

CSC 347 - Concepts of Programming Languages

Scala Classes

Instructor: Stefan Mitsch



Learning Objectives

? Complex data in object-oriented programming?

- Identify classes and objects in Scala
- Use class parameters to initialize, query, and modify object values
- Use Scala objects to implement singletons
- Describe methods



Classes in C++, Java, and Python

C++

```
1 class Number {
2     private:
3         // fields
4         int value;
5
6     public:
7         // constructor
8         Number(int v) : value(v) {}
9         // destructor
10        ~Number() {}
11
12        // methods
13        int getValue() {
14            return value;
15        }
16
17        // operator overloading
18        Number operator+(const Number& n) {
19            return Number(value + n.value);
20        }
21 }
```

Java

```
1 class Number {
2     // fields
3     private int value;
4
5     // constructor
6     public Number(int v) {
7         value = v;
8     }
9     // optional finalizer (unreliable!)
10    public void finalize() {}
11
12    // methods
13    int getValue() {
14        return value;
15    }
16
17    // no operator overloading
18    Number add(Number n) {
19        return new Number(value + n.value);
20    }
21 }
```

Python

```
1 class Number:
2     # fields, no visibility modifiers
3     int value
4
5     # constructor
6     def __init__(self, v):
7         self.value = v
8
9     # optional finalizer (unreliable!)
10    def __del__(self):
11        pass
12
13    # methods
14    def getValue(self):
15        return self.value
16
17    # operator overloading
18    def __add__(self, n):
19        return Number(self.value + n.value)
```



Classes

- Define a class `C`

```
1 class C (f1:Int, val f2:Int, var f3:Int):  
2    //...
```

- Has one constructor with parameters `f1`, `f2`, `f3`
- Instantiate the class `C`

```
1 val c = new C (2, 3, 5)
```

- Instance of `C` is heap allocated
- `c` is a reference to instance



Class Parameter Details

- Class parameters simultaneously declare constructors and accessor methods
- Components of the class body are `public` by default

Scala

```
1 class C1 (x:Int):  
2   def double() = x+x  
3 end C1
```

- `x` private immutable
- public constructor
- public `double`

Expressed in Java

```
1 public class C1 {  
2   private final int x;  
3  
4   public C1(int x) { this.x = x; }  
5  
6   public int double() { return x+x; }  
7 }
```



Class Parameter Details

- Immutable class parameters can be initialized but not modified

Scala

```
1 class C2 (val x:Int):  
2   def double() = x+x  
3 end C2
```

- `x` public immutable

Expressed in Java

```
1 public class C2 {  
2   public final int x;  
3  
4   public C2(int x) { this.x = x; }  
5  
6   public int double() { return x+x; }  
7 }
```



Class Parameter Details

- Mutable class parameters can be changed even after initialization

Scala

```
1 class C3 (var x:Int):  
2   def double() = x+x  
3 end C3
```

- `x` public mutable

Expressed in Java

```
1 public class C3 {  
2   public int x;  
3  
4   public C3(int x) { this.x = x; }  
5  
6   public int double() { return x+x; }  
7 }
```



Class Body

- Class body contains
 - Field declarations
(`val` or `var`)
 - Constructor code
 - Method definitions

```
1 class C (f1:Int, val f2:Int, var f3:Int):  
2   val f4 = f1 * f2  
3   var f5 = f2 * f3  
4  
5   println ("Constructing instance of C")  
6  
7   def m (x:Int) : Int =  
8     // cannot reassign to f1, f2, f4  
9     f3 = f3 + 1  
10    f5 = f5 + 1  
11    f1 * f3 * x  
12  end m  
13 end C
```



Methods vs. Fields

- `val` field: `val x = 1` ; strict, initialized once
- `def` non-parameterized method: `def y = 2` ; non-strict, executed every time
- `lazy val` : memoized `def` , initialized on demand

Scala

```
1 class C:  
2   val x = 1  
3   def y = 2  
4   lazy val z = x + y  
5 end C
```

Expressed in Java

```
1 public class C {  
2   public final int x = 1;  
3  
4   public int y() { return 2; }  
5  
6   private Integer z = null;  
7   public int z() {  
8     if (z == null) z = x + y();  
9     return z;  
10  }  
11 }
```



Objects

- `object` declares a single instance, accessible through the object name
- Language support for the [singleton design pattern](#)

```
1 object C:  
2     var count = 0  
3 end C  
4  
5 C.count = C.count + 1
```



Objects

- Singleton objects `object` are instantiated on program startup
- Method `main` must be declared in an `object`

Java

```
1 public class C {  
2     public static void main (String[] args) {  
3         //...  
4     }  
5 }
```

Scala

```
1 object C:  
2     def main (args:Array[String]) : Unit =  
3         //...  
4     end main  
5 end C
```



Companion Objects

Java: static vs. non-static

- `static` components belong to a class
- All other belong to instance of class

```
1 class C {
2     private int f1 = 0;
3     private static int f2 = 0;
4
5     public int m1 () {
6         C.f2 = 3;
7         return f1;
8     }
9
10    public static int m2 (C other) {
11        other.f1 = 5;
12        return f2;
13    }
14 }
```

Scala: Companion object replaces `static`

- All data and methods belong to objects
- Companion related to other instances

```
1 class C:
2     private var f1 = 0
3     def m1 () =
4         C.f2 = 3
5         f1
6 end C
7
8 object C:
9     private var f2 = 0
10    def m2 (other: C) =
11        other.f1 = 5
12        f2
13 end C
```



Coding Exercise

- `Matrix` class



Summary

- Class parameters and factory methods replace constructors
- Builtin support for the Singleton design pattern

Classes

```
1 class C
2   (    f: Int,
3     val g: Int,
4     var h: Int)
```

- `f` private immutable
- `g` public immutable
- `h` public mutable

Objects

- Replace `static`

```
1 object 0:
2   val n = 10
3 end 0
4
5 val n = 0.n
```

Companion Objects

```
1 class X:
2   private val n = 10
3
4 object X:
5   def getN(x: X) = x.n
6
7 val n = X.getN(new X)
```

- Have access to `private` elements of companions