Software Development

Making More Complicated Programs!

Tim Jackman

BU Summer Challenge

July 11th, 2025

- So far we've seen how to use the basic Java language features to create rudimentary programs.

- So far we've seen how to use the basic Java language features to create rudimentary programs.
- Real-life software applications are a lot more complex than what we've seen

- So far we've seen how to use the basic Java language features to create rudimentary programs.
- Real-life software applications are a lot more complex than what we've seen
- May involve many classes and components working together to create a complex program

- So far we've seen how to use the basic Java language features to create rudimentary programs.
- Real-life software applications are a lot more complex than what we've seen
- May involve many classes and components working together to create a complex program
- Today we are going to learn about some of the key concepts for *Object-Oriented Design*

- So far we've seen how to use the basic Java language features to create rudimentary programs.
- Real-life software applications are a lot more complex than what we've seen
- May involve many classes and components working together to create a complex program
- Today we are going to learn about some of the key concepts for *Object-Oriented Design*
    - OOD is an approach for creating software using the concepts of *classes* and *objects*

- So far we've seen how to use the basic Java language features to create rudimentary programs.
- Real-life software applications are a lot more complex than what we've seen
- May involve many classes and components working together to create a complex program
- Today we are going to learn about some of the key concepts for *Object-Oriented Design*
    - OOD is an approach for creating software using the concepts of *classes* and *objects*
    - Java is *designed* to be used for Object-Oriented Program

- So far we've seen how to use the basic Java language features to create rudimentary programs.
- Real-life software applications are a lot more complex than what we've seen
- May involve many classes and components working together to create a complex program
- Today we are going to learn about some of the key concepts for *Object-Oriented Design*
  - OOD is an approach for creating software using the concepts of *classes* and *objects*
  - Java is *designed* to be used for Object-Oriented Program
- We will also quickly see some advanced Java features that are useful

Encapsulation

- We've already touched on one aspect of Encapsulation with access modifiers

Encapsulation

- We've already touched on one aspect of Encapsulation with access modifiers
- Encapsulation is the idea that *data* should be bundled with the *methods* that operate on it

Encapsulation

- We've already touched on one aspect of Encapsulation with access modifiers
- Encapsulation is the idea that *data* should be bundled with the *methods* that operate on it
- Using access modifiers allow us to sensibly control how data can be used in our program, to keep things from getting messy

Encapsulation

- We've already touched on one aspect of Encapsulation with access modifiers
- Encapsulation is the idea that *data* should be bundled with the *methods* that operate on it
- Using access modifiers allow us to sensibly control how data can be used in our program, to keep things from getting messy
    - Class: "I am responsible for my data, if you want to use it you have to do it my way so nothing goes wrong"

Encapsulation

- We've already touched on one aspect of Encapsulation with access modifiers
- Encapsulation is the idea that *data* should be bundled with the *methods* that operate on it
- Using access modifiers allow us to sensibly control how data can be used in our program, to keep things from getting messy
    - Class: "I am responsible for my data, if you want to use it you have to do it my way so nothing goes wrong"
- Aligns with the <u>fundamental</u> problem-solving technique: Breaking It Down Into Smaller Parts

Composition

- Imagine you wanted to repair a complicated machine like a car. Would you start trying to rebuild it all at once?

Composition

- Imagine you wanted to repair a complicated machine like a car. Would you start trying to rebuild it all at once?
    - We'd start by replacing the wheels, repairing the engine, etc.

Composition

- Imagine you wanted to repair a complicated machine like a car. Would you start trying to rebuild it all at once?
  - We'd start by replacing the wheels, repairing the engine, etc.
- We design programs by breaking the problem apart and building basic classes to represent the basic data we need

Composition

- Imagine you wanted to repair a complicated machine like a car. Would you start trying to rebuild it all at once?
    - We'd start by replacing the wheels, repairing the engine, etc.
- We design programs by breaking the problem apart and building basic classes to represent the basic data we need
- We then *compose* these base classes to build more complicated data types

Inheritance

- We've seen how *classes* are blueprints for objects and how they bundle what's the same about two objects of the same type

Inheritance

- We've seen how *classes* are blueprints for objects and how they bundle what's the same about two objects of the same type
- Can we do the same thing with two classes that share similarities?

