作业参考

https://github.com/glebd/scala-course/tree/master/workspace
http://codegists.com/code/scala-quickcheck/

submit iamiman94@hotmail.com 5KDZ2CFx9227cpGR

useful shortcut

Here you can find a collection of shortcuts you will find useful for Scala development. Replace `Ctrl` by `CMD` if you are using a Mac.

- `Alt+Shift+X` `S`: Run As Scala Application. It is quicker than the popup menu because it does not need to scan every Launcher to see what type of Launches can be acted upon.
- `Alt+Shift+X` `T`: Run As JUnit. Also quicker than a popup menu for the same reasons.
- `Ctrl+Space`: completion (using code, templates,...)
- `Alt` `+` `/`: complete word (unless you have chosen to integrate word completion in completion proposals above)
- `Ctrl` `+` `/`: toggle comment of a block (the selected line(s) or current line)
- `Ctrl-3`: Quick Access – a huge time saver. Opens up a dialog with incremental search on all commands available on the platform
- `Ctrl-Shift-R`: Open Resource – opens up a dialog with incremental search on all files in the workspace
- `Ctrl-.`: Go to the next error in the current editor
- `F2`: Show the error for the position under the cursor
- `F3`: Navigate to definition (same as `Ctrl-click` on an identifier)
- `Ctrl-O`: Quick Outline – opens up a dialog with incremental search on all definitions in the current editor
- `Ctrl-J`: Incremental search
- `F11` or `CMD-F11` (Mac only): Launch the debugger. By default it tries to launch the current file. I configure Eclipse to always launch the last application.
- `Ctrl-F11` or `CMD-Shift-F11` (Mac only): Launch the application (with no debugger attached)

Ctrl+ shift + f10 / shift + f10　运行 scala 文件

- Ctrl + /: toggle comment of a block (the selected line(s) or current line)

If writing a bad import, Alt + enter to correct it

Alt + f12 ： open console

Rig

课外补充

http://www.cnblogs.com/elaron/archive/2013/01/18/2866142.html

scala 是以实现 scaleable language 为初衷设计出来的一门语言。官方中，称它是 object-oriented language 和 functional language 的混合式语言。并且，scala 可以和 java 程序无缝拼接，因为 scala 文件编译后也是成为.class 文件，并且在 JVM 上运行。不过，我更关心的是它的 scaleable（扩展性）

Tuple -可以将不同类型的数据存储在一个数组中

singleton objects - scala 中没有静态方法和属性，全部由 singleton object（单例对象）来替代

trait - scala 中的类 interface 物，但是可以拥有方法体，并且可以在类实例化时混入，而不需为了包装某个类而产生子类

Println string

Output function println can use string directly:
def str = ”abc”; println(str)
We can directly write
println(”abc”)

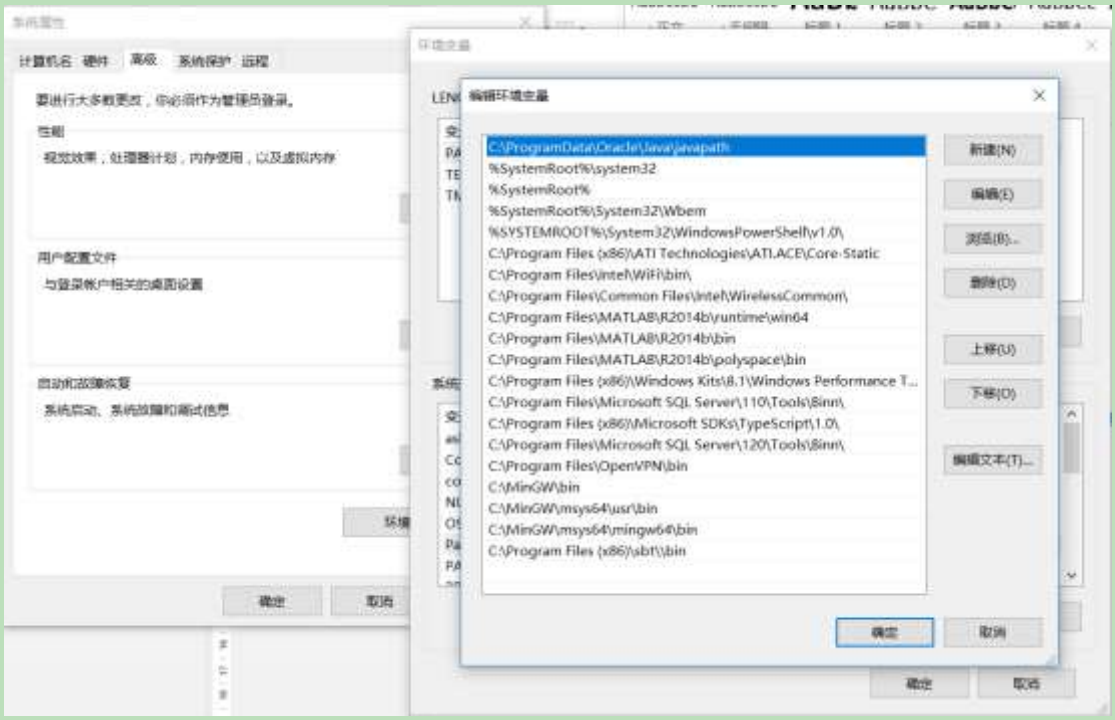
list 操作

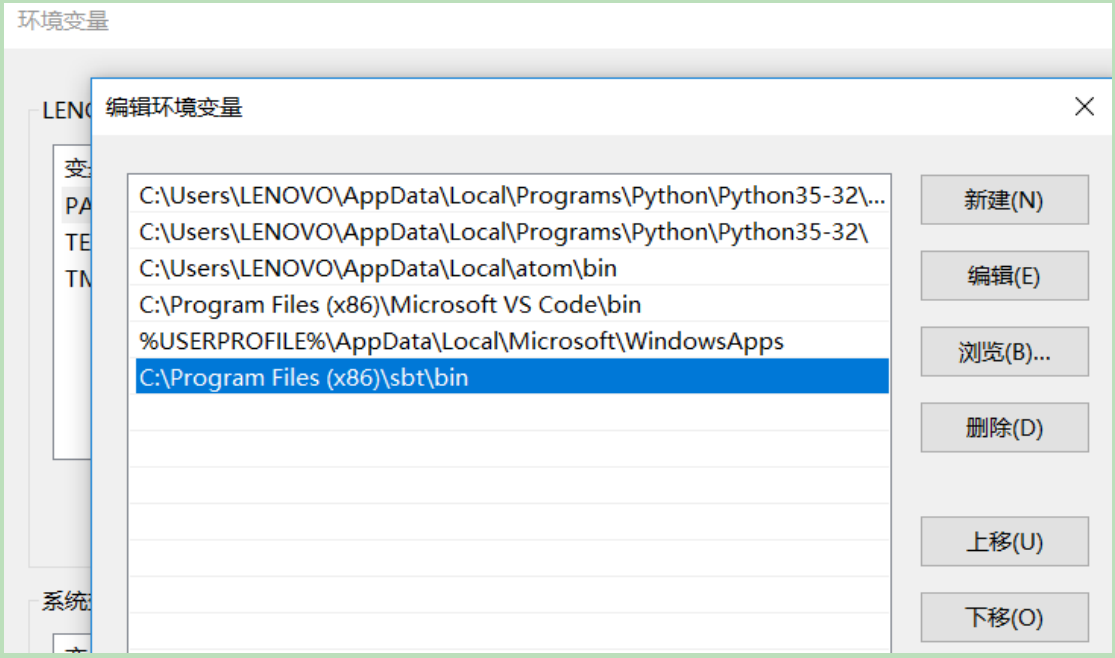
http://www.yiibai.com/scala/scala_lists.html

Week1

Installment

1. http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html
2. 安装 JDK，配置好 path 环境变量后 cmd 中输入 java –version 显示类似：
3. java version "1.8.0_102"
4. Java(TM) SE Runtime Environment (build 1.8.0_102-b14)
5. Java HotSpot(TM) 64-Bit Server VM (build 25.102-b14, mixed mode)

6. 

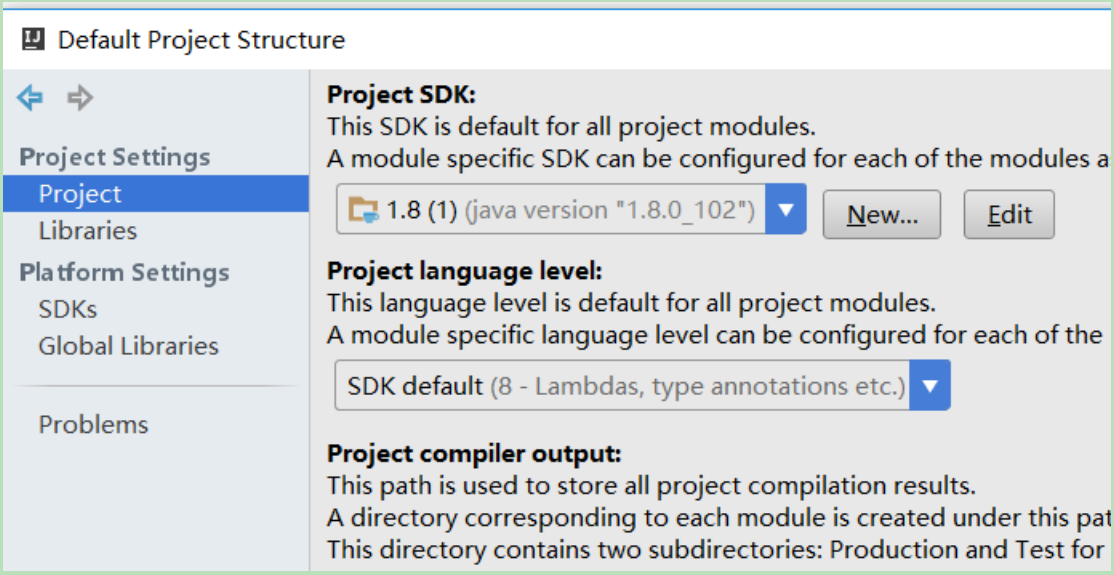
7. http://www.scala-sbt.org/release/docs/Setup.html
8. 安装 sbt 后仍然需要设置新建环境变量，完成后 cmd 中输入 sbt about 让其更新。最后完成可以看到版本是 0.13.12
9. 

10. Java HotSpot(TM) 64-Bit Server VM warning: ignoring option MaxPermSize=256m; support was removed in 8.0
11. [info] Set current project to lenovo (in build file:/C:/Users/LENOVO/)
12. [info] This is sbt 0.13.12
13. [info] The current project is {file:/C:/Users/LENOVO/}lenovo 0.1-SNAPSHOT
14. [info] The current project is built against Scala 2.10.6
15. [info] Available Plugins: sbt.plugins.IvyPlugin, sbt.plugins.JvmPlugin, sbt.plugins.CorePlugin,

sbt.plugins.JUnitXmlReportPlugin

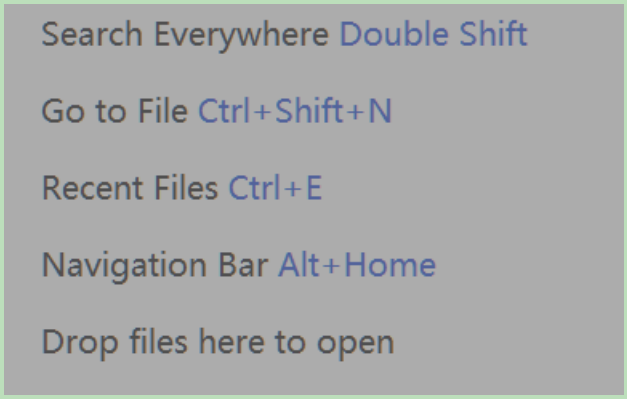
16. [info] sbt, sbt plugins, and build definitions are using Scala 2.10.6
17. https://www.jetbrains.com/idea/download/#section=windows
18. 安装 intellij idea

配置

19. https://www.jetbrains.com/help/idea/2016.2/configuring-global-project-and-module-sdks.html
20. 配置 **Configure → Project defaults → Project structure** and add the JDK
21. 

22. C:\Program Files\Java\jdk1.8.0_102

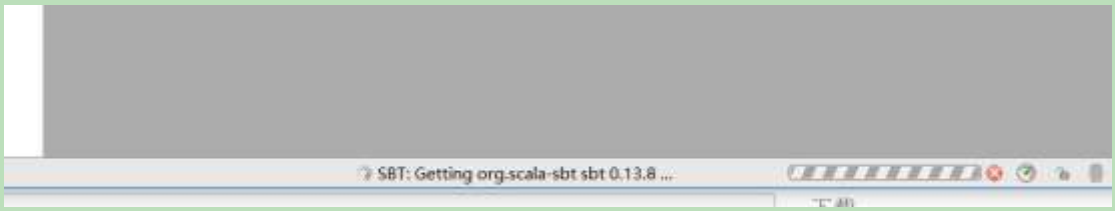
Short cut



下载 scala IDE

http://scala-ide.org/download/sdk.html

create a new project: scala→ sbt and wait for the configuration being done.（creating src director）

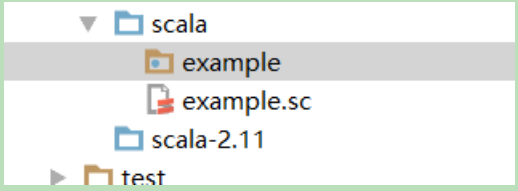


最后错误

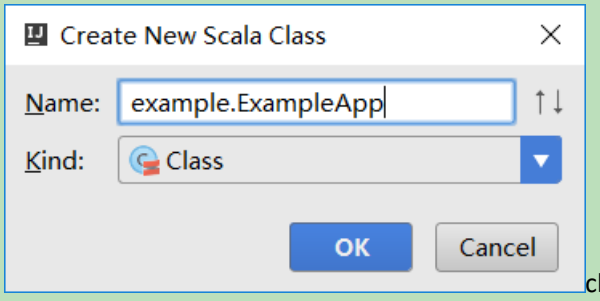
[warn] [FAILED] org.scala-sbt#compiler-interface;0.13.8!compiler-interface.jar(src): (0ms) [warn] ==== typesafe-ivy-releases: tried [warn] https://repo.typesafe.com/typesafe/ivy-releases/org.scala-sbt/compiler-interface/0.13.8/srcs/compiler-interface-sources.jar [warn] ==== sbt-plugin-releases: tried [warn] https://repo.scala-sbt.org/scalasbt/sbt-plugin-releases/org.scala-sbt/compiler-interface/0.13.8/srcs/compiler-interface-sources.jar [warn] ==== local: tried [warn] C:\Users\LENOVO\.ivy2\local\org.scala-sbt\compiler-interface\0.13.8\srcs\compiler-interface-sources.jar [warn] ==== jcenter: tried [warn] https://jcenter.bintray.com/org/scala-sbt/compiler-interface/0.13.8/compiler-interface-0.13.8-sources.jar [warn] ==== public: tried [warn] https://repo1.maven.org/maven2/org/scala-sbt/compiler-interface/0.13.8/compiler-interface-0.13.8-sources.jar [warn] :: [warn] :: FAILED DOWNLOADS :: [warn] :: ^ see resolution messages for details ^ :: [warn] :: [warn] :: org.scala-sbt#compiler-interface;0.13.8!compiler-interface.jar(src) [warn] ::

Create a worksheet and object

新建一个文件夹，在其中新建 scala class，选择为 object 类型

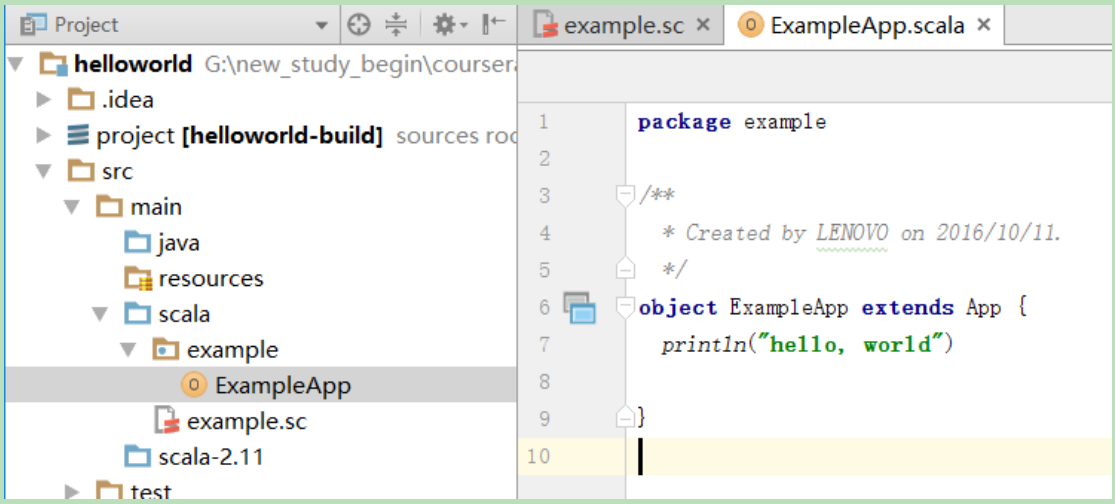


建立完 worksheet (example.sc)后，需要建立一个在 example 下的 class

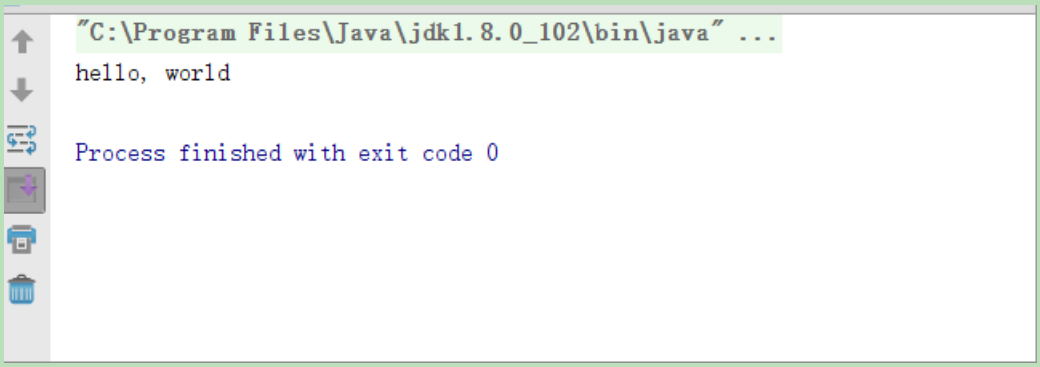


class 选择为 object

Rig

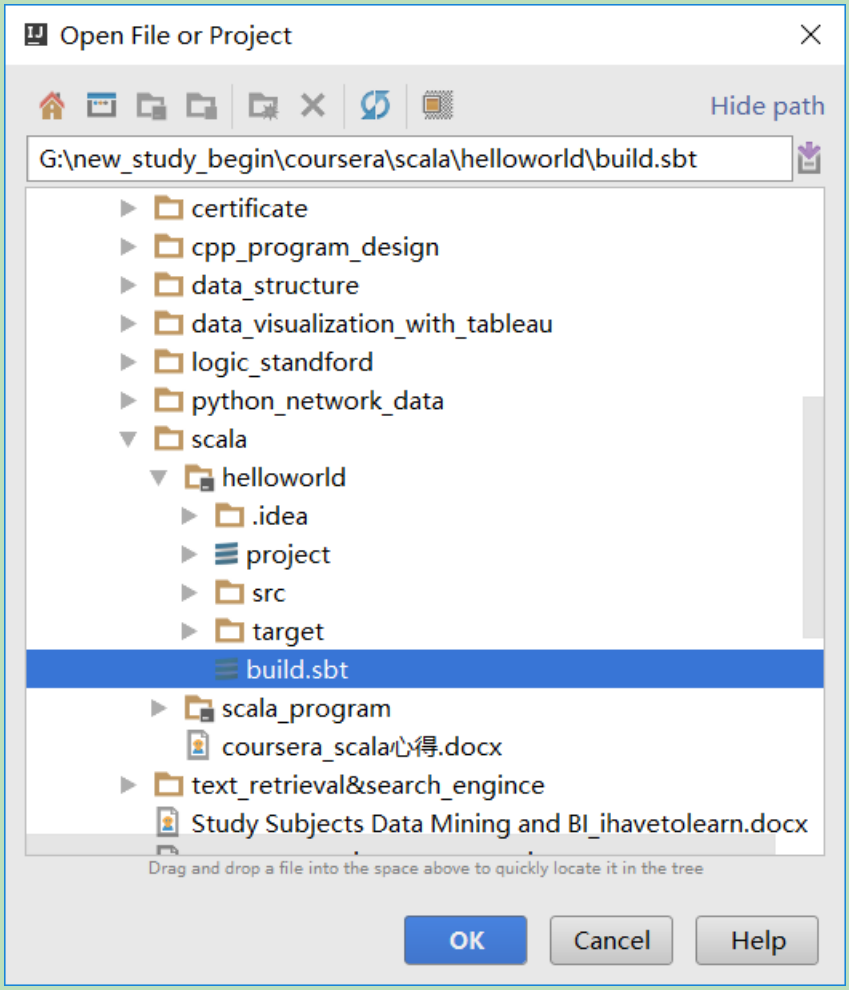


Ctrl+ shift + f10 运行



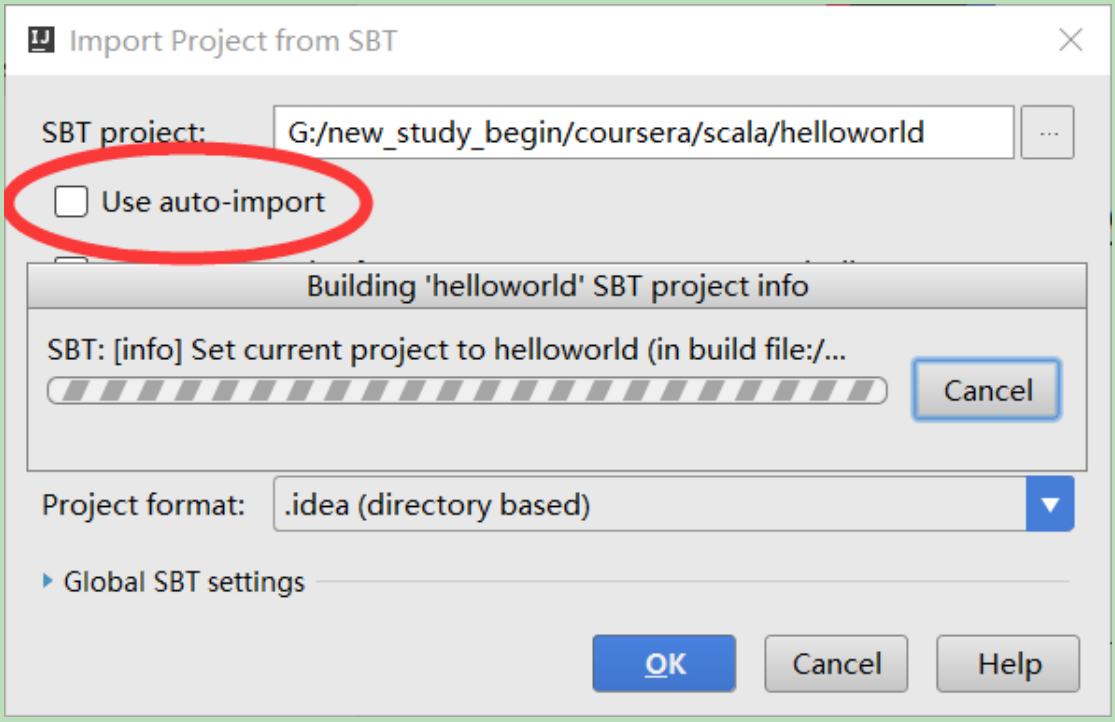
Rig

Open a project



IntelliJ IDEA SBT support synchronizes the project with your build file, so when you change Scala version you're going to use, or add a library, your project is updated accordingly. For the next time, you can avoid this step by checking off the option "Use auto-import" in Step 7.

Rig



强制修改使用的 libraryDependency

在 build.sbt 最后加上一句

```
libraryDependencies += "org.scalatest" %% "scalatest" % "2.2.6" % "test"
```

在 intellij 中打开终端

Alt + f12
G:\new_study_begin\coursera\scala\helloworld>sbt
Java HotSpot(TM) 64-Bit Server VM warning: ignoring option MaxPermSize=256m; su
pport was removed in 8.0
[info] Loading project definition from G:\new_study_begin\coursera\scala\hellow
orld\project
[info] Set current project to helloworld (in build file:/G:/new_study_begin/cou
rsera/scala/helloworld/)

project windows

view----tool windows—alt + 1

Rig

run a project

1. alt+ f12 在 intellij idea 中打开 terminal，其他命令或者快捷键可以用 help—find action 找到
2. 输入 sbt，然后变成在 sbt shell 中了，接下来进行 run，结束输入 exit 或者按下 ctrl+d。注意到此时的 pwd 是在含有 build sbt 的 base directory 了。所以 run 才会有效果
3. 即：G:\new_study_begin\coursera\scala\helloworld>ls
   build.sbt project src target
   成功范例
   G:\new_study_begin\coursera\scala\helloworld>sbt
   Java HotSpot(TM) 64-Bit Server VM warning: ignoring option MaxPermSize=256m; support was removed in 8.0
   [info] Loading project definition from G:\new_study_begin\coursera\scala\helloworld\project
   [info] Set current project to helloworld (in build file:/G:/new_study_begin/coursera/scala/helloworld/)
   > run
   [info] Running example.ExampleApp
   hello, world
   [success] Total time: 0 s, completed 2016-10-11 13:47:11
   > exit

*Sbt

sbt is an open-source build tool for Scala and Java projects, similar to Java's Maven and Ant. Its main features are: Native support for compiling Scala code and integrating with many Scala test frameworks

Running the Scala Interpreter inside SBT

The Scala interpreter is different than the SBT command line.
However, you can start the **Scala interpreter inside sbt** using the *`console`* task.
Scala REPL can only start up if there are no compilation errors in your code.
Cmd (in any directory)
Sbt
Console
Then the remainder become “scala>”,and we can input:
Println(“on, hey!”);

compile

The compile task will compile the source code of the assignment which is located in the directory src/main/scala.

The directory src/test/scala contains unit tests for the project. In order to run these tests in sbt, you can use the test command.

If your project has an object with a main method (or an object extending the trait App), then you can run the code in sbt easily by typing run. In case sbt finds multiple main methods, it will ask you which one you'd like to execute.

Open sbt shell under base directory, then input compile or test or run

basic knowledge

hierarchy

1. you'll see a `build.sbt` declared in the top-leve directory, that is, the base directory.

Source path:

```
src/
   main/
      resources/
         <files to include in main jar here>
      scala/
         <main Scala sources>
      java/
         <main Java sources>
   test/
      resources
         <files to include in test jar here>
      scala/
         <test Scala sources>
      java/
         <test Java sources>
```

2. Other directories in src/ will be ignored. Additionally, all hidden directories will be ignored.

**Classes, Traits, Objects and Packages

Classes

Classes in Scala are very similar to classes in Java. They are templates containing fields and methods. Like in Java, classes can be instantiated using the new construct, there can be many “instances” (or “objects”) of the same class.
In Scala there exists a special kind of class named case classes. You will learn about case classes during the course.
Classes in Scala **cannot have static members.** You can use objects (see below) to achieve similar functionality as with static members in Java.

Traits

Traits are like interfaces in Java, but they can also contain concrete members, i.e. method implementations or field definitions.

Objects

Object in Scala are like classes, but for **every object definition there is only one single instance**. It is **not possible** to create instances of objects using new, instead you can just access the members (methods or fields) of an object using its name.

Packages

Adding a statement such as package foo.bar at the top of a file makes the code in a file part of the package foo.bar. You can then do import foo.bar._ to make everything from package foo.bar available in your code. The content of a package can be scattered across many files. If you define a class MyClass in package foo.bar, you can import that specific class (and not anything else from that package) with import foo.bar.MyClass.
In Scala, everything can be imported, not only class names. So for instance if you have an object baz in package foo.bar, then import foo.bar.baz._ would import all the members of that object.
Hello, World! in Scala
Here are two ways to define a program which outputs “Hello, World!” in Scala:

```
object HelloWorld extends App {
   println("Hello, World!")
}
```

Or

Rig

```
object HelloWorld {
   def main(args: Array[String]) {
      println("Hello, World!")
   }
}
```

In Scala, the main or entry point method is defined in an object. An object can be made executable by either **adding extending the type App** or **by adding a method def main(args: Array[String])**.

Source Files, Classfiles and the JVM

Scala source code is stored in text files with the extension .scala. Typically Scala programmers create one source file for each class, or one source file for a class hierarchy: In fact, Scala allows multiple classes and objects to be defined in the same source file.

- The name of a Scala source file can be chosen freely, but it is recommended to use the name of a class which is defined in that file.
- Package hierarchies should be reflected in directory structure: a source file defining class C in package foo.bar should be stored in a subdirectory as foo/bar/C.scala. Scala does not really enforce this convention, but some tools such as the Scala IDE for eclipse might have problems otherwise.

The scala compiler compiles .scala source files to .class files, like the Java compiler. Classfiles are binary files containing machine code for the Java Virtual Machine. In order to run a Scala program, the JVM has to know the directory where classfiles are stored. This parameter is called the "classpath".

If you are using eclipse or sbt to compile and run your Scala code, you don't need to do any of the above manually - these tools take care of invoking the Scala compiler and the JVM with the correct arguments.

External Documentation

There is a large number of online resources available for learning Scala. The Learning Resources wiki page contains a few links that we think are the most useful for our class.

**Submitting Solution to Coursera

The sbt task submit allows you to submit your solution for the assignment. It will pack your source code into a .jar file and upload it to the coursera servers. Note that the code can only be submitted if there are no compilation errors.

Warnings: Before proceeding:

- Make sure that you run sbt in the root folder of the project (where the *build.sbt* file is).
- **Make sure that the console line is `>` and not *`scala>`*. Otherwise, you're inside the Scala console and not the sbt shell.**

The submit tasks takes two arguments: your Coursera e-mail address and the submission token. **NOTE**: the submission token is **not your login password**. Instead, it's a special password generated by coursera for every assignment. It is available on the Assignments page.

Rig

Press alt+f12 into cmd mode, then use console, input sbt

> submit iamiman94@hotmail.com WYzHuKrENKDR4iBZ
[info] Packaging /Users/luc/example/target/scala-2.11/parprog-example_2.11-1.0.0
-sources.jar ...
[info] Done packaging.
[info] Compiling 1 Scala source to /Users/luc/example/target/scala-2.11/classes
...
[info] Connecting to coursera. Obtaining challenge...
[info] Computing challenge response...
[info] Submitting solution...
[success] Your code was successfully submitted: Your submission has been
accepted and will be graded shortly.
[success] Total time: 6 s, completed Aug 10, 2012 10:35:53 PM
>

Learning resource

Quick References

- Scala Standard Library API
- Scala School!: A Scala tutorial by Twitter
- A Tour of Scala: Tutorial introducing the main concepts of Scala
- Scala Overview on StackOverflow: A list of useful questions sorted by topic

Week 1

- Martin's talk at OSCON 2011: Working Hard to Keep it Simple (slides)
- Books:
- ***Structure and Interpretation of Computer Programs***. *Harold Abelson and Gerald J. Sussman. 2nd edition. MIT Press 1996.* - [**Full text available online**].
- ***Programming in Scala***. Martin Odersky, Lex Spoon and Bill Venners. 3nd edition. Artima 2016.http://www.artima.com/shop/programming_in_scala_3ed
- ***Programming in Scala***. *Martin Odersky, Lex Spoon and Bill Venners. 2nd edition. Artima 2010.* - [**Full text of 1st edition available online**].Artima has graciously provided a 25% discount on the 2nd edition of*Programming in Scala* to all participants of this course. To receive the discount, simply visithttp://www.artima.com/shop/programming_in_scala_2ed and during checkout, please use the coupon code: COURSERA-ODERSKY to have the discount applied.
- ***Scala for the Impatient***. *Cay Horstmann. Addison-Wesley 2012.* - [**First part available for download**.]
- ***Scala in Depth***. *Joshua D. Suereth. Manning 2012.* - [**Available for purchase**].
- ***Programming Scala***. *Dean Wampler and Alex Payne. O'Reilly 2009.* [**Available for purchase**].

