CSCI 334:
Principles of Programming Languages

Lecture 21: OO III & Tech. communication

Instructor: Dan Barowy

Williams

Topics

Topics

Virtual dispatch

Topics

Virtual dispatch

How to give a good talk

Topics

Virtual dispatch

How to give a good talk

Project Q&A

SWELL user testing

This weekend, 30-40 minutes.

`https://bit.ly/2EgWKgi`

It would be a real help if you have the time!

Refresher: Dynamic Dispatch

Refresher: Dynamic Dispatch

- Dynamic dispatch is the OO mechanism for polymorphism.

Refresher: Dynamic Dispatch

- Dynamic dispatch is the OO mechanism for polymorphism.
- Functions ("methods") are always bound to an object (or class).

Refresher: Dynamic Dispatch

- Dynamic dispatch is the OO mechanism for polymorphism.
- Functions ("methods") are always bound to an object (or class).
- A method is called ("dispatched") by sending a "message" to the "selector" of an object.

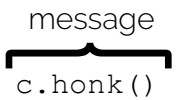
Refresher: Dynamic Dispatch

- Dynamic dispatch is the OO mechanism for polymorphism.
- Functions ("methods") are always bound to an object (or class).
- A method is called ("dispatched") by sending a "message" to the "selector" of an object.

```
c.honk()
```

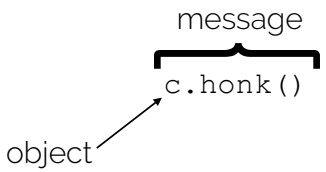
Refresher: Dynamic Dispatch

- Dynamic dispatch is the OO mechanism for polymorphism.
- Functions ("methods") are always bound to an object (or class).
- A method is called ("dispatched") by sending a "message" to the "selector" of an object.



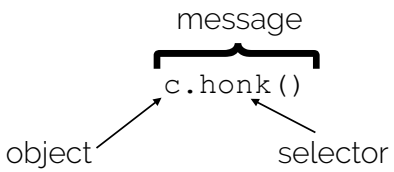
Refresher: Dynamic Dispatch

- Dynamic dispatch is the OO mechanism for polymorphism.
- Functions ("methods") are always bound to an object (or class).
- A method is called ("dispatched") by sending a "message" to the "selector" of an object.



Refresher: Dynamic Dispatch

- Dynamic dispatch is the OO mechanism for polymorphism.
- Functions ("methods") are always bound to an object (or class).
- A method is called ("dispatched") by sending a "message" to the "selector" of an object.



Dynamic Dispatch

- Dynamic dispatch is an algorithm for finding an object's method corresponding to a given selector name.

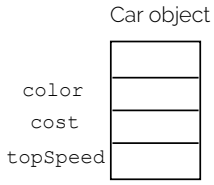
Dynamic Dispatch

- Dynamic dispatch is an algorithm for finding an object's method corresponding to a given selector name.

Car object

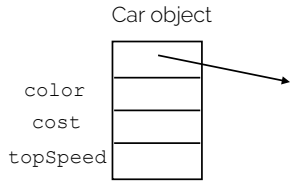
Dynamic Dispatch

- Dynamic dispatch is an algorithm for finding an object's method corresponding to a given selector name.



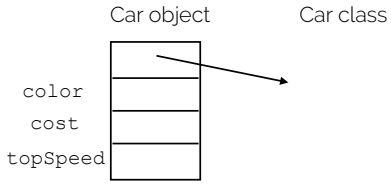
Dynamic Dispatch

- Dynamic dispatch is an algorithm for finding an object's method corresponding to a given selector name.



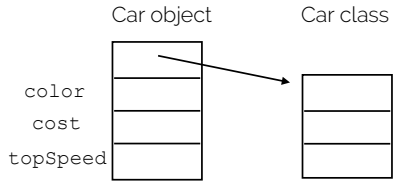
Dynamic Dispatch

- Dynamic dispatch is an algorithm for finding an object's method corresponding to a given selector name.



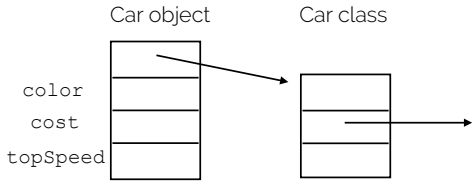
Dynamic Dispatch

- Dynamic dispatch is an algorithm for finding an object's method corresponding to a given selector name.



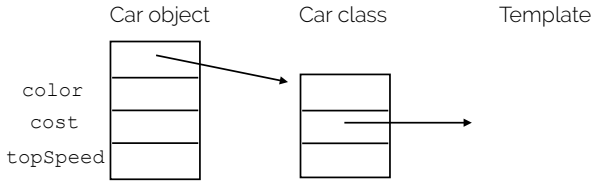
Dynamic Dispatch

- Dynamic dispatch is an algorithm for finding an object's method corresponding to a given selector name.



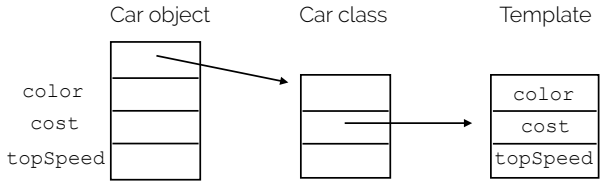
Dynamic Dispatch

- Dynamic dispatch is an algorithm for finding an object's method corresponding to a given selector name.



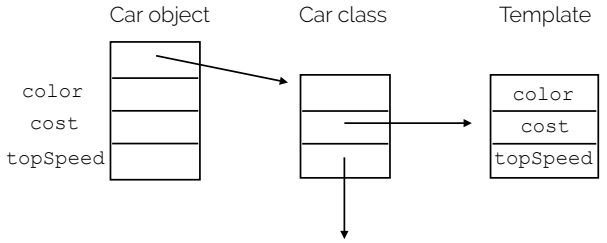
Dynamic Dispatch

- Dynamic dispatch is an algorithm for finding an object's method corresponding to a given selector name.



