Program Design Class

classes - a blueprint
- has properties or attributes
- has behaviors or methods

static is added if the function does not use the attributes of a class

instance is an object created from a class blueprint

constructor is a special method or behavior inside every class that creates and initalizes instances

constructor is always the same name of the class
no need to write return in constructor

the this keyword helps our program make a distinction between the attribute variable and the parameter variable.

! when you create a class and intance - it is a data type

The "this" keyword is used to make a distinction between an attribute variable and a parameter.

1) Create Project
2) in main, right click
3) Select new class

instance method - need an instance first
static method - does not need an instance

If you access the method through an instance it is a non static method ie. string methods instance variables

non static variables - variables that are different from each object dynamically assigned in instatntiation.

static class variables - do not change per instance they hold a value for the whole class to use

Static variable is accessed using classname.varname

classes - organizes data, has poperties and methods
- has constructor
- a blueprint

instance variables and methods
- can be accessed by instance and . operator

static variables and methods
- no need for instance to be accessed

encapsulation

- bind state and behavior together into a single unit
- combine code and data acting on that data

benefits of encapsulation

- Prevent classes from becoming tightly coupled
- easily modify the inner workings of one class without affecting the rest of the program
- We need a clean interface between a given class and the rest of the program.
- Everything can't have direct access
- clear pathways for classes to communicate
- less code changes required for refactoring change
- less likely that an attribute would be overwritten w/ null or invalid. unexpectedly

Access Modifiers

↝ Allows you to restrict access to certain variables or methods

1. Private - only to class they live in
2. Public - accessible anywhere w/in the program
3. Protected - visible to package and all subclass
4. no modifier - visible in the package it lives in.

inheritance

↝ allows us to create class hierarchies where classes inherits properties and behaviors from other classes.

2 key Players of inheritance

superclass	subclass
• inherits from	• inherits properties
• parent class	• child class

for example:

Employee → Salesperson
parent class → childclass

Is a relationship

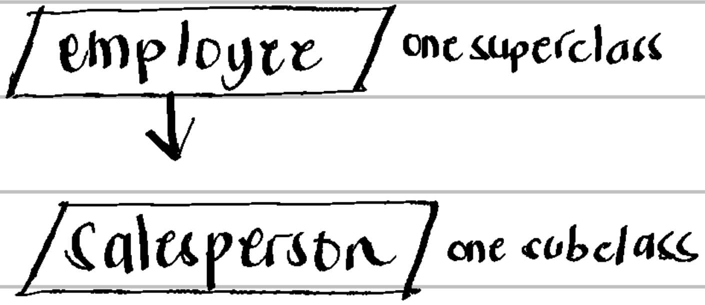
↝ the salesperson "is an" employee
↝ all salesperson instances are also employees
↝ not all employee instances are salespersons

Benefits of inheritance

↝ Promotes code reusability and scalability
↝ common properties and methods can be written in one class (ie employee)
↝ other classes inherits from the common class and add on unique functionality (ie Salesperson)

