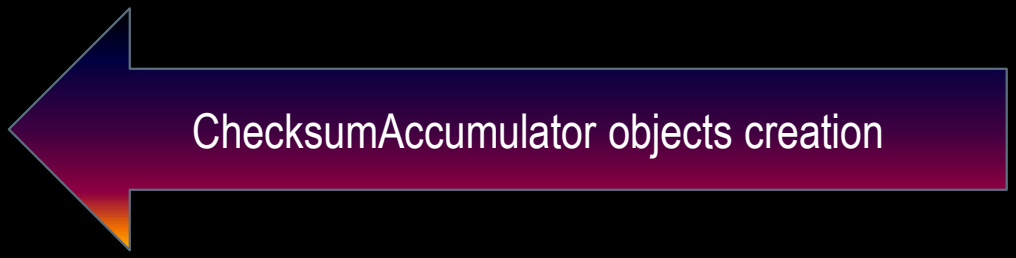
19CSE313 – PRINCIPLES OF PROGRAMMING LANGUAGES

Classes and Objects

CLASSES, FIELDS, AND METHODS

```
class ChecksumAccumulator {
// class definition goes here
}
```

```
new ChecksumAccumulator
```

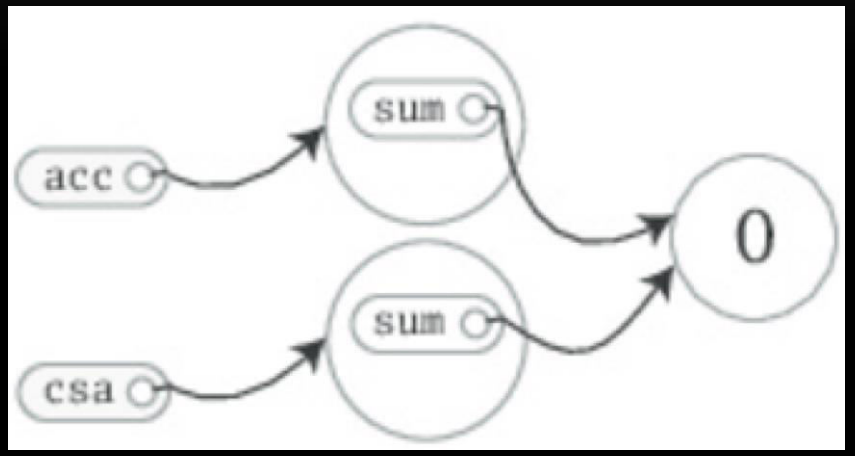


MEMBERS (FIELDS AND METHODS)

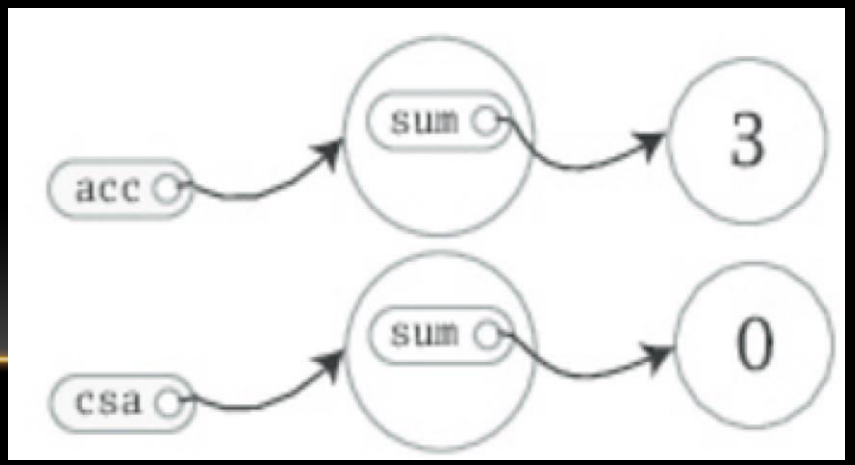
```
class ChecksumAccumulator {
var sum = 0
}
```

Two Instantiations

- val acc = new ChecksumAccumulator
- val csa = new ChecksumAccumulator
- acc.sum = 3



Since sum is var



PRIVATE FIELDS

```
class ChecksumAccumulator {
private var sum = 0
}
```

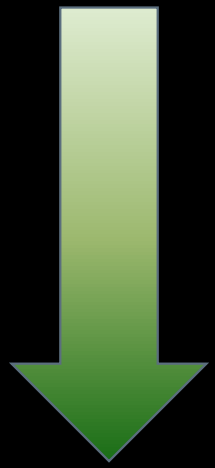
```
val acc = new ChecksumAccumulator
acc.sum = 5 // Won't compile, because sum is private
```

- Private fields can only be accessed by methods defined in the same class
- Members are made public in Scala is by not explicitly specifying any access modifier
- Public is Scala's default access level