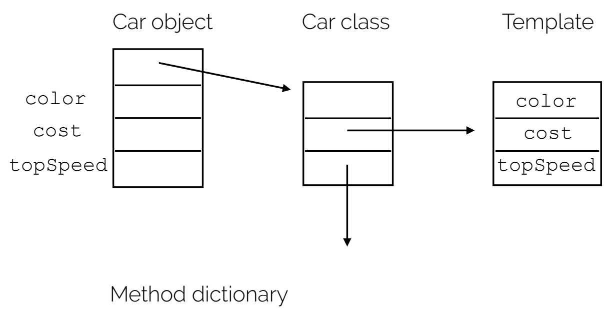
Dynamic Dispatch

- Dynamic dispatch is an algorithm for finding an object's method corresponding to a given selector name.



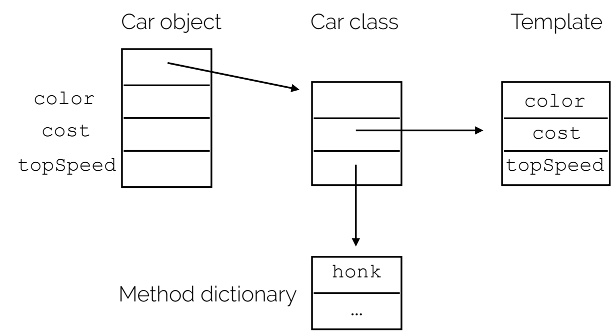
Dynamic Dispatch

• Dynamic dispatch is an algorithm for finding an object's method corresponding to a given selector name.



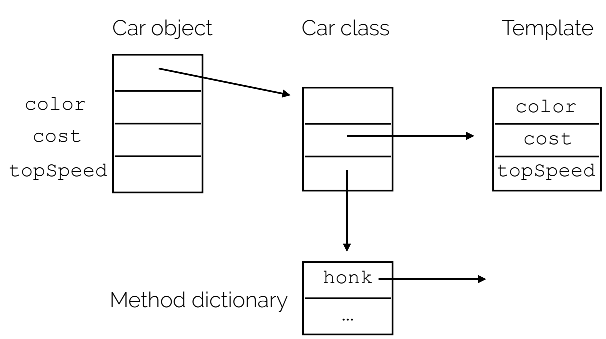
Dynamic Dispatch

• Dynamic dispatch is an algorithm for finding an object's method corresponding to a given selector name.



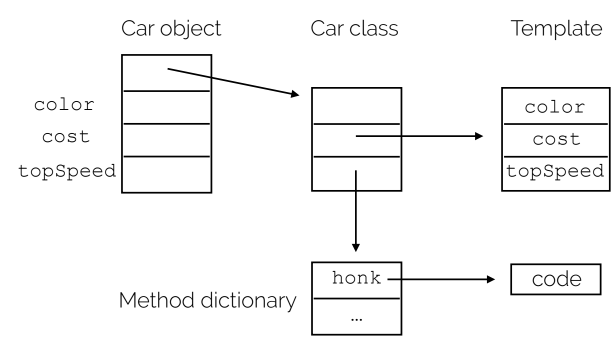
Dynamic Dispatch

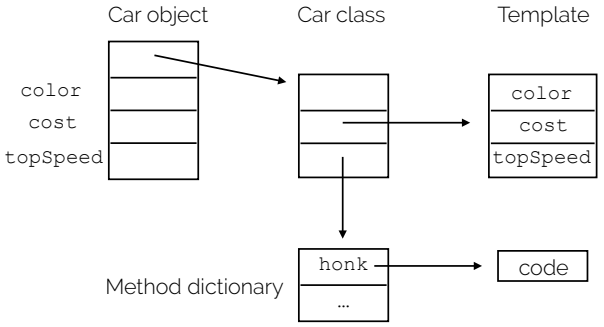
• Dynamic dispatch is an algorithm for finding an object's method corresponding to a given selector name.



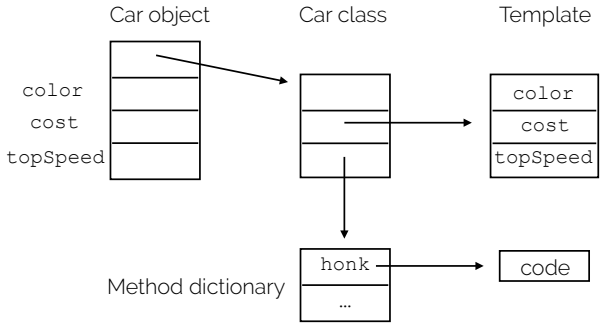
Dynamic Dispatch

• Dynamic dispatch is an algorithm for finding an object's method corresponding to a given selector name.