Different Types of inheritance

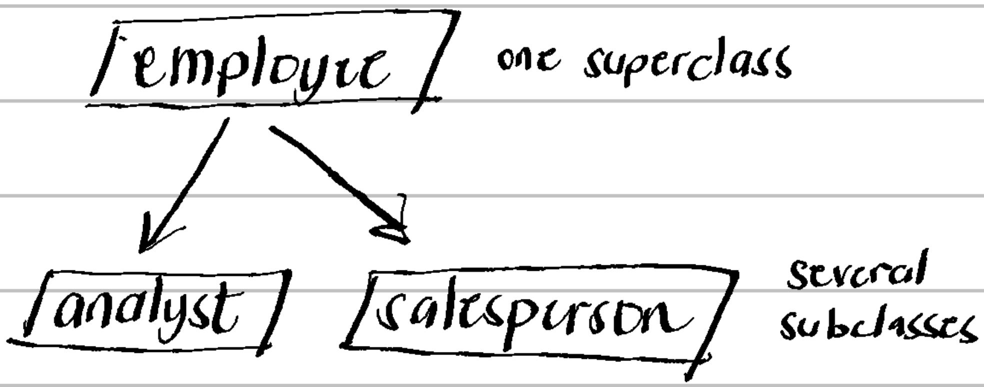
Single level inheritance



one superclass

one subclass

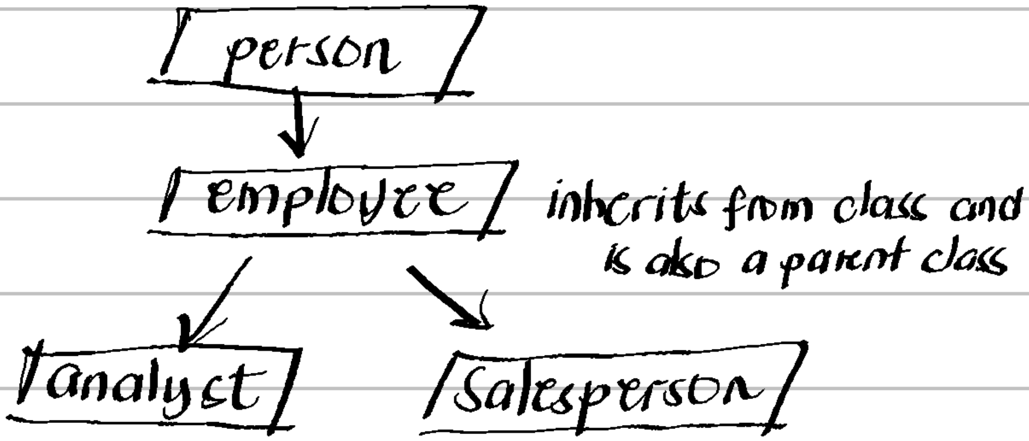
Hierarchical Inheritance



one superclass

several subclasses

multilevel inheritance



inherits from class and is also a parent class

for example:

attribute name is a person class attribute
attribute id is an employee class attribute

other types of inheritance:

- multiple and hybrid can cause unneccessary complexity and rarely used
- removed from java

- in java a class can only have one super class, but multiple subclasses
- if multiple super classes are needed use multilevel inheritance
- abstract classes and interfaces are also used to share code
- we can achieve inheritance by using the keyword extends

Polymorphism

- is the ability for an object or function to take many forms
- helps write reusable code and reduce complexity
- makes code more flexible by providing multiple ways to use similar functionality

2 types of Polymorphism in Java

1. **Runtime** → allows for re using the functionalities Of a given class, and override it with new functionality as needed while leveraging the superclass implementation
2. **Compile-time** → Java decides what method to use based on the input's type and the number Of parameter used at compile time, hence compile-time polymorphism

- overloaded methods are faster because it is bonded during compile time an earlier phase than run time

- overloaded methods also keep everything in one place, while overriden method jumps all over the code.
- ~> also just depends on needs

Usage of method override and method overload

Compile-time polymorphism - method overload

- ~> another way to input to the same functionality

runtime polymorphism - method override

- ~> you might want to change a few method implementations but keep the same core functionality.

Abstraction

- ~> helps us hide implementation complexity
- ~> is like a pod coffee machine - just pop the pod and it makes coffee. do not have to know the technical details of the machine

- Java supports abstract classes and interfaces
- all you need to know is the input, output, and general idea of what the system does.
- No need to get bogged down by the details
- Easier to contribute to

Abstract Classes

- allows us to add abstraction
- like a template class where some of the functionality is not implemented yet.
- you cannot instantiate an abstract class
- other classes can extend abstract class and implement the appropriate functionality

◆ this allows us to place the algorithm in one place and other concrete class can use it w/o worrying implementation. just override the abstract method

interfaces

↝ is a set of method signatures for to-be implemented functionality

↝ its a specification for a set of behaviors without implementation

↝ interfaces cannot be instantiated

↝ uses implements keyword

↝ adding a new method to interface will force all classes using the interface to conform

◆ abstraction leaves implementation to concrete type while promising functionality

Patterns in Java

1) create interface with method specification
2) create an abstract class that implements the interface with base implementations and some abstract methods
3) create a concrete class that implements the abstract methods.

↝ with this pattern you get base implementation for free in the concrete class and complexity is reduced.

↝ leaving specific implementation to the concrete class

Beware of code smells:

- Class bloat
- long method
- god object
- feature envy

Single responsibility principle - states that a class should have a single, well defined responsibility within software system.