Ca' Foscari
University
of Venice

Classes, fields and methods

Object oriented programming, module 1

Pietro Ferrara

pietro.ferrara@unive.it

Classes

- This is why object-oriented programming languages
  - Were introduced
  - Became quickly highly popular
  - After 3 decades they are still the most popular programming languages
    - Maybe… ☺️
- As the name suggested, analogy with real world object
  - Objects have a state
  - Objects provide some functionalities

Real world objects

- Consider for instance a car
- A car has a state representing
  - How much fuel is stored in it
  - Its speed
  - Its direction
- Different actions
  - Brake
  - Steer
  - Refuel
- Actions change the state of the car

Wooclap

https://app.wooclap.com/PO12324

Please help me! (question n. 3)

Modelling real world objects

- A class models real world objects
  - Fields capture the state
  - Methods capture the actions
  - Methods changes the state of the object
- The structure is fixed
  - Once defined, cannot be changed
  - Cannot create/remove fields on-the-fly
- IMPORTANT: other programming languages (not strictly OO) are more flexible
  - JavaScript, Python: out of scope here

Fields

Methods

```
class Car {
  double speed;
  double fuel;
  void refuel(double amount) {
    fuel += amount;
  }
  void accelerate(double a) {
    speed += a;
    fuel -= a*FUEL_CONS;
  }
  void fullBreak() {
    speed = 0.0;
  }
}
```

Instances and method invocation

- Classes can be instantiated into objects
  - An object has actual values for fields
  - Method invocation changes these values
- A class can be instantiated many times
- Each instance has its own state
- Local variables (myCar, yourCar) contains a reference to an object
  - Aka, an instance of class Car

```
Car myCar = new Car();
//myCar: fuel = 0.0, speed = 0.0

myCar.refuel(34.5);
//myCar: fuel = 34.5, speed = 0.0

myCar.accelerate(90.3);
//myCar: fuel = 33.9, speed = 90.3

myCar.fullBreak();
//myCar: fuel = 33.9, speed = 0.0

Car yourCar = new Car();
//yourCar: fuel = 0.0, speed = 0.0
//myCar: fuel = 33.9, speed = 0.0
```

Abstraction and interfaces

- A class defines a contract specifying the interface of the objects
  - Method signature represents the structure
  - Method semantics (meaning) needs to be documented externally
- This allows to abstract away the internal implementation

```
class Car {
  //Add the given amount to the fuel tank
  void refuel(double amount) {…}
  //Increment the speed
  void accelerate(double a) {…}
  //Stop the car
  void fullBreak() {…}
}
```

Interactions between objects

- We add a class representing the fuel type
- And then another class representing a tank
- And then we can modify the Car class!