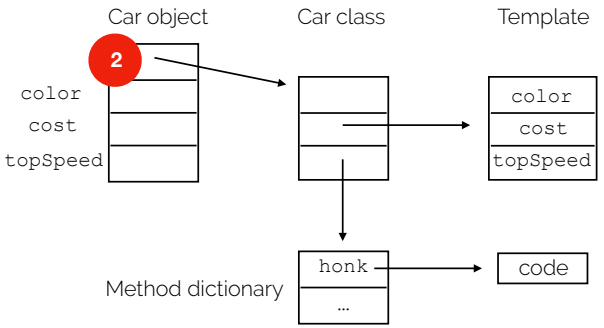


1 Call `c.honk();`



1 Call `c.honk();`

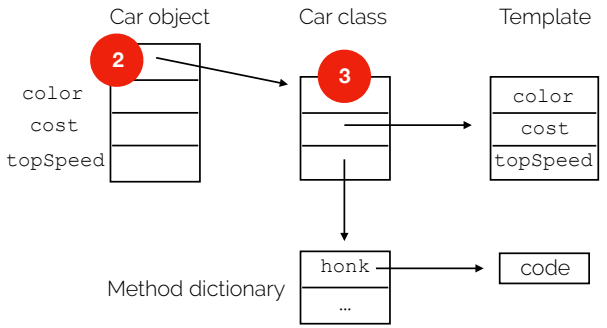
2 `honk` message dispatched to `c`



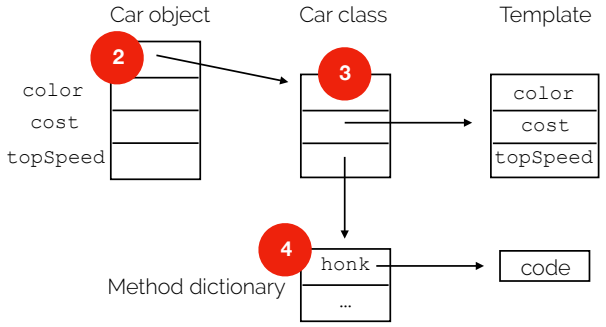
1 Call `c.honk();`

2 `honk` message dispatched to `c`

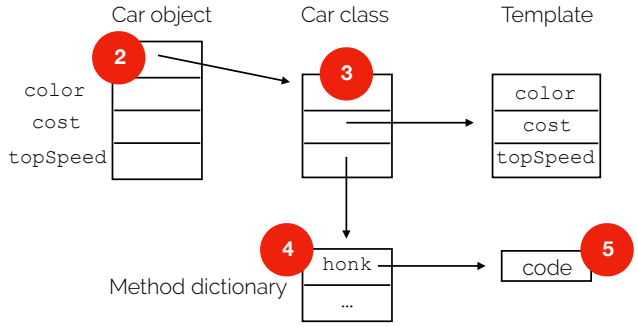
3 `honk` message forwarded to `Car`



1. Call `c.honk();`
2. `honk` message dispatched to `c`
3. `honk` message forwarded to `Car`
4. `honk` message lookup in method dictionary

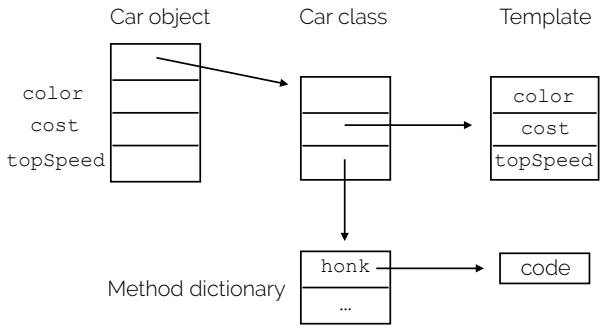


1. Call `c.honk();`
2. `honk` message dispatched to `c`
3. `honk` message forwarded to `Car`
4. `honk` message lookup in method dictionary
5. `honk` executed.



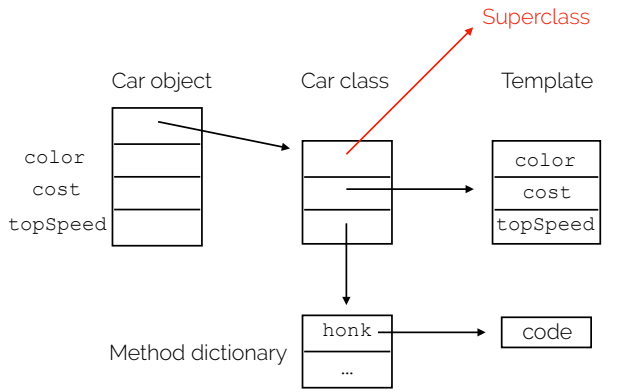
Inheritance

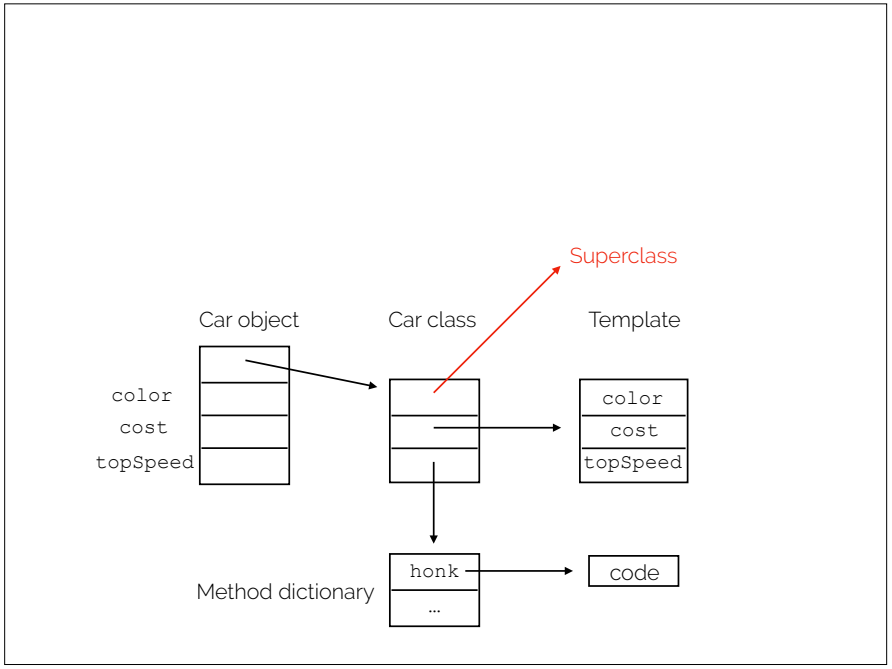
- One small change enables inheritance.

