI\*width\*width);

}

}

public class Program {

public static void main(String[ ] args) {

Scanner sc = new Scanner(System.in);

int x = sc.nextInt();

int y = sc.nextInt();

Square a = new Square(x);

Circle b = new Circle(y);

a.area();

b.area();

}

}

# Exceptions

Una excepción es un problema que ocurre durante la ejecución del programa. Las excepciones causan la terminación anormal del programa.

El manejo de excepciones es un mecanismo poderoso que maneja los errores en tiempo de ejecución para mantener el flujo normal de la aplicación.

Una excepción puede ocurrir por muchas razones diferentes. Algunos ejemplos:

- Un usuario ha introducido datos no válidos.

- No se puede encontrar un archivo que necesita ser abierto.

- Se ha perdido una conexión de red en medio de las comunicaciones.

- Memoria insuficiente y otros problemas relacionados con los recursos físicos.

Como puede ver, las excepciones son causadas por errores del usuario, errores del programador o problemas de recursos físicos. Sin embargo, un programa bien escrito debe manejar todas las posibles excepciones.

# Exception Handling

Las excepciones se pueden detectar mediante una combinación de las palabras clave **try y catch**.

Se coloca un bloque try/catch alrededor del código que podría generar una excepción.

**Syntax:**

**try** {   
//some code   
} **catch (Exception e)** {   
//some code to handle errors   
}

Una declaración de captura implica declarar el tipo de excepción que está tratando de capturar. Si se produce una excepción en el bloque try, se comprueba el bloque catch que sigue al try. Si el tipo de excepción que ocurrió se enumera en un bloque catch, la excepción se pasa al bloque catch de la misma manera que se pasa un argumento a un parámetro de método.

El tipo de excepción se puede utilizar para capturar todas las posibles excepciones.

El siguiente ejemplo demuestra el manejo de excepciones al intentar acceder a un índice de matriz que no existe:

Texto

Descripción generada automáticamente

Sin el bloque try/catch, este código debería bloquear el programa, ya que a[5] no existe.

Observe la declaración (Exception e) en el bloque catch: se usa para capturar todas las excepciones posibles.

# throw

La palabra clave throw le permite generar manualmente excepciones a partir de sus métodos. Algunos de los numerosos tipos de excepción disponibles incluyen IndexOutOfBoundsException, IllegalArgumentException, ArithmeticException, etc.

Por ejemplo, podemos lanzar una ArithmeticException en nuestro método cuando el parámetro es 0.

Texto

Descripción generada automáticamente

La declaración throws en la definición del método define el tipo de Excepción(es) que el método puede lanzar.

A continuación, la palabra clave throw lanza la excepción correspondiente, junto con un mensaje personalizado.

Si llamamos al método div con el segundo parámetro igual a 0, lanzará una excepción ArithmeticException con el mensaje "División por cero".

Se pueden definir varias excepciones en la instrucción throws mediante una lista separada por comas.

Un solo bloque de prueba puede contener varios bloques de captura que manejan diferentes excepciones por separado.

**Example:**

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

Todos los bloques catch deben ordenarse del más específico al más general.

Siguiendo las excepciones específicas, puede usar el tipo de excepción para manejar todas las demás excepciones como la última captura.

# Threads

Java es un lenguaje de programación de subprocesos múltiples. Esto significa que nuestro programa puede hacer un uso óptimo de los recursos disponibles al ejecutar dos o más componentes al mismo tiempo, con cada componente manejando una tarea diferente.

Puede subdividir operaciones específicas dentro de una sola aplicación en subprocesos individuales que se ejecutan en paralelo.

El siguiente diagrama muestra el ciclo de vida de un hilo.

Imagen de la pantalla de un celular con letras

Descripción generada automáticamente con confianza baja

Hay dos formas de crear un hilo.

1. Ampliar la clase Thread

Herede de la clase Thread, anule su método run() y escriba la funcionalidad del hilo en el método run().

Luego, crea un nuevo objeto de su clase y llama a su método de inicio para ejecutar el hilo.

Ejemplo:

Texto

Descripción generada automáticamente

Como puede ver, nuestra clase Loader extiende la clase Thread y anula su método run().

Cuando creamos el objeto obj y llamamos a su método start(), las declaraciones del método run() se ejecutan en un subproceso diferente.

Cada subproceso de Java se prioriza para ayudar al sistema operativo a determinar el orden en el que programar los subprocesos. Las prioridades van del 1 al 10, y cada subproceso tiene como valor predeterminado la prioridad 5. Puede establecer la prioridad del subproceso con el método setPriority().

La otra forma de crear subprocesos es implementar la interfaz Runnable.

Implemente el método run(). Luego, cree un nuevo objeto Thread, pase la clase Runnable a su constructor e inicie el Thread llamando al método start().

