IMPLEMENTING CLASSES

Benedict R. Gaster / @cuberoo_

UWE BRISTOL
University of the West of England

bettertogether

CLASSES

- A description of a common properties (attributes and methods) of a set of objects
- A concept
- Class is defined as part of a Scala program
- Examples include: Person, Car, Planet

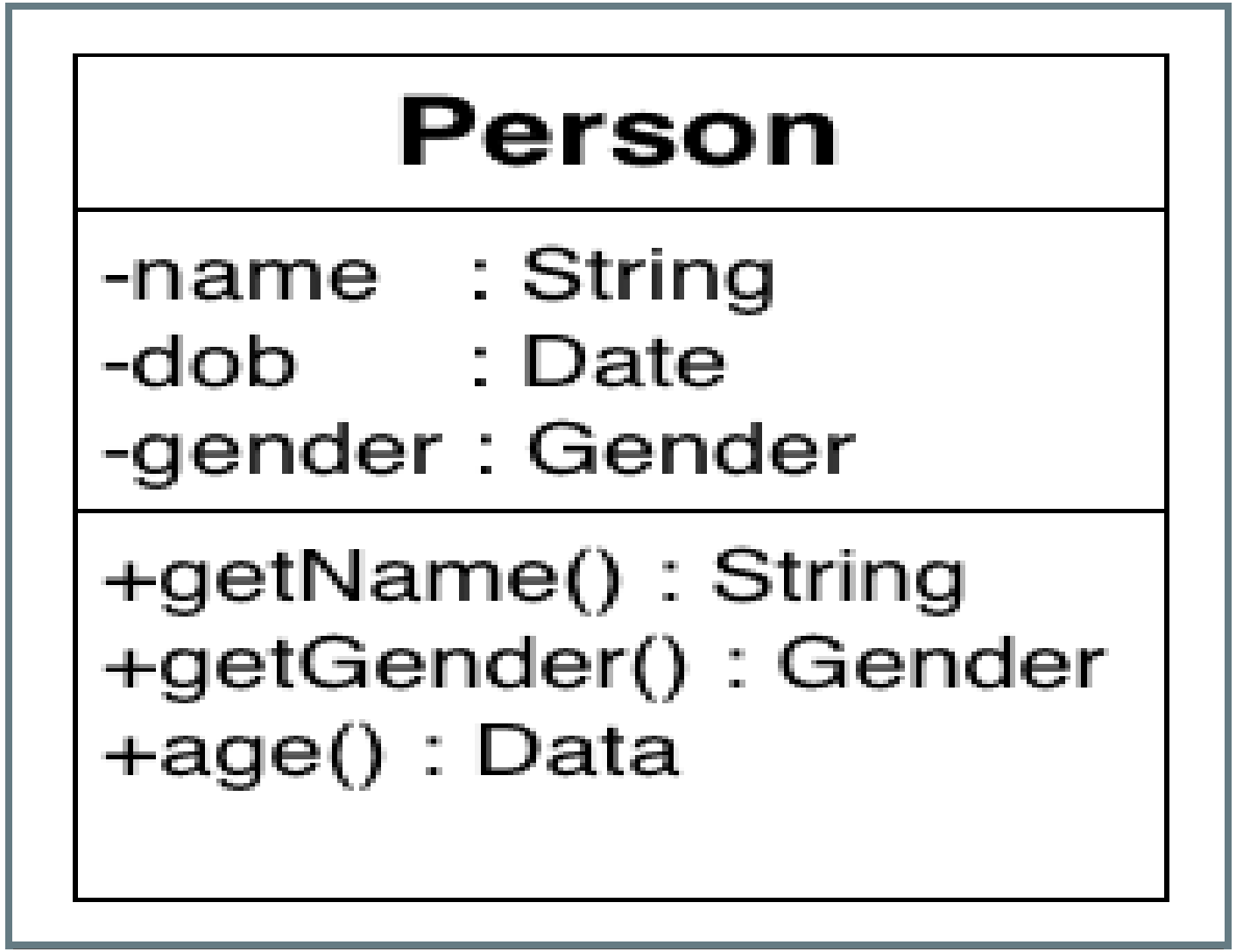
OBJECTS

- A representation of the properties of a single instance of a class
- An object is part of a programs data and its execution
- Examples include: Earth, Mars, and Barack obama

