|  |  |  |
| --- | --- | --- |
| Java | uitleg | extra |
| Main | Hoofdklasse |  |
| Klasse | Een concrete klasse is een volledig geïmplementeerde klasse die objecten kan creëren en instanties kan worden. Het heeft definitieve implementaties voor al zijn methoden en kan rechtstreeks worden gebruikt om objecten te maken en functionaliteit te bieden. Een concrete klasse kan alle eigenschappen, methoden en gedragingen bevatten die nodig zijn om specifieke functionaliteit te realiseren. |  |
| Methode | Functie/actie |  |
| Constructor (no void) | Blauwdruk |  |
| Interface (heeft nestend klasse) | specificeert welke acties een object | Keyword: Implements  public interface MyInterface {  }  public class MyClass implements MyInterface { |
| Superklasse(over erven) | Overkoepelende klasse |  |
| Subklasse (heeft nested klasse) | Onderdeel van superklasse | Keyword: extends  public interface MyConstructor {  }  public class MyClass exstends MyConstructor { |
| Abstracte klasse | Een concrete klasse is een volledig geïmplementeerde klasse die objecten kan creëren en instanties kan worden. Het heeft definitieve implementaties voor al zijn methoden en kan rechtstreeks worden gebruikt om objecten te maken en functionaliteit te bieden. Een concrete klasse kan alle eigenschappen, methoden en gedragingen bevatten die nodig zijn om specifieke functionaliteit te realiseren. |  |
| Generic klasse | een generieke klasse zien als een catalogus die verschillende soorten objecten kan bevatten. Net zoals een catalogus verschillende items kan bevatten, kan een generieke klasse verschillende typen objecten bevatten. | public class ToyBox<T> {} |  |  |
| Static klasse | Een statische klasse in Java is een klasse die direct kan worden gebruikt zonder dat er een instantie van de klasse hoeft te worden gemaakt. Statische klassen worden vaak gebruikt voor hulpprogramma's, utiliteitsfuncties of constanten die geen specifieke instantiegegevens nodig hebben.  Heeft niet altijd waarden en kan niet meer worden geïnstantieerd na aanmaken. | public class MathUtils {  public static int add(int a, int b) {  return a + b;  }  public static int subtract(int a, int b) {  return a - b;  }  public static int multiply(int a, int b) {  return a \* b;  }  public static double divide(int a, int b) {  if (b != 0) {  return (double) a / b;  } else {  throw new ArithmeticException("Division by zero is not allowed.");  }  }  } |
| **Getters en setters zijn methoden die worden gebruikt om toegang te krijgen tot (getters) en waarden te wijzigen (setters) van de private eigenschappen van een klasse. Ze worden gebruikt om de encapsulatie te behouden en de directe toegang tot de interne staat van een object te beperken.**  Het principe achter getters en setters is om een gecontroleerde manier te bieden om de waarden van de private eigenschappen van een klasse te lezen en te wijzigen. Hierdoor kunnen we de interne staat van een object beschermen en ervoor zorgen dat eventuele validatie- of consistentiecontroles worden toegepast.  Door getters en setters te gebruiken, kunnen we de toegang tot de interne staat van een object beheren en ervoor zorgen dat eventuele logica of validatie wordt toegepast bij het lezen en wijzigen van de waarden. Dit helpt bij het handhaven van de consistentie en integriteit van objecten en draagt bij aan een betere modulaire en onderhoudbare code. |  |  |
| Getter | Een getter is een methode die wordt gebruikt om de waarde van een private eigenschap op te halen. Het heeft meestal de vorm getType getPropertyName(). Bijvoorbeeld, als we een private eigenschap name hebben van het type String, zou de getter er als volgt uitzien: | Keyword: get  public String getName() {  return name;  } |
| Getter | Een setter is een methode die wordt gebruikt om de waarde van een private eigenschap in te stellen. Het heeft meestal de vorm void setPropertyName(setType value). Bijvoorbeeld, als we de private eigenschap name willen wijzigen, zou de setter er als volgt uitzien: | Keyword: set  public void setName(String value) {  name = value;  } |
| void | Het keyword void wordt gebruikt om aan te geven dat een methode geen returnwaarde heeft. Met andere woorden, een methode met de void returntype voert een bepaalde actie uit, maar retourneert geen waarde.  Als we het keyword void gebruiken bij de definitie van een methode, betekent dit dat de methode geen waarde teruggeeft en wordt aangeroepen om een bepaalde taak uit te voeren.  Het gebruik van void als returntype is nuttig voor methoden die geen resultaat hoeven te retourneren, maar in plaats daarvan iets moeten doen, zoals het afdrukken van een bericht, het wijzigen van een eigenschap van een object of het uitvoeren van een berekening zonder het resultaat terug te geven. | public void printName() {  System.out.println("John Doe");  } |

# voorbeeld

public static void main(String[] args) {

}

// Concrete klasse

public class Car {

// Eigenschappen

private String brand;

private String model;

// Constructor

public Car(String brand, String model) {

this.brand = brand;

this.model = model;

}

// Methode

public void drive() {

System.out.println("The " + brand + " " + model + " is driving.");

}

}

// Interface

public interface Animal {

void makeSound();

}

// Superklasse

public class Vehicle {

protected String color;

public Vehicle(String color) {

this.color = color;

}

public void start() {

System.out.println("The vehicle starts.");

}

}

// Subklasse

public class Car extends Vehicle {

private String brand;

public Car(String color, String brand) {

super(color);

this.brand = brand;

}

public void drive() {

System.out.println("The " + color + " " + brand + " car is driving.");

}

}

// Abstracte klasse

public abstract class Shape {

public abstract void draw();

}

// Generic klasse

public class Box<T> {

private T item;

public void addItem(T item) {

this.item = item;

}

public T getItem() {

return item;

}

}

// Statische klasse

public class MathUtils {

public static int add(int a, int b) {

return a + b;

}

public static int subtract(int a, int b) {

return a - b;

}

public static int multiply(int a, int b) {

return a \* b;

}

public static double divide(int a, int b) {

if (b != 0) {

return (double) a / b;

} else {

throw new ArithmeticException("Division by zero is not allowed.");

}

}

}

// Teken de elementen

public class Main {

public static void main(String[] args) {

// Concrete klasse

Car car = new Car("Toyota", "Camry");

car.drive();

// Interface

Animal animal = new Animal() {

@Override

public void makeSound() {

System.out.println("The animal makes a sound.");

}

};

animal.makeSound();

// Subklasse

Car car = new Car("Blue", "Toyota");

car.start();

car.drive();

// Abstracte klasse

Shape shape = new Shape() {

@Override

public void draw() {

System.out.println("Drawing a shape.");

}

};

shape.draw();

// Generic klasse

Box<String> box = new Box<>();

box.addItem("Item 1");

String item = box.getItem();

System.out.println("Box item: " + item);

// Statische klasse

int sum = MathUtils.add(5, 3);

System.out.println("Sum: " + sum);

}

}