Rig

Week 2

- Glossary of Scala and FP terms, for any issues with terminology.

Week 3+

- Scala By Example, for more examples which illustrate concepts covered in the lectures.
- Scala Cheatsheet, for a quick reference covering pattern matching syntax, for-comprehension syntax, and more.

Advantage of IDEA compared to eclipse

http://www.oschina.net/news/26929/why-intellij-is-better-than-eclipse

*使用心得（增强体验）

http://www.ituring.com.cn/article/37792

1. In Scala, the standard is to indent using 2 spaces (no tabs).

2. #6 Common Subexpressions

Expression :

You should avoid unnecessary invocations of computation-intensive methods. For example

```
this.remove(this.findMin).ascending(t + this.findMin)
```

invokes the this.findMin method twice. If each invocation is expensive (e.g. has to traverse an entire data structure) and does not have a side-effect, you can save one by introducing a local value binding:

```
val min = this.findMin

this.remove(min).ascending(t + min)
```

This becomes even more important if the function is invoked recursively: in this case the method is not only invoked multiple times, but an exponential number of times.

Semicolons in Scala are only required when writing multiple statements on the same line. Writing unnecessary semicolons should be avoided, for example:

the return statements can simply be dropped.

#11 Avoid mutable local Variables

Rig

Since this is a course on functional programming, **we want you to get used to writing code in a purely functional style, without using side-effecting operations.** You can often rewrite code that uses mutable local variables to code with helper functions that take accumulators.

Instead of:

```
def fib(n: Int): Int = {
   var a = 0
   var b = 1
   var i = 0
   while (i < n) {
      val prev_a = a
      a = b
      b = prev_a + b
      i = i + 1
   }
   a
}
```

Prefer;

```
def fib(n: Int): Int = {
   def fibIter(i: Int, a: Int, b: Int): Int =
      if (i == n) a else fibIter(i+1, b, a+b)
   fibIter(0, 0, 1)
}
```

#12 Eliminate redundant “If” Expressions

Instead of ： ”if (cond) true else false”
you can simply write

“cond”
(Similarly for the negative case).

可能有帮助的网站

1. 解决 Maven、sbt 无法下载依赖包的问题
http://dblab.xmu.edu.cn/blog/maven-network-problem/

homework

1. When working on an assignment, it is important that you don't change any already implemented method, class or object names or types. Doing so will prevent our automated grading tools from working and you have a high risk of not obtaining any points for your solution.

Fix max function in example.Lists._

Double shift to fine Lists， remember to keep the parentheses balanced.

```
def max(xs: List[Int]): Int = {
  if (xs.isEmpty) {
    throw new java.util.NoSuchElementException()
  }
  val tailMax = if (xs.tail.isEmpty) xs.head else max(xs.tail)
  if (xs.head >= tailMax) {
    xs.head
  }
  else tailMax
}
```

run in console

use cmd and set the current directory is base directory(build.sbt is in it)
input:
sbt
console
then it becomes:
scala> import example.Lists._
import example.Lists._

scala> max(List(1,3,2))
res1: Int = 3

run by create new object

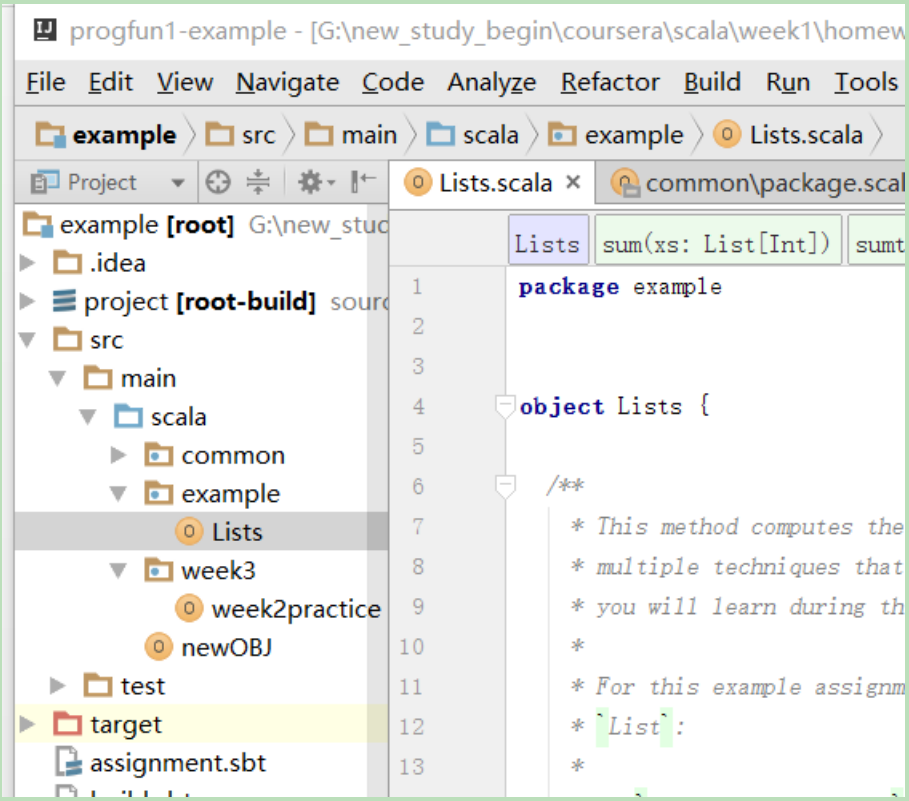
right-click on the package example in**src/main/scala** and select “New” - “Scala Object”; in IDEA, new a Scala class and then change the type into object.
Name it with anything u like, then write the code below in it:
import example.Lists

Rig

```
/**
  * Created by LENOVO on 2016/10/12.
  */
object newOBJ extends App {
  println(Lists.max(List(1,3,2)))
}
```

without import it can't resolve "Lists".

File architecture



Under scala, every directory is a package with many different objects， it seems that you can only run the newOBJ with ctrl + shift + f10. If u create new object under scala, it can't be run.

But u can create a lots of new objects with new defined functions in the example or common program, then use " import example.objectname" or " example.objname.funcname" to import object or function.

Use" objname.funcName" or " funcName" respectively to call the functions in newOBJ body.

Max and sum function.

```
package example


object Lists {

  /**
    * This method computes the sum of all elements in the list xs. There are
    * multiple techniques that can be used for implementing this method, and
    * you will learn during the class.
    *
    * For this example assignment you can use the following methods in class
    * `List`:
    *
    *  - `xs.isEmpty: Boolean` returns `true` if the list `xs` is empty
    *  - `xs.head: Int` returns the head element of the list `xs`. If the list
    * is empty an exception is thrown
    *  - `xs.tail: List[Int]` returns the tail of the list `xs`, i.e. the the
    * list `xs` without its `head` element
    *
    * ''Hint:'' instead of writing a `for` or `while` loop, think of a recursive
    * solution.
    *
    * @param xs A list of natural numbers
    * @return The sum of all elements in `xs`
    */
  def sum(xs: List[Int]): Int = {//using tail recursive
    def sumtail(ss:List[Int], intTail:Int): Int = {
      if (ss.isEmpty) (intTail) else sumtail(ss.tail, (intTail + ss.head))
    }
    if (xs.isEmpty) 0 else (sumtail(xs.tail, xs.head))


  }

  /**
    * This method returns the largest element in a list of integers. If the
    * list `xs` is empty it throws a `java.util.NoSuchElementException`.
    *
    * You can use the same methods of the class `List` as mentioned above.
    *
    * ''Hint:'' Again, think of a recursive solution instead of using looping
    * constructs. You might need to define an auxiliary method.
```

```
    *
    * @param xs A list of natural numbers
    * @return The largest element in `xs`
    * @throws java.util.NoSuchElementException if `xs` is an empty list
    */
  def max(xs: List[Int]): Int = {
    if (xs.isEmpty) {
      throw new java.util.NoSuchElementException()
    }
    val tailMax = if (xs.tail.isEmpty) xs.head else max(xs.tail)
    if (xs.head >= tailMax) {
      xs.head
    }
    else tailMax
  }
}
```

class1

programming paradigm

Paradigm: In science, a paradigm describes distinct concepts or thought patterns in some scientific discipline.

Main programming paradigms:
- imperative programming
- functional programming
- logic programming

These three types are Orthogonal to it:
- object-oriented programming

Imperative programming

Imperative programming is about
- modifying mutable variables,
- using assignments
- and control structures such as if-then-else, loops, break, continue, return.

pure imperative programming is limited by the “Von

Rig

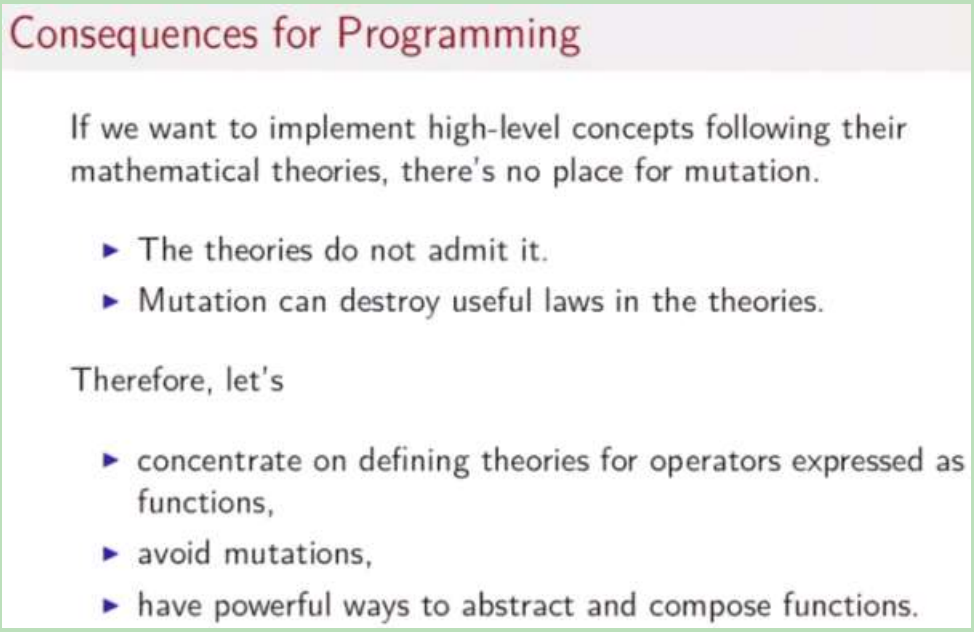
Neumann" bottleneck:
One tends to conceptualize data structures word-by-word.

We need other techniques for defining high-level abstractions!
Such as collections, polynomials, geometric shapes, strings, documents.
Ideally: Develop theories of collections, shapes, strings, …

Theroy

A theory consists of
▸ one or more **data types**
▸ **operations** on these types
▸ laws that describe the **relationships** between values and operations
Normally, **a theory does not describe mutations!**

. So what's important is that a theory mathematics does not describe mutations. A mutation means that I have changed something while **keeping the identity of the thing the same**.(change the calculation of the same operators, like operator reload in C++)



Functional programming

In a restricted sense, functional programming (FP) means programming without mutable variables, assignments, loops, and other imperative control structures.

Rig

In particular, functions in a FP language are **first-class citizens.**
This means
▸ they can be defined anywhere, including inside other functions
▸ like any other value, they can be passed as parameters to functions and returned as results
▸ as for other values, there exists a set operators to compose Functions

Book recommended

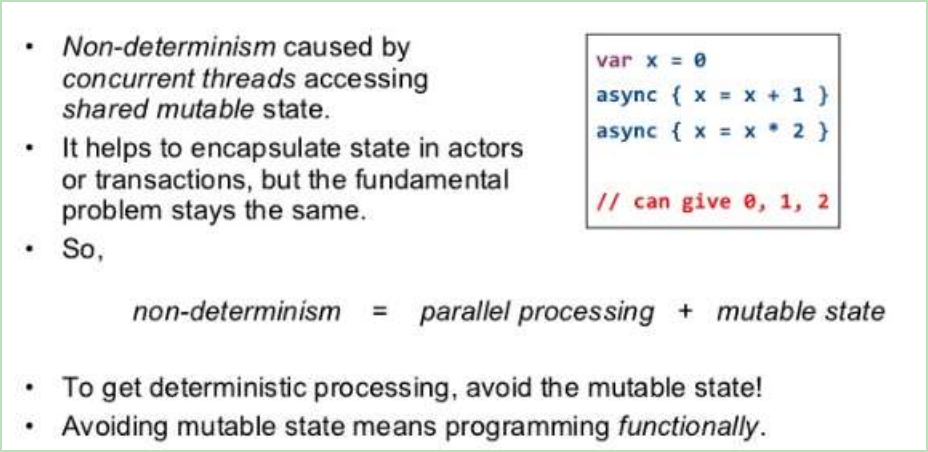
Structure and Interpretation of Computer Programs. Harold Abelson and Gerald J. Sussman. 2nd edition. MIT Press 1996.
A classic. Many parts of the course and quizzes are based on it, but we change the language from Scheme to Scala.

Programming in Scala. Martin Odersky, Lex Spoon, and Bill Venners. 2nd edition. Artima 2010.

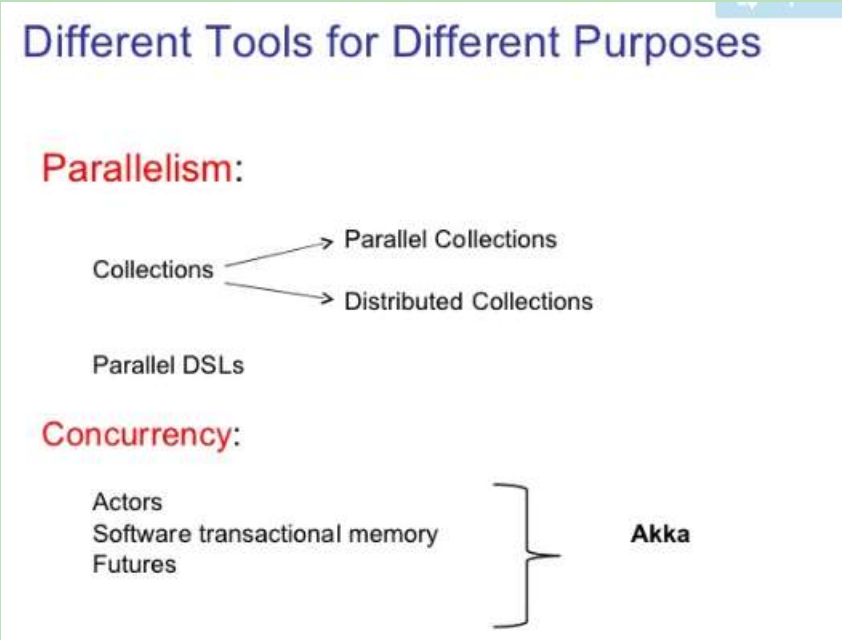
**"Working Hard to Keep It Simple

Scala is very good for exploiting parallelism on multi call and cloud computing.
https://www.youtube.com/watch?v=3jg1AheF4n0

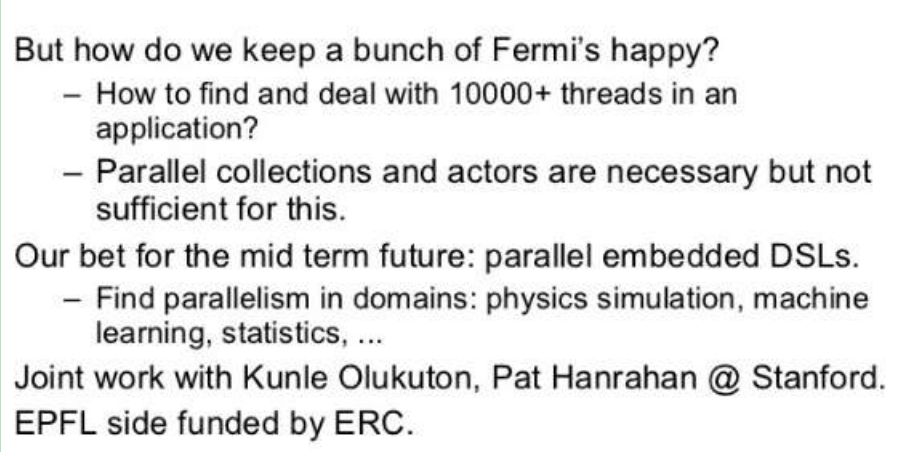
functional programming is inherently avoiding variable mutation.



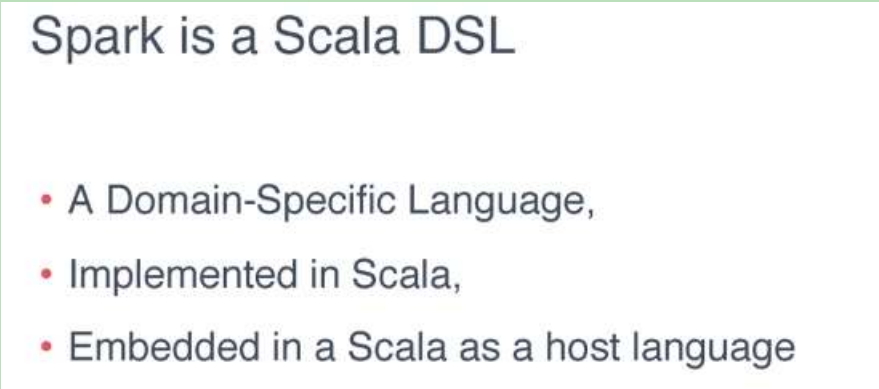
Scala provide both mutable variable and immutable value.



Collection： encapsulated data structure, like set, list and so on



https://www.youtube.com/watch?v=NW5h8d_ZyOs



I think the following video a great full introduction of scala system

Rig

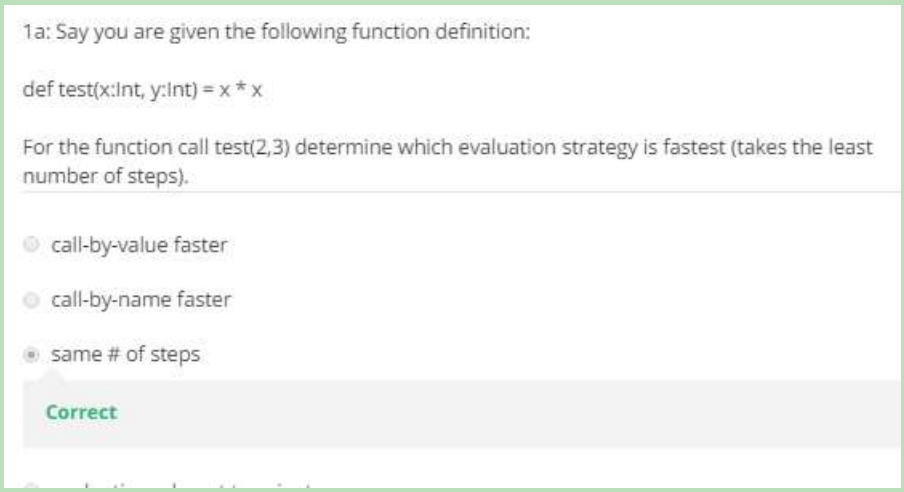
https://www.youtube.com/watch?v=NW5h8d_ZyOs

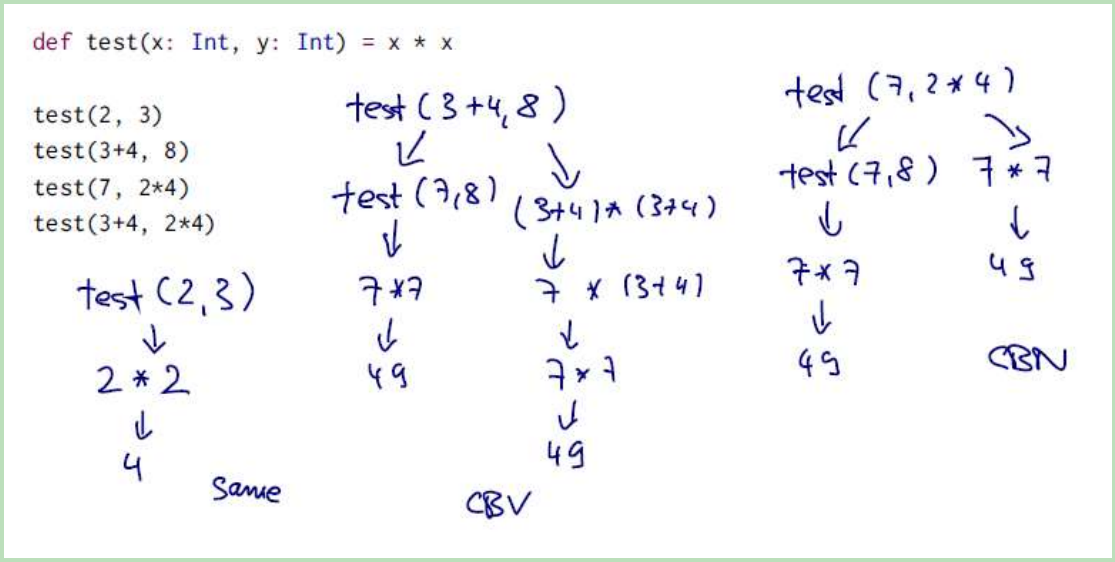
class 2

What's important though is to know that number calculus as a model, and the substitution model, can be applied only to expressions that do not have a side effect. Now, what is a side effect?

When we deal with “c++” in cpp, we need to return the value of c and then plus c with 1, therefore we need something, like store the current value of this identifier.

In other words the expression C++ has a side effect on the current value of the variable. And that side effect cannot be expressed by the substitution mode





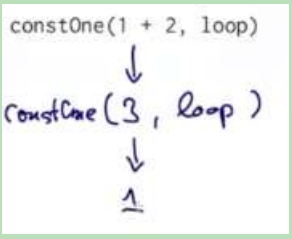
Class3

CBN and CBV

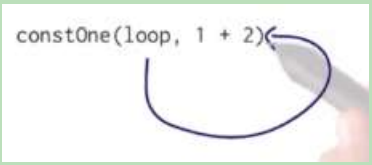
Call by value is often exponentially more efficient than call by name because it avoids this repeated recomputation of argument expressions that call by name tails.
The other argument for call by value is that it plays much nicer with, imperative effects and side effects because you tend to know much better when expressions will be evaluated.
Since **Scala i**s also an, an imperative side, **call by value is the standard choice.**

But if the type of a function parameter starts with => it uses
Def constOne(x:Int, y:Int) = 1 ----------call by value
call-by-name.
Example:
def constOne(x: Int, y: => Int) = 1

Let's trace the evaluations of
constOne(1+2, loop)



second parameter is call – by – name, therefore it stays as a name during first substitution model step. Loop would ignore because of it's a parameter.
My conclusion: this can use in the situation that one of the parameter has a chance of become a loop, and we need to ignore it.
and
constOne(loop, 1+2)



here we have to reduce the first argument because it's call by value, and we get into the same situation that loop produces to itself.

Class 4

**Expression not a statement

It looks like a if-else in Java, but is used for expressions, not statements.
Example:
def abs(x: Int) = if (x >= 0) x else -x
x >= 0 is a predicate, of type Boolean.

**Loop and call-by-name

```
def loop:Boolean = loop//this will invoke a warning not a exception
val a = 0;
def or(x:Boolean, y : =>Boolean) = if(!x) y else true
def and(x:Boolean, y : => Boolean): Boolean = if(x) y else true

or(a == 0, loop)

and(false, loop)
or(a>0, loop)
and(a ==0, loop)

worksheet result
loop: loop[] => Boolean
a: Int = 0
or: or[](val x: Boolean,val y: => Boolean) => Boolean
and: and[](val x: Boolean,val y: => Boolean) => Boolean

res0: Boolean = true

res1: Boolean = true

last two statement can't not be terminated as they are looping.
If I change the and definition as :
```

def and(x:Boolean, y : Boolean): Boolean = **if**(x) y **else true**

then:

and(**false**, loop) can be looping during compilation because y is passed as a value and once invoke and function this will loop.

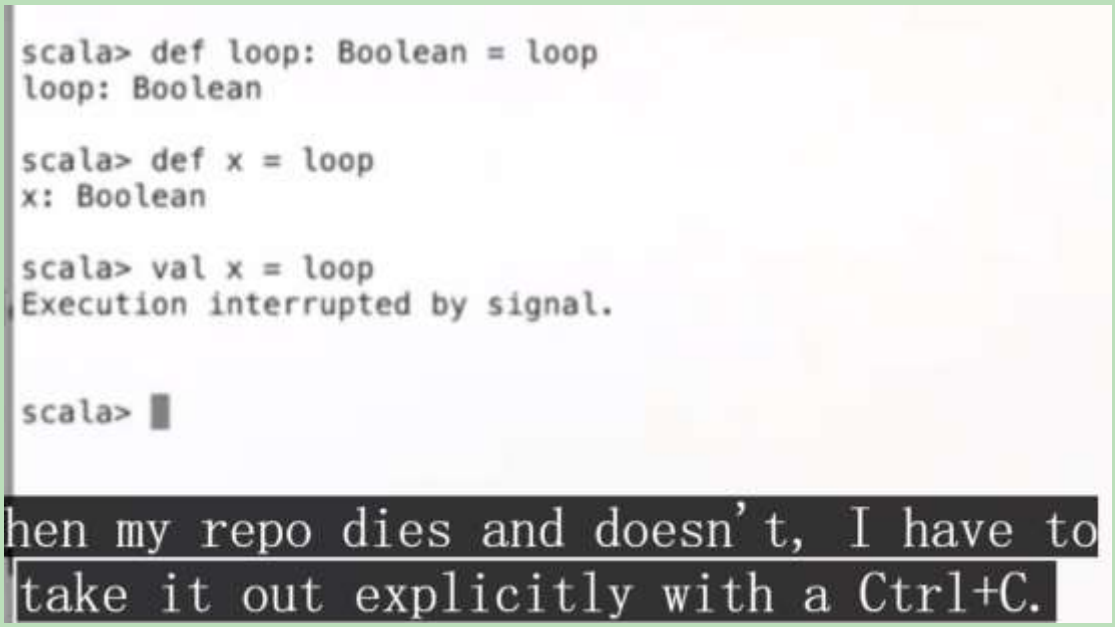
So does or function, once y is passed as a value, all will crash in calculation y.

```
def or(x:Boolean, y :Boolean) = if(!x) y else true
```

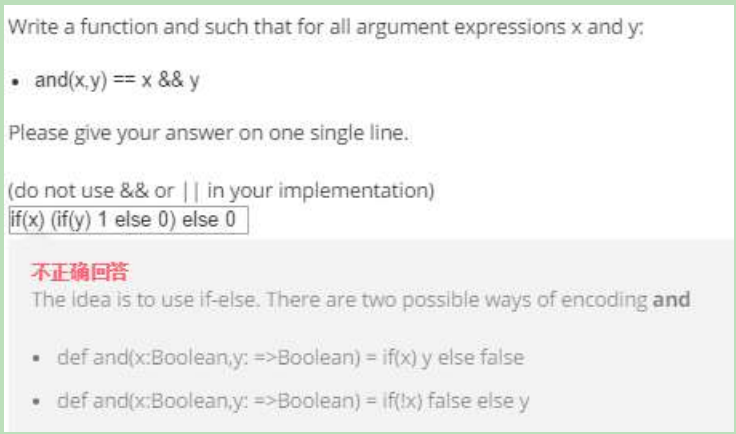
Rig

```
or(a == 0, loop)
```

We just defined another name for loop, whereas if I defined val x equals loop



Quiz



class 5

def sqrtIter(guess: Double, x: Double): **Double** =
if (isGoodEnough(guess, x)) guess
else sqrtIter(improve(guess, x), x)

the result type (double) is required as scala will look through the body to find the type of result. And else part invoke sqrtIter again so you have to name the type to make it explicit.

Rig

CODE Sqrt function

```
def abs(x:Double) = if (x < 0) -x else x

def sqrtIter(guess: Double, x: Double): Double =
  if (isGoodEnough(guess, x)) guess
  else sqrtIter(improve(guess, x), x)

def isGoodEnough(guess: Double, x: Double) = abs(x/guess/guess - 1)<0.0001

def improve(guess: Double, x: Double) =(guess + x / guess) / 2

def sqrt(x: Double) = sqrtIter(1.0, x)

sqrt(1)
sqrt(2)
sqrt(1e-6)
sqrt(1e60)
```

isGoodEnough function is important and should use a proportional distance instead of exact distance.

Long expression span lines

There are two ways to overcome this problem.
You could write the multi-line expression in parentheses, because
semicolons are never inserted inside (...):
(someLongExpression
+ someOtherExpression)
Or you could write the operator on the first line, because this tells
the Scala compiler that the expression is not yet finished:
someLongExpression +
someOtherExpression

tail-recursive

As Donald Knuth has said, premature optimization is the source of all evil.

Hw

CODE Parentheses balance

!!! use else when using if, it's a expression not a statement and there is always have to be a return value.
In this case if you don't use else for the first if clause, when stack is negative, the complier will continue check the rest of body. And execute the rest then return a great result.

```
def balance(chars: List[Char]): Boolean = {
  def checkStack(stack:Int, rightList:List[Char]): Boolean ={
    if (stack < 0) false//everytime use a if remember use else,
    else{
      if (rightList.isEmpty) {
        if(stack == 0) true else false}
      else{
        if (rightList.head == '(') checkStack(stack + 1, rightList.tail)
        else if (rightList.head == ')') checkStack(stack - 1, rightList.tail)
        else checkStack(stack, rightList.tail)
      }
    }
  }
  checkStack(0, chars)
}
```

*CODE recursive enumeration

```
Write a recursive function that counts how many different ways you can make change for an amount, given a list of coin denominations. For example, there are 3 ways to give change for 4 if you have coins with denomination 1 and 2: 1+1+1+1, 1+1+2, 2+2.
def countChange(money: Int, coins: List[Int]): Int = {
  def check(lastMoney: Int, lastCoins: List[Int]): Int = {
    if (lastCoins.isEmpty) 0
    else {
```

```
      if (lastMoney == lastCoins.head) 1
      else if (lastMoney >= lastCoins.head)
        check(lastMoney - lastCoins.head, lastCoins) + check(lastMoney, lastCoins.tail)
      else 0
    }
  }
  check(money, coins.sorted)
}
```

week2

high-order function

function can take function handler as parameter, first order function only takes data types parameters.

Functional languages treat functions as first-class values.

Code:

```
def sumInts(a:Int, b:Int): Int = {
  if (a > b) 0 else (a + sumInts(a + 1, b))
}
def cube(x:Int): Int = x*x*x

def sumCubes(a:Int, b:Int) : Int = {
  if (a > b) 0 else (cube(a) + sumCubes(a + 1, b))
}
def sum(f:Int => Int, a: Int, b: Int): Int =
  if (a>b) 0 else f(a) + sum(f, a+ 1, b)
println(sumCubes(1,10))
println(sum(cube, 1, 10))//Higer-Order-Functions
```

Anonymous function syntax

Several ways to define cube function:

```
1.  def sum(f:Int => Int, a: Int, b: Int): Int =
  if (a>b) 0 else f(a) + sum(f, a+ 1, b)

println(sum(((x:Int) =>x*x*x), 1, 10))
```

```
    def sumCubes2(a:Int, b:Int) = sum(x=> x*x*x, 1, 10)
2.    def sum(f:Int => Int)(a:Int, b:Int):Int =
        if (a>b) 0 else f(a) + sum(f) (a + 1, b)
    //  如果你定义 sum 成为上面这样就不是返回一个函数，则
    //  不能用来定义 sumInts 和 sumCubes
    //  def sumInts = sum(x=>x)
    //  def sumCubes = sum(x=>x*x*x)
      def fact(x:Int):Int = {
        if (x == 1) 1
        else x*fact(x - 1)
      }

      println(sum(fact)(1,10))
      println(sum((x:Int) =>x)( 1, 10))
    //  println(sum(x:Int=>x*x*x)( 1, 10))this can't be compiled through since it does
    not
      //use () in x:Int
```

We can omit the variable types when the compiler can resolve it by somethings we define before this var.