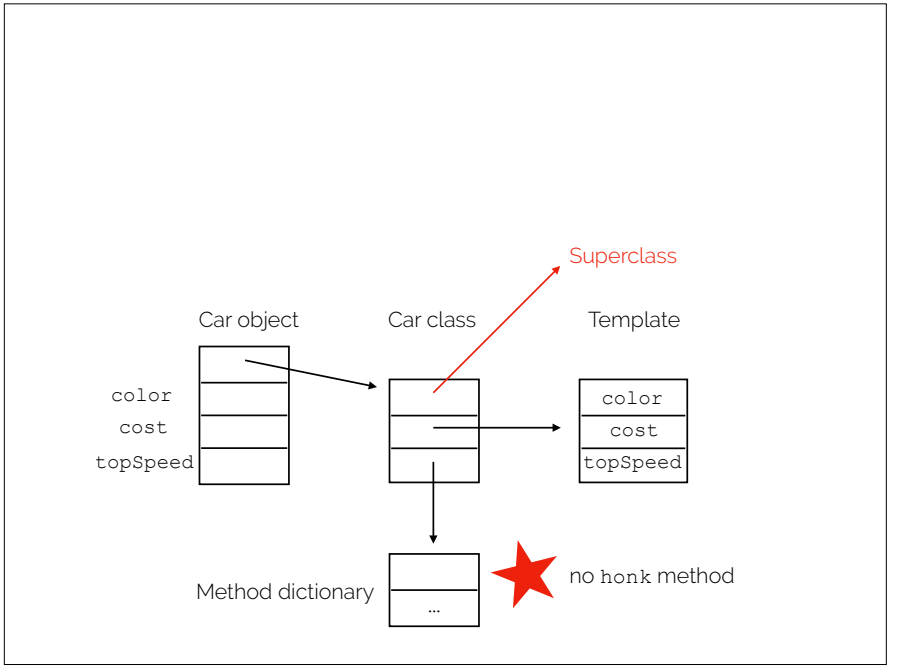


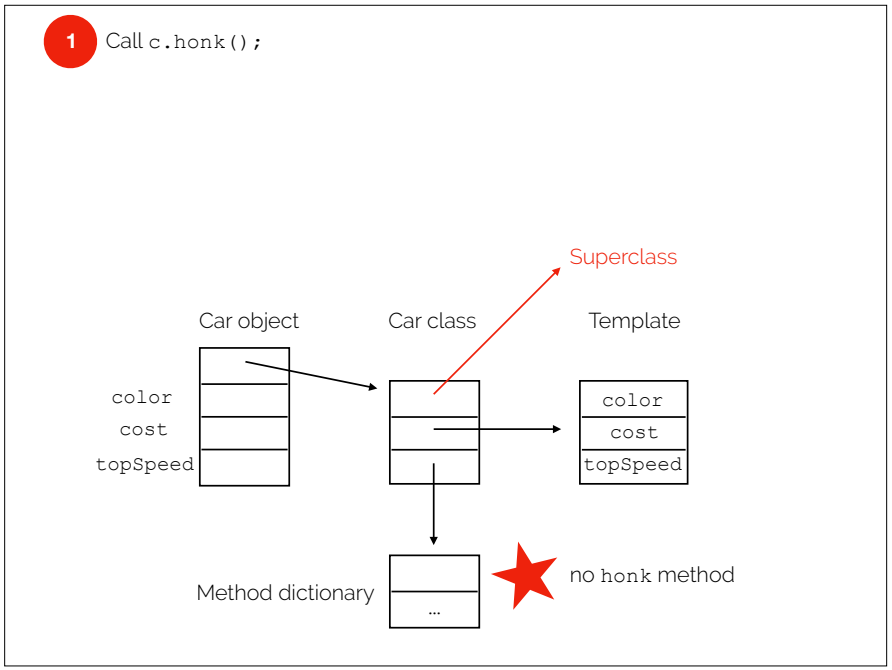
Inheritance

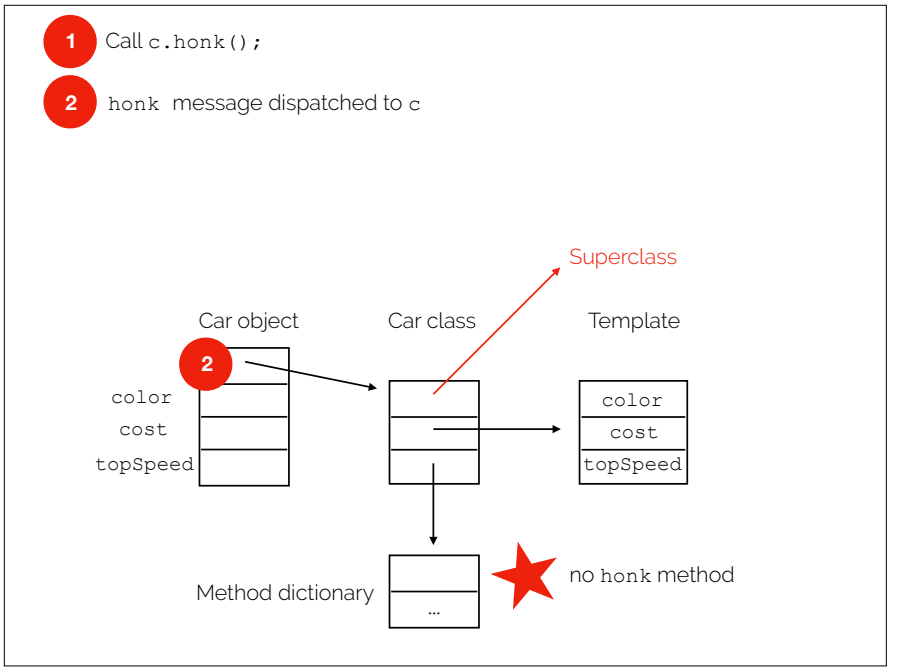
- One small change enables inheritance.



