

Poll Everywhere

PollEv.com/javabear

text javabear to 22333



```
public class Interval {  
    private int left, right;  
  
    public Interval(int left, int right) {  
        this.left = left; this.right = right;  
    }  
  
    public void translate(int shift) {  
        this.left += shift; this.right += shift;  
    }  
  
    public String toString() {  
        return "(" + left + "," + right + ")";  
    }  
}
```

What is the value of `a.toString()` at the end of this code snippet?

```
Interval a = new Interval(1, 5);  
Interval b = a;  
b.translate(3);  
b = new Interval(2, 4);
```

Poll Everywhere

PollEv.com/javabear

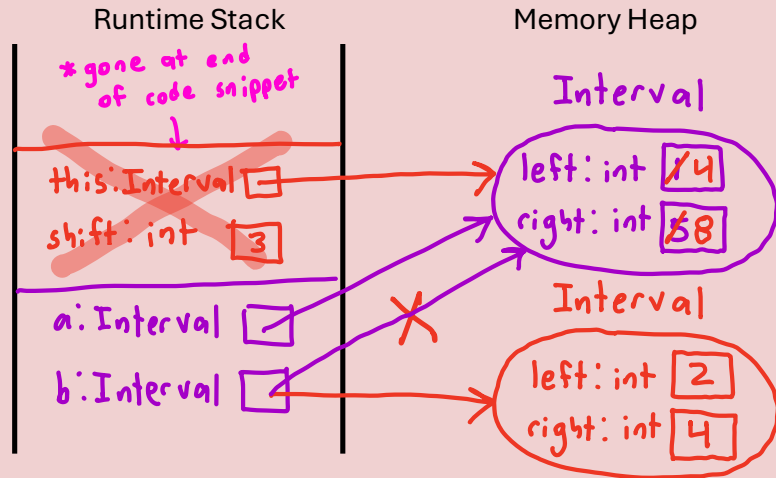
text javabear to 22333



```
public class Interval {  
    private int left, right;  
  
    public Interval(int left, int right) {  
        this.left = left; this.right = right;  
    }  
  
    public void translate(int shift) {  
        this.left += shift; this.right += shift;  
    }  
  
    public String toString() {  
        return "(" + left + "," + right + ")";  
    }  
}
```

```
Interval a = new Interval(1, 5);  
Interval b = a;  
b.translate(3) ← visible through a  
b = new Interval(2, 4);
```

"(4,8)"





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?

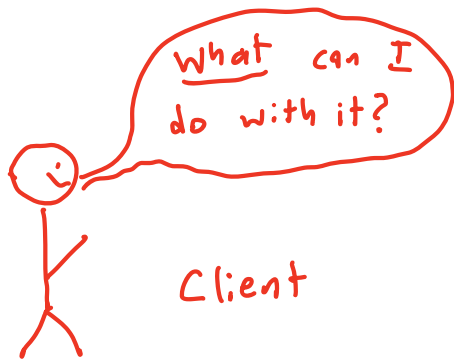
No. The way the driver interacts with the cars is nearly the same; they offer drivers the same interface (set of exposed features)

Other real-world interfaces:

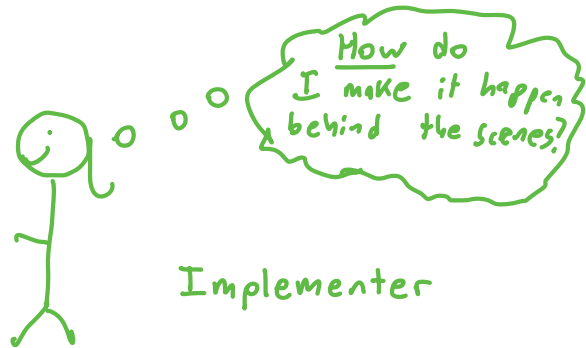
- Power grid: just plug into outlet of right shape
- Data cables, file formats, etc.

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).

- 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

Type that models a contract between clients + implementers



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.

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. (but can also define other methods, too)



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).

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

Our Javadoc rules for overridden methods:

If the "higher" specs match **exactly**: No need for new Javadoc
`@Override` "pulls down" spec

If the definition **refines** the "higher" specs: New complete Javadoc
additional details/ guarantees

Poll Everywhere

PollEv.com/javabear

text **javabear** to 22333



When it *refines* the specs of an overridden method, what should a class be allowed to do?

