Ca' Foscari
University
of Venice

Dispatching method calls

Object oriented programming, module 1

Pietro Ferrara

pietro.ferrara@unive.it

Ca' Foscari University of Venice

Method invocations and field accesses

- Pattern to access object members
  - <receiver>.<member signature>
- Method signature is:
  - Fields: field name
  - Methods: method name, list of parameters with their declared type
- Look up receiver's dynamic type
- Receiver is sometimes implicit
  - this (in non-static methods)
  - The class where the statement is

```
public class Vehicle {
  private double speed;
  public void accelerate(double a) {
    speed += a;
  }
}
public class Car extends Vehicle {
  private FuelType fuelType;
  private double fuel;
  public void accelerate(double a) {
    super.accelerate(a);
    this.fuel -= a * fuelType.fuelConsumption;
  }
}
public class Bicycle extends Vehicle
```

this.speed

Field speed in Vehicle class

Vehicle.accelerate(double)

this.fuelType

FuelType.fuelConsumption

Ca' Foscari University of Venice

Methods overriding

- Subclasses can override methods
  - Hide the behavior of the superclass
- super.<component> gives access to the implementation in the superclass
- Avoid to duplicate code in subclasses
  - speed+=a in accelerate
- Avoid to expose implementation details to subclasses
  - speed can be private

```
public class Vehicle {
  private double speed;
  public void accelerate(double a) {
    this.speed += a;
  }
}
public class Car extends Vehicle {
  private FuelType fuelType;
  private double fuel;
  public void accelerate(double a) {
    super.accelerate(a);
    this.fuel -= a * fuelType.fuelConsumption;
  }
}
public class Bicycle extends Vehicle {}
```

Ca' Foscari University of Venice

Simple method resolution

1. Extract the dynamic type of the receiver of the method call
   - There is always one! (this is implicit)
2. Look for a method in this class whose signature matches the given one
3. If found return it;
4. If not found, repeat 2 with the superclass of the current class

```
int race(Vehicle v1, Vehicle v2, double length) { …
 while(true) { …
  v1.accelerate(random());
  v2.accelerate(random());
 }
}
```

Static: Vehicle
Dynamic: ? Depend on the execution!

```
race(new Car(), new Truck(), 100);
```

```
public class Vehicle {
public void accelerate(double a) {…}
}
public class Car extends Vehicle {
 public void accelerate(double a) {…}
}
public class Truck extends Car {…}
```

Supporting method overriding

- A method overrides a superclass method if it hides its signature
- Dynamic type of the receiver
  - Select the "most specific" implementation
  - Aka, the lowest one in the type hierarchy
- Choosing the static type would not work!
  - Need to statically know exactly what is called
- race does not know about Car, Truck, …
  - Code specialization!

```
race(new Car(), new Bicycle());
race(new Car(), new Truck());
```

```
int race(Vehicle v1, Vehicle v2, double length) {
 v1.fullStop();
 v2.fullStop();
 double distanceV1 = 0, distanceV2=0;
 while(true) {
  distanceV1 += v1.getSpeed();
  distanceV2 += v2.getSpeed();
  if(distanceV1 >= length || distanceV2 >= length) {
   if(distanceV1 > distanceV2) return 1;
   else return 2;
  }
  v1.accelerate(Math.random()*10.0);
  v2.accelerate(Math.random()*10.0);
 }
}
```

Ca' Foscari University of Venice

Problems with method overloading

- Method signatures can "overlap"
  - Same name
  - Different arguments
    - One subtype of the other
- How the JRE chooses what to execute?
  - Static dispatching of method arguments
  - "Nearest" static type of arguments
  - Implementation choice

```
Vehicle v1 = new Car();
Vehicle v2 = new Car();
common_race.race(v1, v2);
```

Static type!!!

```
public class Racing {
 private double length;
 public Racing(double length) {
  this.length = length;
 }
 public int race(Car v1, Car v2) {…}
 public int race(Vehicle v1, Vehicle v2) {…}
 public int race(Car v1, Vehicle v2) {…}
}

Racing common_race = new Racing(100);
common_race.race(new Car(), new Car());
common_race.race(new Bicycle(), new Bicycle());
common_race.race(new Car(), new Bicycle());
common_race.race(new Bicycle(), new Car());
```

Ca' Foscari University of Venice

Static dispatch of method calls

- The call is resolved at compile time
  - Only static types are known
- This does not allow polymorphism

*polymorphism is the provision of **a single interface** to entities of different types or the use of a single symbol to represent multiple different types (Wikipedia)*

Need to explicitly refer to Car
We need to use two specific interfaces!