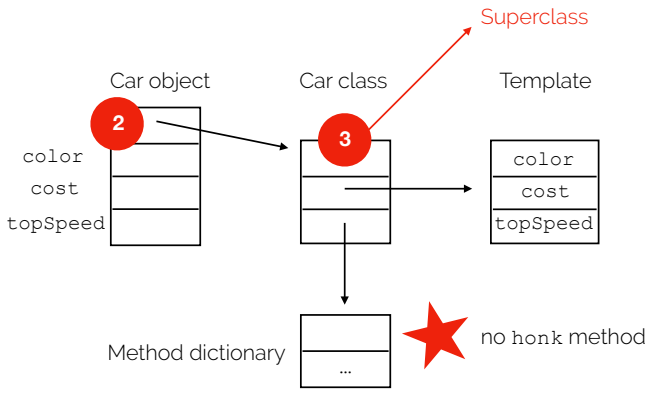

Superclass
Car object
Car class
Template
color
cost
topSpeed
color
cost
topSpeed
Method dictionary
honk
...
code


Superclass
Car object
Car class
Template
color
cost
topSpeed
color
cost
topSpeed
Method dictionary
...
no honk method

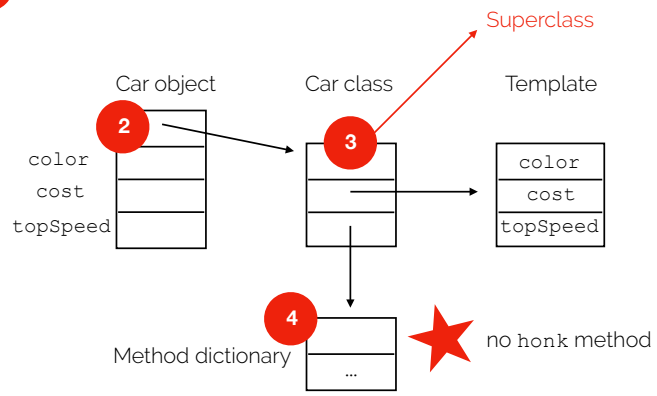

1
Call c.honk();
Superclass
Car object
Car class
Template
color
cost
topSpeed
color
cost
topSpeed
Method dictionary
...
no honk method


1
Call c.honk();
2
honk message dispatched to c
Superclass
Car object
Car class
Template
2
color
cost
topSpeed
color
cost
topSpeed
Method dictionary
...
no honk method

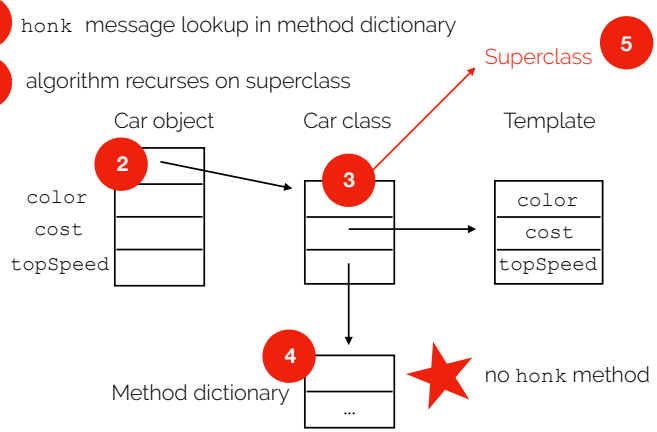
1. Call `c.honk();`
2. `honk` message dispatched to `c`
3. `honk` message forwarded to `Car`



1. Call `c.honk();`
2. `honk` message dispatched to `c`
3. `honk` message forwarded to `Car`
4. `honk` message lookup in method dictionary



1. Call `c.honk();`
2. `honk` message dispatched to `c`
3. `honk` message forwarded to `Car`
4. `honk` message lookup in method dictionary
5. algorithm recurses on superclass



Question

How expensive is dynamic dispatch?

Cost

Cost

1.dereference object

Cost

1.dereference object
2.defererence class

Cost

1.dereference object
2.defererence class
3.dereference method dictionary

Cost

1.dereference object
2.defererence class
3.dereference method dictionary
4.dereference method

Cost

1.dereference object
2.defererence class
3.dereference method dictionary
4.dereference method

for each class or superclass

Cost

1.dereference object
2.defererence class
3.dereference method dictionary
4.dereference method

for each class or superclass

O(n) method lookup

C++

Efficient object oriented programming.

C++

Efficient object oriented programming.

“Only pay for what you use”

“Only pay for what you use”

“Only pay for what you use”

What does this mean?

“Only pay for what you use”

What does this mean?

In Java, OO & other features are “always on”

"Only pay for what you use"

What does this mean?

In Java, OO & other features are "always on"

Even when they are not needed

"Only pay for what you use"

What does this mean?

In Java, OO & other features are "always on"

Even when they are not needed

```
class Math {
 public static average(int[] nums) {
   int sum = 0;
   for (int i = 0; i < nums.length; i++) {
     sum += nums[i];
   }
   return (double sum) / nums.length;
 }
}
```

What happens when a Java program starts?

What happens when a Java program starts?

1. boot up the Java Virtual Machine (JVM)

What happens when a Java program starts?

1. boot up the Java Virtual Machine (JVM)
   a. allocate Java heap, stack, and global var areas

What happens when a Java program starts?

1. boot up the Java Virtual Machine (JVM)
   a. allocate Java heap, stack, and global var areas
   b. start up garbage collector

What happens when a Java program starts?