Anonymous Functions are Syntactic Sugar

An anonymous function $(x_1 : T_1, \ldots, x_n : T_n) \Rightarrow E$ can always be expressed using def as follows:

$$\{\ \text{def } f(x_1 : T_1, \ldots, x_n : T_n) = E; f\ \}$$

where f is an arbitrary, fresh name (that's not yet used in the program).

- One can therefore say that anonymous functions are *syntactic sugar*.

Currying

https://zh.wikipedia.org/wiki/%E6%9F%AF%E9%87%8C%E5%8C%96
把接受多个参数的函数变换成接受一个单一参数（最初函数的第一个参数）的函数，并且返回接受余下的参数而且返回结果的新函数的技术。这个技术由 Christopher Strachey 以逻辑学家哈斯凯尔·加里命名的，尽管它是 Moses Schönfinkel 和戈特洛布·弗雷格发明的。

```
var foo = function(a) {
   return function(b) {
      return a * a + b * b;
   }
}
```

这样调用上述函数：(foo(3))(4)，或直接 foo(3)(4)。

Converting a function with multiple arguments into a function with a single argument that returns another function.

```
def f(a: Int, b: Int): Int // uncurried version (type is (Int, Int) => Int)
def f(a: Int)(b: Int): Int // curried version (type is Int => Int => Int)
```

Int => Int => Int is a omission of parentheses version of Int => (Int => Int)

Reason

Look again at the summation functions:

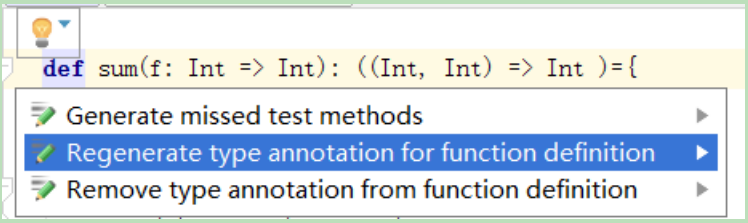
```
def sumInts(a: Int, b: Int) = sum(x => x, a, b)
def sumCubes(a: Int, b: Int) = sum(x => x * x * x, a, b)
def sumFactorials(a: Int, b: Int) = sum(fact, a, b)
```

Question

Note that a and b get passed unchanged from sumInts and sumCubes into sum.

Can we be even shorter by getting rid of these parameters?

Solution: function returning functions



this option will help u remove the redundant parenthese.

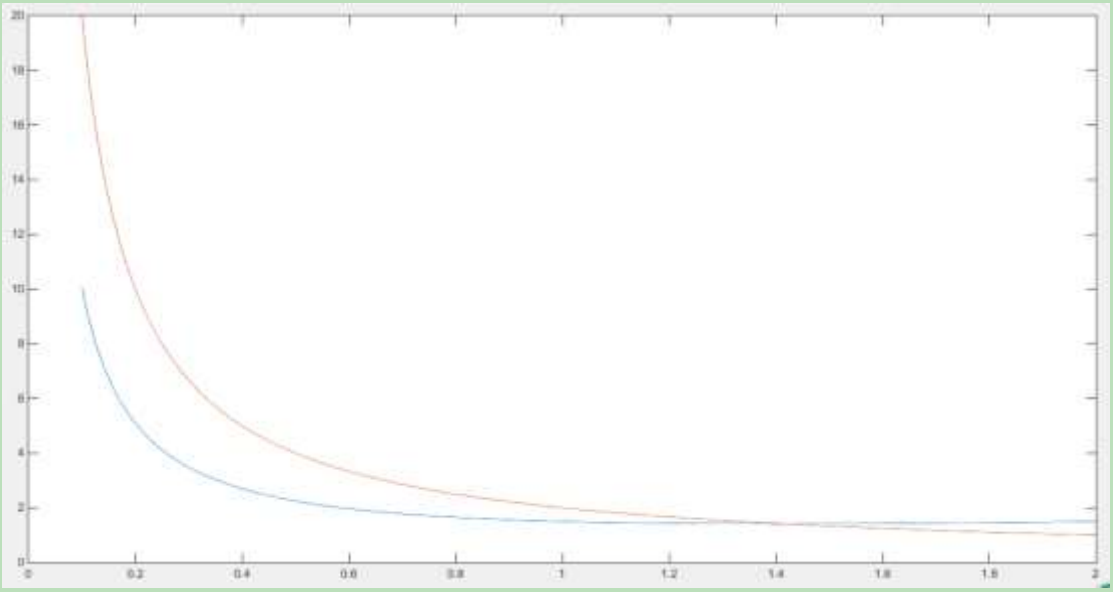
Rig

```
  def sum(f: Int => Int): (Int, Int) => Int ={

  def sumF(a: Int, b: Int): Int =
    if (a > b) 0
    else f(a) + sumF(a + 1, b)

  sumF
}

  def sumInts = sum(x=>x)
  def sumCubes = sum(x=>x*x*x)
  def fact(x:Int):Int = {
    if (x == 1) 1
    else x*fact(x - 1)
  }
  def sumFacts = sum(fact)

  println(sumFacts(1,10))
  println(sumCubes(1, 10))
  println(sumInts(1, 10))
```

fixed point and sqrt function

```
/*L2.3*/
val tolerantError = 0.000001
def abs(x:Double): Double = if(x>=0) x else -x
def isClosedEnough(x:Double, y:Double) = abs(x-y) < tolerantError
def fixedPoint(f:Double => Double)(firstGuess: Double) = {
  def iterate(guess:Double): Double = {
    val next = f(guess)
    println(next)
    if (isClosedEnough(next, guess)) next
    else iterate(next)
  }
  iterate(firstGuess)
}
def sqrt1(x:Double) = fixedPoint((y:Double) => x/y)(1)//this function will oscillate
due to the function choose.
def sqrt2(x:Double) = fixedPoint((y:Double) => (y + x/y)/2)(1)
println(" ans = " + sqrt2(2))
```

Explanation of difference of convergence



Since y = f(x) = 2/x is symmetric to y = x line, and y = (x + 2/x)/2 is asymmetric to the diagonal line, which prevents the repeated oscillation.

averageDamp + fixedfind

We said we derive that the solving the equations for square roots means taking the fixed point of this function(y = x/y). We determined that we needed to take averages to dampen the oscillation(which cause by y =f(y) = x/y), so we use this average damp function in the middle, and we are done

```
import math.abs

object exercise {
  val tolerance = 0.0001                          > tolerance : Do
  def isCloseEnough(x: Double, y: Double) =
    abs((x - y) / x) / x < tolerance              > isCloseEnough:
  def fixedPoint(f: Double => Double)(firstGuess: Double) = {
    def iterate(guess: Double): Double = {
      val next = f(guess)
      if (isCloseEnough(guess, next)) next
      else iterate(next)
    }
    iterate(firstGuess)
  }                                               > fixedPoint: (f
  fixedPoint(x => 1 + x/2)(1)                     > res0: Double =
  def averageDamp(f: Double => Double)(x: Double) = (x + f(x))/2
                                                  > averageDamp: (
  def sqrt(x: Double) =
    fixedPoint(averageDamp(y => x / y))(1)        > sqrt: (x: Doub
  sqrt(2)                                         > res1: Double =
}
```

Abstraction function

```
   def product(f:Int => Int)(x:Int, y:Int): Int =
      if (x > y) 1 else f(x)*product(f)(x + 1, y)
    def fl(x:Int) = x*x*x;
    def factorial(x:Int) = product(fl)(1, x)

    //abstract generalization
    def generalProduct(f:Int =>Int, combineFunc:(Int, Int)=> Int, zero:Int): (Int, Int)
=>Int = {
      def generalFunc(a:Int, b:Int):Int ={
        if (a>b) zero
        else combineFunc(f(a), generalFunc(a + 1, b))
      }
      generalFunc
  }
  //create sumInt sumCubes
  def combineAdd(a:Int, b:Int) = a+b
//  def sumInts(a:Int, b:Int) = generalProduct((x:Int)=>x, combineAdd, 0)//获得的是函
数，ab 成为无意义参数
  def sumInts = generalProduct((x:Int)=>x, combineAdd, 0)
  def productInts = generalProduct((x:Int) => x, (a:Int, b:Int) => a*b, 1 )
```

```
println(sumInts(1, 10))
println(productInts(1, 10))
```

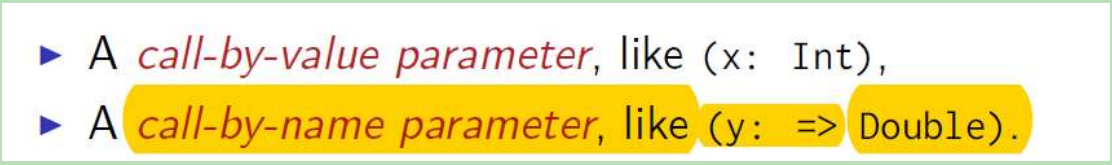
class 4

EBNF

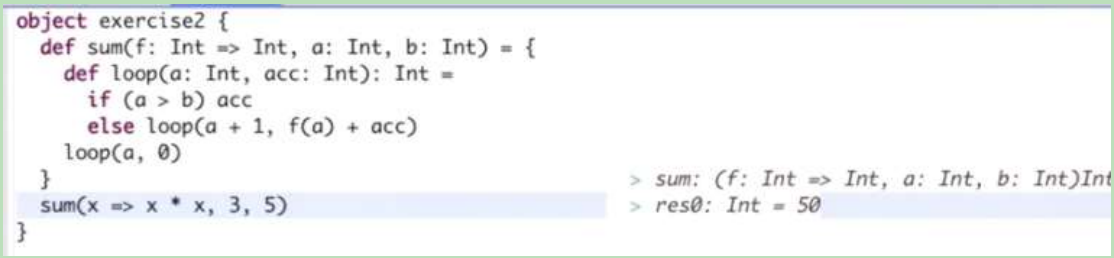
L2.4 is a great example in standard expression of language syntax, but it would be a bit comfound. Below, we give their context-free syntax in Extended Backus-Naur
form (EBNF), where
| denotes an alternative,
[...] an option (0 or 1),
{...} a repetition (0 or more).

Syntax

???



Tail recursive function



Class 6

Client view and data abstraction

Clients observe exactly the same behavior in each case.

Rig

This ability to choose different implementations of the data without affecting clients is called data abstraction.
It is a cornerstone of software engineering.

This and that identifying object

def less(that: Rational) =
numer * that.denom < that.numer * denom

equivalent way to formulate less as follow:
def less(that: Rational) =
this.numer * that.denom < that.numer * this.denom

require and assert :for catching exception

```
class Rational(x: Int, y: Int) {
require(y > 0, ”denominator must be positive”)
...
}
```

If the condition passed to require is false, an IllegalArgumentException is thrown with the given message string.

```
val x = sqrt(y)
assert(x >= 0)
```

Like require, a failing assert will also throw an exception, but it’s a different one: **AssertionError** for assert, **IllegalArgumentException** for require.

This reflects a difference in intent:

1. Require is used to enforce a precondition on the caller of a function.
2. Assert is used as to check the code of the function itself.

Primary constructor

- In Scala, a class implicitly introduces a constructor. This one is
- called the *primary constructor* of the class.
- The primary constructor
- ▸ takes the parameters of the class
- ▸ and executes all statements in the class body (such as the
- require a couple of slides back).

Rig

Auxiliary constructors: like reload constructors in Cpp

Scala also allows the declaration of auxiliary constructors.
These are methods named this
Example :
Adding an auxiliary constructor to the class Rational.
class Rational(x: Int, y: Int) {
def this(x: Int) = this(x, 1)
...
}
new Rational(2) > 2/1

Class and object

The modifier override declares that toString redefines a method that already exists(in the class java.lang.Object)

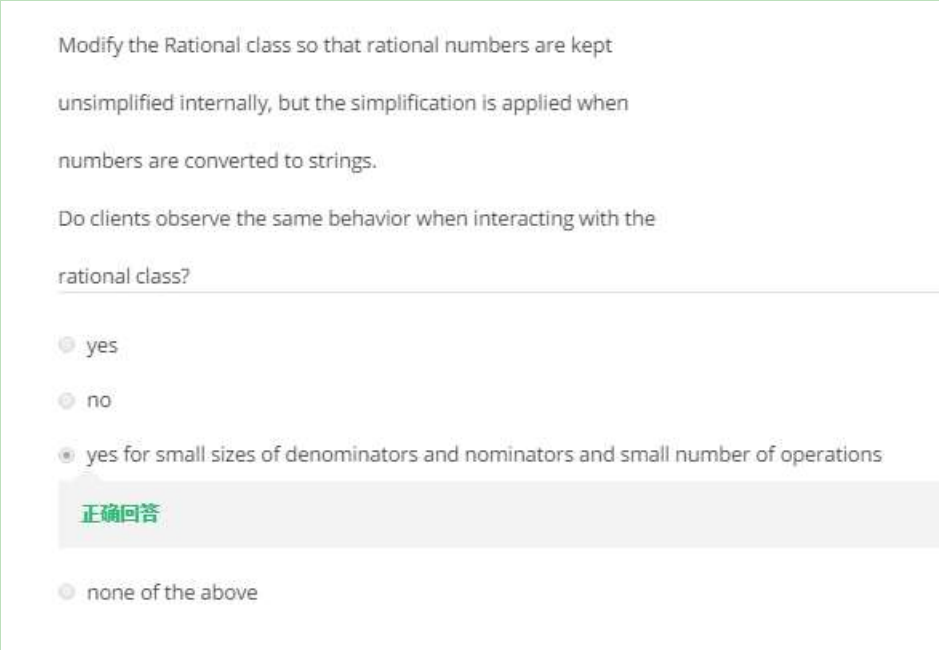
```
class Rational(a:Int, b:Int){
  def numer = a
  def denom = b
  def add(r:Rational): Rational =
    new Rational(numer*r.denom + denom*r.numer, denom*r.denom)
  override def toString = numer + "/" + denom
}



  def reduceRational(a:Rational, b:Rational): Rational =
    new Rational(a.numer*b.denom - a.denom*b.numer, a.denom*b.denom)
  def multipleRational(a:Rational, b:Rational): Rational =
    new Rational(a.numer*b.numer, a.denom*b.denom)
  def devideRational(a:Rational, b:Rational): Rational =
    new Rational(a.numer*b.denom, a.denom*b.numer)
  def isEqualRational(a:Rational, b:Rational) = (a.numer*b.denom == a.denom*b.numer)

  val x = new Rational(1, 2)
  val y = new Rational(3, 4)
  x.numer
  x.denom
  println(x.add(y) + "\n" + multipleRational(x, y))
```

Class example:Rational

```
class Rational(a:Int, b:Int){
  private def gcd(x:Int, y:Int): Int = if(y==0) x else gcd (x, x%y)
  private val g = gcd(a, b)
    def numer = if(b<0) -a else a
    def denom = if (b<0) -b else b


    def add(r:Rational): Rational =
      new Rational(numer*r.denom + denom*r.numer, denom*r.denom)
    override def toString = numer + "/" + denom
    def neg:Rational = new Rational(-numer, denom)// take no parameters
  }



    def reduceRational(a:Rational, b:Rational): Rational =
      new Rational(a.numer*b.denom - a.denom*b.numer, a.denom*b.denom)
    def multipleRational(a:Rational, b:Rational): Rational =
      new Rational(a.numer*b.numer, a.denom*b.denom)
    def devideRational(a:Rational, b:Rational): Rational =
      new Rational(a.numer*b.denom, a.denom*b.numer)
    def isEqualRational(a:Rational, b:Rational) = (a.numer*b.denom == a.denom*b.numer)

    val x = new Rational(1, -2)
    val y = new Rational(-3, 4)
    x.numer
    x.denom
    println(x.add(y) + "\n" + multipleRational(x, y) + "\n" + x.neg)
```

Denominator exception (Like a reload)

```
class Rational(x: Int, y: Int) {
  require(y != 0, "denominator must be nonzero")

  def this(x: Int) = this(x, 1)
```



And, so we might exceed the maximal number for an integer which is a bit more than two billion. For that reason, it actually, it's actually better to always normalize numbers as early as possible because that means that we **can perform more computations without running into arithmetic overflows.**

Class 7

Evaluation procedure (求值过程)

How is an instantiation of the class new C(e1*; :::;* em) evaluted?

Answer: The expression arguments e1*; :::;* em are evaluated like the arguments of a normal function. That's it.
The resulting expression, say, new C(v1*; :::;* vm), is already a value.
Class C(x1,…xm) { …def f(y1,…yn) = b … }
We may use :

New C(v1,… vm).f(w1,.. wm)

How about the evaluation processing:

Answer: The expression $\texttt{new C}(v_1, \ldots, v_m).\texttt{f}(w_1, \ldots, w_n)$ is rewritten to:

$$[w_1/y_1, \ldots, w_n/y_n][v_1/x_1, \ldots, v_m/x_m][\texttt{new C}(v_1, \ldots, v_m)/\texttt{this}]\,\texttt{b}$$

There are three substitutions at work here:

- the substitution of the formal parameters $y_1, \ldots, y_n$ of the function $\texttt{f}$ by the arguments $w_1, \ldots, w_n$,
- the substitution of the formal parameters $x_1, \ldots, x_m$ of the class $\texttt{C}$ by the class arguments $v_1, \ldots, v_m$,
- the substitution of the self reference *this* by the value of the object $\texttt{new C}(v_1, \ldots, v_n)$.

class C (x1,...,xm) {

Here are two example:

1. [] in the middle means nothing to process with “numer”;
2. Use this and that to represent created objects.

```
new Rational(1, 2).numer
→ [1/x, 2/y] [] [new Rational(1,2)/this] x
= 1

new Rational(1, 2).less(new Rational(2, 3))
→ [1/x, 2/y] [newRational(2,3)/that] [new Rational(1,2)/this]
    this.numer * that.denom < that.numer * this.denom
= new Rational(1, 2).numer * new Rational(2, 3).denom <
    new Rational(2, 3).numer * new Rational(1, 2).denom
↠ 1 * 3 < 2 * 2
↠ true
```

Infix notation

All method with a parameter can be used like a infix operator.

It is said that we can write calculation like the left ways in place of right ones:

```
r add s                                    r.add(s)
r less s          /* in place of */        r.less(s)
r max s                                    r.max(s)
```

Scala advantage: relaxed identifiers

Operators can be used as identifiers.

Thus, an identifier can be:

- *Alphanumeric*: starting with a letter, followed by a sequence of letters or numbers
- *Symbolic*: starting with an operator symbol, followed by other operator symbols.
- The underscore character '_' counts as a letter.(**all the operators have a fixed precedence level identified by its start symbol)**
- Alphanumeric identifiers can also end in an underscore, followed by some operator symbols.

Examples of identifiers:

x1 * +?%& vector_++ counter_=

```
  val x = new Rational(1, 3)                    > x : Rational = 1/3
  val y = new Rational(5, 7)                    > y : Rational = 5/7
  val z = new Rational(3, 2)                    > z : Rational = 3/2
  x.numer                                       > res0: Int = 1
  x.denom                                       > res1: Int = 3
  x - y - z                                     > res2: Rational = -79/42
  y + y                                         > res3: Rational = 10/7
  x < y                                         > res4: Boolean = true
  x max y                                       > res5: Rational = 5/7
  new Rational(2)                               > res6: Rational = 2/1
}

class Rational(x: Int, y: Int) {
  require(y != 0, "denominator must be nonzero")

  def this(x: Int) = this(x, 1)

  private def gcd(a: Int, b: Int): Int = if (b == 0) a else gcd(b, a % b)
  private val g = gcd(x, y)
  def numer = x / g
```

```
  def numer = x / g
  def denom = y / g

  def < (that: Rational) = numer * that.denom < that.numer * denom

  def max(that: Rational) = if (this < that) that else this

  def + (that: Rational) =
    new Rational(
      numer * that.denom + that.numer * denom,
      denom * that.denom)

  def neg: Rational = new Rational(-numer, denom)

  def - (that: Rational) = this + that.neg

  override def toString = numer + "/" + denom
```

Prefix operator

We might want to replace neg with a prefix minus. If you look at the error message, “-that” is a kind of unary underscore minus, which is different from the infix minus operators and not a member of rational.
Solution: we have to call it unary minus.

```
def unary_- : Rational = new Rational(-numer, denom)
```

Here we need to use a space to separate – and :, otherwise colon will become a part of identifier and it turns out compile error.

Precedence order of operator

(all letters) **the lowest precedence is every operator that starts with a letter**
|
^
&
< >
= !
:
+ -
* / %
(all other special characters)

Full parenthized version： precedence order

Provide a fully parenthized version of

```
a + b ^? (c ?^ d) less a ==> b | c
```

Every binary operation needs to be put into parentheses, but the structure of the expression should not change.

Solution： start from the highest priority operator, and use parenthesis to include the right

```
((a + b) ^? (c ?^ d)) less ((a ==> b) | c)
```

Attention: since in the precedence list , operator starts with ^ has a lower priority than operator

Rig

starts with ?

*Class Rational

```
class Rational(a:Int, b:Int){
   require(b != 0, "denominator must be nonzero")
   private def gcd(x:Int, y:Int): Int = if(y==0) x else gcd (x, x%y)
   private val g = gcd(a, b)
   def numer = if(b<0) -a else a
   def denom = if (b<0) -b else b


   def + (r:Rational): Rational =
      new Rational(numer*r.denom + denom*r.numer, denom*r.denom)
   override def toString = numer + "/" + denom
   def neg:Rational = new Rational(-numer, denom)// take no parameters
   def - (that:Rational): Rational = {
      new Rational(this.numer*that.denom - this.denom*that.denom, this.denom*that.denom)
   }
   def / (that:Rational): Rational = {
      new Rational(this.numer*that.denom, this.denom*that.numer)
   }
   def * (that:Rational): Rational = {
      new Rational(this.numer*that.numer, this.denom*that.denom)
   }
   def unary_- : Rational = new Rational(-numer, denom)

}
   val a = new Rational(1, 2);
   val b = new Rational(3, 4);
   println(a)
   println(b)
   println(a, b)
   println(a+b, a-b, a*b,"hahaha", a/b, -a)
```

homework

Code dealing with Set

```
package funsets


/**
 * 2. Purely Functional Sets.
 */
object FunSets {
  /**
   * We represent a set by its characteristic function, i.e.
   * its `contains` predicate.
   */
  type Set = Int => Boolean//look like this this a type

  /**
   * Indicates whether a set contains a given element.
   */
  def contains(s: Set, elem: Int): Boolean = s(elem)

  /**
   * Returns the set of the one given element.
   */
    def singletonSet(elem: Int): Set = x => (x == elem)//Set already regulate the type
of x is Int!!! this is a way to create a set " x => (decision expression)"


  /**
   * Returns the union of the two given sets,
   * the sets of all elements that are in either `s` or `t`.
   */
    def union(s: Set, t: Set): Set = x=>(contains(s, x) || contains(t, x))

  /**
   * Returns the intersection of the two given sets,
   * the set of all elements that are both in `s` and `t`.
   */
    def intersect(s: Set, t: Set): Set = x => (contains(s, x) && contains(t, x))

  /**
   * Returns the difference of the two given sets,
```

```
  * the set of all elements of `s` that are not in `t`.
  */
   def diff(s: Set, t: Set): Set = x => (contains(s, x) && !contains(t, x))

  /**
  * Returns the subset of `s` for which `p` holds.
  */
   def filter(s: Set, p: Int => Boolean): Set =
//    x=> (!contains(s, x) | contains(intersect(s, p), x)) !!!
  x => contains(s, x) && p(x)


  /**
  * The bounds for `forall` and `exists` are +/- 1000.
  */
  val bound = 1000

  /**
  * Returns whether all bounded integers within `s` satisfy `p`.
  */
   def forall(s: Set, p: Int => Boolean): Boolean = {
//      val useSet = filter(s, p)//! Set is a val???!!!
    def iter(a: Int): Boolean = {
      if (a < -bound) true
      else if (contains(s, a)) p(a) && iter(a - 1)
      else iter(a - 1)
    }
    iter(bound)
  }

  /**
  * Returns whether there exists a bounded integer within `s`
  * that satisfies `p`.
  */
   def exists(s: Set, p: Int => Boolean): Boolean = {
//      val useSet = filter(s, p)
      def iter(a:Int) : Boolean = {
        if (a < -bound) false
        else if (contains(s, a)) p(a) || iter(a - 1)
        else iter(a - 1)
      }
      iter(bound)
    }
```

```
  /**
   * Returns a set transformed by applying `f` to each element of `s`.
   */
    def map(s: Set, f: Int => Int): Set = x => exists(s, y => x == f(y))

  /**
   * Displays the contents of a set
   */
  def toString(s: Set): String = {
    val xs = for (i <- -bound to bound if contains(s, i)) yield i
    xs.mkString("{", ",", "}")
  }

  /**
   * Prints the contents of a set on the console.
   */
  def printSet(s: Set) {
    println(toString(s))
  }
}
```

week3

class1

abstract class

abstract class IntSet {
def incl(x: Int): IntSet
def contains(x: Int): Boolean
}
If try

Object insets{
  New IntSet;
}

It will fail
We can’t instantiate an abstract class

We need to extend the base class.

Rig

Class extension

```
class Empty extends IntSet {
def contains(x: Int): Boolean = false
def incl(x: Int): IntSet = new NonEmpty(x, new Empty, new Empty)
}
```

Empty and NonEmpty both extend the class IntSet
This implies that the types Empty and NonEmpty conform to the type IntSet
▸ an object of type Empty or NonEmpty can be used wherever an object of type IntSet is required.

Base class and subclass

The direct or indirect superclasses of a class C are called base classes of C.
So, the base classes of NonEmpty are IntSet and **Object**.

Redefine

```
Abstract calss Base{
    Def foo = 1
    Def bar:Int
}

Class sub extends Base{
    Override def foo = 2
    Def bar = 3
}
```

You can't def the abstract variable by override.it needs a new definition.
Bar can either use override before def or not, but foo need to use override

Difference between class Empty and object Empty

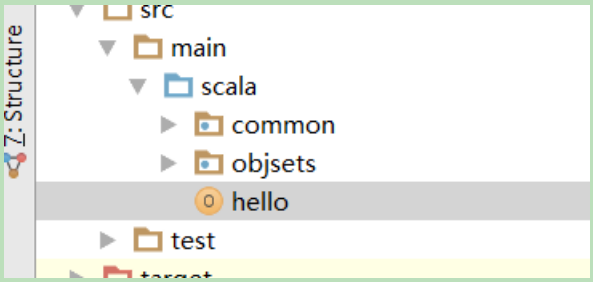
object Empty extends IntSet {
def contains(x: Int): Boolean = false
def incl(x: Int): IntSet = new NonEmpty(x, Empty, Empty)
}

So this object definition is exactly the same as the class definition, except that instead of the keyword class, you use object. And that will define a singleton object named Empty. There is only one of them, and you **don't need or can create that explicitly with new.** So the object Empty simply exists. Technically it will be created. The first time you reference it.

**Standalone program

可以在 object 内写完，直接按 ctrl+shift+f10 久运动的
But it is also possible to create standalone applications in Scala.
Each such application contains an object with a main method.
For instance, here is the “Hello World!” program in Scala.
object Hello {
def main(args: Array[String]) = println(”hello world!”)
}
Once this program is compiled, you can start it from the command line with
> scala Hello

Create a object under scala directory



Ctrl + shift + f10 to run this single object.

Dynamic method dispatch

```
class NonEmpty(elem: Int, left: IntSet, right: IntSet) extends IntSet {
def contains(x: Int): Boolean =
if (x < elem) left contains x
else if (x > elem) right contains x
else true
def incl(x: Int): IntSet =
if (x < elem) new NonEmpty(elem, left incl x, right)
else if (x > elem) new NonEmpty(elem, left, right incl x)
else this
}
```

Empty contains 1
! [1/x] [Empty/this] false
= false

(new NonEmpty(7, Empty, Empty)) contains 7
! [7/elem] [7/x] [new NonEmpty(7; Empty; Empty)/this]
if (x < elem) this.left contains x
else if (x > elem) this.right contains x else true
= if (7 < 7) new NonEmpty(7, Empty, Empty).left contains 7
else if (7 > 7) new NonEmpty(7, Empty, Empty).right
contains 7 else true
! true

This means that the code invoked by a method call depends on the runtime type of the object that contains the method.

Something to Ponder？？？

Dynamic dispatch of methods is analogous to calls to higher-order functions.
Question:
Can we implement one concept in terms of the other?
- Objects in terms of higher-order functions?
- Higher-order functions in terms of objects?

Class 2

**Package

Classes and objects are organized in packages.

To place a class or object inside a package, use a package clause at the top of your source file.
package progfun.examples
object Hello { ... }
This would place Hello in the package progfun.examples.

> scala progfun.examples.Hello

Import

Imports come in several forms:
```
import week3.Rational // imports just Rational
import week3.{Rational, Hello} // imports both Rational and Hello
import week3._ // imports everything in package week3
```
The first two forms are called *named imports*.
The last form is called a *wildcard import*.
You can import from either a package or an object.

Or you just write the method name and press “alt + enter”

*Automatic imports

Some entities are automatically imported in any Scala program.
These are:
- All members of package scala
- All members of package java.lang
- All members of the singleton object scala.Predef.

