Scala Class Review

• Every class has a **primary constructor** which is defined with the class definition.

```scala
class Person(val name: String, var age: Int){}
```

//define a class, its primary constructor and two instance variables.

defined class Person
scala> val John=new Person("John",20);
scala> John.name
res25: String = John
scala> John.age
res27: Int = 20
scala> John.age=30
John.age: Int = 30
scala> John.name="Jo"
<console>:10: error: reassignment to val
John.name="Jo"
```

Class Inheritance

• Like Java, Scala use `extends` to create a sub-class

```scala
class Employee (name: String, age: Int, var salary: Double) extends Person(name, age){}
```

//super-class constructor is called before calling the sub-class constructor

Defined class Employee
scala> val Tom=new Employee("Tom",18,0.0)
Tom: Employee = Employee@73a2335d
scala> Tom.name
res8: String = Tom
scala> Tom.age
res9: Int = 18
scala> Tom.salary
res10: Double = 0.0
```

Overriding Fields and Methods

• The top of the Scala class hierarchy is `Scala.AnyRef`, which all Scala classes inherit directly or indirectly.

• The keyword **override** is required to override an **instance variable** or a **method**:

```scala
class SecretAgent(val codename: String, age: Int) extends Person(codename,age){
override val name="secret";
override val toString= "unknown"
}
```

Defined class SecretAgent
scala> val OO7=new SecretAgent("OO7",100)
OO7: SecretAgent = unknown
scala> OO7.name
res6: java.lang.String = secret
scala> OO7.codename
res7: String = OO7
Abstract Class

- Similar to Java, an abstract class can not be instantiated, usually because one or more of its methods are not defined.
  
  abstract class Person (val name: String, var age: Int)
  -
    def id: Int
    // no method body, hence an abstract method, the abstract keyword is not needed

Extends Abstract Class

- A subclass that extends an abstract class needs to define all abstract methods in its class definition:
  scala> class Student(name: String, age: Int) extends Person(name, age)
  -
    def id = name.hashCode
  // override keyword is not required

Variables in Matching Pattern

- If the case keyword is followed by a variable name, the match expression is assigned to that variable:
  var str = "3"
  for (i = 0 to str.length)
    str(i) match {
      case '+' => sign = 1
      case '-' => sign = -1
      case _ => sign = 0
    }
  // case _ is equivalent to default.
  // no need to use break

Pattern Matching

- Scala match is a better Java switch
  scala
  switch (ch) {
    case '+' => sign = 1
    case '-' => sign = -1
    case _ => sign = 0
  }
  // case _ is equivalent to default.
  // no need to use break

Variables in Matching Pattern

- If the case keyword is followed by a variable name, the match expression is assigned to that variable:
  var str = "3"
  for (i = 0 to str.length)
    str(i) match {
      case '+' => sign = 1
      case '-' => sign = -1
      case ch => digit = Character.digit(ch, 10)
    }
  // ch = '3', digit = 3
Case Classes

- Case classes are a special kind of classes that are defined to use in pattern matching.

```scala
abstract class Expr // Expr is an abstract class
case class Number(n: Double) extends Expr
case class Sum(e1: Expr, e2: Expr) extends Expr
// Number and Sum are case classes.
val e = Expr() //
// e1, e2, e3, e4 are case objects.
val e1 = Number(10) // e1 is 10
val e2 = Sum(e1, Number(2)) // e2 is 10+2
val e3 = Sum(e2, e2) // e3 is?
val e4 = Sum(e1, e3) // e4 is?
```

Pattern Matching for Case Object

```scala
def eval(e: Expr): Double = e match{
  case Number(n) => n
  case Sum(e1, e2) => eval(e1) + eval(e2)
}
```

Trace Recursive Calls

```scala
def eval(e: Expr): Double = e match{
  case Number(n) => {println(n); n}
  case Sum(e1, e2) => {println(e1 +", " + e2); eval(e1) + eval(e2)}
}
```