Object Oriented Programming

Java Programming for Absolute Beginners Phase I

Here We Go…

- As technology has grown and has been applied to different types of problems, different **paradigms** were invented to approach these various problems.
- There are many types of programming paradigms, and they are used to classify languages based on their features and behaviors.
- A language is not confined to a single paradigm, but some may argue otherwise.
- Java was built for the use of **Object Oriented Programming**

So What is an Object Anyways?

- An object is an abstract idea for solving problems.
- An object is defined by its **state** and **behavior**, state being information about an individual object, and behavior being what that object can do.
- The state of an object is all the data that makes up that object. For example, the state of a bank account object would be something like a routing number, balance, account holder name, etc.
- The behavior of an object can act on the sate of the object, manipulating its own state, the state of an other object, or simply performing actions based on those sates. Some examples for behavior in a bank account object would be deposit, withdraw, check balance.
- Note how I have not talked about programming yet. Objects should not be thought of as programming concepts, as they can be used to solve problems outside of computing.

The Three Pillars of OOP

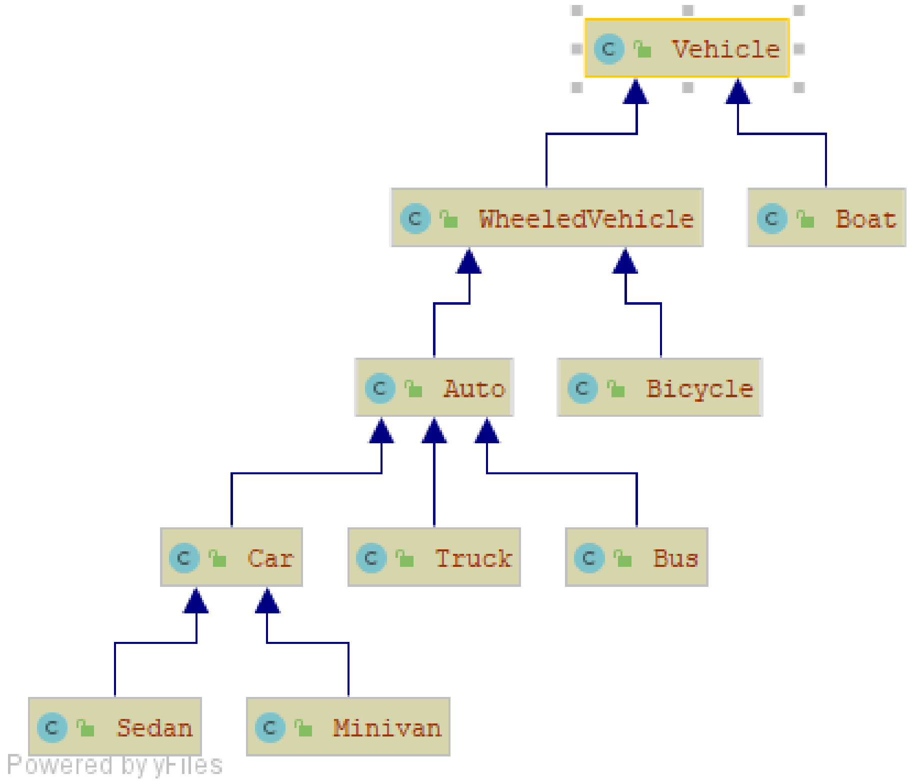
- Encapsulation
- Inheritance
- Polymorphism

Encapsulation

- **Encapsulation** is the process of combining the state and behavior of an object into a single unit.
- In Java, we perform Encapsulation in the form of a **class**.
- A class is the code that will implement an object.
- A class is like a blueprint, and objects are created from classes, note that they are **not** the same thing.
- In Java, an object is an *instance* of a class.
- There can be several instances of the same class.
- The state of the object is defined by the **data fields**, or **instance variables** in a class.
- The behavior of the object is defined by the **methods** in a class.
- We have written several simple classes already!