Inheritance

- We've seen how *classes* are blueprints for objects and how they bundle what's the same about two objects of the same type
- Can we do the same thing with two classes that share similarities?
- Imagine we've designed a `Car` class and a `Truck` class. Both may have similar methods like `drive()` or similar fields like `fuel-efficiency`

Inheritance

- We've seen how *classes* are blueprints for objects and how they bundle what's the same about two objects of the same type
- Can we do the same thing with two classes that share similarities?
- Imagine we've designed a `Car` class and a `Truck` class. Both may have similar methods like `drive()` or similar fields like `fuel-efficiency`
- Java allows us to create a "parent" class *Vehicle* that bundles the shared fields/methods `Car` and `Truck`

Inheritance

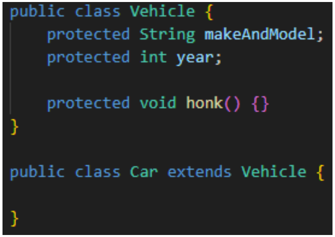
- We've seen how *classes* are blueprints for objects and how they bundle what's the same about two objects of the same type
- Can we do the same thing with two classes that share similarities?
- Imagine we've designed a `Car` class and a `Truck` class. Both may have similar methods like `drive()` or similar fields like `fuel-efficiency`
- Java allows us to create a "parent" class *Vehicle* that bundles the shared fields/methods `Car` and `Truck`
    - We write these fields/methods in the superclass (parent) class and any subclass (child) class *inherits* them automatically

Inheritance

- We've seen how *classes* are blueprints for objects and how they bundle what's the same about two objects of the same type
- Can we do the same thing with two classes that share similarities?
- Imagine we've designed a `Car` class and a `Truck` class. Both may have similar methods like `drive()` or similar fields like `fuel-efficiency`
- Java allows us to create a "parent" class *Vehicle* that bundles the shared fields/methods `Car` and `Truck`
    - We write these fields/methods in the superclass (parent) class and any subclass (child) class *inherits* them automatically
- This idea is called *Inheritance* and allows us to clearly and efficiently reuse code

Inheritance In Java

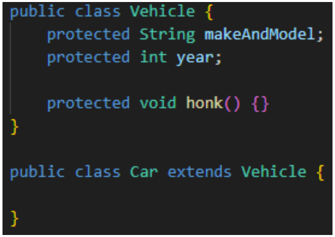
- We use the keyword `extends` to declare a class is a subclass of another



```
public class Vehicle {
    protected String makeAndModel;
    protected int year;

    protected void honk() {}
}

public class Car extends Vehicle {

}
```

Inheritance In Java

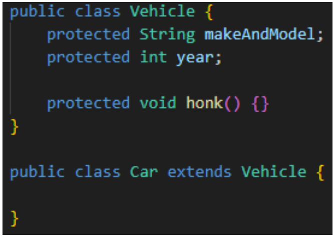
- We use the keyword `extends` to declare a class is a subclass of another



```java
public class Vehicle {
    protected String makeAndModel;
    protected int year;

    protected void honk() {}
}

public class Car extends Vehicle {

}
```

Inheritance In Java

- We use the keyword `extends` to declare a class is a subclass of another

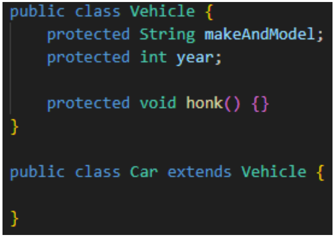


```java
public class Vehicle {
    protected String makeAndModel;
    protected int year;

    protected void honk() {}
}

public class Car extends Vehicle {

}
```