- Object oriented programming languages rely on dynamic dispatching

```
class Vehicle {
  public void accelerate(double a) {…}
}
class Car extends Vehicle {
  public void accelerate(double a) {…}
}

Vehicle v1 = new Vehicle();
Vehicle v2 = new Car();

v1.accelerate(100);
v2.accelerate(100);
((Car) v2). accelerate(100);
```

Ca' Foscari University of Venice

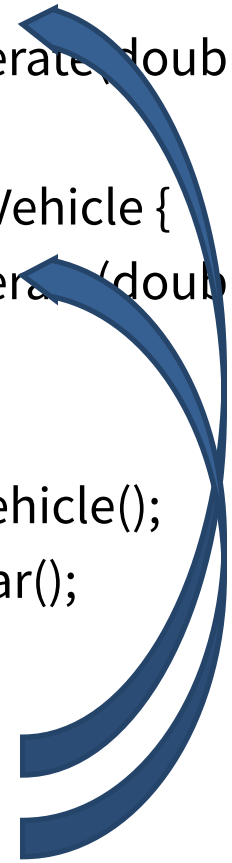
Dynamic dispatch of method calls

- The call is resolved at runtime
  - Dynamic types are known as well
- Automatically dispatch the method call to the most "specific" implementation
- This enables polymorphism
  - No need to know the subclass
    - Even its existence!
- Identify the target of the call at runtime
  - Overhead during the execution

```
class Vehicle {
  public void accelerate(double a) {…}
}
class Car extends Vehicle {
  public void accelerate(double a) {…}
}

Vehicle v1 = new Vehicle();
Vehicle v2 = new Car();

v1.accelerate(100);
v2.accelerate(100);
```



Ca' Foscari University of Venice

Overloading

- Overriding:
  - A method with exactly the same signature
  - A method hides another method in the superclass
- Overloading:
  - Several methods with the same name, different signatures, different implementations
- Do not mix up the two concepts!
  - Things will become more complex soon…

Override

Overload

```
public class Vehicle {
  private double speed;
  public void accelerate(double a) {…}
}
public class Car extends Vehicle {
  private FuelType fuelType;
  private double fuel;
  public void accelerate(double a) {…}
  public void refuel(double amount) {
    fuel += amount;
  }
  public void refuel(FuelTank tank) {
    fuel += tank.getAmount();
  }
}
```

Ca' Foscari University of Venice

Matching method arguments

- In the same class, we might have two methods
  - With the same name
  - With different arguments
- How to choose the invoked one?
  - Number of arguments
  - Matching the types of arguments
- Method arguments are **statically** matched in Java
  - Use static type of the arguments, not the dynamic one!
  - So-called single dispatch
  - C# applies multiple dispatch, aka relies on dynamic types to match method arguments

```
public class Racing {
  int race(Vehicle v1, Vehicle v2) {…}
  int race(Car v1, Car v2) {…}
  int race(Car v1, Vehicle v2) {…}
  int race(Vehicle v1, Truck v2) {…}
}
Racing racing = …;
Car v1 = new Car();
Car v2 = new Car();
Truck v3 = new Truck();
Vehicle v4 = v1;
Vehicle v5 = v3;
racing.race(v1, v2);//(Car, Car)
racing.race(v1, v5);//(Car, Vehicle)
racing.race(v4, v5);//(Vehicle, Vehicle)
racing.race(v1, v3);//(Car, Truck) ???
```

Java algorithm simplified *

1. Extract the dynamic type of the receiver of the method call
2. Collect all the methods in the current class and in the superclasses
3. Return the method in this class whose signature matches the given one, where the matching is performed by
   - Same name
   - "Nearest" arguments looking to the static type
     - No ambiguity is possible: if two method signatures match the static arguments with the same "distance", the code does not compile

* That is, ignoring quite a lot of things, like: parameter types, autoboxing, variable number of arguments, etc..

Ca' Foscari University of Venice

Invocation of static methods

- If static method are invoked through class names, no ambiguity!
- Otherwise, static methods are resolved…
  - … with static dispatching!
- General rule: do not access static members via instances
  - But (unfortunately) this is possible!
- The runtime environment executes the static method in the static type
  - Different from invoking instance methods

```
public class Racing {
   public static void foo() {
     System.out.println("Racing 1");
  }
}
public class Racing2 extends Racing {
   public static void foo() {
     System.out.println("Racing 2");
  }
}
Racing2.foo(); //Racing 2
Racing.foo(); //Racing 1
Racing racing = new Racing();
racing.foo(); //Racing 1
Racing2 racing2 = new Racing2();
racing2.foo(); //Racing 2
Racing racing3 = racing2;
racing3.foo(); //Racing 1
```

Materials

- Lecture notes: Chapter 10
- Arnold & others: 9.6
- https://docs.oracle.com/javase/specs/jls/se11/html/jls-15.html#jls-15.12 (very deep -> complex!)
  - Ignoring variable number of arguments and generics
- Dynamic dispatch: https://medium.com/ingeniouslysimple/static-and-dynamic-dispatch-324d3dc890a3