Lecture 9: Interfaces and Polymorphism

CS 2110

September 23, 2025

Today's Learning Outcomes

39. Implement an interface using a given state representation according to its specifications.

40. Compare and contrast *static types* and *dynamic types*.

41. Identify three scenarios where subtype substitution is permitted.

42. Explain the benefits of leveraging *polymorphism* in object-oriented code.

43. 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 Matt have been worried?

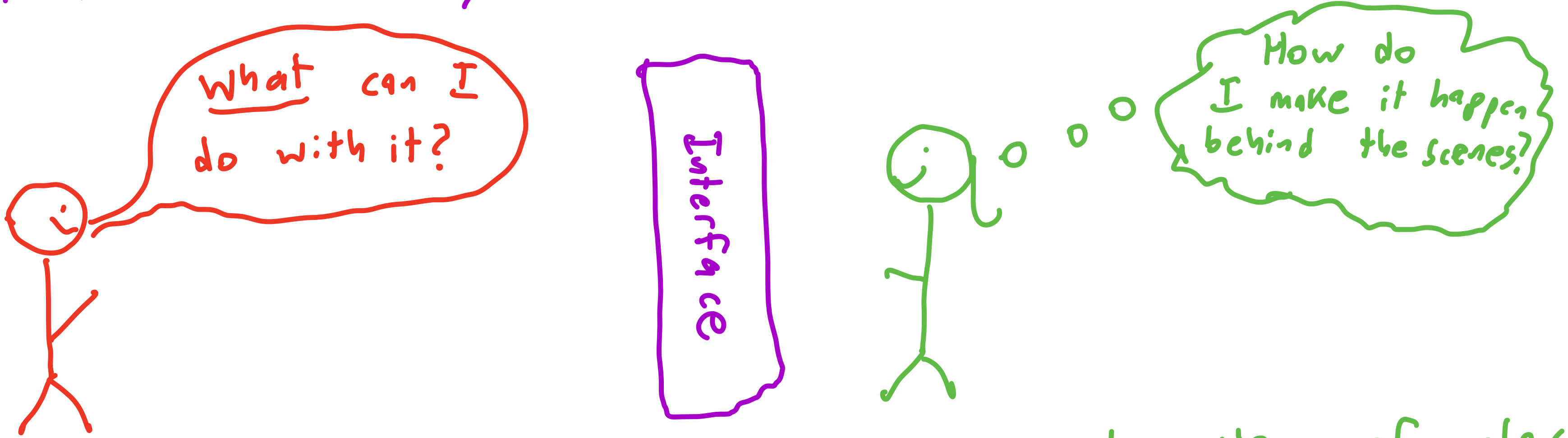
No. The way the driver interacts with the cars is nearly the same; they offer drivers the same interface (set of exposed features)

Other real-world interfaces:

- Power grid: just plug into outlet of right shape
- Data cables, file formats, etc.

Abstraction Barriers

Interfaces present an <u>abstraction barrier</u> between implementer and client of a system



API (application programming interface) view

- method signatures and specifications

source-code view of class

- state representation
- invariants
- method bodies

Interfaces in Java

- New Java construct that is an alternative to a class.

- Assigns a new type name to a collection of guaranteed behaviors without committing to how these behaviors are implemented
  - contains only (public) method signatures and specs
  - no fields
  - no method bodies

  } "behind the scenes" details

Models a contract between clients + implementers

Coding Demo: `Account` interface

Implementing an Interface

Interfaces don't have state (no fields), so can't be constructed.
Classes provide blueprints for objects that can be constructed.

We link a class to an interface using the "implements" keyword:

```
public class CheckingAccount implements Account {
    ...
}
```

To fulfill its end of the Account contract, CheckingAccount must provide method bodies for all Account methods that meet their spec.

Coding Demo: `CheckingAccount` class

Specifications and @Override

The @Override annotation signifies that a class' method definition is based on declaration from "higher up" (e.g. in an interface it implements)

- must match signature exactly* } contract
- must conform to the specifications }