- `protected` is an access keyword that means only this class or its subclasses

Inheritance In Java

- We use the keyword `extends` to declare a class is a subclass of another

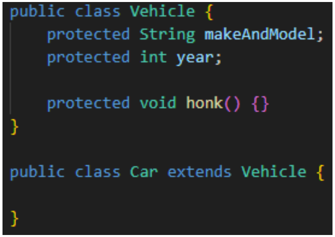


```
public class Vehicle {
    protected String makeAndModel;
    protected int year;

    protected void honk() {}
}

public class Car extends Vehicle {

}
```

- `protected` is an access keyword that means only this class or its subclasses
- Subclasses do not inherit private fields

Inheritance In Java

- We use the keyword `extends` to declare a class is a subclass of another

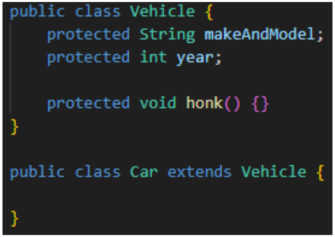


```java
public class Vehicle {
    protected String makeAndModel;
    protected int year;

    protected void honk() {}
}

public class Car extends Vehicle {

}
```

- `protected` is an access keyword that means only this class or its subclasses
- Subclasses do not inherit private fields
- Subclasses can *override* (redefine) inherited fields/methods

Inheritance In Java

- We use the keyword `extends` to declare a class is a subclass of another



```java
public class Vehicle {
    protected String makeAndModel;
    protected int year;

    protected void honk() {}
}

public class Car extends Vehicle {

}
```

- `protected` is an access keyword that means only this class or its subclasses
- Subclasses do not inherit private fields
- Subclasses can *override* (redefine) inherited fields/methods
- Subclasses can call parent methods with `super`

Inheritance in Java

- Classes can only extend one class

Inheritance in Java

- Classes can only extend one class
- By default, every class extends the Object class (except Object)

Inheritance in Java

- Classes can only extend one class
- By default, every class extends the Object class (except Object)
- Every class has `toString()`, `equals()`

Inheritance in Java

- Classes can only extend one class
- By default, every class extends the Object class (except Object)
- Every class has `toString()`, `equals()`
- Object has a default implementation that might not be useful, some classes override these some don't

Interfaces

- Superclasses are *blueprints* for other classes but are *very* detailed

Interfaces

- Superclasses are *blueprints* for other classes but are *very* detailed
    - They provide their own implementations with the idea their subclasses will use them

Interfaces

- Superclasses are *blueprints* for other classes but are *very* detailed
  - They provide their own implementations with the idea their subclasses will use them
- Does it make sense to have a Vehicle object? A Vehicle is an abstract concept?
  - Cars are concrete, Vehicles are not

Interfaces

- Superclasses are *blueprints* for other classes but are *very* detailed
    - They provide their own implementations with the idea their subclasses will use them
- Does it make sense to have a Vehicle object? A Vehicle is an abstract concept?
    - Cars are concrete, Vehicles are not
- What about if we want to just give a “rough sketch” of a class?

Interfaces

- Superclasses are *blueprints* for other classes but are *very* detailed
  - They provide their own implementations with the idea their subclasses will use them
- Does it make sense to have a Vehicle object? A Vehicle is an abstract concept?
  - Cars are concrete, Vehicles are not
- What about if we want to just give a “rough sketch” of a class?
  - Anything following this “rough sketch” should have these methods that do these things