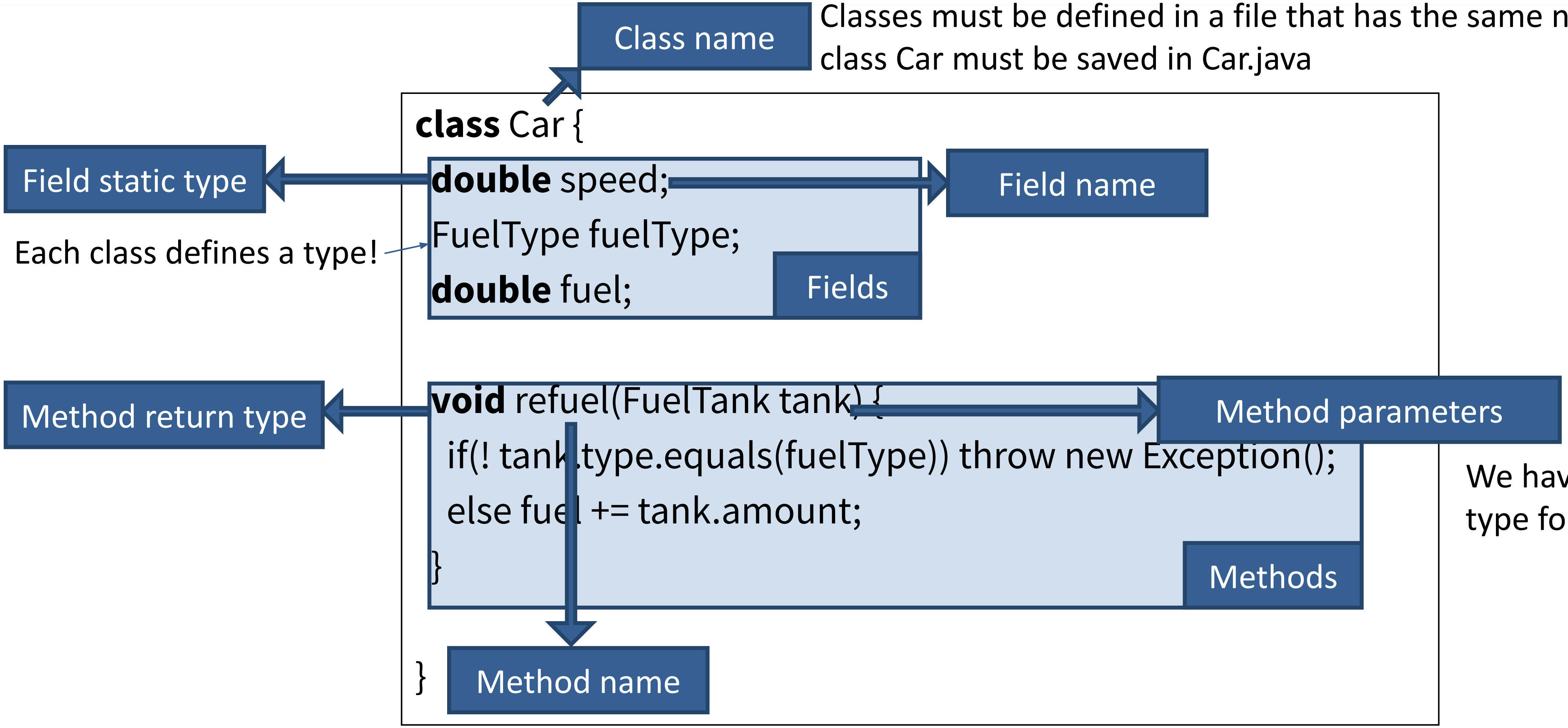
```
class Car {
  double speed;
  FuelType fuelType;
  double fuel;
  void refuel(FuelTank tank) {
    if(! tank.type.equals(fuelType)) throw new Exception();
    else fuel += tank.amount;
  }
}
```

```
class FuelType {
  String name;
  double costPerLiter;
  double FUEL_CONS;
}


FuelType diesel = new
      FuelType(“diesel“, 1.3, 0.3);


FuelType petrol = new
      FuelType(“petrol“, 1.5, 0.5);
class FuelTank {
  FuelType type;
  double amount;
}
```

Defining a class - Example



Classes must be defined in a file that has the same name class Car must be saved in Car.java

```
class Car {
 double speed;
 FuelType fuelType;
 double fuel;

 void refuel(FuelTank tank) {
  if(! tank.type.equals(fuelType)) throw new Exception();
  else fuel += tank.amount;
 }
}
```

Each class defines a type!

