Java Part 6: Classes and Objects

So far our programs used separate values, but a business thinks in things: an employee, an invoice, a customer. A class describes such a thing once, and each object made from it carries its own data. Four steps build a good class: blueprint, construct, protect and behave. In this part, Salma, an HR specialist at a manufacturing company, uses them to represent employees in code, and each step answers one practical question.

1. Blueprint

A class is a blueprint that records what a thing knows and what it can do. What it knows is stored in fields, which are variables inside the class. What it can do is written as methods. An object is one thing made from the class with the keyword new, and every object has its own copy of the field values. The variable that holds an object really holds a reference to it, so two variables can point to the same object, and a variable that points to nothing is null. Calling a method on null stops the program with a NullPointerException. The question for this step is: what does this thing know, and what can it do?

Example: A car rental company describes a vehicle once, in a class with three fields: a plate, a model and a daily rate. From this one blueprint it creates every car in its fleet with the keyword new. The first car object holds one plate and one rate, and the second holds a different plate and a different rate. Changing the daily rate of one car leaves all the other cars untouched, because each object keeps its own copy of the fields.

At work: Salma keeps employee data in three parallel lists: names, salaries and departments. When one employee leaves, a name is removed from one list but not from the others, and the salaries slide onto the wrong names. She creates an Employee class with fields for the name, the department and the salary. Now each employee is one object, and the data of one person can never drift apart.

2. Construct

A constructor is a special method that runs when we create an object with new. It has the same name as the class and no return type, and its job is to give the fields their first values. The word this refers to the object being built or used, which lets us tell a field from a parameter with the same name. If we write no constructor, Java supplies an empty one, but as soon as we write our own, the empty one disappears unless we add it back. A class can have several constructors with different parameters, just like overloaded methods. A constructor that demands the essential data makes it impossible to create an incomplete object. The question for this step is: what must an object have before it can exist?

Example: A library requires a title and an author before it creates a book record. The constructor of the Book class takes both values as parameters and stores them in the fields with the word this. A line that tries to create a book with no title does not compile, so an incomplete record can never exist. The code that creates books stays short and clear, because one line gives the book everything it needs.

At work: Salma decides that an employee must always have a name, a department and a salary. She writes a constructor with three parameters and uses this to store each one in its field. An attempt to create an Employee with no name no longer compiles. A second constructor takes only the name and fills in a default department for new hires.

3. Protect

Encapsulation means hiding the fields and controlling access to them through methods. We mark fields private, so only code inside the class can touch them, and we offer public methods to read or change

This document is a property of Asaseeyat Academy - © Copy Right Protected.

them. A method that reads a field is called a getter, and a method that changes it is called a setter. Because every change goes through a method, the class can check the value first and refuse a negative salary. This keeps the rules of the object in one place, instead of trusting every caller to be careful. Not every field needs a setter: a field that never changes after creation, such as an employee number, needs only a getter. The question for this step is: who may read or change this field, and under which rules?

Example: A bank account class keeps its balance private. The only ways to change the balance are the deposit and withdraw methods, and the withdraw method refuses an amount larger than the balance. If another part of the program tries to set the balance directly, the code does not compile. So no other code in the program can break that rule, and the rule is written in one place only.

At work: Salma marks the fields private, for example `private BigDecimal salary;`, and adds getters for all of them. She does not write a plain setter for the salary, but a method named changeSalary that refuses a negative amount. The employee number gets only a getter, because it never changes. Her colleagues can now use Employee without being able to break its rules.

4. Behave

Methods inside the class work on the data of one object. These instance methods can use the fields directly, so a method such as raiseSalary needs only the percentage and changes the salary of the object it is called on. A method named toString returns a readable description, and Java calls it automatically when we print an object or join it to text. A static method or a static field belongs to the class itself, not to one object, for example a counter of how many employees were created. We keep objects together in collections, such as a list of Employee, and a for each loop then asks each object to do its work. Good classes are small, with a clear name that is a noun, and methods that read like actions. The question for this step is: what should this object be able to do by itself?

Example: A shop models a product with a name, a price and a method that applies a discount. When the method is called on one product, it changes the price of that product only, and the other products keep their own prices. Printing the product shows its name and its price in readable text, thanks to toString. The shop code simply asks each product to discount itself, and never calculates the price from outside.

At work: Salma adds a raiseSalary method that takes a percentage and updates the salary of that employee. She adds toString so that printing an employee shows the name, the department and the salary. The payroll team keeps a list of Employee objects, and a for each loop asks each one to apply the yearly raise. A report of the whole department now takes a few lines instead of three parallel lists.

Conclusion

To recap, a class is a blueprint with fields and methods, and an object is one thing made from it with its own data. A constructor makes sure that every object starts complete, and the word this separates a field from a parameter. Private fields with controlled methods protect the rules of the object, and behaviour lives in the object itself. Salma's employees now travel as single objects that keep their data together. Next time, we will see how classes share behaviour through inheritance and interfaces. A good habit is to design the class from the nouns of the business problem and its methods from the verbs.

Sources

- Oracle: The Java Tutorials, Classes and Objects – https://docs.oracle.com/javase/tutorial/java/javaOO/index.html
- Oracle: The Java Tutorials, Providing Constructors for Your Classes – https://docs.oracle.com/javase/tutorial/java/javaOO/constructors.html
- Oracle: The Java Tutorials, Controlling Access to Members of a Class – https://docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html
- dev.java: Learn Java – https://dev.java/learn/

- Java is a trademark of Oracle and/or its affiliates. This video is independent training material and is not affiliated with Oracle.

This document is a property of Asaseeyat Academy - © Copy Right Protected.