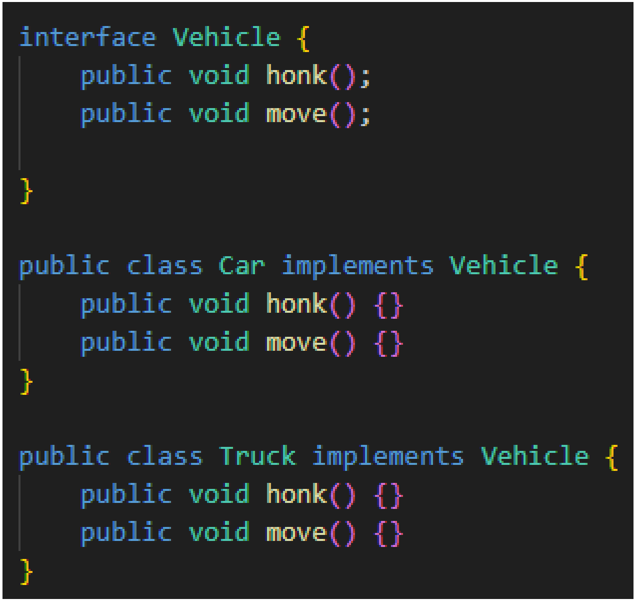
Interfaces

- Superclasses are *blueprints* for other classes but are *very* detailed
  - They provide their own implementations with the idea their subclasses will use them
- Does it make sense to have a Vehicle object? A Vehicle is an abstract concept?
  - Cars are concrete, Vehicles are not
- What about if we want to just give a “rough sketch” of a class?
  - Anything following this “rough sketch” should have these methods that do these things
  - The implementing class following the sketch is responsible for ALL implementation

Interfaces

- Superclasses are *blueprints* for other classes but are *very* detailed
  - They provide their own implementations with the idea their subclasses will use them
- Does it make sense to have a Vehicle object? A Vehicle is an abstract concept?
  - Cars are concrete, Vehicles are not
- What about if we want to just give a “rough sketch” of a class?
  - Anything following this “rough sketch” should have these methods that do these things
  - The implementing class following the sketch is responsible for ALL implementation
- In Java these are called *interfaces*

Interfaces in Java


```java
interface Vehicle {
    public void honk();
    public void move();

}

public class Car implements Vehicle {
    public void honk() {}
    public void move() {}
}

public class Truck implements Vehicle {
    public void honk() {}
    public void move() {}
}
```

Interfaces in Java

- We use the `implements` keyword to declare a class implements an interface

Interfaces in Java

- We use the `implements` keyword to declare a class implements an interface
- Classes can implement as many interfaces as they want

Interfaces in Java

- We use the `implements` keyword to declare a class implements an interface
- Classes can implement as many interfaces as they want
- The power of interfaces is they allow us to use implementing classes like a black-box

Interfaces in Java

- We use the `implements` keyword to declare a class implements an interface
- Classes can implement as many interfaces as they want
- The power of interfaces is they allow us to use implementing classes like a black-box
- In particular, we can write methods using interfaces

Interfaces in Java

- We use the `implements` keyword to declare a class implements an interface
- Classes can implement as many interfaces as they want
- The power of interfaces is they allow us to use implementing classes like a <u>black-box</u>
- In particular, we can write methods using interfaces
  - Our code only get access to the methods of the interface

Interfaces in Java

- We use the `implements` keyword to declare a class implements an interface
- Classes can implement as many interfaces as they want
- The power of interfaces is they allow us to use implementing classes like a black-box
- In particular, we can write methods using interfaces
  - Our code only get access to the methods of the interface
  - But it will work with *any* of the classes implementing the interface

Interfaces in Java

- We use the `implements` keyword to declare a class implements an interface
- Classes can implement as many interfaces as they want
- The power of interfaces is they allow us to use implementing classes like a black-box
- In particular, we can write methods using interfaces
  - Our code only get access to the methods of the interface
  - But it will work with *any* of the classes implementing the interface
  - We can freely swap around our code we're using without breaking anything!

Generics

- Interfaces allow us to write somewhat general code that would work with any implementing class

Generics

- Interfaces allow us to write somewhat general code that would work with any implementing class
- But what if we write a truly general piece of code that would work with *any* class

Generics

- Interfaces allow us to write somewhat general code that would work with any implementing class
- But what if we write a truly general piece of code that would work with *any* class
- We could cast everything up to Objects but we lose a lot of information (no fields, no methods)