Here are the fully qualified names of some types and functions which you have seen so far:
Int scala.Int
Boolean scala.Boolean
Object java.lang.Object

Rig

require scala.Predef.require
assert scala.Predef.assert

learn more about scala library

Scaladoc
You can explore the standard Scala library using the scaladoc web pages.
You can start at
www.scala-lang.org/api/current

implement several supertypes

In Java, as well as in Scala, a class can only have one superclass.
A trait is declared like an abstract class, just with trait instead of abstract class.

```
trait Planar {
def height: Int
def width: Int
def surface = height * width
}
```

Classes, objects and traits can inherit from at most one class but Arbitrary many traits.
Example:

```
class Square extends Shape with Planar with Movable ...
```

value type unit

http://outofmemory.cn/scala/data-types

Byte	8bit 的有符号数字，范围在-128 -- 127
Short	16 bit 有符号数字，范围在-32768 -- 32767
Int	32 bit 有符号数字，范围 -2147483648 到 2147483647

Rig

Long	64 bit 有符号数字，范围-9223372036854775808 到 9223372036854775807
Float	32 bit IEEE 754 单精度浮点数
Double	64 bit IEEE 754 双精度浮点数
Char	16 bit Unicode 字符. 范围 U+0000 到 U+FFFF
String	字符串
Boolean	布尔类型
Unit	表示无值，和其他语言中 void 等同
Null	空值或者空引用
Nothing	所有其他类型的字类型，表示没有值
Any	所有类型的超类，任何实例都属于 Any 类型
AnyRef	所有引用类型的超类

上表中列出的数据类型都是对象，也就是说 scala 没有 java 中的原生类型。在 scala 是可以对数字等基础类型调用方法的。

scala 基础数据类型的表示方式

整数类型的表示方式：

0
035
21
0xFFFFFFFF
0777L
如果未指定类型，通常数字被解析成了 Int 类型，如果表示 Long，可以在数字后面添加 L 或者小写 l 作为后缀。

浮点数的表示方式

0.0
1e30f
3.14159f
1.0e100
.0
如果浮点数后面有 f 或者 F 后缀时，表示这是一个 Float 类型，否则就是一个 Double 类型的。

Scala 的 Boolean 类型表达为 true 和 false

字符类型

在 scala 中字符类型表示为：半角单引号括住的字符，如下

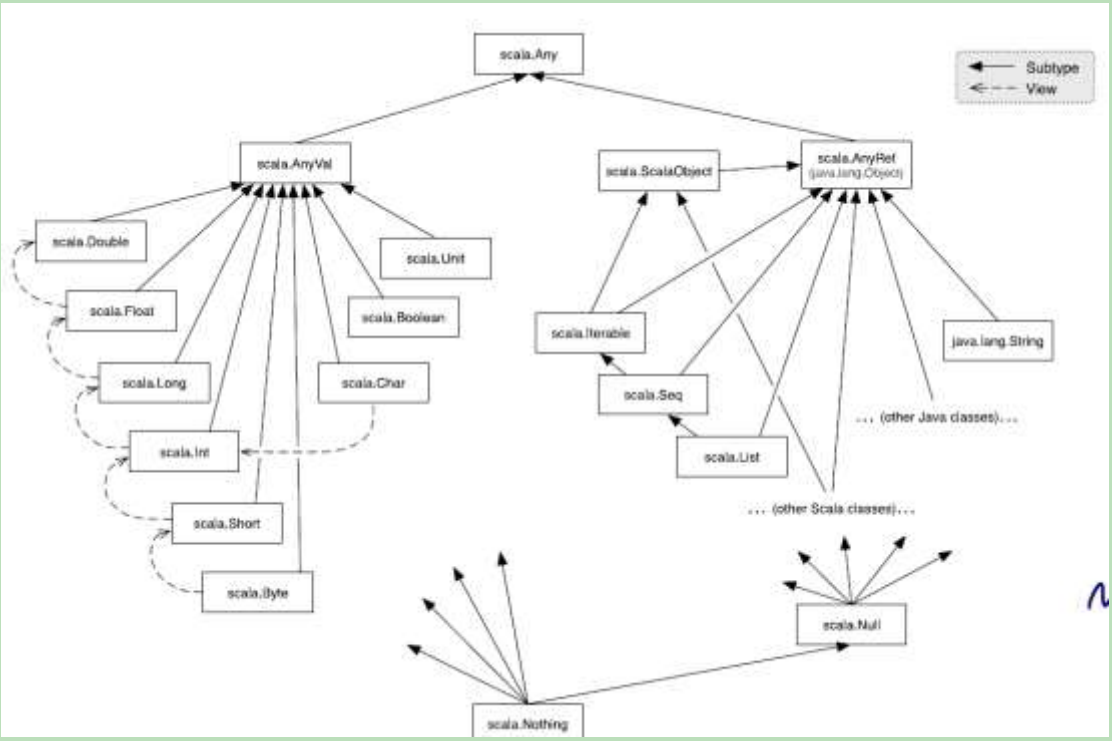
'a'
'\u0041'
'\n'
'\t'
其中\表示转移字符，其后可以跟 u0041 数字或者\r\n 等固定的转移字符

字符串表示方法

在 scala 中单行字符串和 java 中的表达方式完全一样，例如"Hi, OutOfMemory.CN",另外 scala 中还有类似于 python 的多行字符串表示方式，用三个双引号表示分隔符，如下：

val s="""这是一个多行字符串
这是第二行
这是第三行"""
scala 的 Null 表示为 null，例如 val n = null

*scala's class Hierarchy diagram



Rig

in this diagram, we see the relationships between the numeric types, they are actually written with dotted arrows, not hard arrows, and that means that the numeric types, they're not strictly in a subtype relationship with each other, but **in a conversion relationship so you can automatically convert and type such as Long to Float,** but it's not really a subtype.

Top Types

At the top of the type hierarchy we find:
Any :the base type of all types
Methods: ‘==‘, ‘!=‘, ‘equals‘, ‘hashCode, ‘toString‘
AnyRef :The base type of all reference types;
It has an Alias of ‘java.lang.Object‘
AnyVal : The base type of all primitive types.

Nothing type

Nothing is at the bottom of Scala’s type hierarchy. **It is a subtype of every other type.**
There is no value of type Nothing.
Why is that useful?
- To signal abnormal termination
- As an element type of empty collections (see next session)

Exceptions

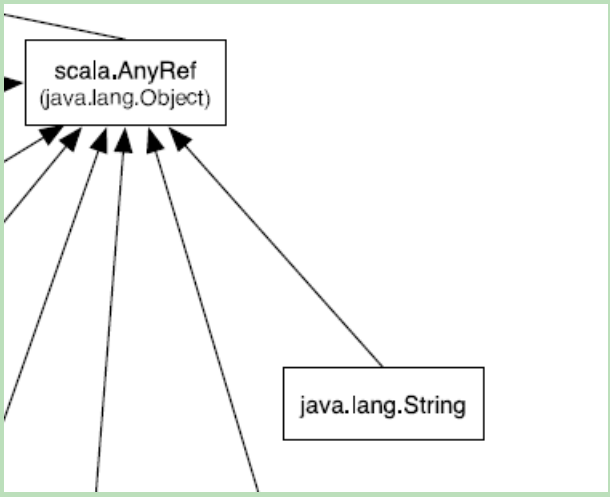
Scala’s exception handling is similar to Java’s.
The expression
throw Exc
aborts evaluation with the exception Exc.
The type of this expression is Nothing.

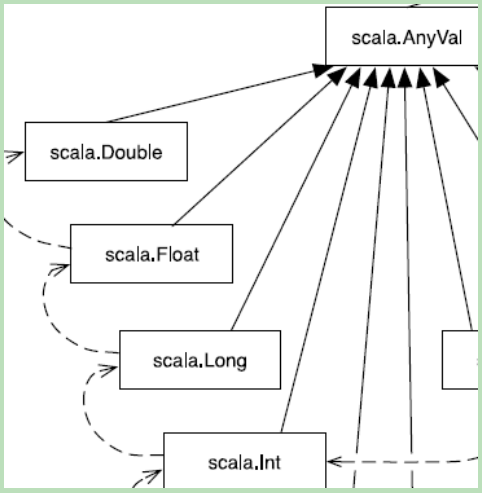
*The Null Type

Every reference class type also **has null as a value.**
The type of null is Null.
Null is a subtype of every class that inherits from Object; it is incompatible with subtypes of AnyVal.
val x = null // x: Null
val y: String = null // y: String
val z: Int = null // error: type mismatch

null is a sub type only of the reference types, not of the value types.(refer to the class hierarchy diagram)

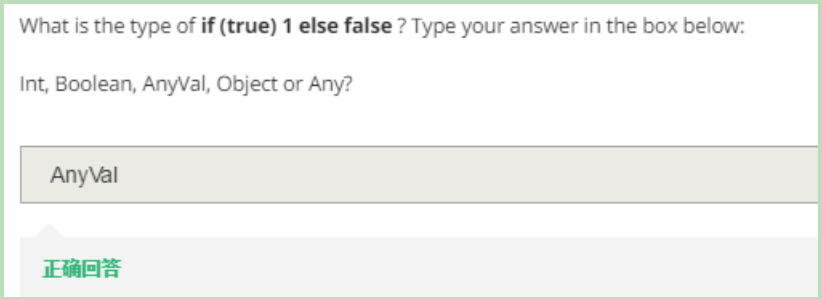
String is a reference types. Int is a val type





QUIZ

So you've seen that null is a sub type only of the reference types, not of the value types



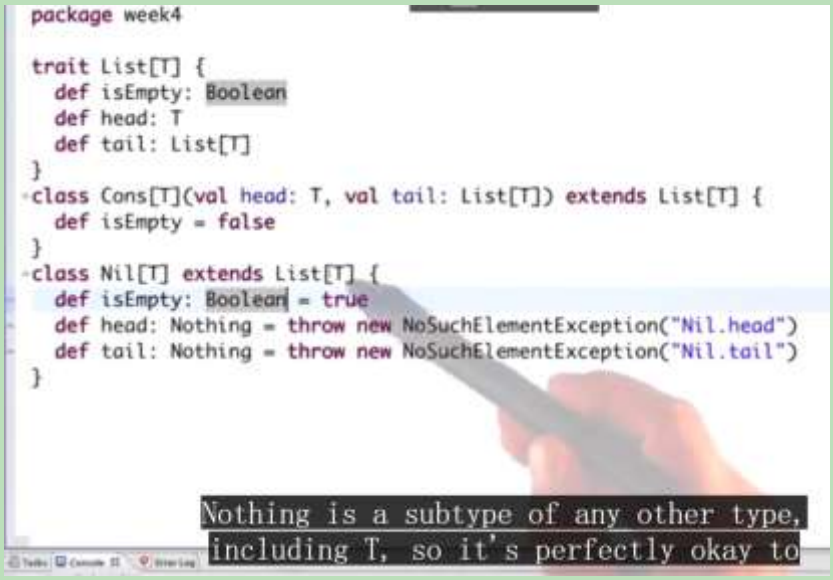
Reason:
e. So we see that we have Int here and we have Boolean here, but both of these values are also values of the supertype. So that will be either AnyVal or Any. And in this case AnyVal is better. **It conveys more information than Any.** So the type check actually picked the type that matched both of these values, AnyVal in that case, and that was as specific as possible. So that's why the type checker answered with AnyVal.

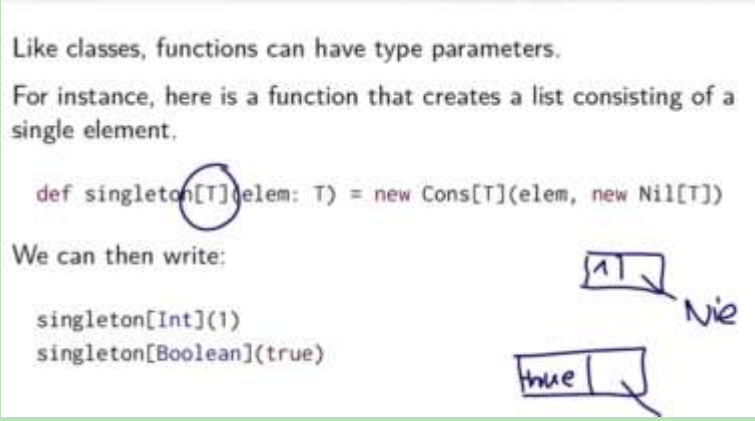
Class 3

Cons-lists---immutable list

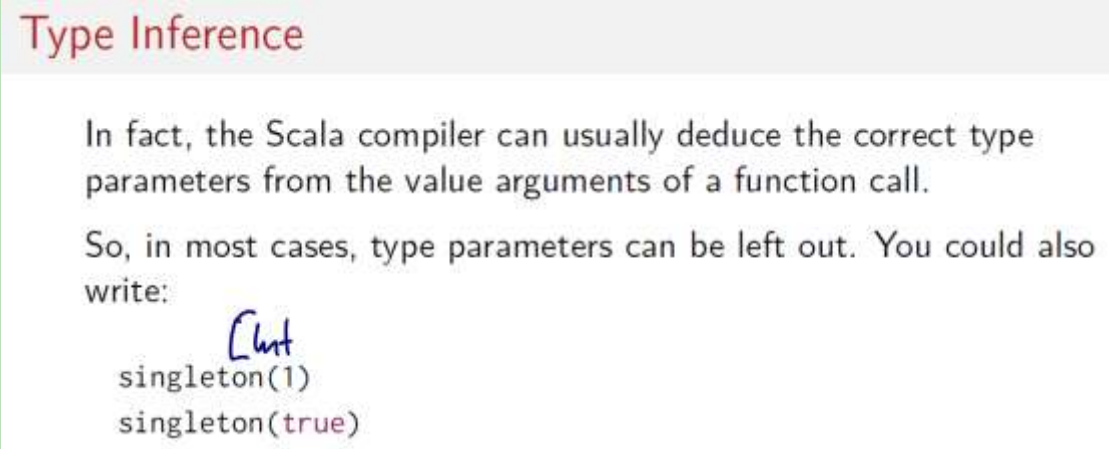
a list which is immutable

nothing is a subtype of any types where we will throw a new exception



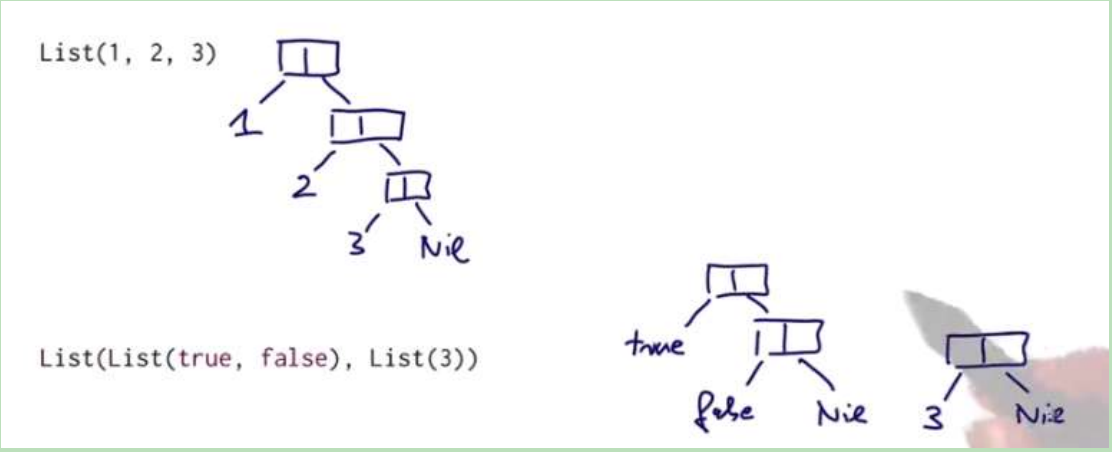


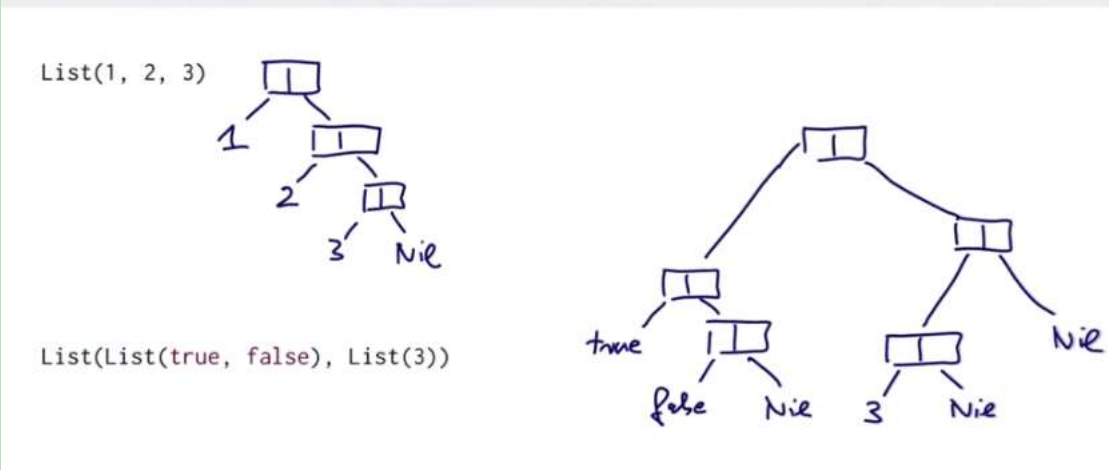
Now we can use type inference, because the complier can know the data type from elem



Rig

Union

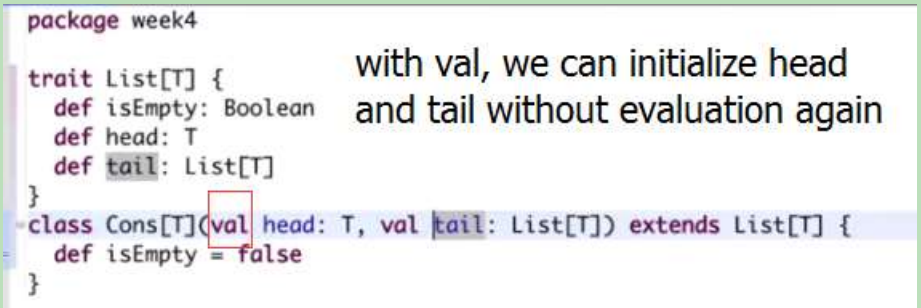




Val and def

The difference between eval and the def concerns only the initialization. Eval is evaluated when the object is first initialized. Where, whereas a def is evaluated each time it is referenced.

we have to put the tail value here in front because otherwise, it wouldn't count as a field, and it wouldn't implement.



Great explanation of “val” prefix In the concept of scope
http://stackoverflow.com/questions/15639078/scala-class-constructor-parameters

Rig

Type erasure

If the type can be inferred correctly like we do in high-order function.

Type Inference

In fact, the Scala compiler can usually deduce the correct type parameters from the value arguments of a function call.

So, in most cases, type parameters can be left out. You could also write:

```
singleton(1)
singleton(true)
```

This means types are only important for the complier to verify that this program satisfies certain correctness properties but they are not relevant for the actual execution.

**Polymorphism like template in C???!!!

Polymorphism

Polymorphism means that a function type comes "in many forms".

In programming it means that

- the function can be applied to arguments of many types, or
- the type can have instances of many types.

We have seen two principal forms of polymorphism:

- subtyping: instances of a subclass can be passed to a base class
- generics: instances of a function or class are created by type parameterization.

The running example for this week is a data structure which is fundamental for many functional languages. That's the immutable list that's built from individual consoles.

Oh, I just forgot, we have to put the tail value here in front because otherwise, it wouldn't count as a field, and it wouldn't implement. So what that shows is that in, Scala val definitions. So field definitions in classes are really special cases of methods. And they can override methods, and they can implement abstract methods and traits. The difference between eval and the def concerns only the initialization. Eval is evaluated when the object is first initialized. Where, whereas a def is evaluated each time it is referenced

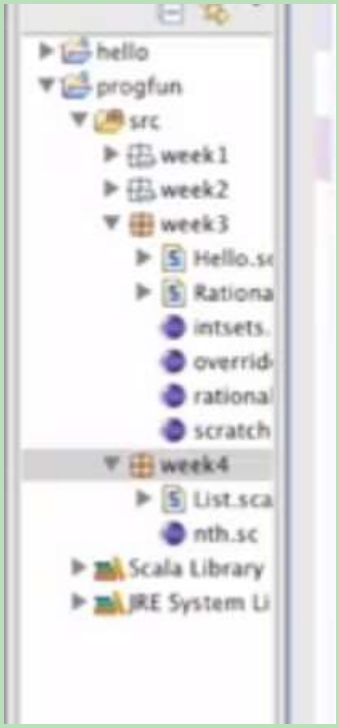
Return nothing:

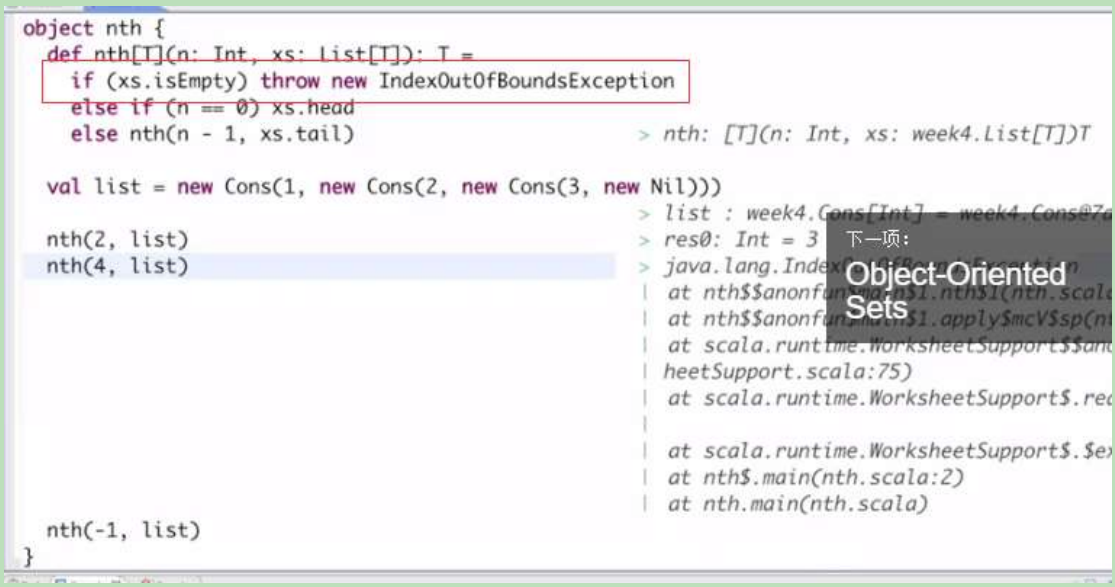
```
trait List[T] {
  def isEmpty: Boolean
  def head: T
  def tail: List[T]
}

class Cons[T](val head: T, val tail: List[T]) extends List[T] {
  def isEmpty = false
}

class Nil[T] extends List[T] {
  def isEmpty = true
  def head = throw new NoSuchElementException("Nil.head")
  def tail = throw new NoSuchElementException("Nil.tail")
}
```

so it's perfectly okay to have a head definition here that returns nothing and that can implement a head, abstract method in the trait list that returns a T, simply because a nothing is a T, a nothing is a subtype of any other type

QUIZ

bulid a worksheet





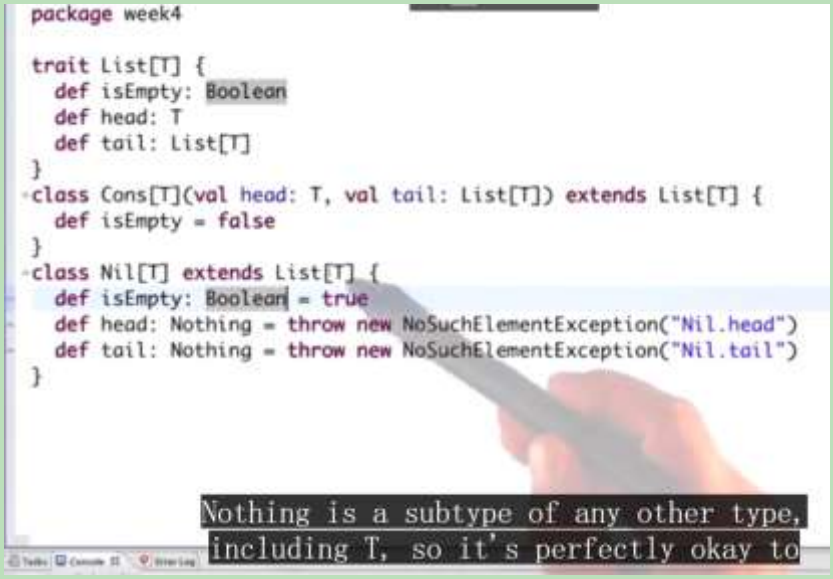
There are a efficiency problem, -1 need to go through all the elements then throw a error like 4 will do.

Abstract class VS trait

A class can only extend one superclass, and thus, only one abstract class. If you want to compose several classes the Scala way is to use mixin class composition: you combine an (optional) superclass, your own member definitions and one or more traits. A trait is restricted in comparison

to classes in that it cannot have constructor parameters (compare the scala reference manual). The restrictions of traits in comparison to classes are introduced to avoid typical problems with multiple inheritance. There are more or less complicated rules with respect to the inheritance hierarchy; it might be best to avoid a hierarchy where this actually matters. ;-) As far as I understand, it can only matter if you inherit two methods with the same signature / two variables with the same name from two different traits.

应该如下面一样再定义子类进行值的初始化



Homework

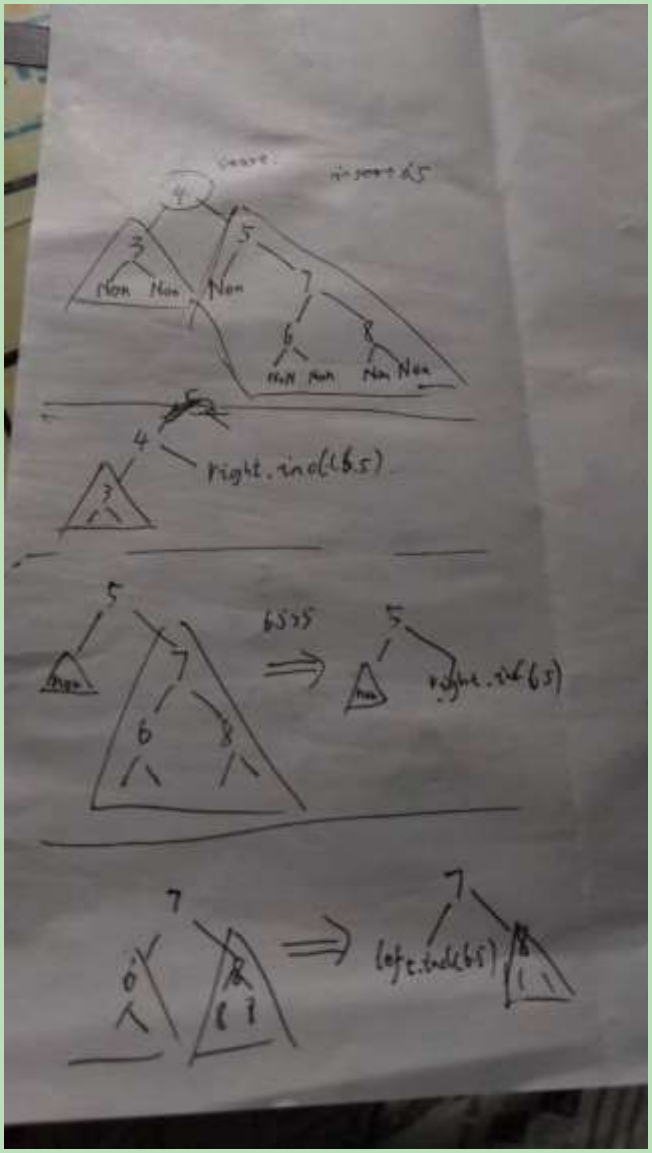
Trait 和优秀的 foreach(f: x=>y)函数

Foreach:从 list 头部运行到尾部
Unit 表示无值，和其他语言中 void 等同

```
trait TweetList {
  def head: Tweet
  def tail: TweetList
  def isEmpty: Boolean
  def foreach(f: Tweet => Unit): Unit =
    if (!isEmpty) {
      f(head)//???难道这 f 函数可以修改，传入的参数可被修改？而不是浅复制
      tail.foreach(f)
    }
}
```

nonEmpty 的 incl 函数

```
def incl(x: Tweet): TweetSet = {
  //只在 nonEmpty 的 filterAcc 和 union 函数中用到
  //是在本身为一颗完整树的情况下，构建一颗新的包含 x 的书
  if (x.text < elem.text) new NonEmpty(elem, left.incl(x), right)//每次 incl 后其实是构造出新的分支出来啊
  else if (elem.text < x.text) new NonEmpty(elem, left, right.incl(x))
  else this
}
```



filterAcc

```
def filterAcc(p: Tweet => Boolean, acc: TweetSet): TweetSet = {
  //树的每个节点有 elem:Int, 还有左右两个 tweetSet,
  //filterAcc 对每个 elem 进行检验，假设是最底层的 elemLowest，下联两个 Empty，l 和 r 都是 new Empty
  //union 后变为 noEmpty, 如果 elem 不符合 p，返回为一个 Empty，其实可以先做 p(elem) 判定，然后判定 empty，减少 new Empty 操作
  //但是仔细想每次都判断，而且很多情况下会是单个 empty 和 noEmpty 组成两个子树
  val l = left.filterAcc(p, acc)
  val r = right.filterAcc(p, acc)
  val lr = l union r
  if (p(elem)) lr.incl(elem) else lr
}
```

week4

class 1

code boolean

try this in scala， this will turn error

```
package idealize.scala
abstract class Boolean{
  def ifThenElse[T](t: => T, e: => T ): T
  def && (x: => Boolean):Boolean = ifThenElse(x, false)
  def || (x: => Boolean):Boolean = ifThenElse(true, x)
  def unary_! : Boolean          = ifThenElse(false, true)
  def == (x: Boolean):Boolean    = ifThenElse(x, x.unary_!)
  //unary_!是一个 prefix, ||\s 可以做
  def == (x:Boolean): Boolean    = ifThenElse(x, !x)
  object true extends Boolean{
    def ifThenElse[T](t: => T, e: => T) = t
  }
  object false extends Boolean{
    def ifThenElse[T](t: => T, e: => T) = e
  }
}
```

Rig

extends object or extends class should be constructed outside of abstract class.

