

Where we left off...

- We'd developed a calculator application, but many of the classes had very similar method bodies
- "Reusing" code via cut-and-paste is tedious and error-prone
- Let's review the situation:

What's wrong with our code?

- Most of the `eval` methods in the `BinaryOp` classes look roughly the same:
 - evaluate left operand, getting a float
 - evaluate right operand, getting a float
 - perform some operation on these two floats

What's wrong with our code?

- Most of the `eval` methods in the `BinaryOp` classes look roughly the same:
 - evaluate left operand, getting a float
 - evaluate right operand, getting a float
 - perform some operation on these two floats

Note that two of these steps are the same for every operation!

- Most of the `eval` methods in the `BinaryOp` classes look roughly the same:
 - evaluate left operand, getting a float
 - evaluate right operand, getting a float
 - perform some operation on these two floats

Note that two of these steps are the same for every operation!

- Most of the `eval` methods in the `BinaryOp` classes look roughly the same:

It would be nice to write these *once* and reuse them!

- perform some operation on these two floats

Motivation: inheritance

- Interfaces enable us to write code without knowing what specific classes it uses
- Interfaces also enable us to write classes that can be used by code that doesn't know about them

Motivation: inheritance

- However, when we declare an interface, we only specify method names, not method implementations
- (Each implementing class provides its own implementations)
- Sometimes, it would be nice to reuse method implementations by *extending* classes

Extending classes

- Often, we would like a class to inherit behavior and fields from another class
- This is the case when you have an “is-a” relationship between two classes
- #1 critical mistake of novice OO programmers: overusing inheritance

Examples:

- A car *is a* vehicle
 - A 1985 Yugo *is a* car (roughly speaking)
- A checking account *is a* bank account
- A platypus *is a* monotreme
 - A monotreme *is a* mammal
 - A mammal *is a* vertebrate
 - ...

“Inherited” behaviors

- Vehicle
 - abstract concept
- Car
 - adds engine, fuel tank data
 - ignition, accelerate, brake, steer methods
- Yugo
 - adds fake leather seats

...or wider and narrower types

- Vertebrates
 - Lots of wildly different animals
- Mammals
 - warm-blooded, milk-producing animals
- Monotremes
 - that's just weird!

What does this mean for us?

- One Java class can “extend” another
- Extending class has all of the fields of the *base class*
 - Can add its own; private is still private
- Extending class has all of the methods of the base class
 - Can add, override, or inherit

Inheritance is a language feature that allows a *derived class* to extend a *base class*.

**How do we declare that a class
extends another class?**

We declare that one class extends another class with the extends keyword.

```
public class MultiClicker
    extends Clicker {
    public void click(int k) {
        for(int i = 0; i < k; i++)
            // can't access clicks!
            click();
        }
    }
}
```

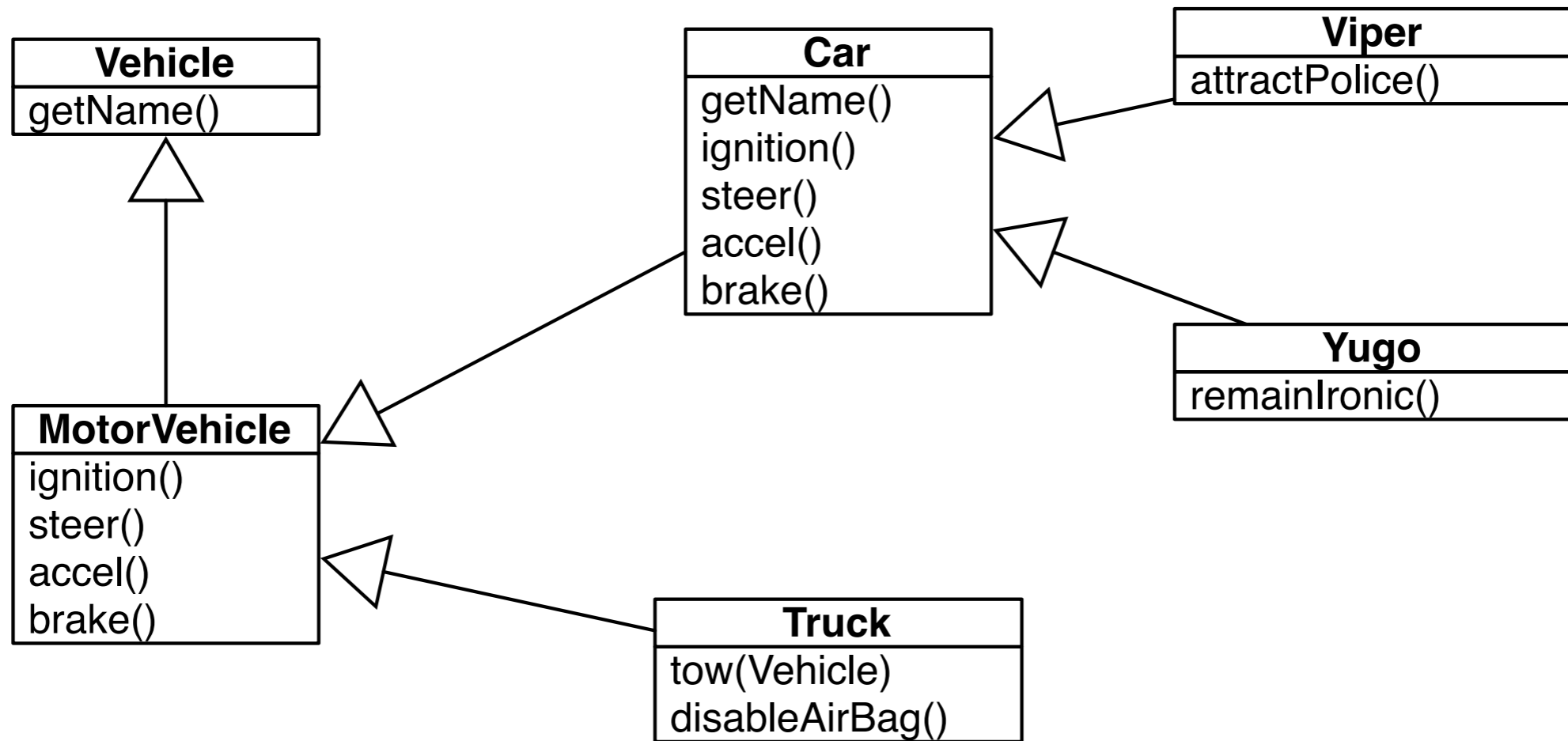
**What does the derived
class “inherit”?**