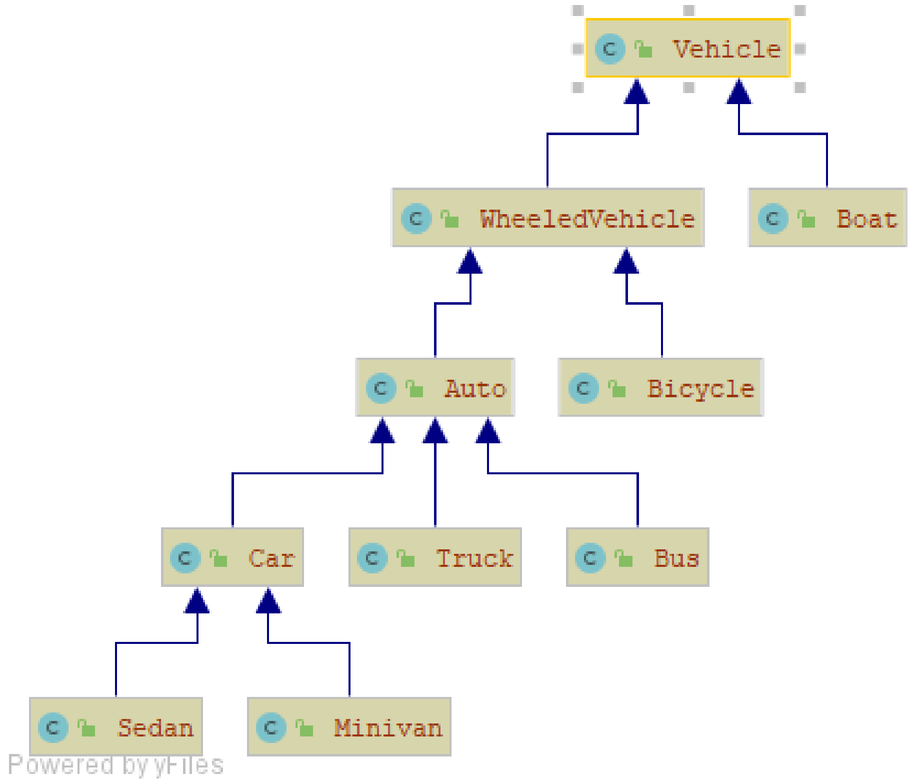
Inheritance

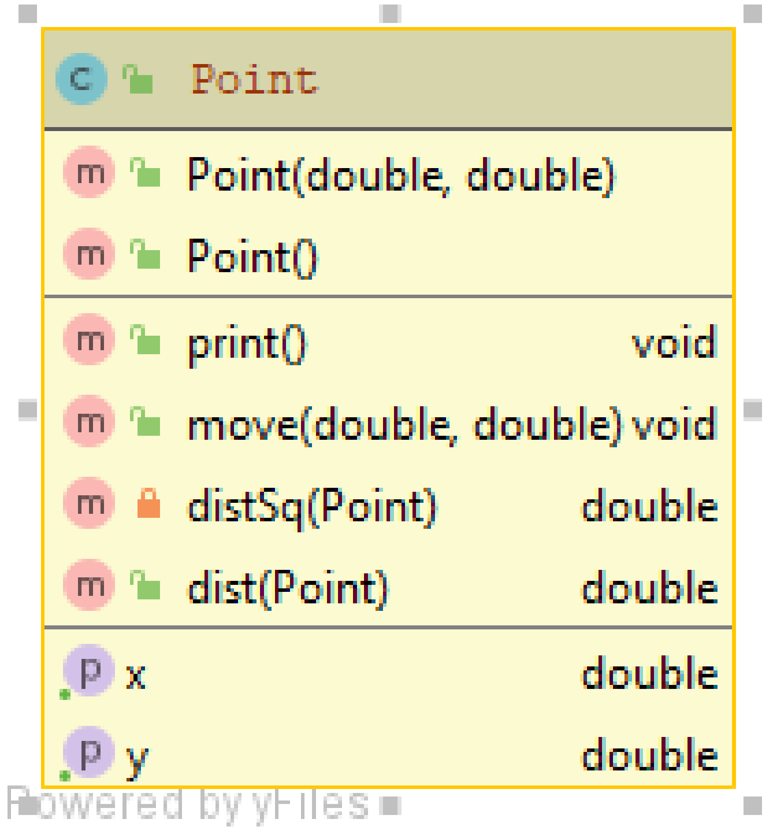
- **Inheritance** is the process of creating a class from an existing class.
- The new class will have all of the original class's state and behavior, but will also add its own, making it bigger.
- The new class is called a **subclass** and the original class is called a **superclass**.
- This is great because it allows for some code to only be written and tested once in a superclass, then used in subclasses.
- A subclass of a class can be a superclass for another class, which leads to an **inheritance hierarchy**.
- A class in Java can only extend one other class, and cannot extend itself.