Add pre-conditions, **Add** post-conditions (A)

Add pre-conditions, **Remove** post-conditions ✗ (B)

Remove pre-conditions, **Add** post-conditions (C)

↙ *Method becomes more robust*

Remove pre-conditions, **Remove** post-conditions ✗ (D)

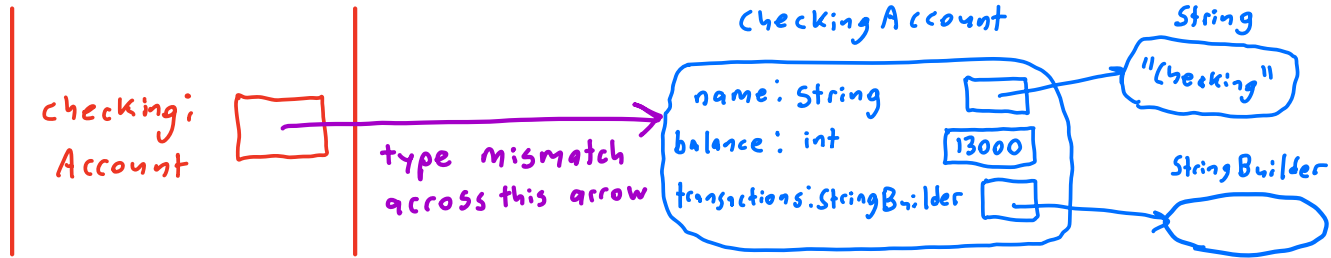


Coding Demo: Client Code with Interfaces



Dynamic vs. Static Types

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



Static Type (Account)

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

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

⇒ We can only call instance methods that exist
CTRR for the static type of the target.

MOST IMPORTANT RULE OF THE COURSE!!!!

Subtype Relationships

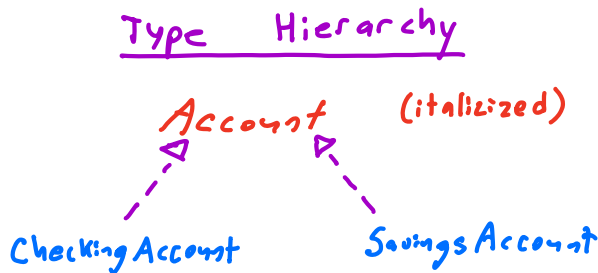
`CheckingAccount` is a **subtype** of `Account` since it's a more specific descriptor.

All `CheckingAccounts` are `Accounts`.

Not all `Accounts` are `CheckingAccounts`.

Notation:

`CheckingAccount <: Account`



Implementing an interface establishes a subtype relationship.

In "type mismatched" variable assignments

dynamic type <: static type

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**

We can assign an **S** object reference to a variable
with static type **T**

"An **Animal** variable can store a **Cat**."

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

```
void f(T param) {...}
```

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

We can pass an **S** object reference as an argument to serve as a **T** parameter.

"If a method expects an **Animal**, it's happy to get a **Cat**."

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

$T \quad f() \quad \{...\}$

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

If a method has a 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/javabear

text javabear to 22333



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

*real-world: A = Animal
types B = Bear*

*no: foo() needs
a B*

```
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()
might return
different A*

B x = foo(b); ~~✗~~ (B)

*supertype
is
fine*

*subtype
is
fine*

A x = bar(b); **(C)**

None of Them (D)

Polymorphism

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

Duplicated code is "smelly"!

– Not as readable, maintainable, etc.

Poly morphic code is able to naturally handle
many shape multiple types of data with the same
code lines.

Interfaces enable subtype polymorphism.

(other varieties coming soon...)



Coding Demo: Many Accounts



Dynamic Dispatch

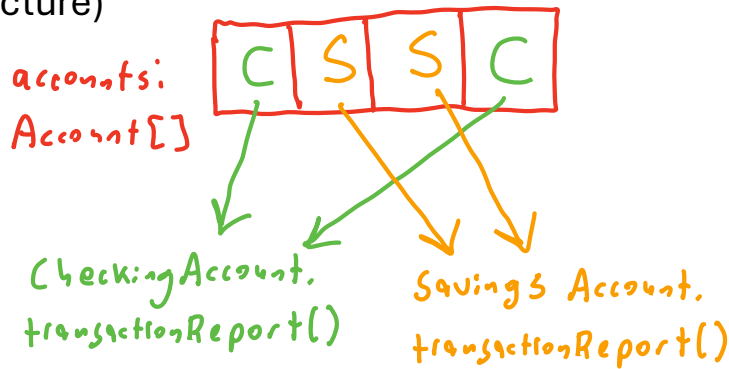
The **dynamic type** of an object determines which "version" of the method is invoked on it at **runtime**.

(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();
}
```



enables subtype polymorphism: same code, different behaviors

Poll Everywhere

PollEv.com/javabear

text **javabear** 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() { ... }  
}
```

```
1 Phone myPixel = new Pixel();  
2 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 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)