Code National

Define national without using primitive class int , which is called peano number

```
abstract class Nat{
  def isZero:Boolean
  def predecessor: Nat
//  def sucessor : Nat
  def sucessor: Nat = new Succ(this)//because this is same for both object zero and
class succ
  def + (that: Nat) : Nat
  def - (that: Nat) : Nat

}
  object Zero extends Nat{
    def isZero = true
    def predecessor = throw new Error("zero has no predecessor")
//    def sucessor = new Succ(Zero)
    def + (that: Nat) : Nat = that
    def - (that: Nat) : Nat = {
//      if (that.isZero) Zero//Zero 是一个 ojbect,不是 class 不能用 new
      if (that.isZero) this//return this is better
      else throw new Error("it's negative")
    }
  }
  class Succ(pre : Nat) extends Nat{
    def isZero = false
//    def predecessor = {
//      if (pre.isZero) Zero
//      else (this - new Succ(Zero))
//    }
    def predecessor = pre// as simple as it can be
//    def sucessor = new Succ(this)
//    def + (that: Nat) : Nat = {
//      if (that.isZero) new Succ(this.predecessor)
//      else (new Succ(this) + that.predecessor)
//    }
    def +(that: Nat):Nat = new Succ(pre + that)

    //!!!need to let it touch zero by recursive
    // so we should use succ(this)
```

```
    override def -(that: Nat): Nat = {
      if (that.isZero) this
//      else (this.predecessor - that.predecessor)
      else new Succ(pre - that.predecessor)//here should use new??? may first way can work
    }
  }
```

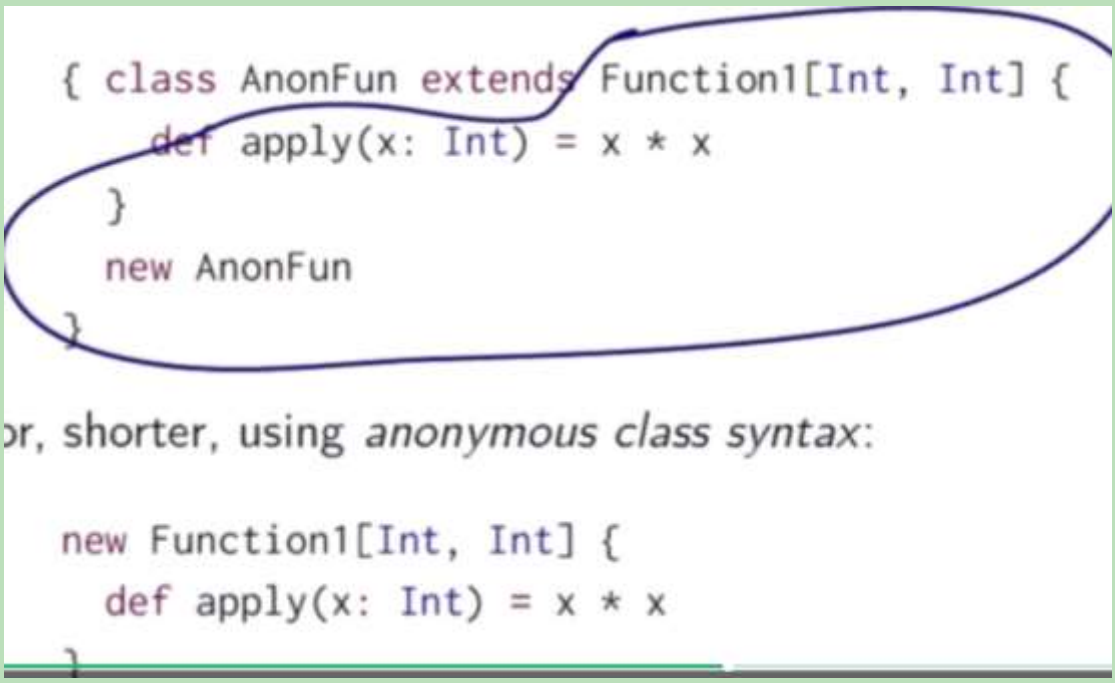
class 2

```
(x: Int) => x * x
```

is expanded to:

```
{ class AnonFun extends Function1[Int, Int] {
    def apply(x: Int) = x * x
  }
  new AnonFun
}
```

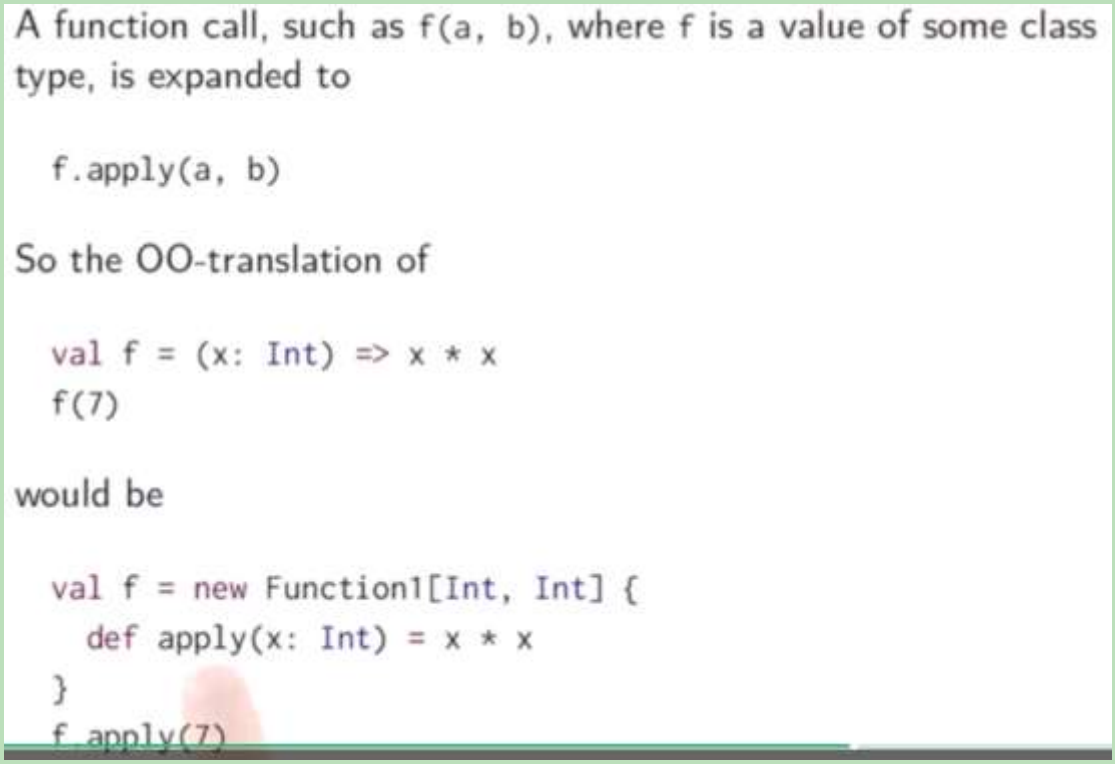
Now all that's needed to do is create an instance of that class. That would be the object that represents this function here.



So that create a new instance of a class that I haven't bothered to give a name. That anonymous class syntax you actually find in Java as well, so both Java and Scala h

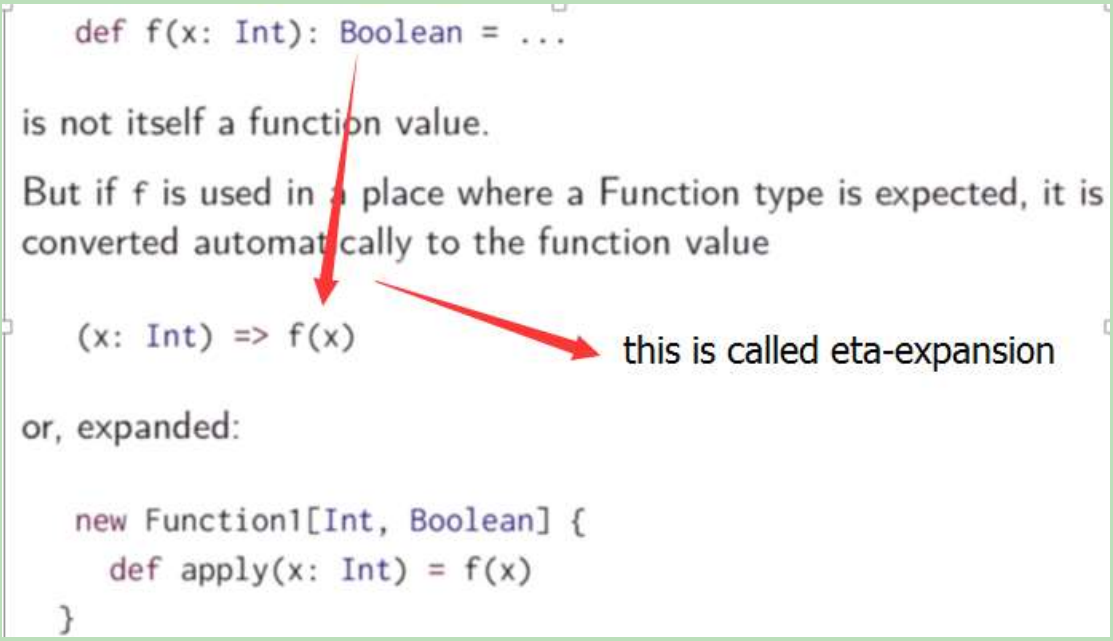
Rig

functions are objects.



Functions really are treated in Scala.

So anything that's defined with a Def are not, themselves, function values. But if the name of a method is used in a place where a function type is expected, it's converted automatically to the function value. And the conversion is just that we create an anonymous function like this one here. Which, where we say, well, give me an argument, and I apply the method to the argument. That anonymous function value we've seen how that expands to this new anonymous class. New function one in boolean with the apply method. So that's how.



So definition of function is that like a object, is is not the function value but is where we say I give a argument and you apply a method on it.

CODE List

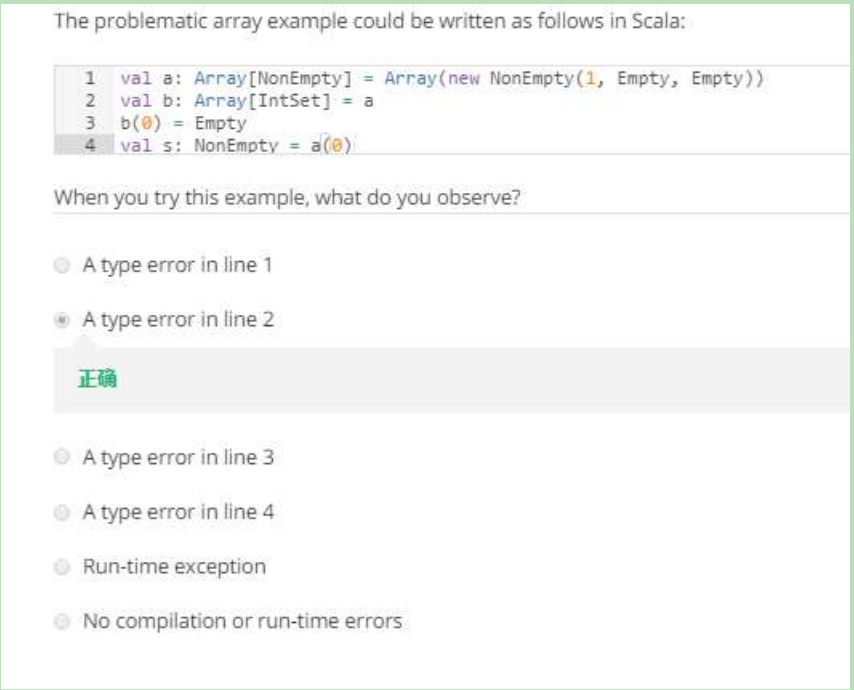
Use object definition with same name , we can define the apply method in it to create some construction method. It a syntax sugar which you can use a apply method instead of using new

```
trait List[T] {
  def isEmpty: Boolean
  def head: T
  def tail: List[T]
}
class Cons[T](val head: T, val tail : List[T]) extends List[T]{
  def isEmpty = false
}
class Nil[T] extends List[T]{
  def isEmpty = true
  def head: Nothing = throw new NoSuchElementException("Nil.head")
  def tail:Nothing = throw new NoSuchElementException("Nil.tail")
}
object List{
  //List(1,2) = List.apply(1, 2)
  def apply[T](x1 : T, x2 : T): List[T] = new Cons(x1, new Cons(x2, new Nil))
  def apply[T]() = new Nil
}
```

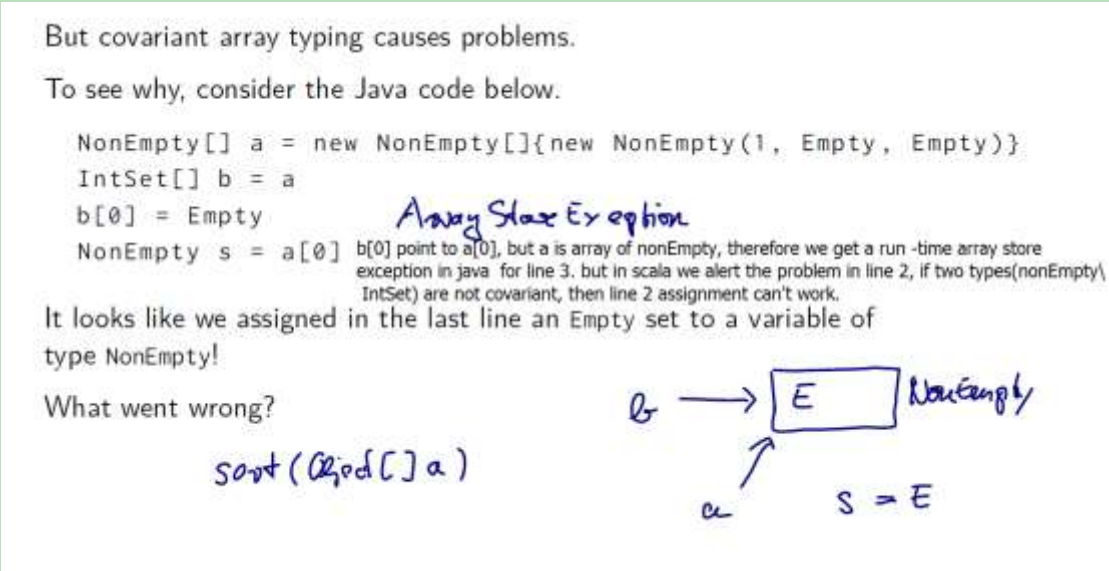
class 3

One was subtyping, which was usually associated with object oriented programming. The other was generics, which came originally from functional programming.

covariant



in Scala, arrays are not covariant. So you would not have a subtype relationship between those two arrays. And that means you will get a type error. It will say, I have found an array of nonempty, but I have expec, expected an array of IntSet.



The Liskov Substitution Principle

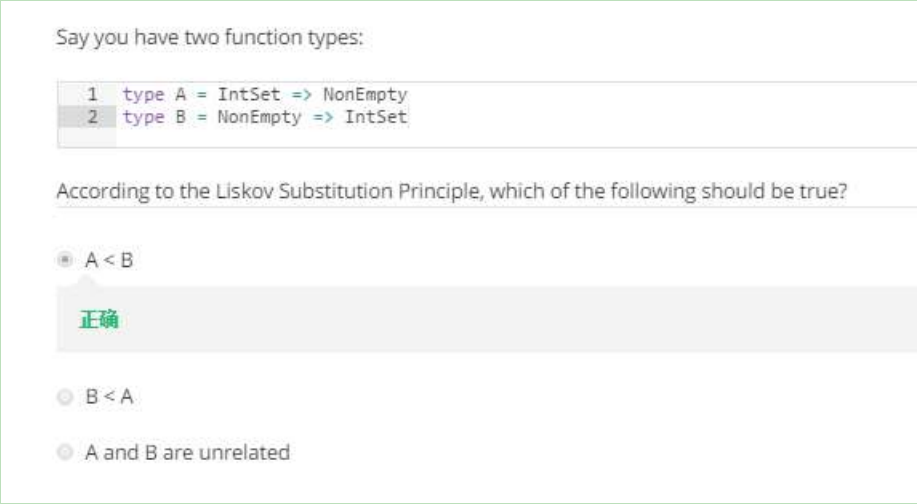
The following principle, stated by Barbara Liskov, tells us when a type can be a subtype of another.
If A <: B, *then everything one can to do with a value of type* B *one should also be able to do with a value of type* A.
[The actual definition Liskov used is a bit more formal. It says:
Let q(x) *be a property provable about objects* x *of type* B.
Then q(y) *should be provable for objects* y *of type* A *where* A <: B.
]

CODE Intset and NonEmpty

```
abstract class IntSet {
def incl(x: Int): IntSet
def contains(x: Int): Boolean
}
class Empty extends IntSet {
def contains(x: Int): Boolean = false
def incl(x: Int): IntSet = new NonEmpty(x, new Empty, new Empty)
}

class NonEmpty(elem: Int, left: IntSet, right: IntSet) extends IntSet {
def contains(x: Int): Boolean =
if (x < elem) left contains x
else if (x > elem) right contains x
else true
def incl(x: Int): IntSet =
if (x < elem) new NonEmpty(elem, left incl x, right)// any member function name can be midfix
else if (x > elem) new NonEmpty(elem, left, right incl x)
else this
}
```

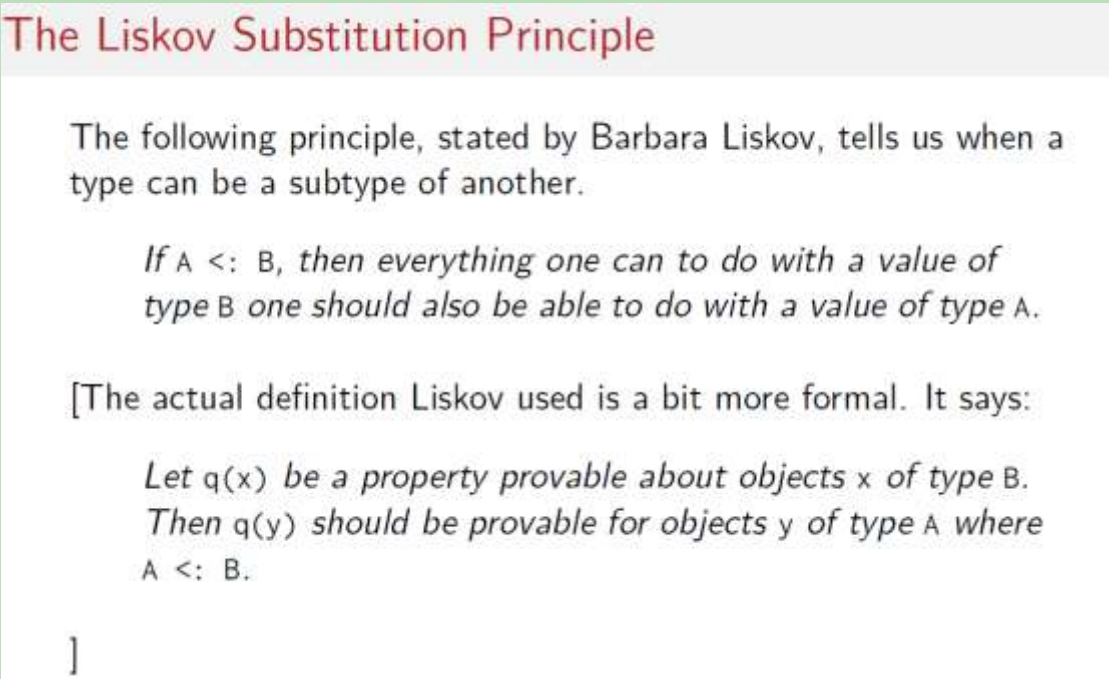
Class 4

Covariant quiz



Become Nonempty is a subclass of Intset, therefore u can successfully give a (nonempty = > intset) to type B, this can also satisfy type A, but when u give a (empty => nonempty) to tybe B, it can't work but this satisfy type A.

Because of liskov substitution principle



This means B is a superclass of A

Rig

Covariant in function

If A2 <: A1 and B1 <: B2, then

A1 => B1 <: A2 => B2

So functions are *contravariant* in their argument type(s) and *covariant* in their result type.

This leads to the following revised definition of the `Function1` trait:

```
package scala
trait Function1[-T, +U] {
  def apply(x: T): U
}
```

Covariant definition in scala

假设 A <: B，可以简单认为 A 是 B 的子类(具体定义是任何 B 满足的性质，A 都满足)

C[A] <: C[B]　　C 是 协变的 covariant

C[A] >: C[B]　　C 是 逆变的 contravariant

C[A]和 C[B]都不是彼此的子类　　C 是 非变的 nonvariant

C 是 AnyRef，即引用类，C[A]代表 C 的元素是 A 类对象，C[A]<:C[B]代表 C[B]是 C[A]基类，因此 C[B]可以赋值给 C[A]

Scala 可以用下面的方式更简单声明容器类的性质

Class C[+A] {..}　　C 是 协变的 covariant

Class C[-A] {...}　　C 是 逆变的 contravariant

Class C[A]{...}　　C 是 非变的 nonvariant

C[+A]代表了假设 A<:B，则 C[A]<:C[B]

Function trait definition in scala

So functions are *contravariant* in their argument type(s) and *covariant* in their result type.

This leads to the following revised definition of the `Function1` trait:

```
package scala
trait Function1[-T, +U] {
  def apply(x: T): U
}
```

Here we de

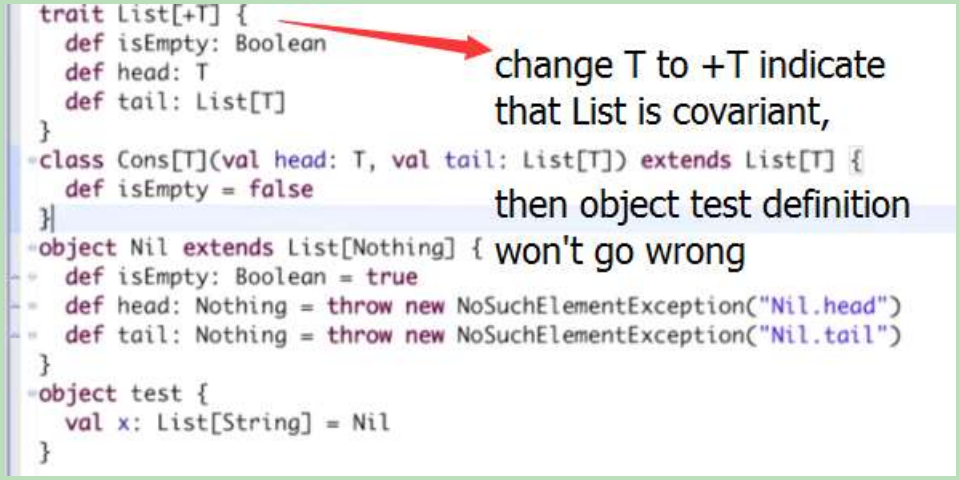
So objects can't have type parameters, because there's only a single instance of them.

```
object Nil extends List[T] {
  def isEmpty: Boolean = true
  def head: Nothing = throw new NoSuchElementException("Nil.head")
  def tail: Nothing = throw new NoSuchElementException("Nil.tail")
}
```

Then we get an error which says that the Type T here is not found. That's true because T is no longer bound as a type parameter. So we have to find another type In which we, which we want to use as the argument type of typelist

List of string = nil type checks, because nil is a list of nothing. Nothing is a subtype of string, and lists are covariants.

Rig



List 定义为[+T]，因此 A 为 B 的子类时，List[A]是 List[B]的子类。

list of non-empty cannot be a sub-type of list of IntSet.

list of non-empty cannot be a sub-type of list of IntSet. Because :

Indeed, the compiler is right to throw out `List` with `prepend`, because it violates the Liskov Substitution Principle:

Here's something one can do with a list `xs` of type `List[IntSet]`:

```
xs.prepend(Empty)
```

But the same operation on a list `ys` of type `List[NonEmpty]` would lead to a type error:

```
ys.prepend(Empty)
           ^ type mismatch
           required: NonEmpty
           found: Empty
```

So, `List[NonEmpty]` cannot be a subtype of `List[IntSet]`.

Fix the prepend

But `prepend` is a natural method to have on immutable lists!

Question: How can we make it variance-correct?

We can use a *lower bound*:

```
def prepend [U >: T] (elem: U): List[U] = new Cons(elem, this)
```

This passes variance checks, because:

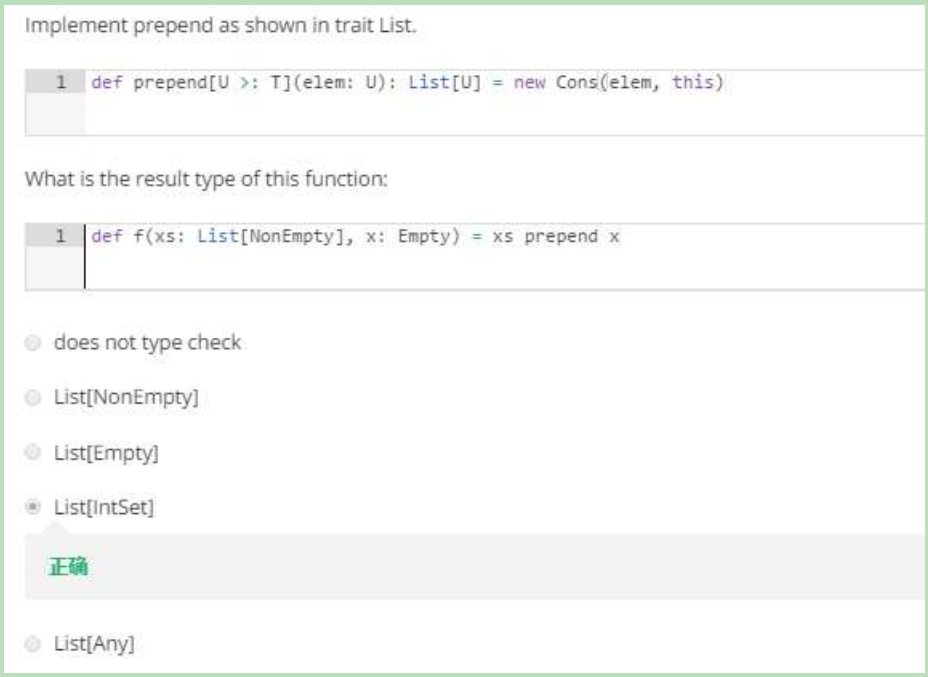
- covariant type parameters may appear in lower bounds of method type parameters
- contravariant type parameters may appear in upper bounds of method

Def perpend(elem: T):List[T] = new Cons(elem, this)
是属于 trait List[+T]的，因此假设此时 this 是一个 List[nonempty],
则 List[T]中的 T 是 nonEmpty 类，所以如果 this.prepend(empty)会导致类型错误，因为 empty

Rig

和 nonEmpty 都是 InSet 的子类，但是 empty 不是 nonEmpty 的子类，不能赋值给需要 nonEmpty 的参数位置。
Def perpend[U:>T](elem: U) :List[U] = new Cons(elem, this)
U 是 T 的基类，因此传入的元素要求是 Inset 类型，empty 是 Inset 子类当然可以传入，得到的 new Cons(empty, this)也是 List【Inset】，毫无问题

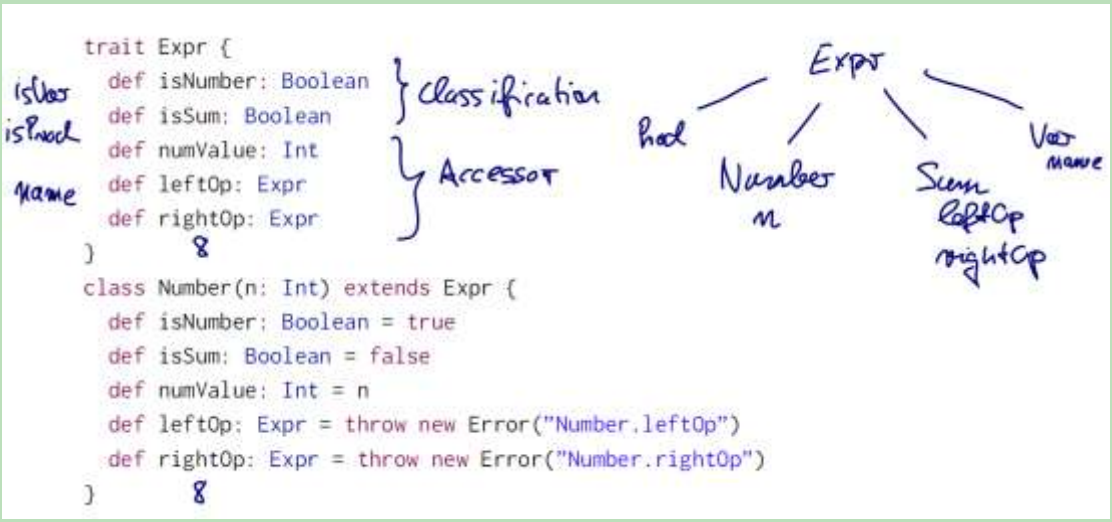
Prepend quiz

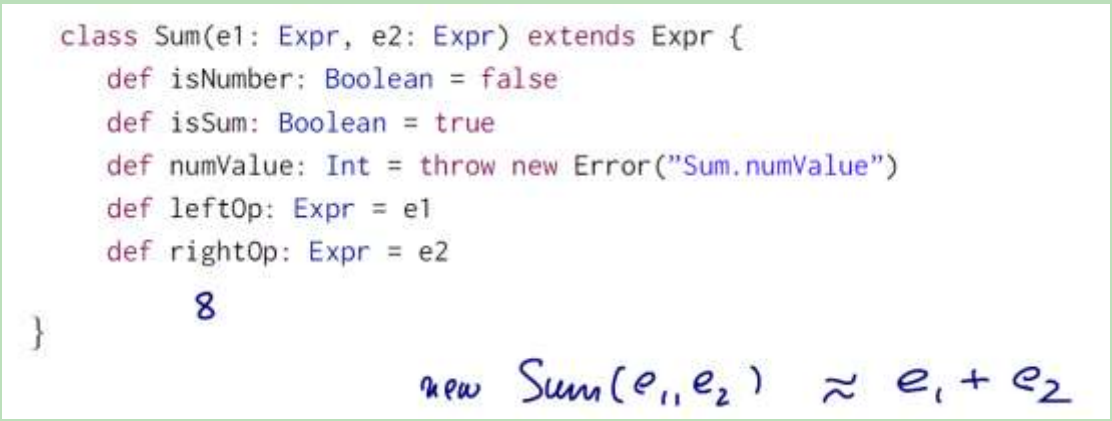


因为这里就使得传入的 T 类型(nonempty)是 U 的子类，这样传入 Empty 便不会出错

Class5 Decomposition

1.Classification and access method





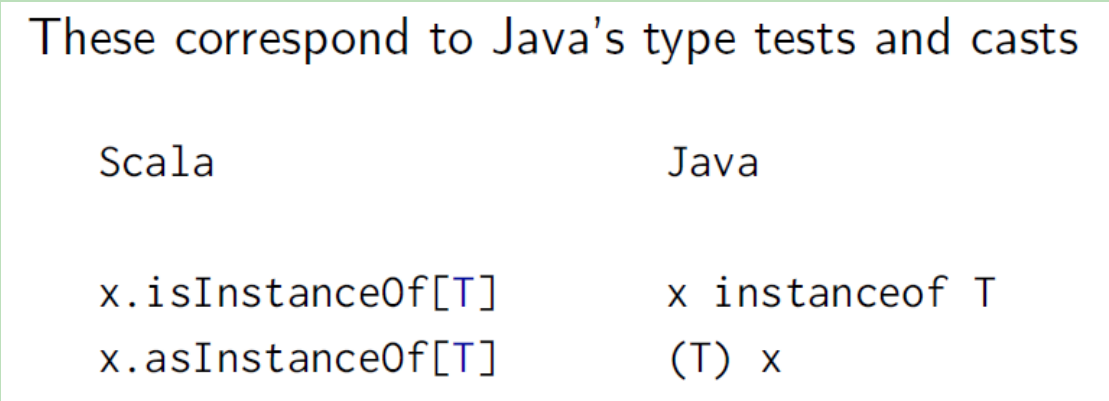
When expr have only two subclass :num and sum, we need the classification method for both subclasses(isNumber, is Sum), when we need to add new subclass: prod and value, we need to add two new subclass classification methods in trait as well as all subclass.
This means quadratic increase of methods

2.Use type test and cast instead of classification and access

actually it turns out that type test and type casts are very **low level and unsafe** and that, in fact,

Rig

Scala has much better solutions.(**showed below, use type test in if clause to assert successful type casting)**



Old version :

```
def eval(e: Expr): Int = {
if (e.isNumber) e.numValue
else if (e.isSum) eval(e.leftOp) + eval(e.rightOp)
else throw new Error("Unknown expression " + e)
}
```

Here's a formulation of the eval method using type tests and casts:

```
def eval(e: Expr): Int =
if (e.isInstanceOf[Number])
e.asInstanceOf[Number].numValue
else if (e.isInstanceOf[Sum])
eval(e.asInstanceOf[Sum].leftOp) +
eval(e.asInstanceOf[Sum].rightOp)
else throw new Error("Unknown expression " + e)
```

use of typetest and typecast is very low level, it is unsafe because, when you do a type-cast you do not know necessarily at runtime whether type-cast will succeed. It might also throw a classcast exception. In this example, here we have actually guarded every type-cast by a type-test, so we can assure statistically that none of these casts will fail, but in general that's not assured. So that's why, we recommend the people to stay away from type-casts.

3. object-oriented decomposition

```
trait Expr {
def eval: Int
}
class Number(n: Int) extends Expr {
def eval: Int = n
}
```

Rig

```
class Sum(e1: Expr, e2: Expr) extends Expr {
def eval: Int = e1.eval + e2.eval
}
```

But this could not create simplify method, which need to use classification and access method again:

a * b + a * c -> a * (b + c)

Non-local simplification

And what if you want to simplify the expressions, say using the rule:

a * b + a * c -> a * (b + c)

Problem: This is a non-local simplification. **It cannot be encapsulated in the method of a single object.**

You are back to square one; you need test and access methods for all the different subclasses.

*Class6 pattern matching

Form of pattern

Patterns are constructed from:

- *constructors*, e.g. Number, Sum,
- *variables*, e.g. n, e1, e2,
- *wildcard patterns* _,(Number(_) act like Number(n) but it does not care about what _ is
- *constants*, e.g. 1, true.

Variables always begin with a lowercase letter.

The same variable name can only appear once in a pattern. So, Sum(x, x) is not a legal pattern.

Names of constants begin with a capital letter, with the exception of the reserved words null, true, false.

An expression of the form

e match *f* case p_1 => e_1 ::: case p_n => e_n *g*

matches the value of the selector e with the patterns p_1; :::; p_n in the order in which they are written.

The **whole match expression is rewritten to the right-hand side of the first case** where the pattern matches the selector *e*.

References to pattern variables are replaced by the corresponding parts in the selector.