Inheritance Pt. 2

- Note in the diagram how the subclasses point up to their super classes.
- These arrows signify the *is-a* relationships within the inheritance hierarchy.
- For example, a Boat *is-a* Vehicle.
- The *is-a* relationship is transitive, so if Bicycle *is-a* Wheeled Vehicle, and Wheeled Vehicle *is-a* Vehicle, then Bicycle *is-a* Vehicle.
- The *is-a* relationship is one directional. A Minivan *is-a* Car, but a Car is not always a Minivan.
- If a class does not explicitly extend another class, then it extends Object.
- All classes have an *is-a* relationship with Object



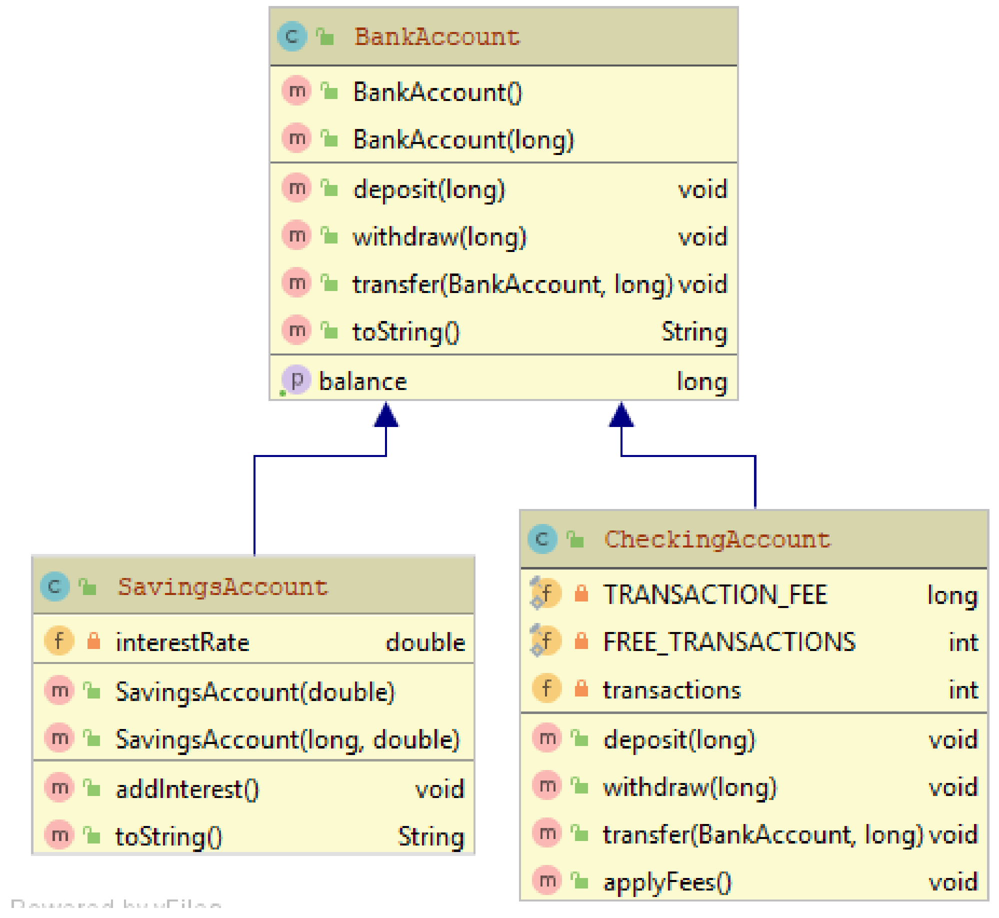


Point

Point(double, double)	
Point()	
print()	void
move(double, double)	void
distSq(Point)	double
dist(Point)	double
x	double
y	double