The derived class inherits all of the fields of the base class and all of its public methods.

(More on fields soon.)

The derived class may...

- *add* methods to the base class,
- *inherit* methods from the base class, or
- *override* methods declared in the base class



```
Vehicle v;  
// ...  
v.getName();
```

Polymorphic method dispatch
chooses the right method based
on the type of the base object.

(Just like with interfaces.)

(How is this different from overloading?)

You can explicitly invoke methods or constructors of your superclass with the `super` keyword.

Any questions?

Example

- Bank accounts are ubiquitous
- Almost every kind of bank account allows you to check your balance and deposit and withdraw funds
- Other kinds have additional capabilities and restrictions

```
public class BankAccount {  
    private float balance;  
  
    public void deposit(float amt) {  
        balance += amt;  
    }  
  
    public void withdraw(float amt) {  
        balance -= amt;  
    }  
  
    public float getBalance() {  
        return balance;  
    }  
}
```

Checking accounts

- Say we wanted to implement a checking account:
 - Just like a bank account, but \$0.10 fees for every withdrawal if balance goes under \$200.
- How would we do this?

```
public class BankAccount {
    private float balance;

    public void deposit(float amt) {
        balance += amt;
    }

    public void withdraw(float amt) {
        balance -= amt;
    }

    public float getBalance() {
        return balance;
    }
}
```

```
public class CheckingAccount
    extends BankAccount {

    public void withdraw(float amt) {
        balance -= amt;
        if (balance < 200.00)
            balance -= 0.10;
    }
}
```

```
public class CheckingAccount
    extends BankAccount {

    public void withdraw(float amt) {
        balance -= amt;
        if (balance < 200.00)
            balance -= 0.10;
    }
}
```

This won't work. Why?

```
public class CheckingAccount
    extends BankAccount {
    private float balance;
    public void withdraw(float amt) {
        balance -= amt;
        if (balance < 200.00)
            balance -= 0.10;
    }
}
```

```
public class CheckingAccount
    extends BankAccount {
    private float balance;
    public void withdraw(float amt) {
        balance -= amt;
        if (balance < 200.00)
            balance -= 0.10;
    }
}
```

This won't work. Why?

Fields

- You can only access private fields in methods declared in the same class
- If you declare a private field in an extending class with the same name as a field in the base class, then you'll have *two different fields!*

However....

You can explicitly invoke methods or constructors of your superclass with the `super` keyword.

```
public class CheckingAccount
    extends BankAccount {
    public void withdraw(float amt) {
        super.withdraw(amt);
        if (getBalance() < 200.00)
            super.withdraw(0.10);
    }
}
```

Fields can be confusing.

Field names are
particular to a class.

CheckingAccount.balance
is a *different field* from
BankAccount.balance.

(An instance of CheckingAccount will have *both*.)

```
class Base {  
    private int x;  
    public void setX(int x) { this.x = x; }  
    public int getX() { return x; }  
}
```

```
class Derived extends Base {  
    private int x;  
    public void setX(int x) { this.x = x; }  
}
```

```
// ...
```

```
    Derived d = new Derived();  
    d.setX(5);  
    System.out.println(d.getX());
```

Object

- Root of “inheritance hierarchy”
- Provides no fields, but several useful methods
 - `toString()`
 - `equals()`
 - `hashCode()`

Everything reachable by a
reference is an Object.

toString()

- Generates a “human-readable” representation of an object’s state
 - What should this include?
- Default implementation isn’t very useful: class name and *hash code*

equals(Object o)

- Returns true if this “is equal to” o, false otherwise.
- What does it mean to be “equal to” an object?
- Default implementation isn’t very useful: `return this == o;`

equals(Object o)

- Should be reflexive:
 - `x.equals(x)`
- transitive:
 - `x.equals(y) && y.equals(z) && x.equals(z)`
- and symmetric:
 - `x.equals(y) && y.equals(x)`

```
Object x, y;  
// ...  
x = y;  
return x == y;
```

```
Object x, y;  
// ...  
x = y;  
return x.equals(y);
```

hashCode()

- Returns a *hash code*, or an `int` that identifies an object
- This is used by a lot of library code
- Default implementation is probably fine; it's extremely thorny to change it.
- This is especially true if you change `equals()`!