



Lecture 9: Interfaces and Polymorphism

CS 2110

September 22, 2026

Today's Learning Outcomes

- 36. Implement an interface using a given state representation according to its specifications.
- 37. Compare and contrast *static types* and *dynamic types*.
- 38. Identify three scenarios where subtype substitution is permitted.
- 39. Explain the benefits of leveraging *polymorphism* in object-oriented code.
- 40. Describe the principle of dynamic dispatch and the compile-time reference rule.

Real-World Interfaces



2014 Toyota Prius C
(Matt's Car)

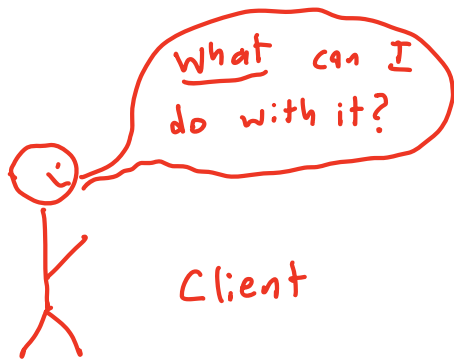


2023 Volkswagen ID.4
(Matt's Rental Car)

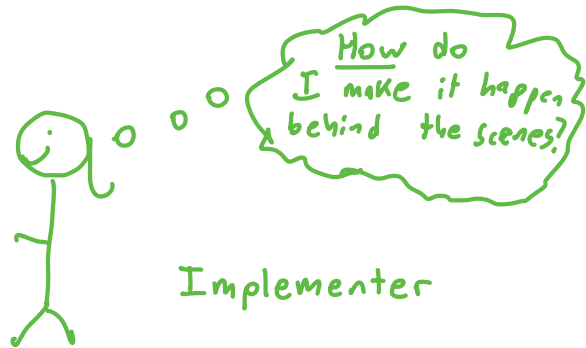
These cars are built and run very differently. Should I be worried when I pick up my rental car?

Software Interfaces and Abstraction Barriers

Interfaces present an abstraction barrier between the **implementer** and the **client** of a software module.



Interface



API (application programming interface)

- method signatures and specifications

source-code view of class

- state representation
- invariants
- method bodies

Interfaces in Java

= a new language construct that offers an alternative way to define a new type (in addition to a class).



Coding Demo: Account interface



Implementing an Interface

Interfaces don't have state (since they don't have fields), so they cannot be instantiated.

Only **classes** provide complete blueprints to construct objects.



Coding Demo: CheckingAccount class



Specifications and @Override

The `@Override` annotation signifies that a method's definition is based on a declaration from a *"higher up"* type (e.g., an interface it implements).

Our Javadoc rules for overridden methods:

If the *"higher"* specs match **exactly**:

If the definition **refines** the *"higher"* specs:

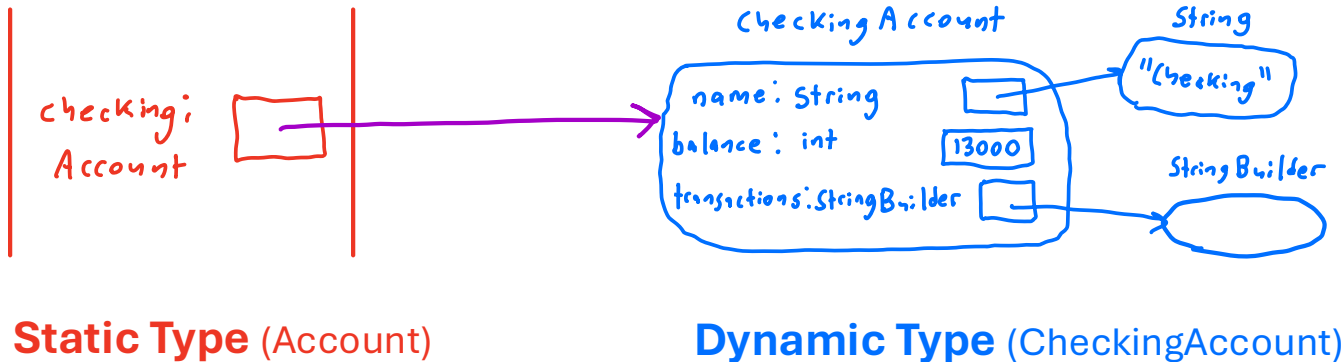


Coding Demo: Client Code with Interfaces



Dynamic vs. Static Types

```
Account checking = new CheckingAccount("Checking", 13000);
```



The Compile Time Reference Rule

A variable's **static type** dictate's the **compiler**'s view of the object it references.
The compiler can't "see" the **dynamic type**, which becomes apparent at **runtime**.

Subtype Relationships

`CheckingAccount` is a **subtype** of `Account` since it's a more specific descriptor.

All `CheckingAccounts` are `Accounts`.

Not all `Accounts` are `CheckingAccounts`.

Notation:

Subtype Substitution

We can often use a **subtype** in place of its **supertype** in our code.

Hint: Use real-world types
to make this easier to
conceptualize!


Cat


Animal

Substitution 1: Assignment

Given two types **S** and **T** with **S** <: **T**

Subtype Substitution

We can often use a **subtype** in place of its **supertype** in our code.

Hint: Use real-world types
to make this easier to
conceptualize!


Cat


Animal

Substitution 2: Parameters

Given two types **S** and **T** with **S** <: **T**

Subtype Substitution

We can often use a **subtype** in place of its **supertype** in our code.

Hint: Use real-world types
to make this easier to
conceptualize!


Cat


Animal

Substitution 3: Return Values

Given two types **S** and **T** with **S** <: **T**

Polymorphism

When we write code, we'd like it to handle as many use cases as possible.

Duplicated code is "*smelly*" !



Coding Demo: Many Accounts



Dynamic Dispatch

The **dynamic type** of an object determines which "version" of the method is invoked on it at **runtime**.

(we'll have more to say about this in the next lecture)

```
Account[] accounts = ...;

// initialize and interact with accounts

for (int i = 0; i < accounts.length; i++) {
    accounts[i].transactionReport();
}
```

accounts:
Account[]



Dynamic vs Static Types: Big Ideas

The **static type** of a variable determines which behaviors can be invoked through that variable.
(Compile-Time Reference Rule)

The **dynamic type** of an object determines how that behavior is actually carried out at runtime.
(Dynamic Dispatch)

Reference Type Coercion

Sometimes, the **Compile-Time Reference Rule** gets in the way, and we need to "adjust the compiler's view" to access a behavior.

```
Account savings = new SavingsAccount("Savings", 230000, 3.0);  
System.out.println(savings.interestRate() + "%");
```

Casting can allow us to adjust the compiler's view, but it's a bit subtle. Casts can fail in different ways at different times.

At **compile-time**:

At **runtime**: