## Java class

```
public class Dog {
```

Fields (attributes)

Constructor

Methods (operations or behaviors)

```
}
```



## Accessibility

Classes, fields and methods must be declared with *access modifiers*.

- Classes and methods are usually declared as `public`—accessible by anyone from anywhere.
- Fields are usually declared as `private`—accessible only within the object.



## Primitive data types

Fields are often defined using variables. In Java, variables must be declared with types. There are four primitive data types:

- `int`—integers: `..., -1, 0, 1, 2, ...`
- `float`, `double`—reals: `..., 0.0, ...`
- `char`—letters: `'a', 'b', 'c', ...`
- `boolean`—boolean values: `true, false`

## Defining fields

```
public class Dog {

  private String breed;
  private char gender;
  private float height;
  private float weight;
  private int age;

  Constructor

  Methods (operations or behaviors)

}
```



## Constructor

- A *constructor* is a special method that instantiates an object.
- Objects are created from a class by invoking a constructor.
- Constructors usually have parameters, and constructors create objects according to the values of parameters.



## Parameters

- Many (but not all) constructors and methods have parameters, just like many functions have parameters.
- In Java, parameters must be declared with types, just like variables must be declared with types.

## Defining a constructor

```
public class Dog {

  private String breed;
  private char gender;
  private float height;
  private float weight;
  private int age;

  public Dog(String b, char g, float h, float w, int a) {
    breed = b;
    gender = g;
    height = h;
    weight = w;
    age = a;
  }

  Methods (operations or behaviors)

}
```



## Methods

- *Methods* define objects' operations, similar to the way functions define basic computational tasks.
- Many (but not all) methods return values, just like many functions return values.
- All methods be declared with return types (which specify the types of values being returned).



## Defining methods

```
public class Dog {

  private String breed;
  private char gender;
  private float height;
  private float weight;
  private int age;

  Constructor

  public void getOlder() {
    age = age + 1;
  }

}
```

## Defining methods

```
public class Dog {

  private String breed;
  private char gender;
  private float height;
  private float weight;
  private int age;

  Constructor

  public int getAge() {
    return age;
  }

}
```



## Accessors

- A method that simply returns the value of a field is called an *accessor* (or *"get" method*).
- An accessor for a field `f` is usually named `getF`.
- An accessor has no parameter.
- The return type of an accessor for a field `f` is the type of `f`.



## Accessors

```
public class Coo {

  private type f;

  ...

  public type getF() {
    return f;
  }

}
```

## Mutators

- A method that simply changes the value of a field by a parameter value is called a *mutator* (or *modifier* or *"set" method*).
- A mutator for a field `f` is usually named `setF`.
- A mutator returns nothing.
- A mutator for a field `f` has a parameter whose type is the same as that of `f`.



## Mutators

```
public class Coo {

  private type f;

  ...

  public type getF() {
    return f;
  }

  public void setF(type f0) {
    f = f0;
  }

}
```



## Method calls

Every method belongs to some object. In general, when we call a method, we must specify both the object and method:

```
object.method(<parameters>)
```

## Internal method calls

To invoke a method within the same object, we can just call by the method name.

```
public class Coo {

  public void moo() {
    noo(5);
  }

  public void noo(int x) {
    ...
  }

}
```



## Internal method calls

Or, to be more explicit, you may also put "`this`" with the method name.

```
public class Coo {

  public void moo() {
    this.noo(5);
  }

  public void noo(int x) {
    ...
  }

}
```



## External method calls

To invoke a method of another object, you must always explicitly specify the object.

In general, from a class `Doo`, to call a `Coo` object `c`'s method `noo()`, put:

```
public class Doo {

  public void poo() {
    Coo c = new Coo();
    c.noo(7);
  }

}
```

## Inheritance

## Inheritance

*Inheritance* is the notion of one class inheriting the fields and methods from another class.

- In Java, inheritance happens when one class is defined as a *subclass* of another.
- Inheritance allows us to organize classes in hierarchical structures.
- Inheritance prevents redundancy.



## Inheritance

**Example.** Consider creating classes for vehicles, cars, trucks, sedans, wagons, etc.

We could create all these classes independently, but they are closely related:

- A car is a special kind of a vehicle, so is a truck.
- A sedan is a special kind of a car, so is a wagon.

## Inheritance

**Example.** A car is a special kind of a vehicle.

- A car should have all the attributes and behaviors of a vehicle.
- Can we define a car as some kind of a vehicle's child?

Yes, we can create cars as a *subclass* of vehicles.



## Inheritance

**Example.** `Car` as a subclass of `Vehicle`.

```
public class Vehicle {

  protected int numOfWheels;
  protected int weight;

  ...

  public void drive() {
    ...
  }

}
```



## Inheritance

```
public class Car extends Vehicle {

  // Everything from Vehicle plus the following:

  protected int numOfDoors;
  protected String bodyColor;

  ...

  public void headHightOn() {
    ...
  }

}
```

## Inheritance

How about constructors? A subclass's constructor must always first call its superclass's constructor.

```
public Car() {
  super();
  numOfDoors = 4;
  bodyColor = "Silver";
}
```