Ejemplo:

Texto

Descripción generada automáticamente

El método Thread.sleep() pausa un hilo durante un período de tiempo específico. Por ejemplo, llamar a Thread.sleep(1000); pausa el hilo por un segundo. Tenga en cuenta que Thread.sleep() lanza una InterruptedException, así que asegúrese de rodearla con un bloque try/catch.

Puede parecer que implementar la interfaz Runnable es un poco más complejo que extender desde la clase Thread. Sin embargo, implementar la interfaz Runnable es la forma preferida de iniciar un subproceso, ya que también le permite extenderse desde otra clase.

# Types of Exceptions

Hay dos tipos de excepciones, marcadas y no marcadas (también llamado tiempo de ejecución). La principal diferencia es que las excepciones verificadas se verifican cuando se compilan, mientras que las excepciones no verificadas se verifican en tiempo de ejecución.

Como se mencionó en nuestra lección anterior, Thread.sleep() lanza una InterruptedException. Este es un ejemplo de una excepción comprobada. Su código no se compilará hasta que haya manejado la excepción.

public class MyClass {   
public static void main(String[ ] args) {   
try {   
**Thread.sleep(1000)**;   
} catch (InterruptedException e) {   
//some code   
}   
}   
}

Hemos visto ejemplos de excepciones no verificadas, que se verifican en tiempo de ejecución, en lecciones anteriores.

Ejemplo (al intentar dividir por 0):

Texto

Descripción generada automáticamente

Es bueno conocer los tipos de excepciones porque pueden ayudarlo a depurar su código más rápido.

# ArrayList

La API de Java proporciona clases especiales para almacenar y manipular grupos de objetos.

Una de esas clases es ArrayList. Las matrices estándar de Java tienen una longitud fija, lo que significa que una vez creadas, no pueden expandirse ni reducirse.

Por otro lado, las ArrayLists se crean con un tamaño inicial, pero cuando se supera este tamaño, la colección se amplía automáticamente.

Cuando se eliminan objetos, ArrayList puede reducir su tamaño. Tenga en cuenta que la clase ArrayList está en el paquete java.util, por lo que es necesario importarla antes de usarla.

Cree una ArrayList como lo haría con cualquier objeto.

import **java.util.ArrayList;**  
//...   
**ArrayList** colors = new **ArrayList**();

Opcionalmente, puede especificar una capacidad y tipo de objetos que contendrá ArrayList:

ArrayList<**String**> colors = new ArrayList<**String**>(**10**);

El código anterior define una ArrayList de cadenas con 10 como tamaño inicial.

ArrayLists almacena objetos. Por lo tanto, el tipo especificado debe ser un tipo de clase. No puede pasar, por ejemplo, int como tipo de objeto. En su lugar, utilice los tipos de clase especiales que correspondan al tipo de valor deseado, como Integer para int, Double para double, etc.

La clase ArrayList proporciona varios métodos útiles para manipular sus objetos.

El método add() agrega nuevos objetos a ArrayList. Por el contrario, el método remove() elimina objetos de ArrayList.

Ejemplo:

Texto

Descripción generada automáticamente

Otros métodos útiles incluyen los siguientes:

- contains (): devuelve verdadero si la lista contiene el elemento especificado

- get(int index): Devuelve el elemento en la posición especificada en la lista

- size(): Devuelve el número de elementos de la lista

- clear(): Elimina todos los elementos de la lista

Nota: Al igual que con las matrices, la indexación comienza con 0.

# LinkedList

LinkedList es muy similar en sintaxis a ArrayList.

Puede cambiar fácilmente una ArrayList a una LinkedList cambiando el tipo de objeto.

Texto

Descripción generada automáticamente

No puede especificar una capacidad inicial para LinkedList.

# LinkedList vs. ArrayList

La diferencia más notable entre LinkedList y ArrayList está en la forma en que almacenan objetos.

ArrayList es mejor para almacenar y acceder a datos, ya que es muy similar a una matriz normal.

LinkedList es mejor para manipular datos, como realizar numerosas inserciones y eliminaciones.

Además de almacenar el objeto, LinkedList almacena la dirección de memoria (o enlace) del elemento que le sigue. Se llama LinkedList porque cada elemento contiene un enlace al elemento vecino.

Imagen que contiene Forma

Descripción generada automáticamente

Puede usar el bucle for mejorado para iterar sobre sus elementos.

Texto

Descripción generada automáticamente

Resumen:

- Utilice una ArrayList cuando necesite un acceso rápido a sus datos.

- Utilice una LinkedList cuando necesite realizar una gran cantidad de inserciones y/o eliminaciones.

# HashMap

Las matrices y las listas almacenan elementos como colecciones ordenadas, con cada elemento dado un índice entero.

HashMap se utiliza para almacenar colecciones de datos como pares de clave y valor. Un objeto se utiliza como clave (índice) para otro objeto (el valor).