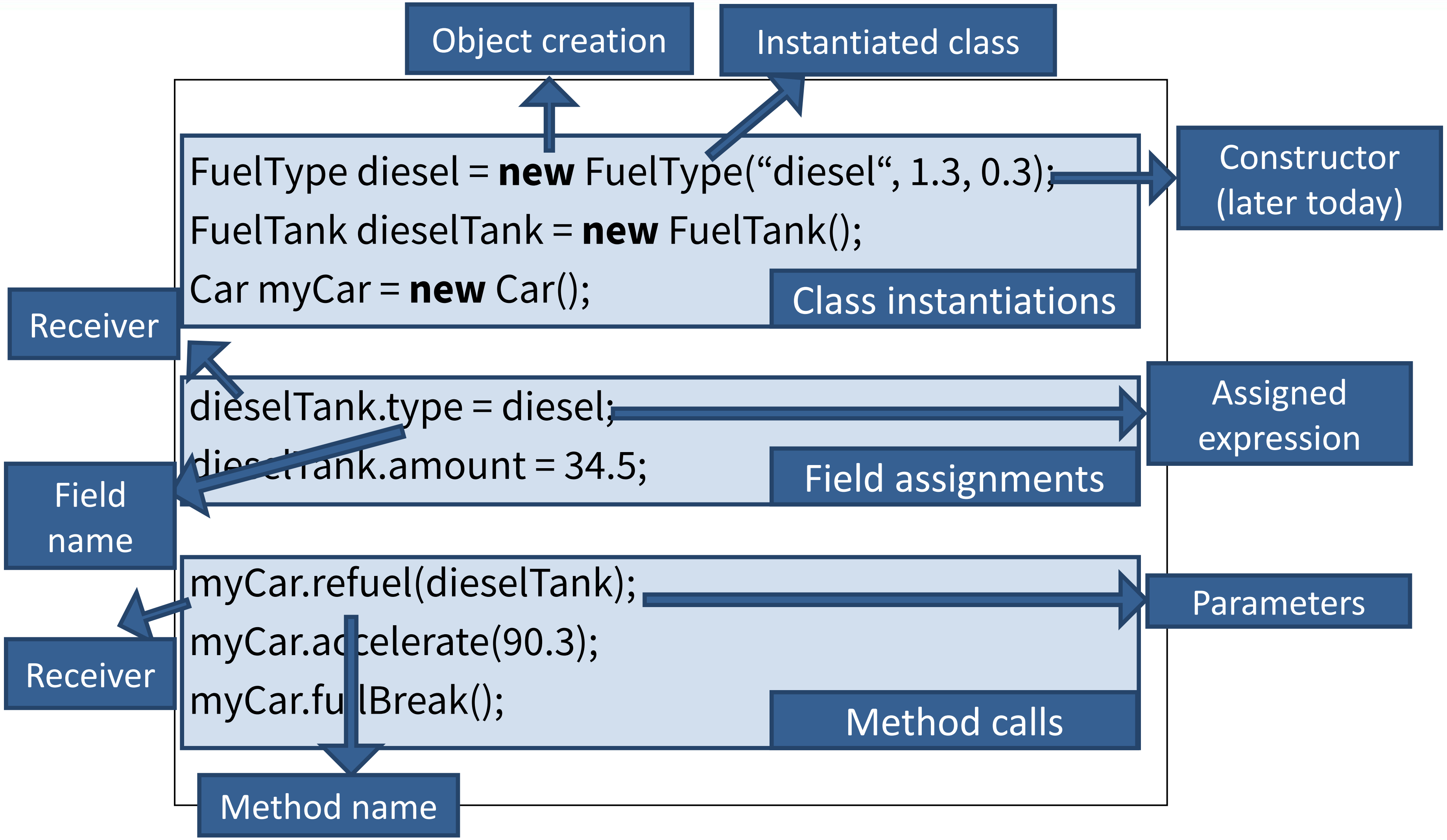
We have name and static type for each parameter

Defining a class - Structure

```
class <class_name> {
 <field1_type> field1_name [ = <initial_value>];
 <field2_type> field2_name [ = <initial_value>];
  …
 <method1_returntype> <method1_name>(<method1_pars>) {
   <method1_body>
 }
 <method2_returntype> <method2_name>(<method2_pars>)
    {
   <method2_body>
 }
 …
}
```

- Definitions of methods and fields can be mixed, but it’s a good practice to have first field and then method definitions
- Naming conventions
- Fields can define an initial value
  - 0, false or null if not defined
- Different modifiers for different components
  - Most of them will be introduced and explained later during the course

Instantiating and using classes Example



Object creation

Instantiated class

```
FuelType diesel = new FuelType(“diesel“, 1.3, 0.3);
FuelTank dieselTank = new FuelTank();
Car myCar = new Car();
```

Constructor (later today)

Class instantiations

Receiver

```
dieselTank.type = diesel;
dieselTank.amount = 34.5;
```

Assigned expression

Field name

Field assignments

```
myCar.refuel(dieselTank);
myCar.accelerate(90.3);
myCar.fullBreak();
```

Parameters

Receiver

Method calls

Method name

```
class Car {
  double speed;
  FuelType fuelType;
  double fuel;
  void refuel(FuelTank tank) {…}
  void accelerate(double a) {…}
  void fullBreak() {…}
}
class FuelType {
  String name;
  double costPerLiter;
  double fuelConsumption;
}
class FuelTank {
  FuelType type;
  double amount;
}
```

Initialization and constructors Example

```
FuelType diesel = new FuelType(“diesel“, 1.3, 0.3);
//This does not compile!!!

FuelType diesel = new FuelType();
diesel.name=“diesel“;
diesel.costPerLiter=1.3;
diesel. fuelConsumption = 0.3;
//This does not compile!!!
```

```
class FuelType {
 String name;
 double costPerLiter;
 double fuelConsumption;
}

class FuelType {
 String name;
 double costPerLiter;
 double fuelConsumption;
 FuelType(String n,
        double cpl, double fc) {
  name = n;
  costPerLiter = cpl;
  fuelConsumption = fc;
 }
}
```

Constructor: “special” method invoked when a class is instantiated.
If a class does not define a constructor, a default one (with no parameters) is added by the compiler.
If a class defines a constructor, the default one is not added.