WHAT'S THE CONNECTION BETWEEN OBJECTS AND CLASSES

- In object-oriented programming we write classes
  - They are static, defined in the source code
  - There is just one defintion for each class
- Objects are create from classes
  - They are created dynaically
  - We say a particular object is an instance of a given class
- A classs contains the "instructions" for how an object will behave

PERSON CLASS DIAGRAM



PERSON SCALA

```
class Person(name : String, dob : Date, gender : Gender) {
   def getName() : String = name
   def getGender() : Gender = gender
}
```

FOR A PERSON CLASS WE HAVE

- An implict default constructor
- Data local to any instance of Person
- Implict instance data declarations

CONSTRUCTORS

- Constructors are used to create objects from classes
  - If we want to provide data at object creation, then we pass these values to a constructor
  - In Scala a **primary** constructor is a classes body and argument list

```
class A(/* classes argument list */) {
    // body of class primary constructor
}
```

- Every class has at least one constructor, its primary constructor

CREATING INSTANCES OF CLASSES, I.E. OBJECTS

- Use the **new** keyword to create a new instance of a class
- In general, new ClassName(argument1, ..., argumentN)

```
new Person("Ben", new Date(1,"Feburary", 1968), new Male())
```

PRIMARY CONSTRUTOR

- Arguments create attributes internal to class
- Private access, i.e. not accessable from outside of class or by subclasses

```
class Foo(s : String)

val f : Foo = new Foo("hello, you!")
res2: Foo = Foo@45fa80c
f.s
console:10: error: value s is not a member of Foo
```

PRIMARY CONSTRUCTOR

- If we want to have public access to constructor arguments, then
  - we create public attributes, initialized to constructor arguments

```
class Foo(s : String) {
   val str : String = s

val f : Foo = new Foo("hello, you!")
res2: Foo = Foo@45fa80c9
f.str
res1: String = hello
```

CONSTRUCTORS CAN BE MANY

- It is possible to have many additional constructors
  - Called auxiliary constructors
  - Each auxiliary constructor is named with the keyword **this** and has a different set of arguments
- Consider the Date type
  - By default it uses the United Kingdon's date format
  - But it is equally valid to use USA's date format

CREATING INSTANCES OF DATE

```
new Date(1,"Feburary", 1968)) // allowed

new Date("Feburary", 1, 1968)) // not (currently) allowed
```

MULTIPLE CONSTRUCTORS FOR DATE

```
class Dates(d : Int, m : String, y: Int) {
  def this(m : String, d : Int, y : Int) = this(d, m, y)
  // other methods
}
```

INHERITANCE

- An object may "inherit" characteristics and behaviour from another
- Subclass **B** inherits from a superclass **A** using the keyword **extends**

```
class B extends A {
  // data and methods for B go here
}
```

INHERITANCE EXAMPLE - CHILD IS A PERSON

```
class Child(name : Name, dob : Date, gender : Gender, school : String)
      extends Person(name,dob,gender)
```

- Pass constructor arguments to superclass, e.g:

```
extends Person(name,dob,gender)
```

INHERITANCE ALLOWS OVERRIDING METHODS

- A subclass can override a superclasses' method using the keyword **override**

OVERRIDABLE DISPLAY METHOD FOR PERSON

```
class Person(name : Name, dob : Date, gender : Gender) {
  def getName() : String = name
  def getGender() : Gender = gender

  def display() : Unit = {
    println(name)
    gender.display()
  }
}
```

OVERRIDES THE DISPLAY METHOD

```
class Child(name : Name, dob : Date, gender : Gender, school : String)
      extends Person(name,dob,gender) {

    override def display() : Unit = {
      super.display()
      println(school)
    }
  }
```