Los métodos put, remove y get se utilizan para agregar, eliminar y acceder a valores en HashMap.

Ejemplo:

Texto

Descripción generada automáticamente

Hemos creado un HashMap con cadenas como claves y enteros como valores.

Utilice el método get y la clave correspondiente para acceder a los elementos de HashMap.

Un HashMap no puede contener claves duplicadas. Al agregar un elemento nuevo con una clave que ya existe, se sobrescribe el elemento anterior.

La clase HashMap proporciona métodos containsKey y containsValue que determinan la presencia de una clave o valor especificado.

Si intenta obtener un valor que no está presente en su mapa, devuelve el valor nulo.

null es un tipo especial que representa la ausencia de un valor.

# Sets

Un Conjunto es una colección que no puede contener elementos duplicados. Modela la abstracción matemática de conjuntos.

Una de las implementaciones del Set es la clase HashSet.

Ejemplo:

Texto

Descripción generada automáticamente

Puede usar el método size() para obtener la cantidad de elementos en el HashSet.

# LinkedHashSet

La clase HashSet no retiene automáticamente el orden de los elementos a medida que se agregan. Para ordenar los elementos, use un LinkedHashSet, que mantiene una lista vinculada de los elementos del conjunto en el orden en que se insertaron.

¿Qué es hash?

Una tabla hash almacena información a través de un mecanismo llamado hash, en el que el contenido informativo de una clave se usa para determinar un valor único llamado código hash.

Entonces, básicamente, cada elemento en HashSet está asociado con su código hash único.

Ha aprendido acerca de los distintos tipos de colecciones que están disponibles en Java, incluidas **Lists**, **Maps**, and **Sets** (listas, mapas y conjuntos). La elección de cuál usar es específica de los datos que necesita almacenar y manipular.

# Sorting Lists

Para la manipulación de datos en diferentes tipos de colecciones, la API de Java proporciona una clase Collections, que se incluye en el paquete java.util.

Uno de los métodos de clase Collections más populares es sort(), que ordena los elementos de su tipo de colección. Los métodos de la clase Collections son estáticos, por lo que no necesita un objeto Collections para llamarlos.

Ejemplo:

Texto

Descripción generada automáticamente

Como puede ver, los elementos se han ordenado alfabéticamente.

Puede llamar a los métodos sort() en diferentes tipos de Listas, como Integers.

Texto

Descripción generada automáticamente

Otros métodos útiles en la clase Collections:

**max**(Colección c): Devuelve el elemento máximo en c según lo determinado por el ordenamiento natural.

**min**(Colección c): Devuelve el elemento mínimo en c según lo determinado por el ordenamiento natural.

**reverse**(List list): Invierte la secuencia en la lista.

**shuffle** (Lista lista): Baraja (es decir, aleatoriza) los elementos de la lista.

# Iterators

Un iterador es un objeto que permite recorrer una colección, obtener o eliminar elementos.

Antes de poder acceder a una colección a través de un iterador, debe obtener uno. Cada una de las clases de colección proporciona un método iterator() que devuelve un iterador al inicio de la colección. Mediante el uso de este objeto iterador, puede acceder a cada elemento de la colección, un elemento a la vez.

La clase Iterator proporciona los siguientes métodos:

**hasNext():** Devuelve verdadero si hay al menos un elemento más; de lo contrario, devuelve falso.

**next():** Devuelve el siguiente objeto y avanza el iterador.

**remove():** elimina el último objeto devuelto por next de la colección.

La clase Iterator debe importarse del paquete java.util.

Ejemplo:

Texto

Descripción generada automáticamente

it.next() devuelve el primer elemento de la lista y luego mueve el iterador al siguiente elemento.

Cada vez que llamas it.next(), el iterador se mueve al siguiente elemento de la lista.

Normalmente, los iteradores se utilizan en bucles. En cada iteración del bucle, puede acceder al elemento de la lista correspondiente.

Ejemplo:

**Texto

Descripción generada automáticamente**

**Forma

Descripción generada automáticamente con confianza media**

Aquí, el ciclo while determina si el iterador tiene elementos adicionales, imprime el valor del elemento y avanza el iterador al siguiente.

# Working with Files

El paquete java.io incluye una clase de archivo que le permite trabajar con archivos.

Para comenzar, cree un objeto File y especifique la ruta del archivo en el constructor.

**import java.io.File;**  
...   
**File** file = new **File**("C:\\data\\input-file.txt");

Con el método exist(), puede determinar si existe un archivo.

import java.io.File;   
  
public class MyClass {   
public static void main(String[ ] args) {   
File x = new File("C:\\sololearn\\test.txt");   
if(x.**exists**()) {   
System.out.println(x.getName() + "exists!");   
}   
else {   
System.out.println("The file does not exist");   
}   
}   
}

El código anterior imprime un mensaje que indica si el archivo existe o no en la ruta especificada.