Rig

*expression problem

Object oriented decomposition solution:

```
trait Expr {
  def eval: Int ;  def show: String
}
class Number(n: Int) extends Expr {
  def eval: Int = n
}
class Sum(e1: Expr, e2: Expr) extends Expr {
  def eval: Int = e1.eval + e2.eval
}
```

Pattern matching method

Refer to the code after this section: case matching for eval & eval defined as a method

Which to choose

So it's a criterion that looks at the future extensibility and the possible extension pass of your system.
If what you do is mostly **creating new subclasses,** then actually the **object oriented decomposition solution** has the upper hand. The reason is that it's very easy and a very local change to just create a new subclass with an eval method, whereas in the functional solution, you'd have to **go back and change the code inside the eval method and add a new case to it**.

On the other hand, if what you do will be **create lots of new methods**. But the **class hierarchy itself will be kept relatively stable**. Then pattern matching is actually advantageous. Because, again, e**ach new method in the pattern matching solution is just a local change**, whether you put it in the base class, or maybe even outside the class hierarchy. Whereas a new method such as show in the object oriented decomposition would require **a new incrementation is each sub class.** So there would be more parts, That you have to touch. So the problematic of this extensibility in two dimensions, where you might want to add new classes to a hierarchy, or you might want to add new methods,

Rig

or maybe both, has been named the expression problem. The name actually comes from this very example of arithmetic expression,

CODE: case matching for eval function

```
/*one solution*/
trait Expr
case class Number(n: Int) extends Expr
case class Sum(e1: Expr, e2: Expr) extends Expr
//下面的如果不注释，会导致 apply 函数重复定义，因为前面的 case class
//object Number{
//  def apply(n: Int) = new Number(n)
//}
//object Sum{
//  def apply(e1: Expr, e2: Expr) = new Sum(e1, e2)
//}
//不定义 case class ，只用 object 是不能使用下面的 match
def eval(e: Expr): Int = e match{
  case Number(n) => n
  case Sum(e1, e2) => eval(e1) + eval(e2)
}
def show(e:Expr): String = e match{
  case Number(n) => n.toString()
  case Sum(e1, e2) => show(e1) + " + " + show(e2)
}
eval(Sum(Number(1), Number(2)))
//eval(Number(Sum(Number(1), Number(2))))//Number 只能传入 int
eval(Number(1)) + eval(Number(2))
show(Sum(Number(2), Number(2)))
```

CODE eval defined as a method of expr

```
/*second solution*/
trait Expr{
  def eval:Int = this match{
    case Number(n) => n
    case Sum(e1, e2) => e1.eval + e2.eval
      //why can't sum work like this:
      //case Sum(e1, e2) => eval(e1) + eval(e2)
      //becasue this is this.eval(e1), but eval does not
      //take any parameter
  }
  def show: String = this match {
    case Number(n) => n.toString()
    case Sum(e1, e2) => e1.show + " + " + e2.show
  }
}
case class Number(n: Int) extends Expr
case class Sum(e1: Expr, e2: Expr) extends Expr

Number(1).eval + Number(2).eval
//show(Number(1).eval + Number(2).eval)// this can't work becasue
//parameter is Int
(Sum(Number(2), Number(2))).show
```

CODE custom show

```
def show: String = this match {
  case Number(n) => n.toString()
  case Sum(e1, e2) => e1.show + " + " + e2.show
  case Prod(e1, e2) => e1 match{
    case Sum(_, _) => "(" + e1.show + ")*" + e2.show
    case Number(n) => n.toString() + e2.show
  }
}
(Prod(Sum(Number(1), Number(2)), Number(1))).show
```

res2: String = (1 + 2)*1

Class 7

List constructor ::

All lists are constructed from:

- the empty list `Nil`, and
- the construction operation `::` (pronounced *cons*):
  `x :: xs` gives a new list with the first element `x`, followed by the elements of `xs`.

For example:

```
fruit = "apples" :: ("oranges" :: ("pears" :: Nil))
nums  = 1 :: (2 :: (3 :: (4 :: Nil)))
empty = Nil
```

new Cons (x, xs)

x :: xs

Operator end in a colon

1. We have the universal convention that **all operators that end in a colon associate to the right.** Whereas all other operators would associate to the left, as usual.

Convention: Operators ending in ":" associate to the right.

A :: B :: C is interpreted as A :: (B :: C).

Example

```
val nums = 1 :: 2 :: 3 :: 4 :: Nil
```

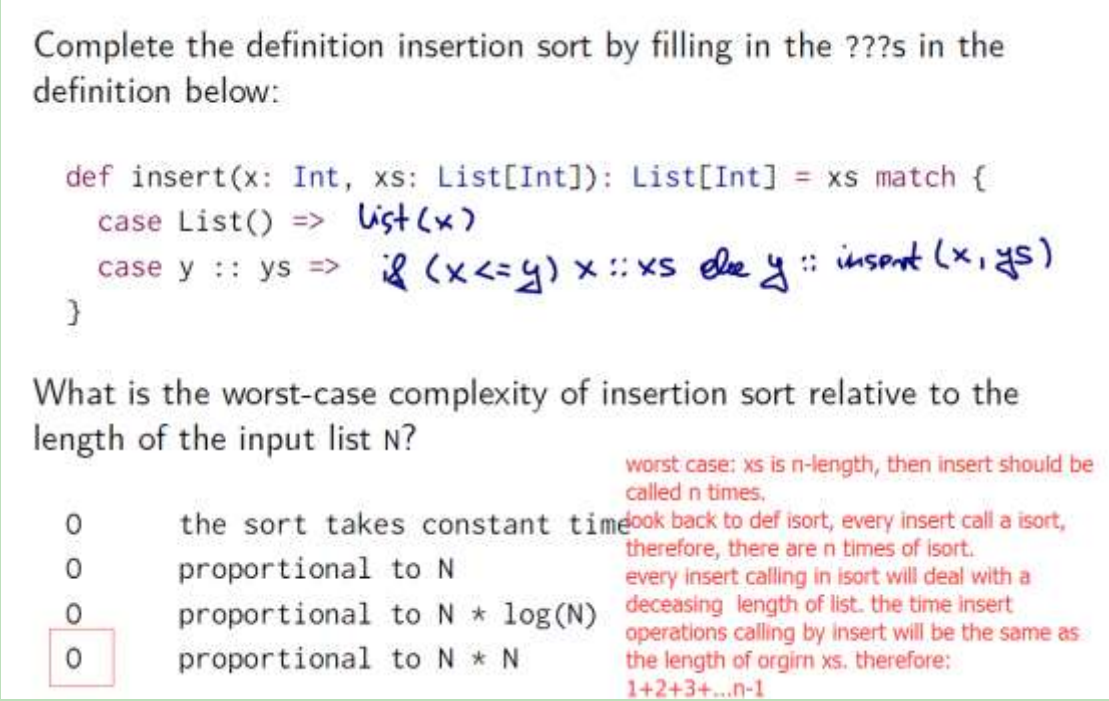
Operators ending in ":" are also different in the they are seen as method calls of the *right-hand* operand.

So the expression above is equivalent to

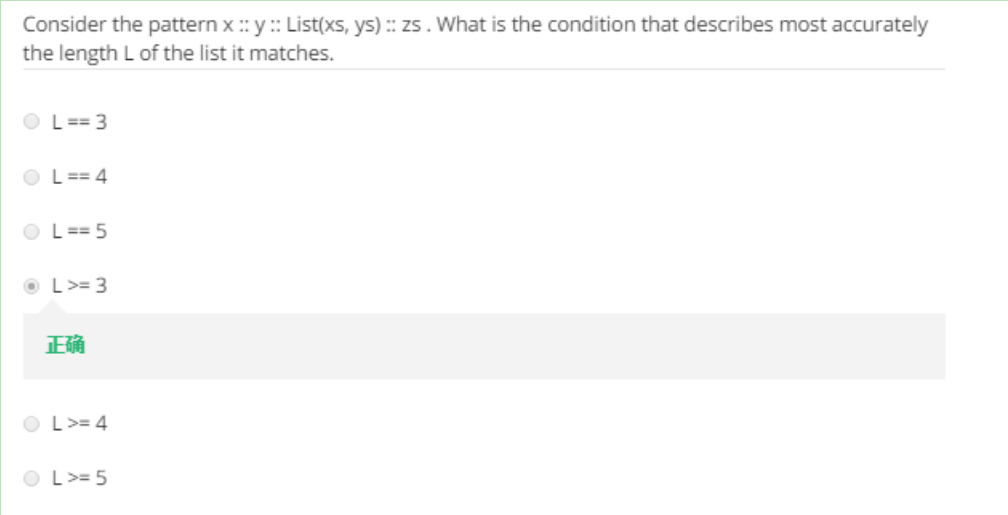
:: ≈ prepend

```
Nil.::(4).::(3).::(2).::(1)
```

Time complexity



quiz



Consider the real value of zs, when it's empty, then l = 3, or l = 4

Week5

Class 1

Rig

Class 2 pair and tuple

```
//pair
val a = (1, 2)
a._1
a._2
val (label, value) = a
label
value
```

The fields of a tuple can be accessed with names _1, _2, ...

So instead of the pattern binding

```
val (label, value) = pair
```

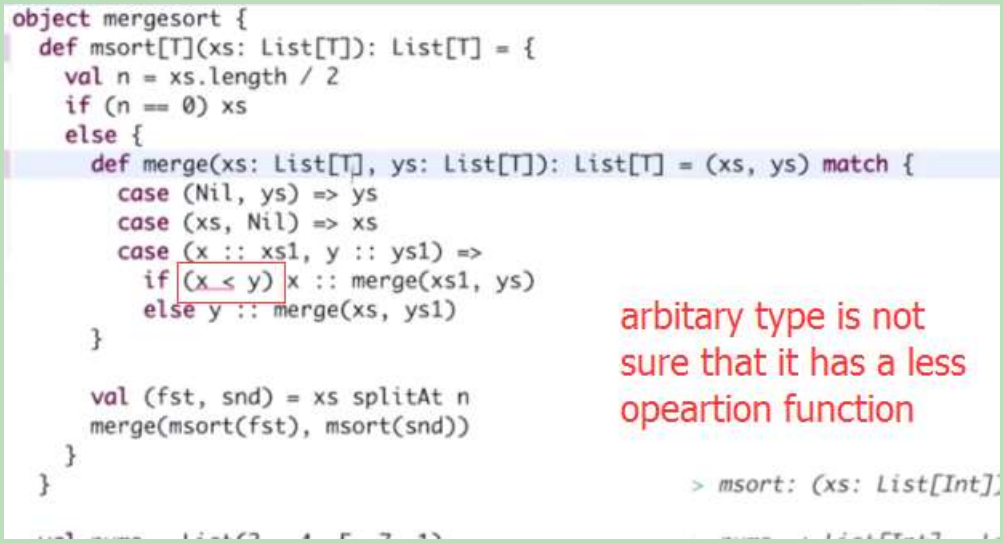
one could also have written:

```
val label = pair._1
val value = pair._2
```

Code tuple matching:归并排序

```
object mergeSort{
  def msort(xs: List[Int]): List[Int] = {
    val n = xs.length / 2
    if (n == 0) xs
    else {
      def merge(xl: List[Int], yl: List[Int]): List[Int] = (xl, yl) match {
        case (Nil, yl) => yl
        case (xl, Nil) => xl
        case (x :: xt, y :: yt) =>
          if (x < y) x :: merge(xt, yl)
          else y :: merge(xl, yt)
      }
      merge(msort(xs.take(n)), msort(xs.drop(n)))
    }
  }
}
println(mergeSort.msort(List(1,334,4, 5,6, 7,3, 4)))
```

class3



Code:: arbitrary type of function definition need [T] after function name and we have to give a definition for less than operation

CODE arbitrary mergeSort

```
object mergeSortT{
  def msort[T](xs: List[T])(lt: (T, T) => Boolean): List[T] = {
    val n = xs.length / 2
    if (n == 0) xs
    else {
      def merge(xl: List[T], yl: List[T]): List[T] = (xl, yl) match {
        case (Nil, yl) => yl
        case (xl, Nil) => xl
        case (x :: xt, y :: yt) =>
          if (lt(x, y)) x :: merge(xt, yl)
          else y :: merge(xl, yt)
      }
      merge(msort(xs.take(n))(lt), msort(xs.drop(n))(lt))
    }
  }
}
mergeSortT.msort(List("a", "b", "c", "aa", "ab"))((x, y) => (x.compareTo(y) < 0))
//parameter used in msort have different types, one of them is function type,
```

```
//compiler can infer the type of x and y from function type
mergeSortT.msort(List(1, 3, 4, 5, 6, 6, 9))((x, y) => (x > y))
```

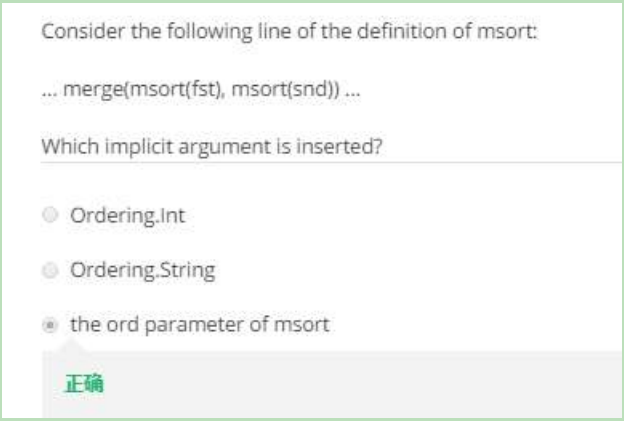
CODE import ordering

```
import math.Ordering
object mergeSortT{
  def msort[T](xs: List[T])(ord: Ordering[T]): List[T] = {
    val n = xs.length / 2
    if (n == 0) xs
    else {
      def merge(xl: List[T], yl: List[T]): List[T] = (xl, yl) match {
        case (Nil, yl) => yl
        case (xl, Nil) => xl
        case (x :: xt, y :: yt) =>
          if (ord.lt(x, y)) x :: merge(xt, yl)
          else y :: merge(xl, yt)
      }
      merge(msort(xs.take(n))(ord), msort(xs.drop(n))(ord))
    }
  }
}
mergeSortT.msort(List("a", "b", "c", "aa", "ab"))(Ordering.String)
//parameter used in msort have different types, one of them is function type,
//compiler can infer the type of x and y from function type
mergeSortT.msort(List(1, 3, 4, 5, 6, 6, 9))(Ordering.Int)
```

*using implicit argument

```
import math.Ordering
object mergeSortT{
  def msort[T](xs: List[T])(implicit ord: Ordering[T]): List[T] = {
    val n = xs.length / 2
    if (n == 0) xs
    else {
      def merge(xl: List[T], yl: List[T]): List[T] = (xl, yl) match {
        case (Nil, yl) => yl
        case (xl, Nil) => xl
        case (x :: xt, y :: yt) =>
          if (ord.lt(x, y)) x :: merge(xt, yl)
          else y :: merge(xl, yt)
```

```
      }
      merge(msort(xs.take(n)), msort(xs.drop(n)))
    }
  }
}
mergeSortT.msort(List("a", "b", "c", "aa", "ab"))
//parameter used in msort have different types, one of them is function type,
//compiler can infer the type of x and y from function type
mergeSortT.msort(List(1, 3, 4, 5, 6, 6, 9))
```



Say, a function takes an implicit parameter of type T.The compiler will search an implicit definition that

- is marked implicit
- has a type compatible with T
- is visible at the point of the function call, or is defined in a companion object associated with T.

If there is a **single (most specific) definition**, it will be taken as actual argument for the implicit parameter. If there are multiple possible type, like

```
mergeSortT.msort(List("a", "b", "c", 1, "ab"))
```

it's an error.

* Class 4 List function

```
object listfun {
  val nums = List(2, -4, 5, 7, 1)                    > nums  : List[Int] = List(2, -4, 5, 7, 1)
  val fruits = List("apple", "pineapple", "orange", "banana")
                                                     > fruits  : List[java.lang.String] = List(a

  nums filter (x => x > 0)                           > res0: List[Int] = List(2, 5, 7, 1)
  nums filterNot (x => x > 0)                        > res1: List[Int] = List(-4)
  nums partition (x => x > 0)                        > res2: (List[Int], List[Int]) = (List(2, 5
}
```

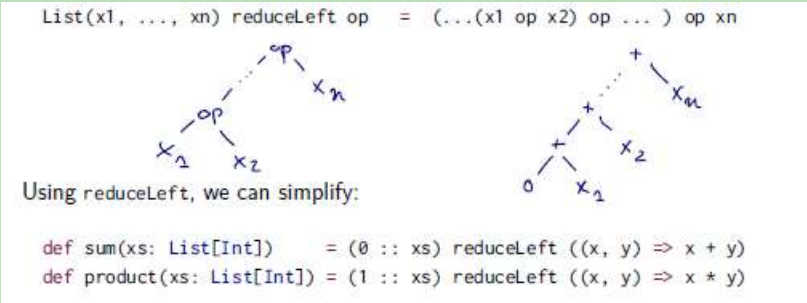
```
  nums filter (x => x > 0)                    > res0: List[Int] = List(2, 5, 7, 1)
  nums filterNot (x => x > 0)                 > res1: List[Int] = List(-4)
  nums partition (x => x > 0)                 > res2: (List[Int], List[Int]) = (List(2,

  nums takeWhile (x => x > 0)                 > res3: List[Int] = List(2)
  nums dropWhile (x => x > 0)                 > res4: List[Int] = List(-4, 5, 7, 1)
  nums span (x => x > 0)
}
```

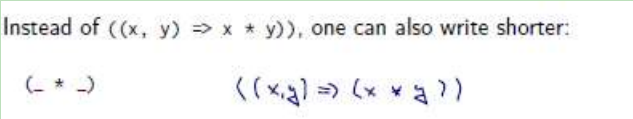
```
def pack[T](xs: List[T]): List[Any] = xs match{
  case Nil => Nil
  case y :: ys => y match{
    case z :: zs => {
//      println(ys.isEmpty, " - ", z)
      if (!(ys.isEmpty)  && (z == ys.head)) {
//        println((ys.head :: (z :: zs)))
        pack((ys.head :: (z :: zs)) :: ys.tail)
      }
      else (z :: zs) :: pack(ys)//key is use :: between two element, then left part
will be a single element
      //if use ++ , then left part will be expended.
    }
    case z => {
//      println( z)
      pack(List(z) :: ys)
    }
  }
}

def pack2[T](xs: List[T]): List[Any] = xs match{
  case Nil => Nil
  case x :: xs1 => {
    val (first, rest) = xs.span(y => y == x)
    first :: pack2(rest)
  }

}

def encode[T](xs: List[T]): List[Any] = xs match{
  case Nil => Nil
  case x :: xs1 => {
    val (first, rest) = xs.span(y => y == x)
    (x, first.length) :: encode(rest)
  }
```

```
}

pack(List("a", "a", "a", "b", "c", "c", "a"))
encode(List("a", "a", "a", "b", "c", "c", "a"))
```

class 5

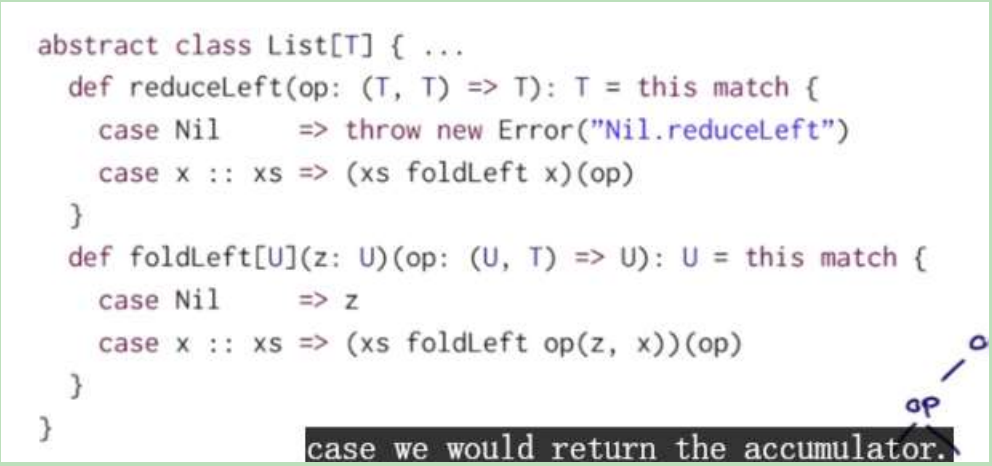
reduceleft






```
def sum1(xs: List[Int]): Int = (0 :: xs) reduceLeft (_ + _)
def sum(xs: List[Int]): Int = (xs foldLeft 0)(_ + _)
sum(List())
sum1(List())//???why can this run without error when input empty list,
```

Here we need xs have almost one elements???not ， because the definitions are showed below:



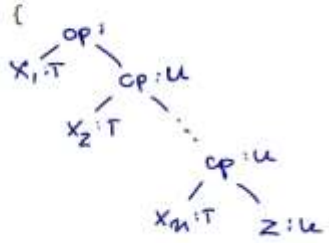
Rig

Code Foldright and reduceright

Keep an eye on the type U and T

They are defined as follows

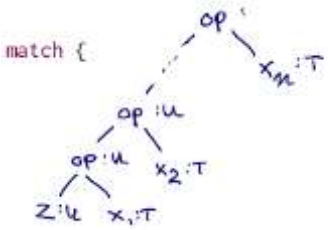
```
def reduceRight(op: (T, T) => T): T = this match {
  case Nil => throw new Error("Nil.reduceRight")
  case x :: Nil => x
  case x :: xs => op(x, xs.reduceRight(op))
}
def foldRight[U](z: U)(op: (T, U) => U): U = this match {
  case Nil => z
  case x :: xs => op(x, (xs foldRight z)(op))
}
```



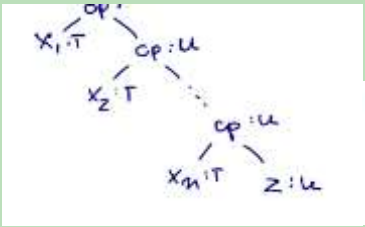

```
def concatFR[T](xs: List[T], ys: List[T]): List[T] = (xs foldRight ys) (_ :: _)
def concatFL[T](xs: List[T], ys: List[T]): List[T] = (xs foldLeft ys) (_ :: _)

concatFL(List(1,2), List(3, 4))//this can’ t work because right parameter of foldLeft ys would be use in the left side of (_ :: _)
concatFR(List(1,2), List(3, 4))
```

Something Interesting

```
def concatFR[T](xs: List[T], ys: List[T]): List[T] = (xs foldLeft ys) ((x, y) => (y :: x))

concatFR(List(1,2,3), List(4,5,6))

List[Int] = List(3, 2, 1, 4, 5, 6)
This can be explained by the definition of two function
Foldleft start from z, process operation of (z, x) , z is last result, x is next element in xs
```




```
def foldLeft[U](z: U)(op: (U, T) => U): U = this match {
  case Nil     => z
  case x :: xs => (xs foldLeft op(z, x))(op)
}
}
```




```
def foldRight[T, U](z: U)(op: (T, U) => U): U = this match {
  case Nil => z
  case x :: xs => op(x, (xs foldRight z)(op))
}
```

Fold right have a huge burden of recursive using of stack.

Test

```
def mapFun[T, U](xs: List[T], f: T => U): List[U] = (xs foldRight List[U]())(f(_) :: _)
def lengthFun[T](xs: List[T]): Int = (xs foldRight 0) ((x, y) => (y + 1))
lengthFun(List(1,2,3))
def op(x: Int):Int = x + 3
mapFun(List(1, 2, 3), op)//u can't use (x => x + 1) to replace op ,since type U is not defined in mapFun
//complier don;t know whether plus operation can work or not
```

**class 6

correctness of program

What does it mean for a function program to be correct? One possible answer would be that the definitions in the program satisfy certain laws. Often these laws are **represented as equalities between terms.**

structual induction

question：

```
(xs ++ ys) ++ zs  =  xs ++ (ys ++ zs)
       xs ++ Nil  =  xs
       Nil ++ xs  =  xs
```

Q: How can we prove properties like these?

A: By *structural induction* on lists.

Our induction hypothesis is:

```
(xs ++ ys) ++ zs  =  xs ++ (ys ++ zs)
```

Base case:

Base case: `Nil`

For the left-hand side we have:

```
(Nil ++ ys) ++ zs

=   ys ++ zs        // by 1st clause of ++
```

For the right-hand side, we have:

```
Nil ++ (ys ++ zs)

=   ys ++ zs        // by 1st clause of ++
```

This case is therefore established.

Induction step

```
def concat[T](xs: List[T], ys: List[T]) = xs match {
  case List() => ys
  case x :: xs1 => x :: concat(xs1, ys)
}
```

2n clause of ++ : x:: (y ++ z) = (x :: y) ++ z

Induction step: `x :: xs`

For the left-hand side, we have:

```
((x :: xs) ++ ys) ++ zs

=   (x :: (xs ++ ys)) ++ zs     // by 2nd clause of ++

=   x :: ((xs ++ ys) ++ zs)     // by 2nd clause of ++

=   x :: (xs ++ (ys ++ zs))     // by induction hypothesis
```

Rig

For the right hand side we have:

```
(x :: xs) ++ (ys ++ zs)

=   x :: (xs ++ (ys ++ zs))     // by 2nd clause of ++
```

So this case (and with it, the property) is established.

Class 7

```
      Nil.reverse  =  Nil                      // 1st clause
(x :: xs).reverse  =  xs.reverse ++ List(x)    // 2nd clause
```

We'd like to prove the following proposition

```
xs.reverse.reverse  =  xs
```

We still need to show:

```
(xs.reverse ++ List(x)).reverse  =  x :: xs.reverse.reverse
```

Trying to prove it directly by induction doesn't work.

We must instead try to *generalize* the equation. For *any* list ys,

after simplify both sides of original equation, we still can get a obvious result and here we try to make some transformation

```
(ys ++ List(x)).reverse  =  x :: ys.reverse
```

Unfold and fold

```
((y :: ys) ++ List(x)).reverse          // to show: =  x :: (y :: ys).reverse
                  ↓ unfold
=   (y :: (ys ++ List(x))).reverse      // by 2nd clause of ++

=   (ys ++ List(x)).reverse ++ List(y)  // by 2nd clause of reverse

=   (x :: ys.reverse) ++ List(y)        // by the induction hypothesis

=   x :: (ys.reverse ++ List(y))        // by 1st clause of ++
                  ↓ fold
=   x :: (y :: ys).reverse              // by 2nd clause of reverse
```

This establishes the auxiliary equation, and with it the main proposition.

fold / unfold method

Test

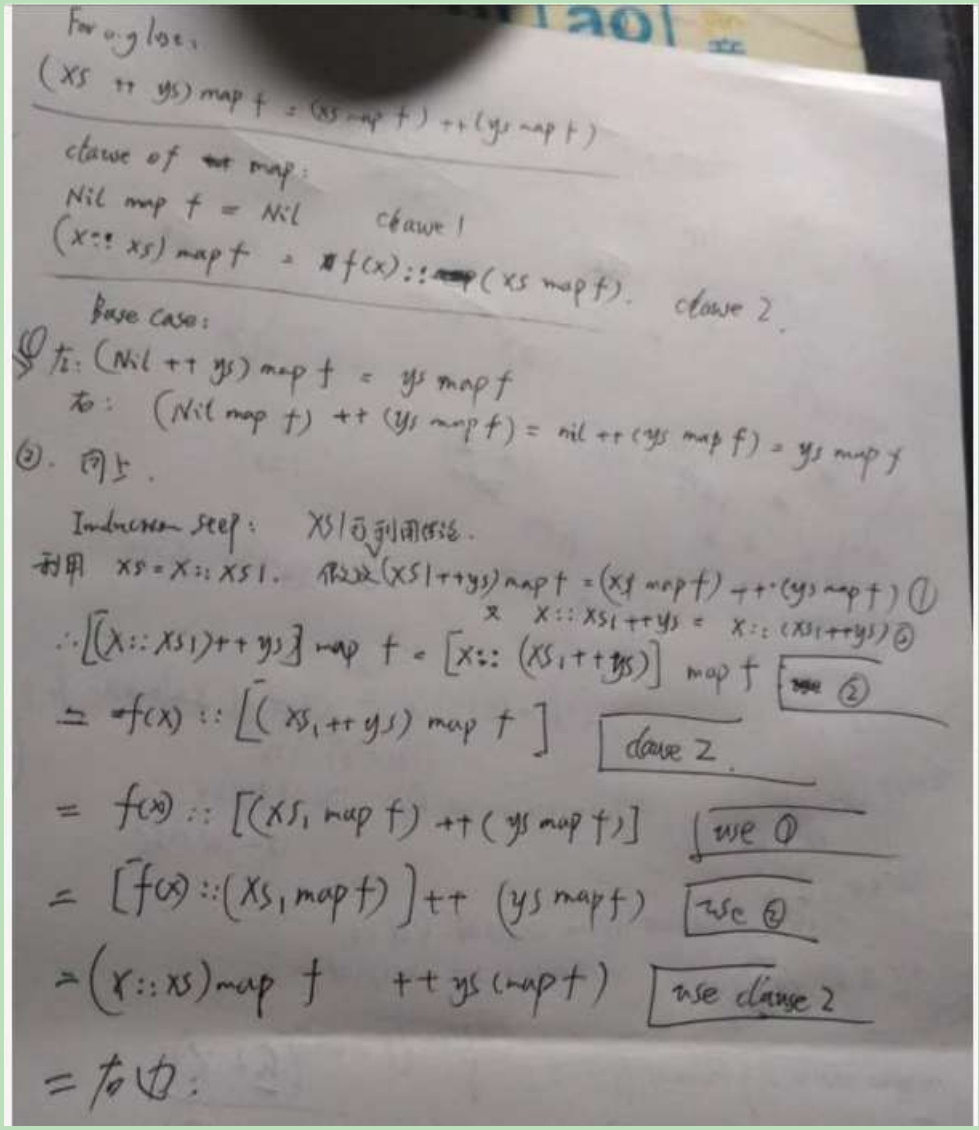
Prove the following distribution law for `map` over concatenation.

For any lists `xs`, `ys`, function `f`:

```
(xs ++ ys) map f  =  (xs map f) ++ (ys map f)
```

You will need the clauses of ++ as well as the following clauses for `map`:

```
      Nil map f  =  Nil
(x :: xs) map f  =  f(x) :: (xs map f)
```

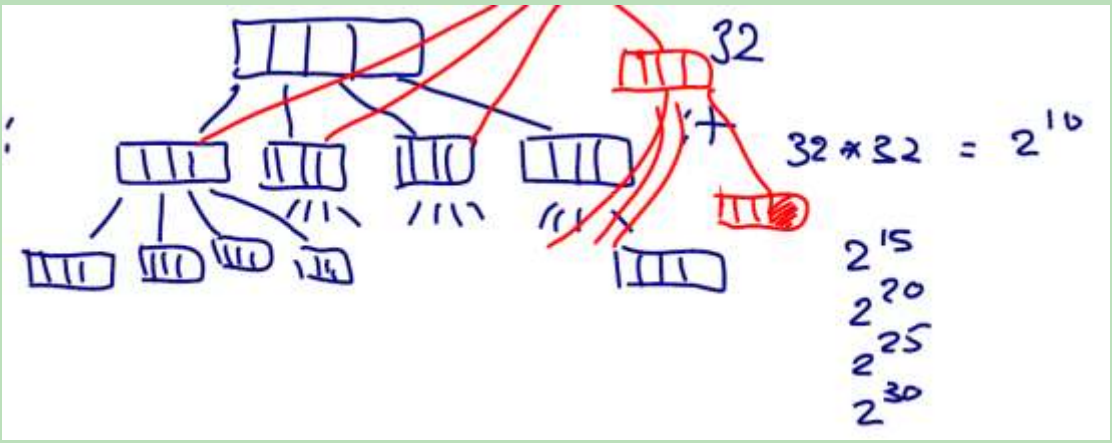


WEEk6

Class1

New collections are immutable.