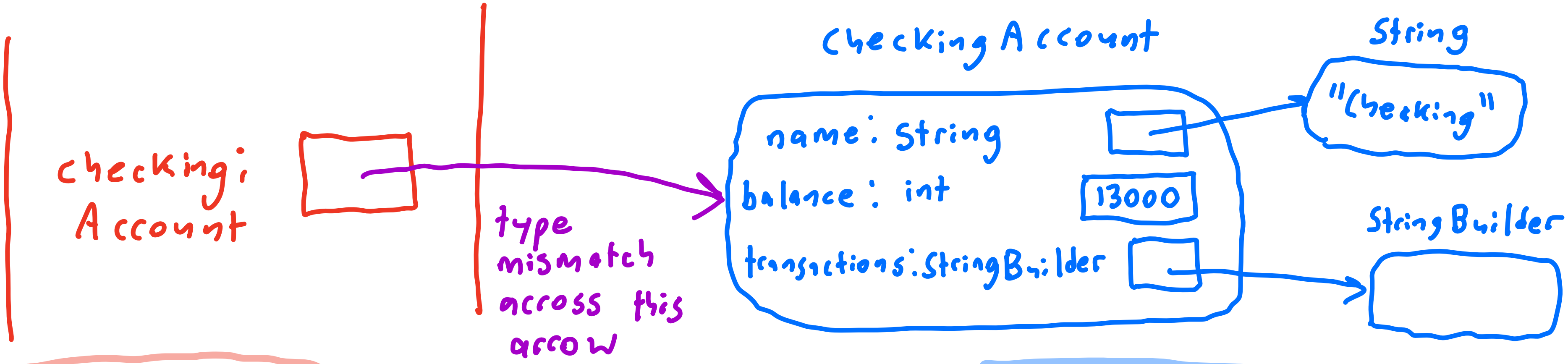
If the higher specs match exactly, no need for new JavaDoc. @Override "pulls down" spec.

If the class definition <u>refines</u> spec (adds new post-conditions) then new complete documentation is needed.

Coding Demo: Client Code with Interfaces

Dynamic vs. Static Types

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



static type (Account)
- describes variable
- telling compiler how it should "view" object ref'd by checking

dynamic type (CheckingAccount)
- describes object
- which blueprint was used at runtime to build this object

The Compile Time Reference Rule

A variable's static type dictates the compiler's view of the object it references.

- Compiler can't "see" dynamic types

The compiler is responsible for enforcing type safety of our programs.

⟹ We can only call methods that exist for the static type of a variable.

CTRR

MOST IMPORTANT RULE OF THE COURSE!!!

Subtype Relationships

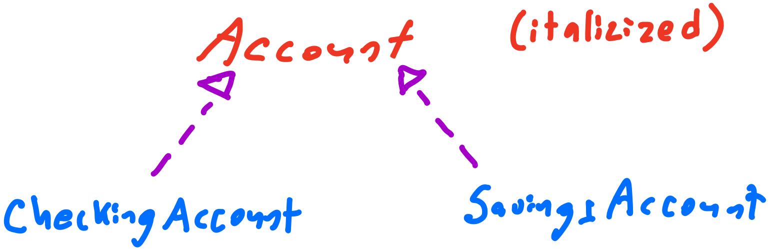
We say CheckingAccount is a <u>subtype</u> of Account since it's a more specific descriptor.

All CheckingAccounts are Accounts

Not all Accounts are CheckingAccounts

Notation CheckingAccount <: Account

Type Hierarchy



Implementing an interface establishes a subtype relationship.

In "type mismatched" variable assignments

dynamic type <: static type

Subtype Substitution

Often, we can use a subtype in place of its supertype.

(Hint: real-world example) Cat <: Animal

1. Assignment — If S <: T, we can assign an S object reference to a variable with static type T

"An Animal variable can store a Cat"

2. Parameters — If S <: T, we can pass an S object reference as an argument to serve as a T parameter

"If a method expected to get an Animal, it's happy to get a Cat"

3. Return value — If S <: T and f() has return type T, it can return an S object reference.

"If a method promises to return an Animal, it's allowed to return a Cat"

Poll Everywhere

PollEv.com/2110fa25 text 2110fa25 to 22333

Suppose that B <: A. Which line of code will compile if it is inserted "// HERE"?

```
static A foo(B b) { ... }

static B bar(A a) { ... }

public static void main(...) {
  A a = new A();
  B b = new B();
  // HERE
}
```

- `A x = foo(a);` (A) — no: foo needs a B
- `B x = foo(b);` (B) — no: foo might return a different A
- `A x = bar(b);` (C) — supertype is fine; subtype is fine
- None of Them (D)

Polymorphism

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

- Avoids code duplication
- Improves readibility, maintainability

Polymorphic code is able to naturally handle multiple types of data with the same code lines.

Poly = many, morphic = shape

Interfaces enable <u>subtype</u> polymorphism.