1. boot up the Java Virtual Machine (JVM)
   a. allocate Java heap, stack, and global var areas
   b. start up garbage collector
   c. start up Just-in-Time perfmon & compiler (JIT)

What happens when a Java program starts?

1. boot up the Java Virtual Machine (JVM)
   a. allocate Java heap, stack, and global var areas
   b. start up garbage collector
   c. start up Just-in-Time perfmon & compiler (JIT)
2. load first class definition (the one with `main`)

What happens when a Java program starts?

1. boot up the Java Virtual Machine (JVM)
   a. allocate Java heap, stack, and global var areas
   b. start up garbage collector
   c. start up Just-in-Time perfmon & compiler (JIT)
2. load first class definition (the one with `main`)
   a. verify bytecode for runtime safety

What happens when a Java program starts?

1. boot up the Java Virtual Machine (JVM)
   a. allocate Java heap, stack, and global var areas
   b. start up garbage collector
   c. start up Just-in-Time perfmon & compiler (JIT)
2. load first class definition (the one with `main`)
   a. verify bytecode for runtime safety
3. load all class defs for linked code (e.g., stdlib)

What happens when a Java program starts?

1. boot up the Java Virtual Machine (JVM)
   a. allocate Java heap, stack, and global var areas
   b. start up garbage collector
   c. start up Just-in-Time perfmon & compiler (JIT)
2. load first class definition (the one with `main`)
   a. verify bytecode for runtime safety
3. load all class defs for linked code (e.g., stdlib)
   a. verify, if necessary

What happens when a Java program starts?

1. boot up the Java Virtual Machine (JVM)
   a. allocate Java heap, stack, and global var areas
   b. start up garbage collector
   c. start up Just-in-Time perfmon & compiler (JIT)
2. load first class definition (the one with `main`)
   a. verify bytecode for runtime safety
3. load all class defs for linked code (e.g., stdlib)
   a. verify, if necessary
4. allocate space for static variables

What happens when a Java program starts?

1. boot up the Java Virtual Machine (JVM)
   a. allocate Java heap, stack, and global var areas
   b. start up garbage collector
   c. start up Just-in-Time perfmon & compiler (JIT)
2. load first class definition (the one with `main`)
   a. verify bytecode for runtime safety
3. load all class defs for linked code (e.g., stdlib)
   a. verify, if necessary
4. allocate space for static variables
5. initialize static variables

What happens when a Java program starts?

1. boot up the Java Virtual Machine (JVM)
   a. allocate Java heap, stack, and global var areas
   b. start up garbage collector
   c. start up Just-in-Time perfmon & compiler (JIT)
2. load first class definition (the one with `main`)
   a. verify bytecode for runtime safety
3. load all class defs for linked code (e.g., stdlib)
   a. verify, if necessary
4. allocate space for static variables
5. initialize static variables
6. execute main

What happens when a Java program starts?

1. boot up the Java Virtual Machine (JVM)
   a. allocate Java heap, stack, and global var areas
   b. start up garbage collector
   c. start up Just-in-Time perfmon & compiler (JIT)
2. load first class definition (the one with `main`)
   a. verify bytecode for runtime safety
3. load all class defs for linked code (e.g., stdlib)
   a. verify, if necessary
4. allocate space for static variables
5. initialize static variables
6. execute main
   a. repeat loading, linking, verifying, allocation, and initialization steps as needed.

What happens when a Java program starts?

1. boot up the Java Virtual Machine (JVM)
   a. allocate Java heap, stack, and global var areas
   b. start up garbage collector
   c. start up Just-in-Time perfmon & compiler (JIT)
2. load first class definition (the one with `main`)
   a. verify bytecode for runtime safety
3. load all class defs for linked code (e.g., stdlib)
   a. verify, if necessary
4. allocate space for static variables
5. initialize static variables
6. execute main
   a. repeat loading, linking, verifying, allocation, and initialization steps as needed.
   b. periodically run the garbage collector

What happens when a Java program starts?

1. boot up the Java Virtual Machine (JVM)
   a. allocate Java heap, stack, and global var areas
   b. start up garbage collector
   c. start up Just-in-Time perfmon & compiler (JIT)
2. load first class definition (the one with `main`)
   a. verify bytecode for runtime safety
3. load all class defs for linked code (e.g., stdlib)
   a. verify, if necessary
4. allocate space for static variables
5. initialize static variables
6. execute main
   a. repeat loading, linking, verifying, allocation, and initialization steps as needed.
   b. periodically run the garbage collector
   c. run the JIT constantly, in a separate thread

"Only pay for what you use"

"Only pay for what you use"

```
class Math {
 public static average(int[] nums) {
   int sum = 0;
   for (int i = 0; i < nums.length; i++) {
     sum += nums[i];
   }
   return (double sum) / nums.length;
 }
}
```

"Only pay for what you use"

```
class Math {
 public static average(int[] nums) {
   int sum = 0;
   for (int i = 0; i < nums.length; i++) {
     sum += nums[i];
   }
   return (double sum) / nums.length;
 }
}
```

We're not using any objects!

“Only pay for what you use”

```
class Math {
 public static average(int[] nums) {
   int sum = 0;
   for (int i = 0; i < nums.length; i++) {
     sum += nums[i];
   }
   return (double sum) / nums.length;
 }
}
```

We’re not using any objects!