Vector



Finding: dig into deep level, complexity: log32(n). 32 is the length of one block

Inserting: we see we have to create a new Object 32 element array for every level of the vector where we did the change here. Here we need to create 3 new blocks.

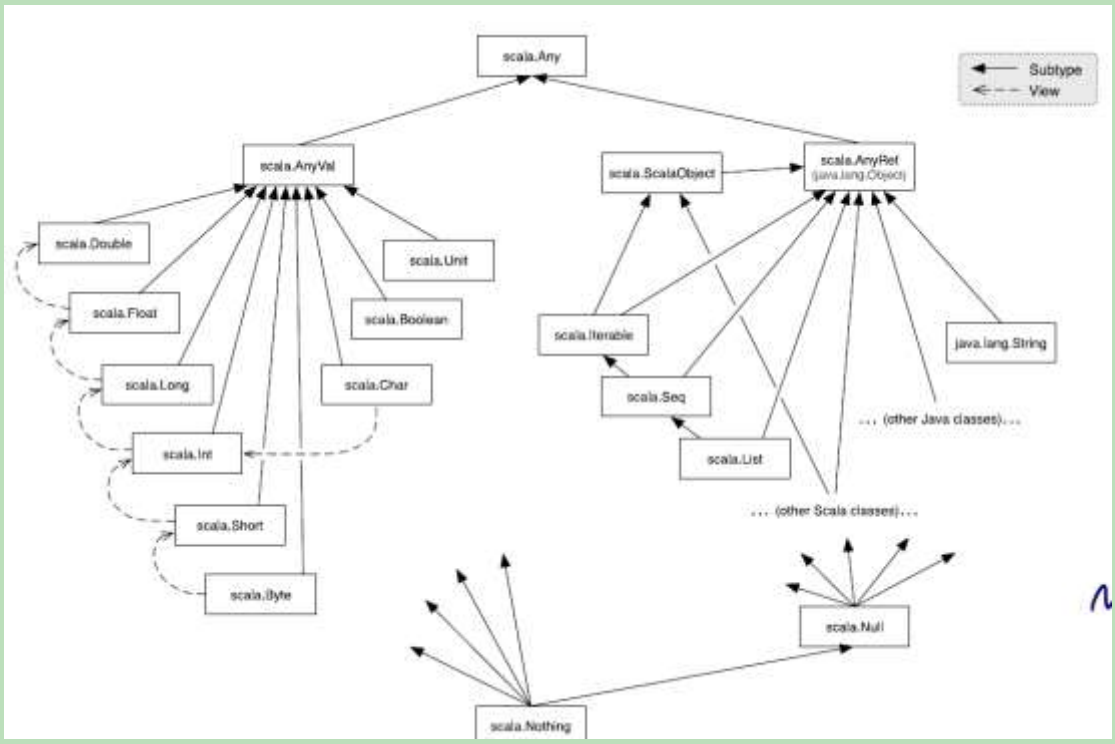
Vectors are created analogously to lists:
val nums = Vector(1, 2, 3, -88)
val people = Vector(”Bob”, ”James”, ”Peter”)

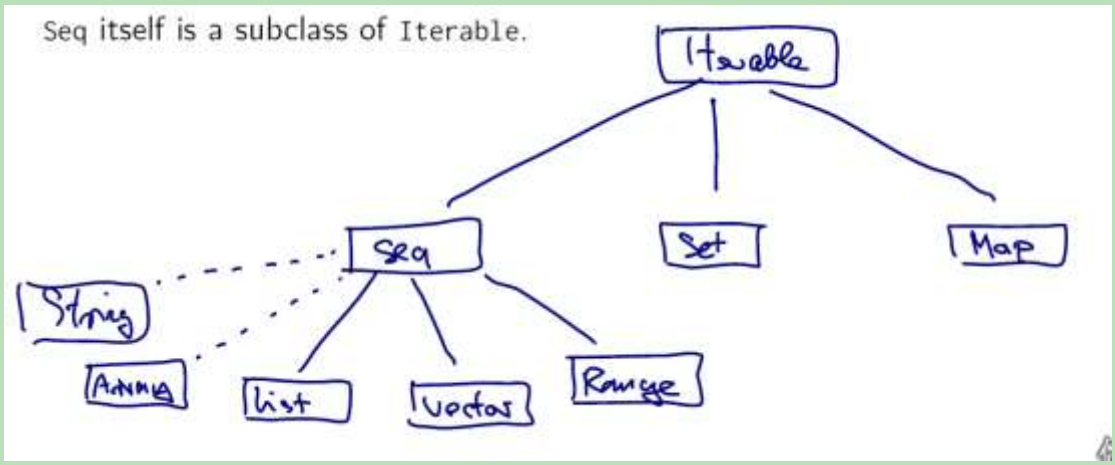
operation of vector

They **support the same operations as lists, with the exception of ::**
Instead of x :: xs, there is
x +: xs Create a new vector with leading element x, followed
by all elements of xs.
xs :+ x Create a new vector with trailing element x, preceded
by all elements of xs.
(Note that the : always points to the sequence.)

Rig

Seq type





draw a dotted line here because they're not really subclasses of sequence. They cannot be that because both **string and array come from the Java universe,** and of course a Java class that doesn't know that at some future time somebody would define a class called Scala.sequence.

range type

val r: Range = 1 until 5
val s: Range = 1 to 5
1 to 10 by 3
6 to 1 by -2
Ranges a represented as single objects with three fields: lower bound, upper bound, step value.

sequence opearation

`xs exists p`	`true` if there is an element `x` of `xs` such that `p(x)` holds, `false` otherwise.
`xs forall p`	`true` if `p(x)` holds for all elements `x` of `xs`, `false` otherwise.
`xs zip ys`	A sequence of pairs drawn from corresponding elements of sequences `xs` and `ys`.
`xs.unzip`	Splits a sequence of pairs `xs` into two sequences consisting of the first, respectively second halves of all pairs.
`xs.flatMap f`	Applies collection-valued function `f` to all elements of `xs` and concatenates the results
`xs.sum`	The sum of all elements of this numeric collection.
`xs.product`	The product of all elements of this numeric collection
`xs.max`	The maximum of all elements of this collection (an `Ordering` must exist)
`xs.min`	The minimum of all elements of this collection

All these are

```
object test{
  val xs = Array(1, 2, 3, 44)
  xs map (x => x*2)
  val s = "Hello World"
  s filter (c => c.isUpper)
  s exists (c => c.isUpper)
  s forall (c => c.isUpper)
  val pairs = List(1, 2, 3) zip s
  pairs.unzip
  s flatMap (c=> List('.', c))
  xs.sum
  xs.max
//  To list all combinations of numbers x and y where x is drawn from
```

Rig

```
//    1..M and y is drawn from 1..N:
  val a = (1 to 3) flatMap ( x=> (4 to 6) map(y => (x,y)))
  println(a)

  def scalarProduct(xs: Vector[Double], ys: Vector[Double]): Double = {
    (xs zip ys).map(xy => xy._1 + xy._2).sum
  }
  def scalarProduct2(xs: Vector[Double], ys: Vector[Double]): Double = {
    (xs zip ys).map{ case (x,y) => x*y}.sum
  }
  scalarProduct(Vector(1, 2, 3), Vector(4,5, 6))
//  scalarProduct2(1 to 3, 4 to 6)
  def isPrime(n: Int): Boolean = (2 until n) forall (d => n % d != 0)

}
```

Class 2

flatten does, essentially, the same thing as concat. It takes a collection of collections, and returns a single collection, containing all the elements in the sub-collections. And if the collection is a sequence, then the combination will be just the concatenation of these sub-collections.

for expression – still immutable value

Syntax of For

A for-expression is of the form

```
for ( s ) yield e
```

where `s` is a sequence of *generators* and *filters*, and `e` is an expression whose value is returned by an iteration.

- A *generator* is of the form `p <- e`, where `p` is a pattern and `e` an expression whose value is a collection.
- A *filter* is of the form `if f` where `f` is a boolean expression.
- The sequence must start with a generator.
- If there are several generators in the sequence, the last generators vary faster than the first.

Instead of `( s )`, braces `{ s }` can also be used, and then the sequence of generators and filters can be written on multiple lines without requiring semicolons.

Flatmap and map

```
val t3 = (1 until 4) flatMap ( i => (1 to i) map ( j => (i, j)))
val t = (1 to 4) map (x => x*x)
```

t3: scala.collection.immutable.IndexedSeq[(Int, Int)] = Vector((1,1), (2,1), (2,2), (3,1), (3,2), (3,3))
t: scala.collection.immutable.IndexedSeq[Int] = Vector(1, 4, 9, 16)
map 函数：

函数式编程都有一个 map 函数，map 函数就像一个加工厂，传入一个函数，利用这个函数将集合里的每一个元素处理并将结果返回。
将 map 和 flatMap 说成 Scala 函数机制的核心都不为过分。

“flatMap “函数的一半功能和 map 函数一样，不过有个要求，传入的函数在处理完后返回值必须是 List， 如果返回值不是 List，那么将出错。也就是说，传入的函数是有要求的——返回值用 List()函数构造成 list 才行。 这样，每个元素处理后返回一个 List，我们得到一个包含 List 元素的 List，flatMap 自动将所有的内部 list 的元素取出来构成一个 List 返回。

Why does the result of map in pairs is vector: using map function with range type need indexed sequence structure, and vector is naturally a indexed sequence, therefore it is automatically transformed into vector

```
object class2{
  def scalarProduct(xs: List[Double], ys: List[Double]):Double = {
    val a = for {
      (i, j) <- xs zip ys
    }yield i*j
    a.sum
  }
  def isPrime(n: Int): Boolean = (2 until n) forall (d => n % d != 0)
  val t1 = for {
    i <- 1 until 4
    j <- 4 until 7
    if isPrime(i + j)
  } yield (i, j)
  val t2 = ((1 until 4) map ( x => (1 until x) map (j => (x, j)) )).flatten
  val t3 = (1 until 4) flatMap ( i => (1 to i) map ( j => (i, j)))

}
println(class2.t2)
println(class2.t3)
```

class 3

CODE queens riddle

```
object nqueens {
  def queens(n: Int): Set[List[Int]] = {
    def placeQueens(k: Int): Set[List[Int]] =
      if (k == 0) Set(List())
      else
        for {
          queens <- placeQueens(k - 1)
          col <- 0 until n
          if isSafe(col, queens)
        } yield col :: queens
    placeQueens(n)
  }                                                    > queens: (n: I

  def isSafe(col: Int, queens: List[Int]): Boolean = {
    val row = queens.length
    val queensWithRow = (row - 1 to 0 by -1) zip queens
    queensWithRow forall {
      case (r, c) => col != c && math.abs(col - c) != row - r
    }
  }                                                    > isSafe: (col:

  queens(4)                                            > res0: Set[Lis
}
```

```
def isSafe(col: Int, queens: List[Int]): Boolean = {
  val row = queens.length
  val queensWithRow = (row - 1 to 0 by -1) zip queens
  queensWithRow forall {
    case (r, c) => col != c && math.abs(col - c) != row - r
  }
                                                       > isSafe: (col: Int, q

def show(queens: List[Int]) = {
  val lines =
    for (col <- queens.reverse)
    yield Vector.fill(queens.length)("* ").updated(col, "X ").mkString
  "\n" + (lines mkString "\n")
}                                                      > show: (queens: List[
queens(4) map show
```

```
(queens(8) map show) mkString "\n"
```

List[String] mkString String

```
print(List("a", "b", "c") mkString "\n")
```

class 4

CODE Polynomial

```
object class4{
  class Poly(val terms: Map[Int, Double]){
    def + (other: Poly) = new Poly(terms ++ (other.terms map adjust))
//    def adjust(term: (Int, Double)): (Int, Double) = {
//      for {
//        (_key, _value) <- terms
//        if _key == term._1
//      } yield (term._1, term._2 + _value)
//    }//这个写法会导致结果是 Map[Int, Double]类型
    def adjust(term: (Int, Double)): (Int, Double) = {
  val (exp, coeff) = term
  terms get exp match {//get 操作为取键，可能为 None or Some(value)
    case Some(coeff1) => exp -> (coeff + coeff1) // ->取 value 的引用可用于修改
    case None => exp -> coeff
  }
}
  override def toString =
    (for {
      (exp, coeff) <- terms.toList.sorted.reverse
    } yield coeff + "x^" + exp) mkString "+"

  }
  val p1 = new Poly(Map(1 -> 2, 3 -> 4, 5 -> 6.2))
  val p2 = new Poly(Map(0 -> 3, 3 -> 7))
  val p3 = p1 + p2
}
println(class4.p3)
```

CODE POLY using Default

```
object class42{
  class Poly(val terms0: Map[Int, Double]){
```

```
    def this(bindings: (Int, Double)*) = this(bindings.toMap)//*means repeated
    val terms = terms0 withDefaultValue 0.0//here use 0 will turn wrong since the
default is not double
    def + (other: Poly) = new Poly(terms ++ (other.terms map adjust))
    //    def adjust(term: (Int, Double)): (Int, Double) = {
    //      for {
    //        (_key, _value) <- terms
    //        if _key == term._1
    //      } yield (term._1, term._2 + _value)
    //    }//这个写法会导致结果是Map[Int, Double]类型
    def adjust(term: (Int, Double)): (Int, Double) = {
      val (exp, coeff) = term
       exp -> (coeff + terms(exp))

    }
    override def toString =
      (for {
        (exp, coeff) <- terms.toList.sorted.reverse
      } yield coeff + "x^" + exp) mkString "+"

  }
  val p1 = new Poly(1 -> 2, 3 -> 4, 5 -> 6.2)
  val p2 = new Poly(0 -> 3, 3 -> 7)
  val p3 = p1 + p2
}
```

**CODE POLY using foldLeft

Using foldLeft is more efficient than using ++

Expression:

I would argue the one with foldLeft is more efficient because what happens here is that each of these bindings will be immediately added to our terms Maps so, we build up the result directly, whereas before (in adjust version), we would create another list of terms that contain the adjusted terms and then we would concatenate this list to the original one.

```
object class43{
  class Poly(val terms0: Map[Int, Double]){
    def this(bindings: (Int, Double)*) = this(bindings.toMap)//*means repeated
    val terms = terms0 withDefaultValue 0.0//here use 0 will turn wrong since the
default is not double
    def + (other: Poly) = new Poly((other.terms foldLeft terms)(addTerm))//default is
original this.terms
```

```
    def addTerm(terms: Map[Int, Double], term: (Int, Double)): Map[Int, Double] = {
//      terms(term._1) -> (terms(term._1) + term._2)
      val (exp, coeff) = term
      val changed = coeff + terms(exp)

//      exp -> (changed)// 如果出现了 terms 中没有的键：exp，这个键其实不会被加入，
//      //这个 a->b 的语句在 adjust 能起作用是因为返的是（Int, Double)，这语句是个返回值
      terms + (exp -> (changed))//这样才能起修改作用

      terms
    }
//    def adjust(term: (Int, Double)): (Int, Double) = {
//      val (exp, coeff) = term
//      exp -> (coeff + terms(exp))
//
//    }
    override def toString =
      (for {
        (exp, coeff) <- terms.toList.sorted.reverse
      } yield coeff + "x^" + exp) mkString "+"

  }
  val p1 = new Poly(1 -> 2, 3 -> 4, 5 -> 6.2)
  val p2 = new Poly(0 -> 3, 3 -> 7)
  val p3 = p1 + p2
}
print(class43.p3)
```

class 5

Map like charCode is function itself.

CODE map letters to numbers

```
object class5 {
  //  val in =
Source.fromURL("http:lamp.epf.ch/files/content/sites/lamp/files/teaching/progfun/linxuwords")
  val rWords = List("we", "you", "here", "-", "jj", "kk")
  val words = rWords filter (word => word forall (chr => chr.isLetter))
```

Rig

```
  //we can filter all of non letter characters.
  val mnem1 = Map("2" -> "ABC", "3" -> "DEF", "4" -> "GHI", "5" -> "JKL",
    "6" -> "MNO", "7" -> "PQRS", "8" -> "TUV", "9" -> "WXYZ"
  )
  val mnem = Map('2' -> "ABC", '3' -> "DEF", '4' -> "GHI", '5' -> "JKL",
    '6' -> "MNO", '7' -> "PQRS", '8' -> "TUV", '9' -> "WXYZ")

  val charCode: Map[Char, Char] =
    for {      (digit, str) <- mnem
      //use mnem1 will triger the problem of type mismatch
      // because "" is string '' is char
      ltr <- str
    } yield ltr -> digit
  //这里不能将 for 循环用括号括起来，不然会提示这个错误，其实就是找不到返回值

  def wordCode(word: String): String =
//    word map charCode//只有大写字母可以匹配
  word.toUpperCase map charCode//转换为大写字母
  //here charCode, a map is function itself
  val wordsForNum: Map[String, Seq[String]] =
    words groupBy wordCode withDefaultValue Seq()
  //groupby is a member function of list

  def encode(number: String): Set[List[String]] ={
    if (number.isEmpty) Set(List())//first we consider boundary case
    else {
      val temp = for {
        split <- 1 to number.length;
        word <- wordsForNum(number take split)
        rest <- encode(number drop split)
      } yield word :: rest
      temp.toSet
    }
  }
  val e1 = encode("4373")

}
print(class5.words)
print(class5.wordCode("hey"))
print(class5.wordsForNum)
print(class5.e1)
```

List(we, you, here, jj, kk)res0: Unit = ()

Rig

439res1: Unit = ()
Map(968 -> List(you), 4373 -> List(here), 55 -> List(jj, kk), 93 -> List(we))res2: Unit = ()
Set(List(here))res3: Unit = ()

advantage of immutable collection in scala

Scala's immutable collections are:
- easy to use: few steps to do the job.
- concise: one word replaces a whole loop.
- safe: type checker is really good at catching errors.
- fast: collection ops are tuned, can be parallelized.
- universal: one vocabulary to work on all kinds of collections.

This makes them a very attractive tool for software development

Homework

Split a sentence into words
http://stackoverflow.com/questions/14469958/how-to-split-sentence-into-words-separated-by-multiple-spaces

course2

week 1

recap1

如何在编译器外执行 scala 代码

Once you start writing some code, you might want to run your code on a few examples to see if it works correctly. We present two possibilities to run the methods you implemented.
Using the Scala REPL
In the sbt console, start the Scala REPL by typing **console**.

> console
[info] Starting scala interpreter...
scala>

The classes of the assignment are available inside the REPL, so you can for instance import all the methods from object **Lists:**

scala> import example.Lists._
import example.Lists._
scala> max(List(1,3,2))
res1: Int = 3
Using a Main Object
Another way to run your code is to create a new **Main** object that can be executed by the Java Virtual Machine.

1. In eclipse, right-click on the package example in **src/main/scala** and select “New” - “Scala Object”
2. Use Main as the object name (any other name would also work)
3. Confirm by clicking “Finish”

In order to make the object executable it has to extend the type **App**. Change the object definition to the following:

```
object Main extends App {
  println(Lists.max(List(1,3,2)))
}
```

Now the **Main** object can be executed. In order to do so in eclipse:

1. Right-click on the file **Main.scala**
2. Select “Run As” - “Scala Application”

You can also run the **Main** object in the sbt console by simply using the command**run.**

Test/assert/throw

```
test("details why one plus one is not three") {
  assert(1 + 1 === 2) // Fix me, please!
}
```

Rig

```
test("intNotZero throws an exception if its argument is 0") {
  intercept[IllegalArgumentException] {//拦截特定异常类型，花括号内为测试语句
    intNotZero(0)
  }
}

def intNotZero(x: Int): Int = {
  if (x == 0) throw new IllegalArgumentException("zero is not allowed")//使用 new 语句创建异常对象并抛出
  else x
}
```

JSON 处理

右键在 main 文件夹下的 scala 文件夹，选择 scala class，命名为 JsonObject，并选择为 scala object. 如果要运行该 object，需要 extends App，右键选择 run

```
object JsonObject extends App {

  abstract class JSON
  case class JSeq(elems: List[JSON]) extends JSON
  case class JObj(bindings: Map[String, JSON]) extends JSON
  case class JNum(num: Double) extends JSON
  case class JStr(str: String) extends JSON
  case class JBool(b:Boolean) extends JSON
  case object JNull extends JSON
  val data = JObj(Map(
    "firstName" -> JStr("John"),
    "address" -> JObj(Map(
      "streetAddress" -> JStr("21 2nd Street"),
      "state" -> JStr("NY"),
      "postalCode" -> JNum(10021)
    )),
    "phoneNumbers"->JSeq(List(
      JObj(Map(
        "type" -> JStr("home"),
        "number" -> JStr("101-110")
      )),
      JObj(Map(
        "type" -> JStr("fax"),
        "number" -> JStr("111-110")
      ))
    ))
```

```
  ))
  def showJson(json:JSON):String = json match{
    case JSeq(elems) =>
      "[" + (elems map showJson).mkString(",\n") + "]"
    case JObj(bindings) =>
      val assocs = bindings map {
        case (key, value) => "\"" + key + "\":" + showJson(value)
      }
      "{" + (assocs.mkString(",\n")) + "}"
    case JNum(num) => num.toString
    case JStr(str) => '\"' + str + '\"'
    case JBool(b) => b.toString
    case JNull => "null"
  }
  println(showJson(data))
}
```

Funtional1 Trait

包含一个传入参数的函数其实是 trait Functional1 的实例对象，包含 apply 函数

```
object addOne extends Function1[Int, Int]{
   def apply(m:Int):Int = m + 1
}
//其实 Functional1[Int, Int]可以简化为(Int => Int)
def func(f: Int => Int, n:Int) :Int = f(n)

val b = func(addOne, 2)//addOne 继承 Function1，可以作为函数参数传入
//val c = func(add, 2)//add 不是 function1 的子类，不能作为函数传入
```

因此 JSON 类中 JObj 的模式匹配：

```
{ case (key, value) => key + ": " + value }
```

expands to the Function1 instance

```
new Function1[JBinding, String] {
def apply(x: JBinding) = x match {
case (key, value) => key + ": " + show(value)
}
}
```

这里是整个语句新建了一个函数，其实就是新建了一个 Functional1 的子类，其 apply 函数会返

回一个字符串。（一般同类名的伴生对象会返回 new class(xx)即新类的对象）

Subclassing function

For instance, maps are functions from keys to values:
trait Map[Key, Value] extends (Key => Value) ...
Sequences are functions from Int indices to values:
trait Seq[Elem] extends Int => Elem
That’ s why we can write
elems(i)
for sequence (and array) indexing.

Partial Matches

我们可以将一个 PM 块
{case “ping” => “Pong”}
看成
Val f:String => String = { case “ping” => “Pong” }
如果尝试 f(“pong”)会返回 matchError
可以利用 partial function 的功能，来判断 case 的情况是否存在

```
val f: PartialFunction[String, String] = { case "ping" => "pong" }
f.isDefinedAt("ping") // true
f.isDefinedAt("pong") // false
```

The partial function trait is defined as follows:

```
trait PartialFunction[-A, +R] extends Function1[-A, +R] {
def apply(x: A): R
def isDefinedAt(x: A): Boolean
}
```

QUIZ

Given the function

```
val f: PartialFunction[List[Int], String] = {
  case Nil => "one"
  case x :: y :: rest => "two"
}
```

What do you expect is the result of

```
f.isDefinedAt(List(1, 2, 3))
```

?

List(1,2,3)匹配了第二种情况

- true
- false

How about the following variation:

```
val g: PartialFunction[List[Int], String] = {
  case Nil => "one"
  case x :: rest =>
    rest match {
      case Nil => "two"
    }
}
```

partial function gives you only applies to the outermost The most matching block, it's not a guarantee that if a function is defined at an argument 。

rest will match 2::3::Nil, even when running will go error in matching rest

```
g.isDefinedAt(List(1, 2, 3))
```

gives:

- true
- false

Recap2 : collection

所有 collection 共用一套通用 method:

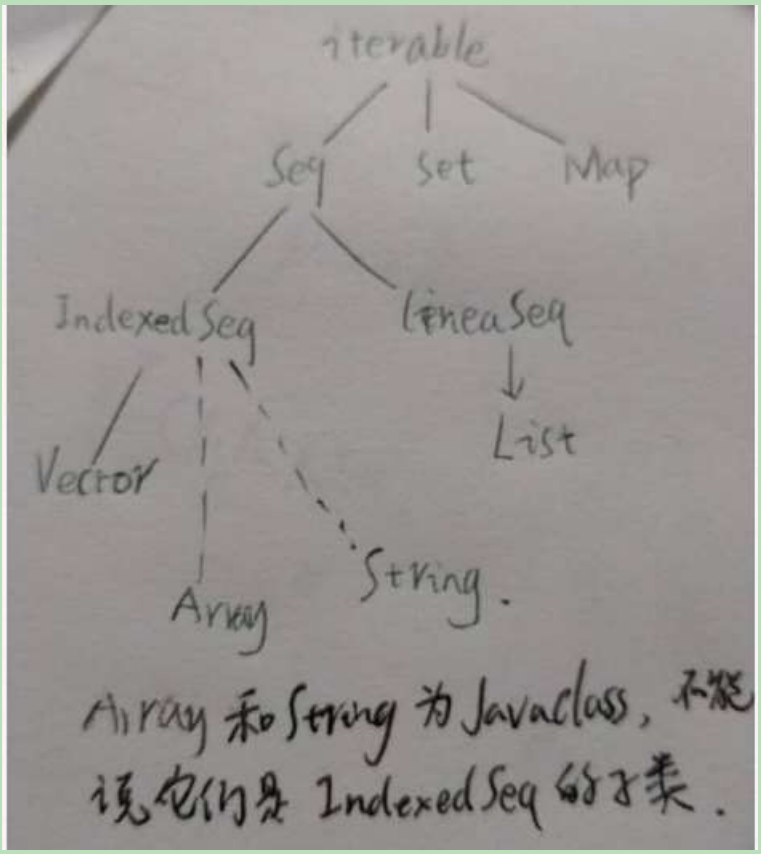
核心： map; flatMap; filter

Rig

还有：foldLeft,foldRight

Scala defines several collection classes:

Structure:



Base Classes

- `Iterable` (collections you can iterate on)
- `Seq` (ordered sequences)
- `Set`
- `Map` (lookup data structure)

Immutable Collections

- `List` (linked list, provides fast sequential access)
- `Stream` (same as List, except that the tail is evaluated only on demand)
- `Vector` (array-like type, implemented as tree of blocks, provides fast random access)

Rig

- `Range` (ordered sequence of integers with equal spacing)
- `String` (Java type, implicitly converted to a character sequence, so you can treat every string like a `Seq[Char]`)
- `Map` (collection that maps keys to values)
- `Set` (collection without duplicate elements)

Mutable Collections

- `Array` (Scala arrays are native JVM arrays at runtime, therefore they are very performant)
- Scala also has mutable maps and sets; these should only be used if there are performance issues with immutable types

For 语句和 map filter flatmap 的关系

```
abstract class List[+T] {
def map[U](f: T => U): List[U] = this match {
case x :: xs => f(x) :: xs.map(f)
case Nil => Nil
}
}
```

Int::List but List ++ List

```
abstract class List[+T] {
def flatMap[U](f: T => List[U]): List[U] = this match {
case x :: xs => f(x) ++ xs.flatMap(f)
case Nil => Nil
}
}
```

```
abstract class List[+T] {
def filter(p: T => Boolean): List[T] = this match {
case x :: xs =>
if (p(x)) x :: xs.filter(p) else xs.filter(p)
case Nil => Nil
}
}
```

Instead of:
(1 until n) flatMap (i =>

Rig

```
(1 until i) filter (j => isPrime(i + j)) map
(j => (i, j)))
one can write:
for {
i <- 1 until n
j <- 1 until i
if isPrime(i + j)
} yield (i, j)
```

```
for (x <- e1) yield e2
```
is translated to
```
e1.map(x => e2)
```

A for-expression
```
for (x <- e1 if f; s) yield e2
```
where f is a filter and s is a (potentially empty) sequence of generators and filters, is translated to
```
for (x <- e1.withFilter(x => f); s) yield e2
```

```
for (x <- e1; y <- e2; s) yield e3
```
where s is a (potentially empty) sequence of generators and filters, is translated into
```
e1.flatMap(x => for (y <- e2; s) yield e3)
```

CODE 利用 for 语句查询

For 语句内容在{}中按行顺序执行，<-符号代表分解循环，=则没有分解是赋值

```
val data: List[JSON] = ...
for {
JObj(bindings) <- data
JSeq(phones) = bindings("phoneNumbers")
JObj(phone) <- phones
JStr(digits) = phone("number")
if digits startsWith  "212"
} yield (bindings("firstName"), bindings("lastName"))
```

```
//把 if b1 != b2 改成 b1.title < b2.title 可以使得输出结果不会重复
//但是如果遇到了写过 3 本书的人，因为 b1>b2, b2 > b3, b1 > b3，这样会导致名字重复出现 3 次
```

```
//解决方法:
// 1. a1.distinct
//2. 把 books 改为 set,猜测其 for 函数体中的<-循环，b1 和 b2 会自动规避重复
```

FOR 语句的翻译

Week1 的 Recap2 一课中，我们提到了 for 语句在 scala 中其实是被翻译为 map flatmap 和 withFilter 构成的语句。
因此如果要对新的的数据结构进行 for 语句查询，考虑完善新结构（collection）的 map flatmap 和 withFilter 函数。
前一节课的对 JSON 结构的 for 查询，是建立在 JObj 是 LIST 结构，List 本身就有这几个函数

This is the basis of the Scala data base connection frameworks
ScalaQuery and Slick.
Similar ideas underly Microsoft's LINQ

其中有一处使用了 withfilter 函数
```
for (x <- e1 if f; s) yield e2
```
改写为
```
for (x <- e1.withFilter(x => f); s) yield e2
```

官方对该函数解释
```
Note: the difference between c filter p and c withFilter p is that the
former
creates a new collection, whereas the latter only restricts the domain of
subsequent map, flatMap, foreach, and withFilter operations.
```

Filter 函数接受一个 collection 和判断函数，产生一个新的 collection。而 withFilter 只在跟 map,flatmap, foreach 搭配时，利用 x = collection.withFilter(func1) 的方式，可以认为 x 其实就是做了限制的 collection，而不是一个新的 collection。当 x map func2 的时候，不会把所有的 x 都传给 map 函数。
这样做应该是编译器内部一种优化提高效率，因为用 filter 需要产生一个新的 collection，再把新 collection 传给 map。而 withFilter 是将一个有限制条件的原 collection 传给 map 等函数。