Initialization and constructors
Structure

```
class <class_name> {
 …
 <class_name>(<constructor1_pars>) {
    < constructor1_body>
 }
 <class_name>(<constructor2_pars>) {
    < constructor2_body>
 }
 …
}
```

```
class FuelType {
  String name;
  double costPerLiter;
  double fuelConsumption;
  FuelType(String n, double cpl, double fc) {
    name = n;
    costPerLiter = cpl;
    fuelConsumption = fc;
  }
}
```

this keyword

```
class Car {
  double speed;
  FuelType fuelType;
  double fuel;
  void refuel(FuelTank tank) {…}
  void accelerate(double a) {…}
  void fullBreak() {…}
}
class FuelTank {
  FuelType type;
  double amount;
  void refuelCar(Car c) {
    c.refuel(this);
  }
}
```

Reference to the current object

- this is a Java keyword
- Pointer to the current object
- Pass a reference to the current object to other methods
- Might be used to access fields and methods of the current object
  - Distinguish fields from local vars
- Invoke a constructor from another constructor
  - Only 1st statement of a constructor

```
class FuelTank {
  FuelType type;
  double amount;
  FuelTank(FuelType type,
       double amount) {
    this.type = type;
    this.amount = amount;
  }
  FuelTank(FuelType type) {
   this(type, 0);
  }
}
```

Types of modifiers

- Access modifiers: fields and methods
  - Only public for classes
  - Will be covered during the course
- Concurrency modifiers: fields and methods
  - Not covered
- Static: fields, methods
  - Will be covered today
- Final: fields, methods, classes
  - Will be covered during the course (fields today)
- Abstract: methods, classes
  - Will be covered during the course

public
no modifier (default)
protected
private

Access

synchronized
volatile

Concurrency

static
final
abstract

Others

Static fields

- Fields of different objects contain different values
  - Fields are “object specific”
- Fields shared among all the instances of a class?
  - static fields
- Accessed using the class name
- Can be accessed also through this or an object reference
  - But this is a bad practice!
    - Guess why!

```
class FuelTank {
 FuelType type;
 double amount;
 static int numberOfTanks = 0;
 FuelTank(FuelType type, double amount) {
   numberOfTanks++;
   this.type = type;
   this.amount = amount;
 }
 FuelTank(FuelType type) {
  this(type, 0);
 }
}

System.out.println(“Created “+
     FuelTank.numberOfTanks+ “tanks“);
```

Incremented each time the class is instantiated
Count the number of tanks already created

Accessed using the class name

Static methods and initialization

- Methods invoked on the class
  - Not on a specific instance!
- They can access only static fields
- And invoke static methods
- Invoked using the class name
  - Also through instances, but please avoid it!
- Static constructor
  - Through a static block that must initialize the static variables

```
class FuelTank {
 FuelType type;
 double amount;
 static int numberOfTanks;
 …
 static void resetTanksCount() {
  numberOfTanks = 0;
 }
 static {
  FuelTank.resetTanksCount();
 }
}
```

Static method

Static constructor

final fields

- A final field is a field that cannot be changed after being initialized
- Constructors must initialize all final fields
- Represent an immutable property
  - Computable when the object is created
  - Like an identifier!