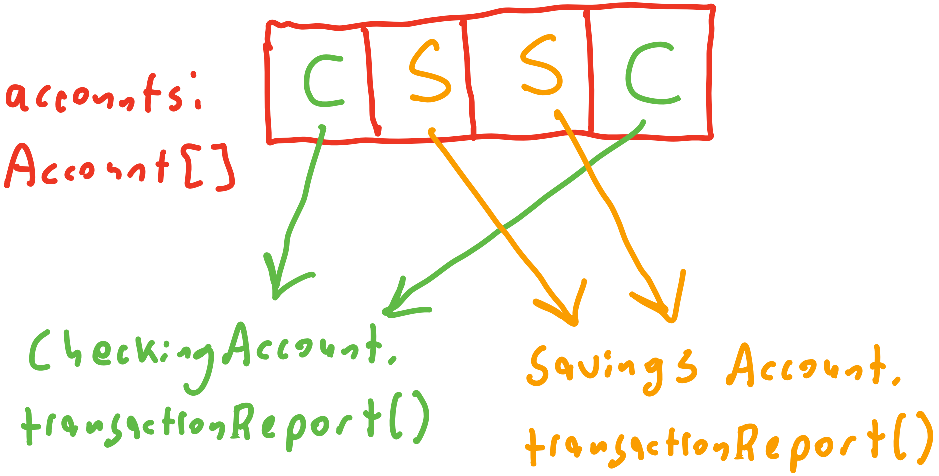
(Other varieties coming soon...)

Coding Demo: Many Accounts

Dynamic Dispatch

The dynamic type of an object determines which "version" of a method gets invoked on it.

(more to say about this next lecture...)



subtype polymorphism

same code, different behaviours

```
Account[] accounts;

// initialize and interact with accounts

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

Poll Everywhere

PollEv.com/2110fa25 text **2110fa25** to **22333**

Given these type declarations (top), what happens when we try to run the following client code (bottom)?

```
interface Phone {
   void makeCall();
   void sendText(); }

class Pixel implements Phone {
  void makeCall() { … }
  void sendText() { … }
  void takePicture() { … } }
```

```
Phone myPixel = new Pixel();
myPixel.takePicture();
```

Compile Time Reference Rule!

- Compiler Error (Line 1) **(A)**
- Compiler Error (Line 2) **(B)**
- Runtime Error **(C)**
- Runs OK (Dynamic Dispatch) **(D)**

Dynamic vs Static Types: Big Ideas

The static type of a variable determines which behaviors can be called on that variable.
(Compile Time Reference Rule)

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

Reference Type Coercion

Sometimes, the CTRR 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(((SavingsAccount) savings).interestRate() + "%");
```

We can use casting to adjust the compiler's view (lower in type hierarchy)

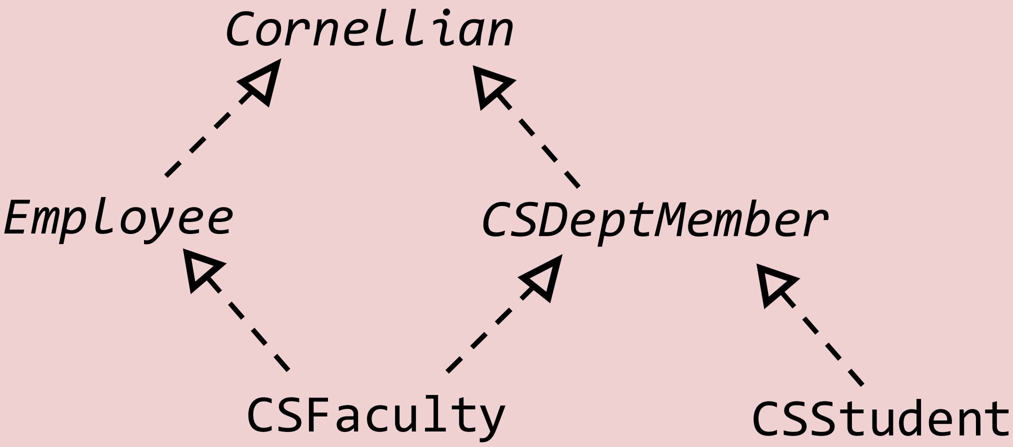
Compile Time: Compiler "trusts cast" if it can possibly succeed

Runtime: Dynamic type determines if cast actually works (or if exception is thrown)

Poll Everywhere

PollEv.com/2110fa25 text **2110fa25** to **22333**

Given the following type hierarchy and variable declarations, which cast will ***not*** compile?




```
Cornellian c;    Employee e;    CSFaculty f;
CSStudent s;     CSDeptMember d;
```

upcast: not needed, but ok

`d = (CSDeptMember) s;` **(A)**

works if c → CSFaculty

`f = (CSFaculty) c;` **(B)**

works if d → CSFaculty

`e = (Employee) d;` **(C)**

no, student can't be Faculty

`f = (CSFaculty) s;` **(D)**