Example 1 UML

Example 2 UML

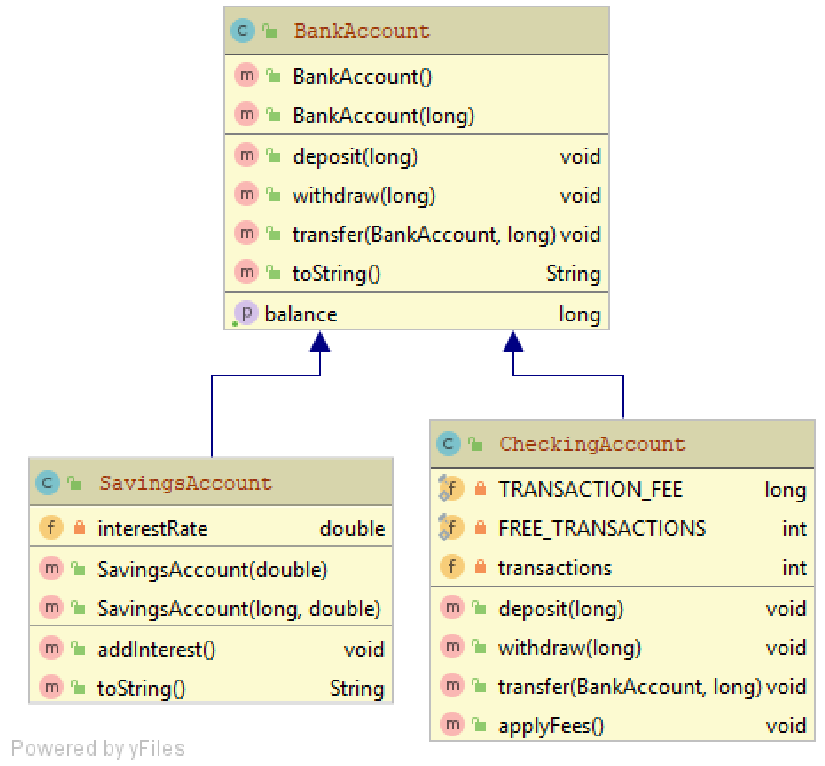


Polymorphism

- The word **Polymorphism** has Greek roots that roughly mean “many forms”
- In Java, this means that an object of one type can take a form of an object of another type.
- But wait! There’s rules to this.
- This only works when there is an *is-a* relationship at play, meaning Polymorphism and Inheritance work hand in hand.
- Basically, this means that a variable of the type of a super class can reference an instance of a subclass.
- Also, when we override a method, we are making it **polymorphic**, and when calling that method, the compiler will search bottom-up for an implementation of that method, so the subclass’s implementation will always be used.

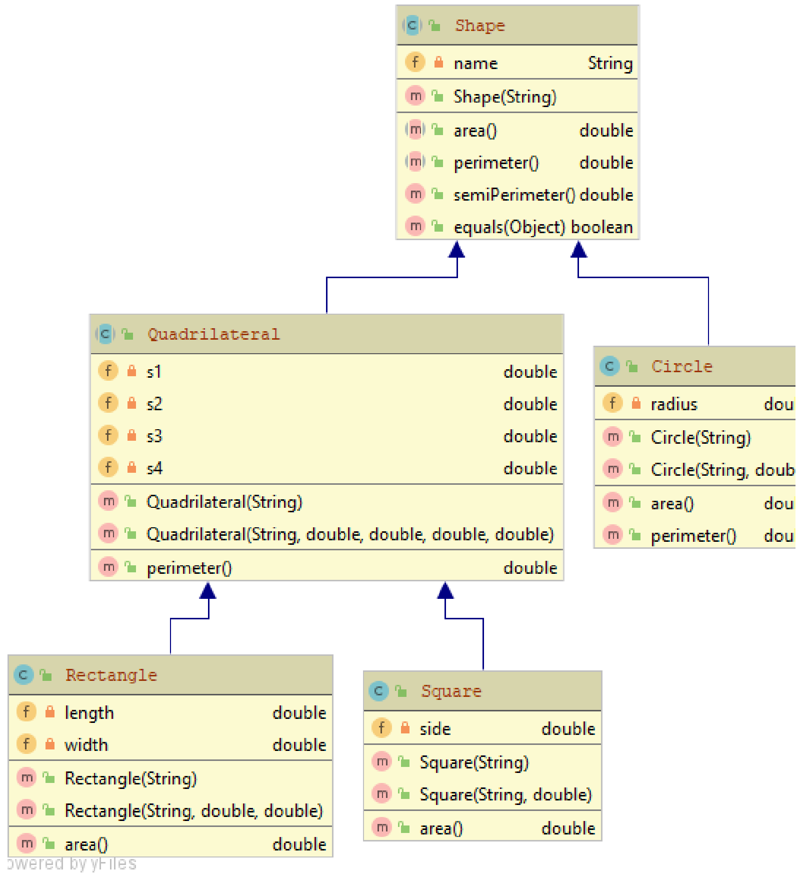
More on Polymorphic Methods

- For instance, take a look at the UML for example 2.
- If I had a BankAccount object and call withdraw, the compiler will use the withdraw method as defined in the BankAccount class.
- However, if I call withdraw on a SavingsAccount object, the compiler will see that there is no such definition in the SavingsAccount class, so it will go up to the superclass and use the definition there. In the case of a deeply nested hierarchy, it will keep searching upwards until a definition is found.
- Now, if I call withdraw on a CheckingAccount object, the compiler will see the definition of withdraw in the CheckingAccount class, and use that method.



Abstract Classes

- An **Abstract Class** is a superclass that represent an abstract object. Meaning that there will be no direct instances of that class.
- To create an object from an abstract class, there must be a subclass.
- Classes that are not abstract are **concrete**.
- Abstract classes contain abstract methods, that define only the signature of the method.
- A concrete class *must* implement all abstract methods in it's super class hierarchy.
- Combining abstract classes and polymorphism is where OOP's true capabilities shine.

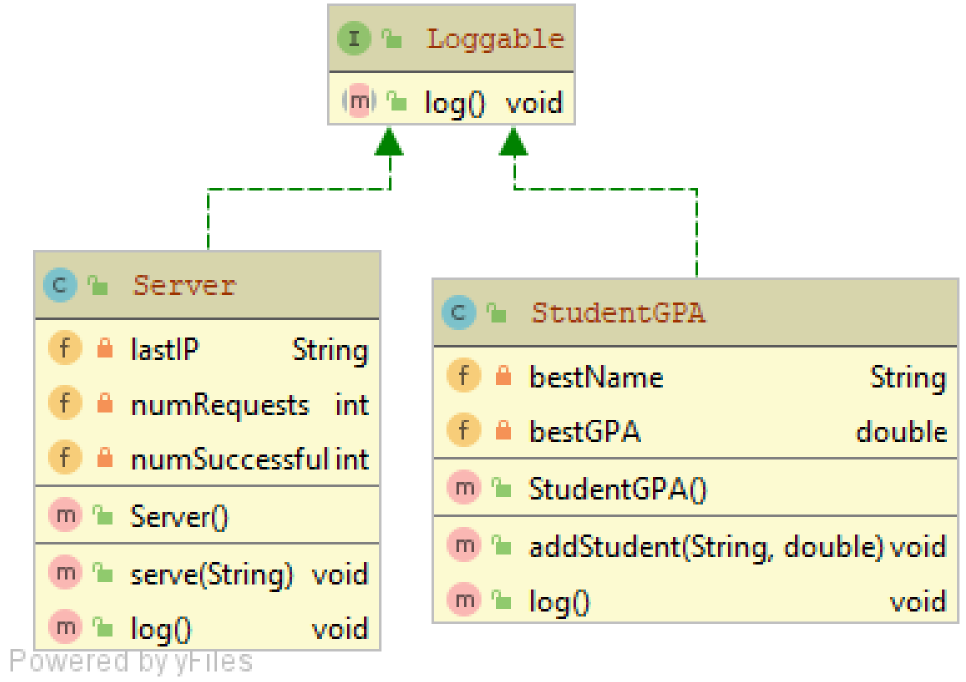


Example 4 UML

Interfaces

- An **Interface** is a definition for some of the behavior of an object.
- Like an abstract class, no instance is created of an interface directly.
- However, an interface does not carry any state, and cannot have any implementations for behavior, unlike an abstract class.
- An interface does not define an object, it is merely a set of methods that need to be *implemented* in a class.
- A class does not extend an interface, a class **implements** an interface.
- Like abstract class, an implementation for all methods in an interface must be found in a concrete class.
- Classes that define completely unlike objects may implement the same interface.
- Classes can implement zero or many interfaces.
- Interfaces can extend other interfaces.
- Interfaces can extend multiple interfaces, unlike classes.

Example 5 UML



Any Questions?