In C++, the “no class” program is as fast as C

“Only pay for what you use”

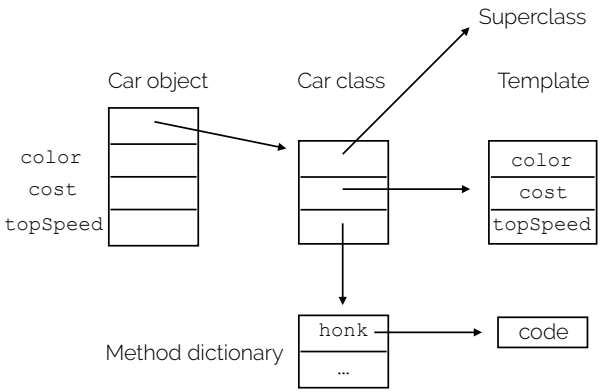
```
class Math {
 public static average(int[] nums) {
   int sum = 0;
   for (int i = 0; i < nums.length; i++) {
     sum += nums[i];
   }
   return (double sum) / nums.length;
 }
}
```

We’re not using any objects!

In C++, the “no class” program is as fast as C

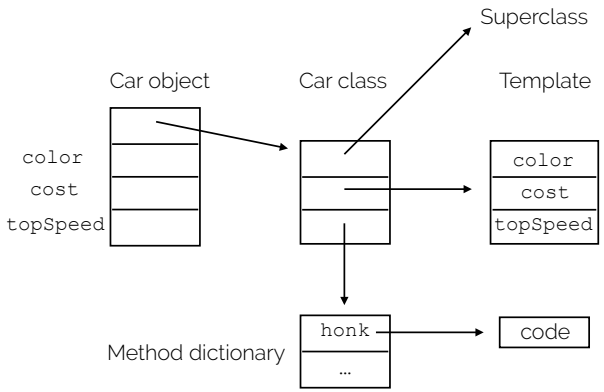
Without classes, C++ is basically C

C++ does OO efficiently



C++ does OO efficiently

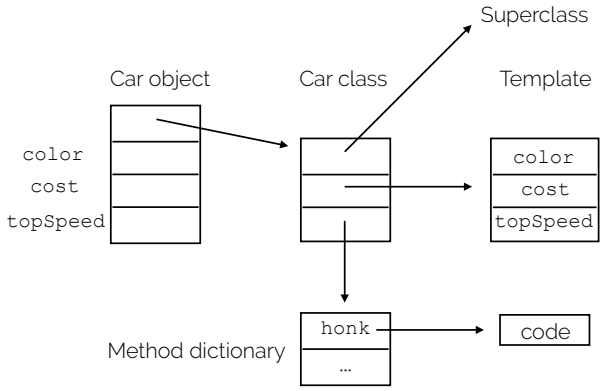
C++ static methods are just C procedures. No classes needed.



C++ does OO efficiently

C++ static methods are just C procedures. No classes needed.

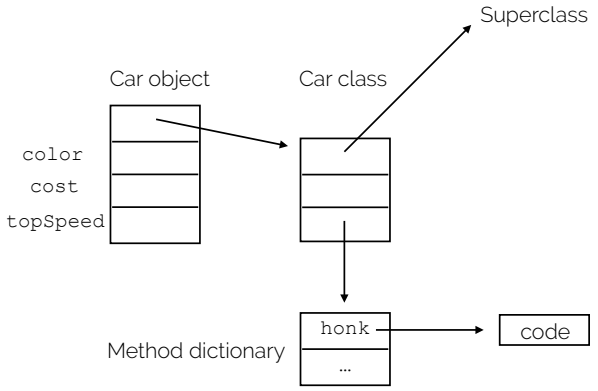
C++ eliminates lookups by computing locations at compile-time.



C++ does OO efficiently

C++ static methods are just C procedures. No classes needed.

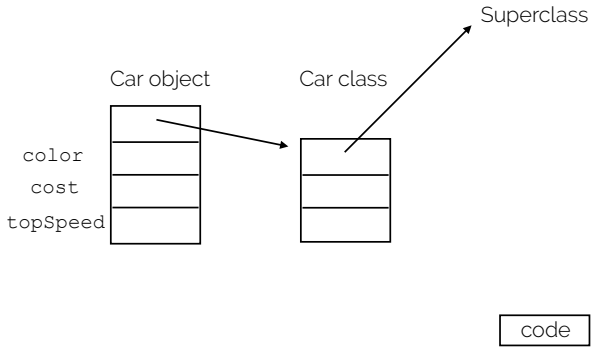
C++ eliminates lookups by computing locations at compile-time.



C++ does OO efficiently

C++ static methods are just C procedures. No classes needed.

C++ eliminates lookups by computing locations at compile-time.

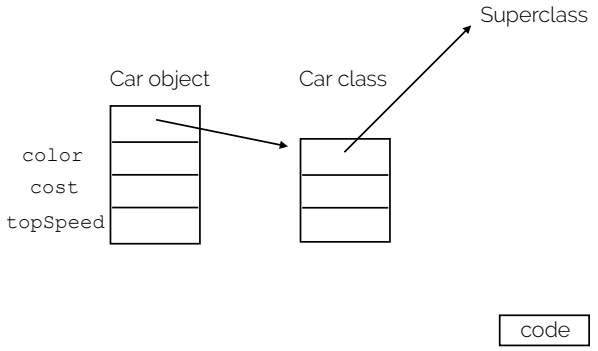


C++ does OO efficiently

C++ static methods are just C procedures. No classes needed.

C++ eliminates lookups by computing locations at compile-time.

C++ also copies any needed superclass method pointers into class

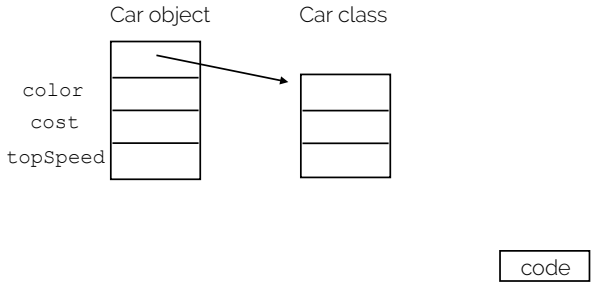


C++ does OO efficiently

C++ static methods are just C procedures. No classes needed.