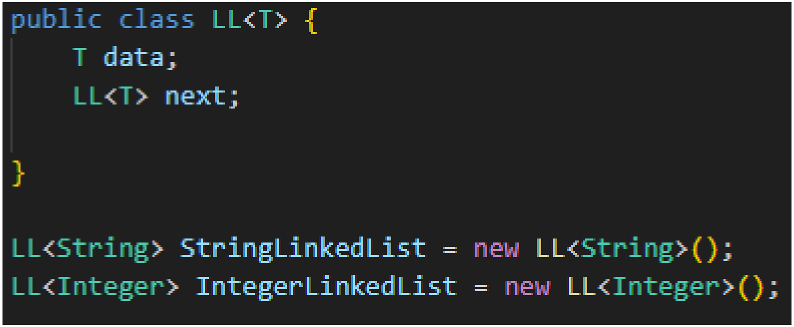
Generics

- Interfaces allow us to write somewhat general code that would work with any implementing class
- But what if we write a truly general piece of code that would work with *any* class
- We could cast everything up to Objects but we lose a lot of information (no fields, no methods)
- For example, we implemented IntegerLL in the homework. But the code would be the same for StringLL, or FileLL, or CarLL

Generics

- Interfaces allow us to write somewhat general code that would work with any implementing class
- But what if we write a truly general piece of code that would work with *any* class
- We could cast everything up to Objects but we lose a lot of information (no fields, no methods)
- For example, we implemented IntegerLL in the homework. But the code would be the same for StringLL, or FileLL, or CarLL
- To solve this problem in programming, we use the concept of *generics*, where we treat the Type like an input in our class definition

Generics in Java



```
public class LL<T> {
    T data;
    LL<T> next;

}

LL<String> StringLinkedList = new LL<String>();
LL<Integer> IntegerLinkedList = new LL<Integer>();
```

Building Software

- How do we use these tools to build complex programs?

Building Software

- How do we use these tools to build complex programs?
- Depending on your application, you might use a different *software architectural pattern*

Building Software

- How do we use these tools to build complex programs?
- Depending on your application, you might use a different *software architectural pattern*
- These are designs for how to tackle a specific problem

Building Software

- How do we use these tools to build complex programs?
- Depending on your application, you might use a different *software architectural pattern*
- These are designs for how to tackle a specific problem
- For example, imagine you want to implement a user interface, how should you structure your code?

Model-View-Controller (MVC)

- MVC is a pattern for UI that separates the program into three pieces: Model, View, and Controller which each do a different task

Model-View-Controller (MVC)

- MVC is a pattern for UI that separates the program into three pieces: Model, View, and Controller which each do a different task
- The Model is the code controls the internal data of the program

Model-View-Controller (MVC)

- MVC is a pattern for UI that separates the program into three pieces: Model, View, and Controller which each do a different task
- The Model is the code controls the internal data of the program
- The View is code that controls what the program looks like

Model-View-Controller (MVC)

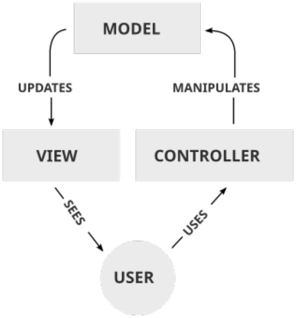
- MVC is a pattern for UI that separates the program into three pieces: Model, View, and Controller which each do a different task
- The Model is the code controls the internal data of the program
- The View is code that controls what the program looks like
- The Controller is the code that takes the user input and

Model-View-Controller (MVC)

- MVC is a pattern for UI that separates the program into three pieces: Model, View, and Controller which each do a different task
- The Model is the code controls the internal data of the program
- The View is code that controls what the program looks like
- The Controller is the code that takes the user input and
- These three components are separated from one another but interact in specific ways

Model-View-Controller (MVC)

- MVC is a pattern for UI that separates the program into three pieces: Model, View, and Controller which each do a different task
- The Model is the code controls the internal data of the program
- The View is code that controls what the program looks like
- The Controller is the code that takes the user input and
- These three components are separated from one another but interact in specific ways


MODEL
UPDATES
MANIPULATES
VIEW
CONTROLLER
SEES
USES
USER