El método getName() devuelve el nombre del archivo.

Tenga en cuenta que usamos barras invertidas dobles en la ruta, ya que una barra invertida debe escaparse en la cadena de ruta.

# Reading a File

Los archivos son útiles para almacenar y recuperar datos, y hay varias formas de leer un archivo.

Una de las formas más sencillas es usar la clase Scanner del paquete java.util.

El constructor de la clase Scanner puede tomar un objeto File como entrada.

Para leer el contenido de un archivo de texto en la ruta "C:\\sololearn\\test.txt", necesitaríamos crear un objeto File con la ruta correspondiente y pasarlo al objeto Scanner.

try {   
**File** x = new **File**("C:\\sololearn\\test.txt");   
**Scanner** sc = new **Scanner**(x);   
}   
catch (FileNotFoundException e) {   
  
}

Rodeamos el código con un bloque try/catch, porque existe la posibilidad de que el archivo no exista.

La clase Scanner hereda del iterador, por lo que se comporta como tal.

Podemos usar el método next() del objeto Scanner para leer el contenido del archivo.

try {   
File x = new File("C:\\sololearn\\test.txt");   
Scanner sc = new Scanner(x);   
**while(sc.hasNext()) {   
System.out.println(sc.next());   
}**sc.close();   
} catch (FileNotFoundException e) {   
System.out.println("Error");   
}

El contenido del archivo se muestra palabra por palabra, porque el método next() devuelve cada palabra por separado.

Siempre es una buena práctica cerrar un archivo cuando termine de trabajar con él. Una forma de hacer esto es usar el método close() de Scanner.

# Reading a File (ejercicio)

## Drag and drop from the options below to read and print the content of the file a.txt, and then close it.

try {

File f = new File("a.txt");

Scanner sc = new Scanner(f);

while (sc.hasNext()) {

String a = sc.next();

String b = sc.next();

System.out.println(a + " " + b);

}

sc.close();

}

catch (Exception e) {

System.out.println("Error");

}

# Creating Files

Formatter, otra clase útil en el paquete java.util, se usa para crear contenido y escribirlo en archivos.

Ejemplo:

**import java.util.Formatter;**  
  
public class MyClass {   
public static void main(String[ ] args) {   
try {   
**Formatter** f = new **Formatter**("C:\\sololearn\\test.txt");   
} catch (Exception e) {   
System.out.println("Error");   
}   
}   
}

Esto crea un archivo vacío en la ruta especificada. Si el archivo ya existe, esto lo sobrescribirá.

Una vez más, debe rodear el código con un bloque try/catch, ya que la operación puede fallar.

# Writing to Files

Una vez que se crea el archivo, puede escribir contenido en él usando el mismo método format() del objeto Formatter.

Ejemplo:

Texto

Descripción generada automáticamente

El método format() formatea sus parámetros de acuerdo con su primer parámetro.

%s significa una cadena y se reemplaza por el primer parámetro después del formato. El segundo %s se reemplaza por el siguiente, y así sucesivamente. Entonces, el formato %s %s %s denota tres cadenas que están separadas por espacios.

Nota: \r\n es el símbolo de nueva línea en Windows.

El código anterior crea un archivo con el siguiente contenido:

# 1 John Smith 2 Amy Brown

¡No olvide cerrar el archivo una vez que haya terminado de escribirlo!

# Bowling Game

You are creating a bowling game!   
The given code declares a **Bowling** class with its constructor and **addPlayer**() method.   
Each player of the game has a name and points, and are stored in the players **HashMap**.   
The code in main takes 3 players data as input and adds them to the game.   
You need to add a **getWinner**() method to the class, which calculates and outputs the name of the player with the maximum points.   
  
**Sample Input:**  
Dave 42   
Amy 103   
Rob 64   
  
**Sample Output:**  
Amy

You need to iterate through the HashMap to find the element with the maximum points and output its corresponding key.

import java.util.\*;

public class Bowling {

HashMap<String, Integer> players;

Bowling() {

players = new HashMap<String, Integer>();

}

public void addPlayer(String name, int p) {

players.put(name, p);

}

//your code goes here

public void getWinner() {

int max = Collections.max(players.values());

for (Map.Entry<String, Integer> entry : players.entrySet()) {

if (entry.getValue().equals(max)) {

System.out.println(entry.getKey());

}

}

}

}

public class Program {

public static void main(String[ ] args) {

Bowling game = new Bowling();

Scanner sc = new Scanner(System.in);

for(int i=0;i<3;i++) {

String input = sc.nextLine();

String[] values = input.split(" ");

String name = values[0];

int points = Integer.parseInt(values[1]);

game.addPlayer(name, points);

}

game.getWinner();

}

}