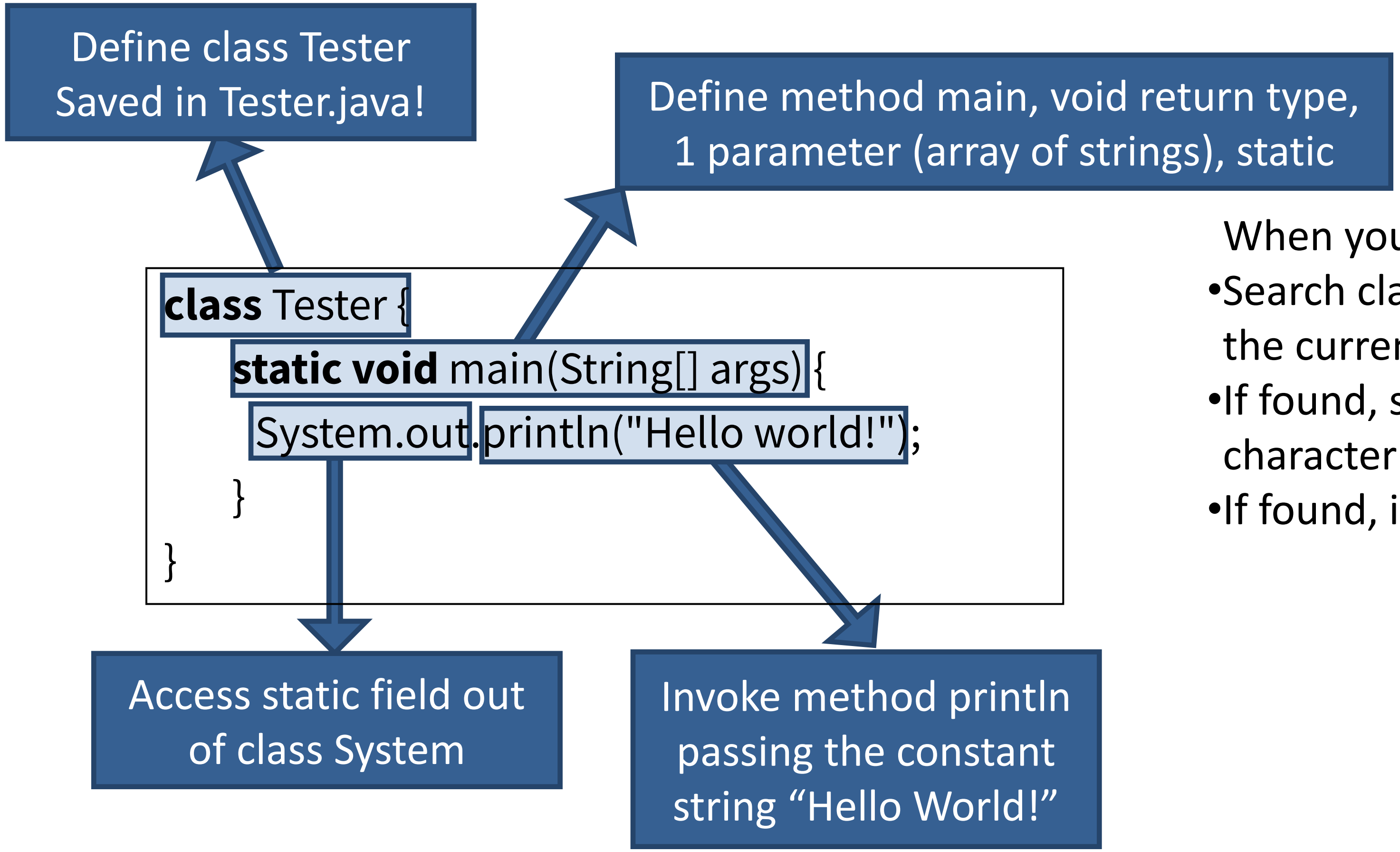
```
class FuelTank {
  FuelType type;
  double amount;
  static int numberOfTanks = 0;
  final int tankId;
  FuelTank(FuelType type, double amount) {
     tankId = numberOfTanks;
     numberOfTanks++;
     this.type = type;
     this.amount = amount;
  }
  FuelTank(FuelType type) {
    this(type, 0);
  }
}
```

Modifiers

- Attachable to class, fields, methods
- Specify some additional behaviors
  - A field that cannot be modified
  - Synchronize when invoking a method
  - A class that is only partially implemented
- Each component can have many modifiers
  - But they should be compatible
- Several different types
  - Some providing essential OO features

```
<class_modifier> class <class_name> {
 <field1_modifier> <field1_type> field1_name;
 <field2_modifier> <field2_type> field2_name;
  …
 <method1_modifier> <method1_returntype>
     <method1_name>(<method1_pars>) {
    <method1_body>
 }
 <method2_modifier> <method2_returntype>
     <method2_name>(<method2_pars>) {
    <method2_body>
 }
 …
}
```

Hello World… again!



Define class Tester Saved in Tester.java!

Define method main, void return type, 1 parameter (array of strings), static

```
class Tester {
    static void main(String[] args) {
      System.out.println("Hello world!");
    }
}
```

Access static field out of class System

Invoke method println passing the constant string “Hello World!”

When you run “java Tester”, the JRE

- Search class Tester from the classes in the current classpath (directory)
- If found, search a method with these characteristics
- If found, it executes it

Java vs. C

C

```
Vector * v = malloc(sizeof(Vector))

v -> x = 5;

*(v+1) = 10;

int* i = malloc(4);

free(v)
```

Java

```
Vector v = new Vector();

v.x = 5;

//NO “FREE“ POINTERS TO THE HEAP

//NO “FREE“ ALLOCATION

//AUTOMATIC GARBAGE COLLECTION
```

Textbook

- Lecture notes: Chapter 2, 3.1-3.3
- Arnold&others:
  - Classes, fields, methods, constructors, modifiers:
    - Sections 2.1, 2.2, 2.4, 2.5, 2.6 (2.6.3, 2.6.5, 2.6.6), 2.7
  - Exercises: 2.2, 2.6 (using Car instead of Vehicle), 2.15, 2.16
- Budd: 4.1, 4.2, 4.3