- A subclass can override refer to superclasses' "overridden" method using the keyword **super**

ABSTRACT CLASSES

- A class **without** one or more method implementations, e.g.
  - An abstract Gender class might provide methods for both genders, but only specific inherited genders, e.g. women, can define the specific behaviour (for example, type of chromosomes)
- Java calls these **interfaces**
  - Sadly this is a very overloaded term!
- Scala calls these **abstract classes**
  - Lots more on this later in the course

ABSTRACT CLASS FOR REPRESENTING GENDER

```
abstract class Gender {
  def isMale() : Boolean
  def isFemale() : Boolean
}
```

GENDER FEMALE

```
class Female extends Gender {
   override def isMale() : Boolean = false
   override def isFemale() : Boolean = true
}
```

GENDER FEMALE

```
class Male extends Genders {
   override def isMale() : Boolean = true
   override def isFemale() : Boolean = false
}
```

AVOID IMPLEMENTING ISMALE WHEN FEMALE

```
abstract class Gender {
  def isMale() : Boolean = false
  def isFemale() : Boolean = false
}

class Females extends Genders {
  override def isFemale() : Boolean = true
}
```

ALTERNATIVES

But what can we do with this expressiveness?

ALTERNATIVES

- Using isMale() and isFemale() allows development of algroithms that depend on this knownlege, but work for all people

PROBLEM STATEMENT

Given a list of people, write functions to

calulate the number of men calculate the number of women

HOW MANY WOMEN

```
def calNumberOfWomen(ps : List[Person]) : Int = {
  var f : Int = 0

  for (p <- ps) {
     ... // what should we put here?
  }

  f
}
```

HOW MANY WOMEN

```
... // what should we put here?
```

GET GENDER FOR PARTICUALAR PERSON

```
val g = p.getGender()
... // what should we put here?
```

SIMPLY ASK IF FEMALE

```
val g = p.getGender()
if (g.isFemale())
   f = f + 1
```

COMPLETE IMPLEMENTATION

```
def calNumberOfWomen(ps : List[Person]) : Int = {
  var f : Int = 0

  for (p <- ps) {
    val g = p.getGender()
    if (g.isFemale())
      f = f + 1
  }

  f
}
```

SCALALIZING A LITTLE

```
def calNumberOfWomen(ps : List[Person]) : Int = {
  var f : Int = 0

  for (p <- ps if p.getGender().isFemale())
      f = f + 1

  f
}
```

AND A LITTLE MORE

```
def calNumberOfWomen(ps : List[Person]) : Int =
   ps.foldRight(0)
               ((n,f) => if (g.getGender().isFemale()) f + 1 else f)
```

We will come back to this later in the course!

CASE CLASSES OR PATTERN MATCHING

A common pattern is to perform a different task for each type of class in a hierarchy Scala provides special syntax and functionality to make this easier using the **case** keyword when defining classes enables pattern matching

ABSTRACT CLASS FOR REPRESENTING GENDER WITH PATTERN MATCHING

```
abstract class Gender
case class Female() extends Gender
case class Male()   extends Gender
```

MATCHING AGAINST A CASE CALSS

- Using the **match** construct we can pattern match against alternative subclasses

```
def isFemale( g : Gender ) : Boolean =
   g match {
      case Female() => true
      case Male()   => false
   }
```

NUMBER OF WOMEN USING PATTERN MATCHING

```
def calNumberOfMaleFemale(ps : List[Person]) : (Int,Int) = {
    var f : Int = 0
    for (p <- ps)
      p.getGender() match {
        case Male()  => ()
        case Female() => f = f + 1
      }
    f
  }
```

OR USING OUR ISFEMALE PREDICATE

```
def calNumberOfWomen(ps : List[Person]) : Int = {
  var f : Int = 0
  for (p <- ps if isFemale(p.getGender()))
      f = f + 1

  f
}
```

- Notice how we are now using the "functional" style of **isFemale(...)**
- Scala's ability to mix functional programming and object-oriented programming, is very powerful!