C++ eliminates lookups by computing locations at compile-time.

C++ also copies any needed superclass method pointers into class

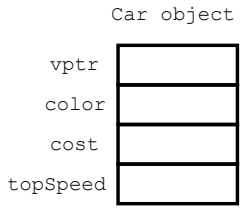


Virtual Dispatch

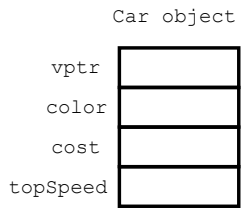
Virtual Dispatch

Car object

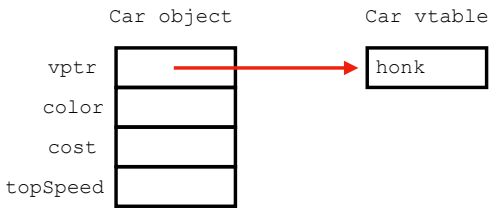
Virtual Dispatch



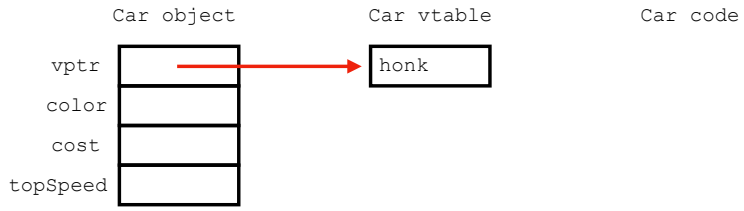
Virtual Dispatch



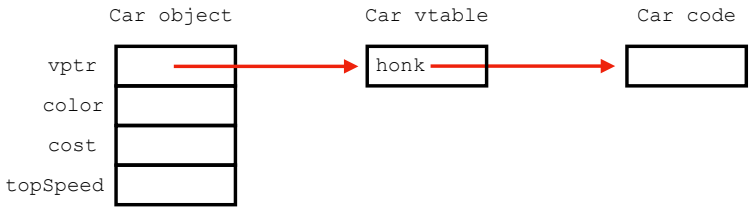
Virtual Dispatch



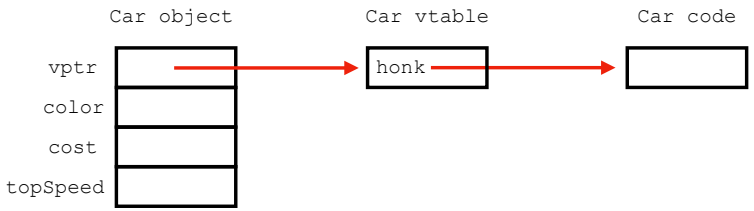
Virtual Dispatch



Virtual Dispatch

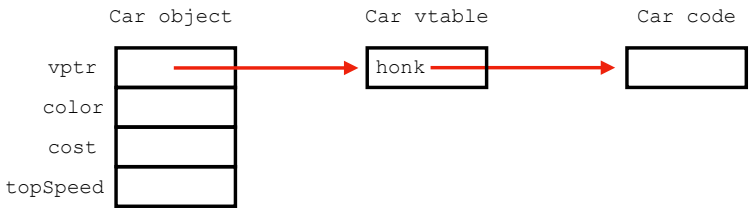


Virtual Dispatch



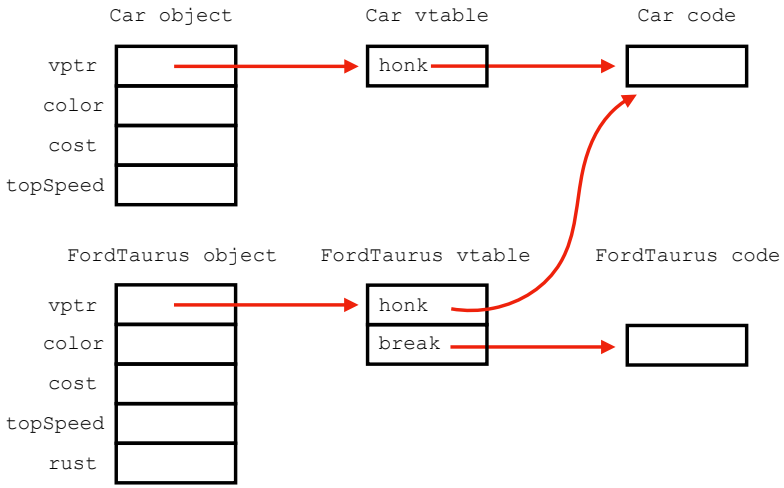
- Functions without the `virtual` keyword are just regular C functions (that also have access to class instance data).

Virtual Dispatch



- Functions without the `virtual` keyword are just regular C functions (that also have access to class instance data).
- C++ virtual dispatch does *never searches* as in SmallTalk; vtable/instance variable offsets known at compile-time.

Virtual Dispatch



Cost

Cost

1.dereference object

Cost

1.dereference object
2.defererence class

Cost

1.dereference object
2.defererence class
3.dereference method dictionary

Cost

1.dereference object
2.defererence class
~~3.dereference method dictionary~~

Cost

1.dereference object
2.defererence class
3.dereference method dictionary

for each class or superclass

Cost

1.dereference object
2.defererence class
3.dereference method dictionary

for each class or superclass

Cost

1.dereference object
2.defererence class
3.dereference method dictionary
4.dereference method

for each class or superclass

Cost

1.dereference object
2.defererence class
3.dereference method dictionary
4.dereference method

for each class or superclass

O(1) method lookup

How to give a good talk

Five tips

1. Have a story

2. Don't "bury the lede"

3. Don't make your audience read

4. Show by example

5. Stay on script

5.1. Finish on time

Project Q&A