METHODS TO ACCESS PRIVATE FIELDS

```
class ChecksumAccumulator {
    private var sum = 0
    def add(b: Byte): Unit = {
      sum += b
    }
    def checksum(): Int = {
      return ~(sum & 0xFF) + 1
    }
}
```

- Any parameters to a method can be used inside the method.
- Method parameters in Scala are vals, not vars and cannot be reassigned



```
def add(b: Byte): Unit = {
b = 1 // This won't compile, because b is a val
sum += b
}
```

A MORE CONCISE CHECKSUMACCUMULATOR

```
// In file ChecksumAccumulator.scala
class ChecksumAccumulator {
    private var sum = 0
    def add(b: Byte): Unit = { sum += b }
    def checksum(): Int = ~(sum & 0xFF) + 1
}
```

It is often better to explicitly provide the result types of public methods declared in a class even when the compiler would infer it for you

- Methods with a result type of Unit, such as ChecksumAccumulator's add method, are executed for their side effects.
- A side effect is generally defined as mutating state somewhere external to the method or performing an I/O action.
- In add's case, the side effect is that sum is reassigned.
- A method that is executed only for its side effects is known as a *procedure*.

SEMICOLON INFERENCE

- In a Scala program, a semicolon at the end of a statement is usually optional if the statement appears by itself on a single line.
- A semicolon is required if you write multiple statements on a single line
- Example: val s = "hello"; println(s)

- Parsing Multi-line statements

```
if (x < 2)
    println("too small")
else
    println("ok")
```

If you want to enter a statement that spans multiple lines, most of the time you can simply enter it and Scala will separate the statements in the correct place.

Treated as single 4-line statement

PARSING AS SINGLE STATEMENT

- Occasionally, however, Scala will split a statement into two parts against your wishes:

  x

  + y

  This parses as two statements x and +y.

- If you intend it to parse as one statement x + y, you can always wrap it in parentheses:

  (x

  + y)

- Alternatively, you can put the + at the end of a line.

  x +

  y +

  z

Whenever you are chaining an infix operation such as +, it is a common Scala style to put the operators at the end of the line instead of the beginning

RULES OF SEMICOLON INFERENCE

- A line ending is treated as a semicolon unless one of the following conditions is true:

1. The line in question ends in a word that would not be legal as the end of a statement, such as a period or an infix operator.

2. The next line begins with a word that cannot start a statement.

3. The line ends while inside parentheses (...) or brackets [...], because these cannot contain multiple statements anyway.

SINGLETON AND COMPANION OBJECTS

- Scala is more object-oriented than Java is that classes in Scala cannot have static members.
- Instead, Scala has *singleton objects*.
- A singleton object definition looks like a class definition, except instead of the keyword class you use the keyword object
- When a singleton object shares the same name with a class, it is called that class's *companion object*.
- You must define both the class and its companion object in the same source file.
- The class is called the *companion class* of the singleton object.
- A class and its companion object can access each other's private members.

CHECKSUMACCUMULATOR SINGLETON OBJECT

```
// In file ChecksumAccumulator.scala
import scala.collection.mutable
object ChecksumAccumulator {            //key word class used instead of object
  private val cache = mutable.Map.empty[String, Int]   //previous calc'd checksums are cached
   def calculate(s: String): Int =      //inputs string and calculates checksum
     if (cache.contains(s))           // check if already present as key in map
       cache(s)                       // if present, return the mapped value
     else {                           //calculate check sum
       val acc = new ChecksumAccumulator         //new check sum instance
           for (c <- s)                                  //cycles through each character
         acc.add(c.toByte)                //converts the character to a Byte and calls add()
       val cs = acc.checksum()                    //invokes checksum() on acc
       cache += (s -> cs)           //string key mapped to checksum value & added to cache
       cs     }   }                 // cs is the result of the method
```

SCALA APPLICATION

```
// In file Summer.scala
import ChecksumAccumulator.calculate
object Summer {
def main(args: Array[String]) = {
for (arg <- args)
println(arg + ": " + calculate(arg))
}
}
```

```
D:\PPL\Scala>scalac ChecksumAccumulator.scala Summer.scala

D:\PPL\Scala>scala Summer of love
of: -213
love: -182
```

CLASSES VS SINGLETON OBJECTS

- Singleton objects cannot take parameters, whereas classes can.
- Singleton object cannot be instantiated with the newkeyword, hence no way to pass parameters to it.
- Each singleton object is implemented as an instance of a *synthetic class* referenced from a static variable, so they have the same initialization semantics as Java statics.
- In particular, a singleton object is initialized the first time some code accesses it.
- A singleton object that does not share the same name with a companion class is called a *standalone object*.
- Standalone objects can be used for many purposes, including collecting related utility methods together or defining an entry point to a Scala application.

THANK YOU