CDOE **产生随机数和随机数应用于测试**

```
/**
 * Created by LENOVO on 2016/12/10.
```

Rig

```
  */
import java.util.Random
object RandomGenerator extends App {
  //随机数产生器
  val rand = new Random//产生器
  val randInt:Int = rand.nextInt
  println(rand)
  println(randInt)

  trait Generator[+T]{
    self => //这是取 self 为“this"的别名，这个 this 指的 Generator trait 的 this
    def generate: T

    def map[S](f:T => S):Generator[S] = new Generator[S]{
      def generate = f(self.generate)
    }
    def flatMap[S](f : T => Generator[S]) : Generator[S] =
      new Generator[S]{
        def generate = f(self.generate).generate
      }
  //解释 flatMap:
  // for( x<-Gx; y<-Gy) yield f(x,y) 翻译后为
  //Gx flatMap (x => for (y <- Gy) yield f(x,y) ) for 再用 map 翻译
  //Gx flatMap (x => Gy map ( y=>f(x,y)) )
  //因此 map 函数需要把先调用 Gy 的 generate 函数产生 y，但由于返回结果必须为 Generator 类，因为 new 一个 generator 类的实例，其 generate 函数可以返回 f(Gy.generate)

  //因为 G[T] flatMap f, f 为 T => Generator[S], 首先要调用 G[T]返回一个 T。所以 flatMap 返回的 generator 类的 generate 函数有 self.generate, 接着使用 f 函数返回 Generator[S]
  }
  val integers = new Generator[Int]{
    val rand = new Random
    def generate = rand.nextInt()
  }
  println(integers)//intergers 仍是一个产生器
  println(integers.generate)//调用其 generate 函数方可得到一个对应的随机值

//  //第一种定义方式
//  val booleans = new Generator[Boolean]{
//    def generate = integers.generate > 0
//  }
//  val pairs = new Generator[(Int, Int)]{
//    def generate = (integers.generate, integers.generate)
//  }
```

```
  //第二种定义方式,我们希望更简洁的表达,如下面
  val booleans = for ( x <- integers) yield (x > 0)
  //T 和 U 是类型名

  def pairs[T,U](t: Generator[T], u: Generator[U]) = for{
    x <- t
    y <- u
  } yield (x ,y)
  val int1 = pairs(integers, integers)
  println("int1:" + int1.generate)
  println(booleans.generate)
  //利用 for 的转换,第二种定义方式可以改下为下面的形式
  //val booleans = integers map (x => x >0)
  //def pairs[T, U](t:Generator[T], u: Generator[U]) =
  // t flatMap( x => u map ( y => (x, y)))
  //因此需要的 trait Genertor 补充 map 和 flatmap 函数

//  def map[S](f: T => S): Generator[S] = new Generator[S] {
//    def generate = f(self.generate)
//  } map 函数结构跟普通的 List 的 map 函数不同，返回的是一个创造器
//  def flatMap[S](f: T => Generator[S]): Generator[S] = new Generator[S] {
//    def generate = f(self.generate).generate
//  }

  //for 的这种定义方式比较奇怪,但其目的是利用 for 语句的见解甩开冗长的 new generator 定义!!!
  //for 语句在 scala 编译器被转换为 map flatmap 函数，因此只需自定义 map 和 flatmap 即可

  def single[T](x : T): Generator[T] = new Generator[T]{
    def generate = x
  }
  def choose(lb: Int, hb: Int):Generator[Int] =
    for (x <- integers ) yield lb + x % (hb - lb)

  def oneOf[T](xs : T*):Generator [T] =
    for (idx <- choose(0, xs.length)) yield xs(idx)
  //oneOf 函数不返回值，而是通过调用 oneOf.generate 返回，一旦启用 generate 就跟
  //for 语句表示的一样，idx 就是一个 choose 返回的 int，然后 xs（idx)返回 Array xs 的元素

  //产生随机整数 list
  def lists: Generator[List[Int]] = for{
    isEmpty <- booleans
    list <- if (isEmpty) emptyLists else nonEmptyLists
  } yield list
```

```
  def emptyLists = single(Nil)
  def nonEmptyLists = for{
    head <- integers
    tail <- lists
  } yield head :: tail

  //!!!随机测试
  def test[T](g: Generator[T], numTimes: Int = 100)
             (testFunction: T => Boolean): Unit = {//Unit代表没有返回值
    for (i <- 0 until numTimes) {//限制循环次数，这个for是没有yield的
      val value = g.generate
      assert(testFunction(value), "test failed for "+value)
    }
    println("passed "+numTimes+" tests")
  }
  println("test result:\n")
  test(pairs(lists, lists)) {
    case (xs, ys) => (xs ++ ys).length > xs.length
  }
//报错：因为可能相等




}
```

对比

```
def map[S](f:T => S):Generator[S] = new Generator[S]{
    def generate = f(self.generate)
  }
  def flatMap[S](f : T => Generator[S]) : Generator[S] =
    new Generator[S]{
      def generate = f(self.generate).generate
    }
}
```

记一次 debug 过程

```
case object Empty extends Cons{
  val head:Nothing = throw new NoSuchElementException
  val tail:Nothing  = throw new NoSuchElementException
}
case class NoEmpty[T](xs: T*) extends Cons[T]{
  val head = xs(0)
  val tail:Cons[T] = xs.tail match{
    case y::ys => NoEmpty(xs.tail)
    case Nil => Empty
  }
}
abstract class Cons[+T]{
 val head:T
  val tail:Cons[T]
  def show: String = this.tail match{
    case NoEmpty(xs) => this.head.toString + "," + this.tail.show
    case Empty => ""
  }
}
val c1 = NoEmpty(1,2,3)
c1.show
```

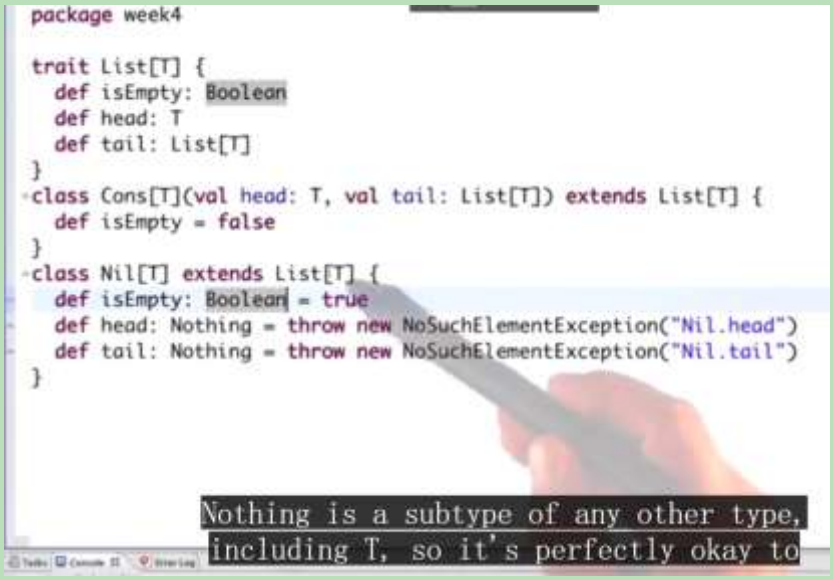
1.首先说明下因为编译内容在 worksheet,并不是 object 中，所以不能 forward reference,必须保证 Empty-NoEmpty-abstract class 的定义顺序

2. 报错： Error:(56, 8) overriding value tail in class Cons of type A$A275.this.Cons[T];
 value tail has incompatible type
  val tail = xs.tail match{
      说明 No Empty 中的 tail 有错

3. 在 val tail 后面加上:Cons[T]后，编译器直接指出了错误根源

```
case class NoEmpty[T](xs: T*) extends Cons[T]{
  val head = xs(0)
  val tail:Cons[T] = xs.tail match{
    case y::ys => NoEmpty(xs.tail)
    case Nil => Empty
  }
}
```

4. 把红线部分改为 xs(2) 编译通过，但是执行后仍会出现 matcherror。原因是（xs:T*）希望传入的是一堆用逗号分隔开的 T 类型值，而不是单独的 Array T，因此红线的地方如果改为 Xs(1),xs(2) …. Xs(n)就可以成功



成功修改

```
case object EmptyCons extends Cons[Nothing]{//这里必须给 Cons 带参数类型，实在只能给 Nothing，而如果改为 case class，编译器则不允许 case class 不带参数
可以case class Empty[T]() extends Cons[T] ，并且在 trait 中把 Empty 改为 Empty()

def isEmpty = true
  def head:Nothing = throw new NoSuchElementException
  def tail:Nothing = throw new NoSuchElementException
}
case class NoEmptyCons[T](head: T, tail: Cons[T]) extends Cons[T]{
  def isEmpty = false
}
trait Cons[+T]{
  def isEmpty:Boolean
  def head:T
  def tail:Cons[T]
  def show:String = this match{
    case NoEmptyCons(x,y) => this.head.toString + "," + this.tail.show
    case EmptyCons => ""
  }
}



val c1 = NoEmptyCons(1,NoEmptyCons(1,EmptyCons))
println(c1.show)
```

成功修改的另一个版本：注意因为 head 会 throw new Exception.则一定不能够用 val head，不然产生 Empty 实例时，会导致计算 head，引起异常，而 def 便如 call by name 一样，不会立即计算

```
case class Empty[T]() extends Cons[T]{
  def head:Nothing = throw new NoSuchElementException
  def tail:Nothing  = throw new NoSuchElementException
}
case class NoEmpty[T](xs:List[T]) extends Cons[T]{
 def head = xs(0)
 def tail:Cons[T] = xs.tail match{
    case  y::ys => NoEmpty(xs.tail)//
    case Nil => Empty()
  }
}
trait Cons[+T]{
  def head:T
  def tail:Cons[T]
  def show: String = this.tail match{
    case NoEmpty(xs) => this.head.toString + "," + this.tail.show
    case Empty() => ""
  }
}
val c1 = NoEmpty(List(1,2,3))
c1.show
```

**monad？？？

一个简单的解释 monad 概念
http://www.ruanyifeng.com/blog/2015/07/monad.html?utm_source=tuicool

. Therefore, map is in Scala a primitive function that is also defined on every monad. to qualify as a monad,

we've also seen that the left unit law itself doesn't really have a good explanation in terms of for-expressions. So that would mean that while Try is not a monad, it is still a type that can be used in a perfectly safe and reasonable way as the carrier type of a for-expression.

关于 monad 的 2 点初略总结：

首先是 monad 的定义：类似下面

```
trait M[T] {
def flatMap[U](f: T => M[U]): M[U]
```

Rig

```
}
def unit[T](x: T): M[T]
```

的结构，必须满足以下几个条件

Associativity:

```
m flatMap f flatMap g == m flatMap (x => f(x) flatMap g)
```

Left unit

```
unit(x) flatMap f == f(x)
```

Right unit

```
m flatMap unit == m
```

2. 满足这些法则的好处：

a 满足 Associativity 可以在 for 语句中嵌套

```
for (y <- for (x <- m; y <- f(x)) yield y
z <- g(y)) yield z

==

 for (x <- m;
y <- f(x)
z <- g(y)) yield z
```

b: Right unit says:

```
for (x <- m) yield x
== m
```

c: Left unit does not have an analogue for for-expressions.

**TRY

```
abstract class Try[+T]
case class Success[T](x: T) extends Try[T]
case class Failure(ex: Exception) extends Try[Nothing]
```

Try 用于在线程或计算机之间传递可能失败的计算结果

可以把计算语句 expr 封装在 expr 中

```
Try(expr) // gives Success(someValue) or Failure(someException)
```

下面是Try的实现:

```
object Try {
```

Rig

```
def apply[T](expr: => T): Try[T] =
try Success(expr)
catch {
case NonFatal(ex) => Failure(ex)
}
```

希望达到的效果：　如果 x,y 计算成功则返回 SuccessX 和 successY，再返回 Success(f(x, y))
否则返回 failure

```
for {
x <- computeX
y <- computeY
} yield f(x, y)
```

相当于：

```
computeX flatmap( x => for( y<- ComputeY) yield f(x,y)
```

再翻译 for 为 map

```
computeX flatmap( x => computeY map (y => f(x,y)) )
```

因此

```
abstract class Try[T] {
def flatMap[U](f: T => Try[U]): Try[U] = this match {
case Success(x) => try f(x) catch { case NonFatal(ex) => Failure(ex) } //sx成功则try success
case fail: Failure => fail
}
def map[U](f: T => U): Try[U] = this match {
case Success(x) => Try(f(x))
case fail: Failure => fail
}}
```

So, for a Try value t,

```
t map f == t flatMap (x => Try(f(x)))
== t flatMap (f andThen Try)
```

不理解

It turns out the left unit law fails.

```
Try(expr) flatMap f  !=  f(expr)
```

Indeed the left-hand side will never raise a non-fatal exception whereas the right-hand side will raise any exception thrown by `expr` or `f`.

Hence, `Try` trades one monad law for another law which is more useful in this context:

> *An expression composed from 'Try', 'map', 'flatMap' will never throw a non-fatal exception.*

Call this the "bullet-proof" principle.

Week2

Class1

Ppt 中的推理过程都是 structure induction

CODE of tree

```
abstract class IntSet {
def incl(x: Int): IntSet
def contains(x: Int): Boolean
def union(other: IntSet): IntSet
}
object Empty extends IntSet {
def contains(x: Int): Boolean = false
def incl(x: Int): IntSet = NonEmpty(x, Empty, Empty)
def union(other: IntSet) = other
}
case class NonEmpty(elem: Int, left: IntSet, right: IntSet) extends IntSet {
def contains(x: Int): Boolean =
if (x < elem) left contains x
else if (x > elem) right contains x
else true
def incl(x: Int): IntSet =
```

Rig

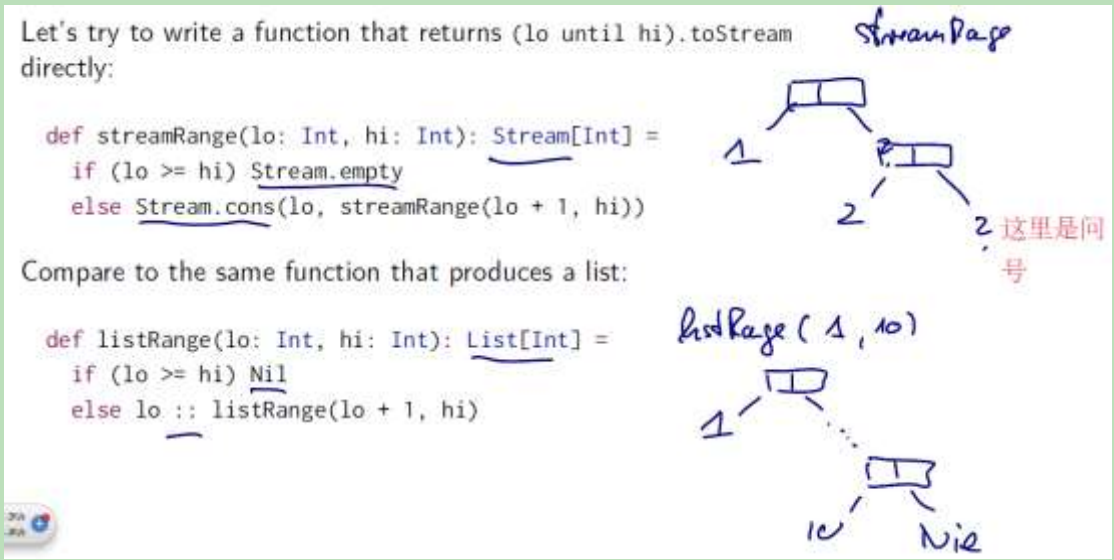
```
if (x < elem) NonEmpty(elem, left incl x, right)
else if (x > elem) NonEmpty(elem, left, right incl x)
else this

def union(other: IntSet): IntSet = (l union (r union (other))) incl x
}
```

proposition

Proposition 1: Empty contains x = false.
Proposition 2: (s incl x) contains x = true
Proposition 3: If x != y then
(xs incl y) contains x = xs contains x.
Proposition 4:
(xs union ys) contains x = xs contains x || ys contains x

Class 2

((1000 to 10000) filter isPrime)(1)
因为计算时只需要第二个元素即可，tail 的计算耗费资源
利用 Stream 结构，一种类似于 List 的 head,tail 组成，但是 tail 为 call by name,因此如果没有请求，tail 不会被计算
(1 to 1000).toStream > *res0: Stream[Int] = Stream(1, ?)*



((1000 to 10000).toStream filter isPrime)(1)
x #:: xs == Stream.cons(x, xs)

stream 的代码

```
trait Stream[+A] extends Seq[A] {
def isEmpty: Boolean
def head: A
def tail: Stream[A]
...
}

如果定义 object
object Stream {
def cons[T](hd: T, tl: => Stream[T]) = new Stream[T] {
def isEmpty = false
def head = hd
def tail = tl
}
val empty = new Stream[Nothing] {
def isEmpty = true
def head = throw new NoSuchElementException("empty.head")
def tail = throw new NoSuchElementException("empty.tail")
}
}
定义 class，关键就在于 cons 的第二个传入参数为 call by name
class Stream[+T] {
...
def filter(p: T => Boolean): Stream[T] =
if (isEmpty) this
else if (p(head)) cons(head, tail.filter(p))
else tail.filter(p)
}

streamRange 函数
def streamRange(lo: Int, hi: Int): Stream[Int] = {
print(lo+" ")
if (lo >= hi) Stream.empty
else Stream.cons(lo, streamRange(lo + 1, hi))
}
Quiz
streamRange(1, 10).take(3).toList
打印出来结果：1 2 3
```

That idea is implemented in a new class which is called the stream. Streams are similar to lists but their tail is evaluated only on demand. So, here's how it can define streams. They are, can be built from a constant Stream.empty, so the empty value and the stream object,

(1 to 1000).toStream > *res0: Stream[Int] = Stream(1, ?)*

The one major exception is ::.

x :: xs always produces a list, never a stream.

There is however an alternative operator #:: which produces a stream.

```
x #:: xs  ==  Stream.cons(x, xs)
```

#:: can be used in expressions as well as patterns.

Class 3

Lazy val

1. the proposed implementation of stream suffers from a serous potential performance problem: if tail is called several times, the corresponding stream will be recomputed each time
2. use lazy value/ (as opposed to *by-name evaluation* in the case where everything is recomputed, and *strict evaluation* for normal parameters and val definitions.)

3. Consider the following program:

```
def expr = {
  val x = { print("x"); 1 }
  lazy val y = { print("y"); 2 }
  def z = { print("z"); 3 }
  z + y + x + z + y + x
}
expr
```

xzyz

If you run this program, what gets printed as a side effect of evaluating `expr`?

0	zyxzyx	●	xzyz
0	xyzz	0	zyzz
0	something else		

Val x will be and only be evaluated as it is defined. Lazy value y will be only evaluated when first being called. Z is a def so it will be evaluated every time being called.

New definition of Cons

```
def cons[T](hd: T, tl: => Stream[T]) = new Stream[T] {
def head = hd
lazy val tail = tl
...
}
```

Store the tail once called.

Proof

Assuming apply is defined like this in stream[T].

```
def apply(n: Int): T =
if (n == 0) head
else tail.apply(n-1)
```

```
def streamRange(a:Int, b:Int): stream[Int] = {
if (a >= b) empty
else cons(a, streamRange(a + 1, b))
```

(streamRange(1000, 10000) filter isPrime) apply 1 可以得到一个 isPrime 函数

整个证明的过程重看!!! 非常详细的展示编译器的编译顺序，由函数名 filter 进行扩展

Rig

成 if elseif else 表达式语句。If(isPrime(c1.head)) 证明了 case class cons 的 head 为 def 型，filter 和 if 语句编译传递为 call by name

```
      C1.filter(isPrime).apply(1)

 -->  (if (C1.isEmpty) C1                    // by expanding filter
       else if (isPrime(C1.head)) cons(C1.head, C1.tail.filter(isPrime))
       else C1.tail.filter(isPrime))
      .apply(1)
```

配合 apply 的定义

```
     C2.filter(isPrime).apply(1)

-->  cons(1009, C2.tail.filter(isPrime)).apply(1)

-->  if (1 == 0) cons(1009, C2.tail.filter(isPrime)).head // by eval. apply
     else cons(1009, C2.tail.filter(isPrime)).tail.apply(0)
```

```
==   C3.filter(isPrime).apply(0)

-->> cons(1013, C3.tail.filter(isPrime)).apply(0)      // by eval. filter

--> 1013                                              // by eval. apply
```

Only the part of the stream necessary to compute the result has been constructed

(streamRange(1000, 10000) filter isPrime) apply 1
因此输出了 1000-10000 中的素数队列中第二个元素

Class 4

无穷序列

```
def from(n:Int):Stream[Int] = n #:: from(n + 1)
val nats = from(0)
val m4s = nats map ( _ * 4)
m4s.take(4).toList
```

利用 a #:: b 中 b 在被调用时才计算的性质 nats 表示了全体自然数

有关 Stream 的 blog

http://blog.csdn.net/cuipengfei1/article/details/40475201

stream 的源码

```
trait MyStream[+A] {
   ......
}

case object Empty extends MyStream[Nothing]

case class Cons[+A](h: () => A, t: () => MyStream[A]) extends MyStream[A]


object MyStream {

   def apply[A](elems: A*): MyStream[A] = {
      if (elems.isEmpty) empty
      else cons(elems.head, apply(elems.tail: _*))
   }

   def cons[A](hd: => A, tl: => MyStream[A]): MyStream[A] = {
      lazy val head = hd
      lazy val tail = tl
      Cons(() => head, () => tail)
   }

   def empty[A]: MyStream[A] = Empty
}
```

Filter

```
trait MyStream[+A] {

   def filter(p: A => Boolean): MyStream[A] = {
      this match {
         case Cons(h, t) =>
            if (p(h())) cons(h(), t().filter(p))
            else t().filter(p)
         case Empty => empty
      }
   }
```

if (p(h())) cons(h(), t().filter(p))
这行代码中我们又用到了小写的 cons，它所接受的参数不会被立即求值。也就是说 filter 一旦找到一个合适的元素，它就不再继续跑了，剩下的计算被延迟了。

比较值得提一下的是：这里的 h()是什么呢？h 是构造 Cons 时的第一个参数，它是什么类型的？()=>A。它就是之前提到的能够生产数据的算法，就是那个能够开出花朵的花苞。在这里我们说 h()，就是在调用这个函数来拿到它所生产的数据，就是让一个花苞开出花朵。

}

```
trait MyStream[+A] {

   . . . . . .

   def take(n: Int): MyStream[A] = {
      if (n > 0) this match {
         case Cons(h, t) if n == 1 => cons(h(), MyStream.empty)
         case Cons(h, t) => cons(h(), t().take(n - 1))
         case _ => MyStream.empty
      }
      else MyStream()
   }

   . . . . . .

}

def toList: List[A] = {
      this match {
         case Cons(h, t) => h() :: t().toList
         case Empty => Nil
      }
   }
```

注意这里小写的 cons
cons，那就意味着作为它的参数的表达式不会被立即求值，那这就意味着计算被放到了函数里，稍后再执行

**CODE 实现 Stream

Object 中可以 forward inference

Rig

```
object MyStream extends App{
//  trait MyStream[+A] extends Seq[A]{

  abstract class MyStream[+A]{//这里定义为 trait 或 abstract class 均可，但是如果 extends Seq[A]，则需要
    //在 head isEmpty 等 def 左边写上 override
    def isEmpty: Boolean
    def head: A
    def tail: MyStream[A]
    def filter(p : A => Boolean): MyStream[A] = {
      if(isEmpty) this
      else if (p(head)) cons(head, tail.filter(p))
      else tail.filter(p)
    }
    def streamRange(lb:Int, hb:Int):MyStream[Int] = {
      print(lb + "")
      if (lb >= hb) MyStream.empty
      else MyStream.cons(lb, streamRange(lb + 1, hb))

    }
    def apply(n :Int):A = {
      if (n == 0) head
      else tail.apply(n - 1)
    }
//    def #:: [T] (n : T, myStream: MyStream[T]) = MyStream.cons(n, myStream)
    def take(n : Int): MyStream[A] = {
      if (n == 0) this
      else cons(head,  tail.take(n - 1))
    }
    def toList(num:Int):List[Any] = {//有没有方法可以判断到底计算到那里
      def getList(list:List[Any], n : Int) : List[Any]= {
        if (n == 0) list
        else getList(this.apply(n) :: list, n - 1)//！！！如果不改成 List【Any】，会因为编译器判断 this.apply(n)的类型不明确
        //因此类型检查不通过
      }
       getList(Nil, num)

    }
  }

//  def from(n:Int):MyStream[Int] = n #:: from(n + 1)
  def from(n:Int):MyStream[Int] = cons(n, from(n + 1))
```

```
  def cons[T](hd: T, tl: => MyStream[T]) = new MyStream[T] {//使用 new MyStream 需要先定义了 trait MyStream
    def isEmpty = false
    def head = hd
    lazy val tail = tl//cons 为 lazy value，被调用时才会计算，且保存计算结果
  }
  val empty = new MyStream[Nothing] {
    def isEmpty = true
    def head = throw new NoSuchElementException("empty.head")
    def tail = throw new NoSuchElementException("empty.tail")
  }
  val a1 = from(0)
  val a2 = a1.take(10).toList(10)//? ? ? 不知道怎么写 toList 函数
  val a3 = a1.apply(10)
  println(a1,a2,a3)
}
```

class 5

```
// States

  type State = Vector[Int]
  val initialState = capacity map (x => 0)

// Moves

  trait Move
  case class Empty(glass: Int) extends Move
  case class Fill(glass: Int) extends Move
  case class Pour(from: Int, to: Int) extends Move

  val glasses = 0 until capacity.length

  val moves =
    (for (g <- glasses) yield Empty(g)) ++
    (for (g <- glasses) yield Fill(g)) ++
    (for (from <- glasses; to <- glasses if from != to) yield Pour(from, to))
```

Create new worksheet to see what moves is available for me.

```
package week7

object test {
  val problem = new Pouring(Vector(4, 7))
  problem.moves


}
```

Rig

```
trait Move {
  def change(state: State): State
}
case class Empty(glass: Int) extends Move {
  def change(state: State) = state updated (glass, 0)
}
case class Fill(glass: Int) extends Move {
  def change(state: State) = state updated (glass, capacity(glass))
}
case class Pour(from: Int, to: Int) extends Move {
  def change(state: State) = {
    val amount = state(from) min (capacity(to) - state(to))
    state updated (from, state(from) - amount) updated (to, state(to) + amount)
  }
}
```

```
  val glasses = 0 until capacity.length

  val moves =
    (for (g <- glasses) yield Empty(g)) ++
    (for (g <- glasses) yield Fill(g)) ++
    (for (from <- glasses; to <- glasses if from != to) yield Pour(from, to))

// Paths

  class Path(history: List[Move]) {
    def endState: State = trackState(history)
    private def trackState(xs: List[Move]): State = xs match {
      case Nil => initialState
      case move :: xs1 => move change trackState(xs1)
    }
  }
}
```

recursive explicit pattern matching solution,

simplified one

```
// Paths

 class Path(history: List[Move]) {
   def endState: State = (history foldRight initialState) (_ change _)


 }
```

```
// Paths

  class Path(history: List[Move]) {
    def endState: State = (history foldRight initialState) (_ change _)
    def extend(move: Move) = new Path(move :: history)
    override def toString = (history.reverse mkString " ") + "--> " + endState
  }

  val initialPath = new Path(Nil)
```

Find solution

Rig

```
val initialPath = new Path(Nil)

def from(paths: Set[Path]): Stream[Set[Path]] =
  if (paths.isEmpty) Stream.empty
  else {
    val more = for {
      path <- paths
      next <- moves map path.extend
    } yield next
    paths #:: from(more)
  }

val pathSets = from(Set(initialPath))

def solutions(target: Int): Stream[Path] =
  for {
    pathSet <- pathSets
    path <- pathSet
    if path.endState contains target
  } yield path
```

```
bject test {
val problem = new Pouring(Vector(4, 9))
problem.moves


problem.solutions(6)
```

```
> problem  : week7.Pourir
> res0: scala.collection.
| ek7.test.problem.Move]
| 1), Pour(1,0))/
```

takes a long time

Exclude explored states

```
def from(paths: Set[Path], explored: Set[State]): Stream[Set[Path]] =
  if (paths.isEmpty) Stream.empty
  else {
    val more = for {
      path <- paths
      next <- moves map path.extend
      if !(explored contains next.endState)
    } yield next
    paths #:: from(more, explored ++ (more map (_.endState)))
  }

val pathSets = from(Set(initialPath), Set(initialState))
```

Endstate method is repeatedly calculated

```
class Path(history: List[Move], val endState: State) {
  def extend(move: Move) = new Path(move :: history, move change endState)
  override def toString = (history.reverse mkString " ") + "--> " + endState
}

val initialPath = new Path(Nil, initialState)
```

, think of what might change in the future and how you could encapsulate this, the implementations from clients so that you could change it in the future without changing any of the client code.

Rig

CODE 枚举所有可能的操作

```
class Pouring(capacity: Vector[Int]) {
    //States
    type State = Vector[Int]
    val initialState = capacity map (x => 0)

    //Moves
    trait Move
    case class Empty(glass: Int) extends Move
    case class Fill(glass: Int) extends Move
    case class Pour(from:Int, to: Int) extends Move

    val glasses  = 0 until capacity.length

    val moves =
      (for (g <- glasses ) yield Empty(g)) ++
        (for (g <- glasses) yield Fill(g)) ++
        (for (from <- glasses; to <- glasses if from != to) yield Pour(from, to))
  }
//  object test extends App{
    val problem = new Pouring(Vector(4, 7))
    println(problem.moves)//所有的 move 操作集合
```

CODE 初步求解

```
class Pouring(capacity: Vector[Int]) {
    //States
    type State = Vector[Int]
    val initialState = capacity map (x => 0)

    //Moves
    trait Move{
      def change(state: State):State
    }
    case class Empty(glass: Int) extends Move{
      def change(state: State) = state updated (glass, 0)
    }
    case class Fill(glass: Int) extends Move{
      def change(state: State) = state updated (glass, capacity(glass))
    }
```

```
    case class Pour(from:Int, to: Int) extends Move{
      def change(state: State) = {
        val amount = state(from) min (capacity(to) - state(to))//min 是 int 的成员函数
        state updated (from, state(from) - amount) updated (to, state(to) + amount)
      }
    }

    val glasses  = 0 until capacity.length

    val moves =
      (for (g <- glasses ) yield Empty(g)) ++
        (for (g <- glasses) yield Fill(g)) ++
        (for (from <- glasses; to <- glasses if from != to) yield Pour(from, to))
    //PatHs
    class Path(history: List[Move]){
//      def endState: State = trackState(history)
//      private def trackState(xs: List[Move]): State = xs match{
//        case Nil => initialState
//        case move :: xs1 => move change trackState(xs1)//move 为主体 change 是 move 的
成员函数，trackState(xs1)生成 State 作为 change 的参数
    //更简洁的 endState 写法：
      def endState: State = (history foldRight initialState)(_ change _)
      def extend(move: Move) = new Path(move :: history)
      override def toString = (history.reverse mkString " ") + "-->" + endState
    }
   val initialPath = new Path(Nil)
   def from(paths: Set[Path]): Stream[Set[Path]] = {
     if (paths.isEmpty) Stream.empty
     else {
       val more = for{
         path <- paths
         next <- moves map path.extend//extend 接受单一 Move,这是把所有可能的 move 都加给
path
       } yield next
       paths #:: from(more)//??
     }

   }
   val pathSets = from(Set(initialPath))

   def solutions(target: Int):Stream[Path] = {
     for {
       pathSet <- pathSets
       path <- pathSet
```

```
      if path.endState contains target
    } yield path
  }

}
//  object test extends App{
    val problem = new Pouring(Vector(4, 9))
    println(problem.moves)//所有的 move 操作集合
    problem.solutions(6)//take a long time to go
```

CODE 剪枝板

```
class Pouring(capacity: Vector[Int]) {
    //States
    type State = Vector[Int]
    val initialState = capacity map (x => 0)

    //Moves
    trait Move{
      def change(state: State):State
    }
    case class Empty(glass: Int) extends Move{
      def change(state: State) = state updated (glass, 0)
    }
    case class Fill(glass: Int) extends Move{
      def change(state: State) = state updated (glass, capacity(glass))
    }
    case class Pour(from:Int, to: Int) extends Move{
      def change(state: State) = {
        val amount = state(from) min (capacity(to) - state(to))//min 是 int 的成员函数
        state updated (from, state(from) - amount) updated (to, state(to) + amount)
      }
    }

    val glasses  = 0 until capacity.length

    val moves =
      (for (g <- glasses ) yield Empty(g)) ++
        (for (g <- glasses) yield Fill(g)) ++
        (for (from <- glasses; to <- glasses if from != to) yield Pour(from, to))
    //PatHs
    class Path(history: List[Move], val endState: State){
```

```
//      def endState: State = trackState(history)
//      private def trackState(xs: List[Move]): State = xs match{
//        case Nil => initialState
//        case move :: xs1 => move change trackState(xs1)//move 为主体 change 是 move 的
成员函数，trackState(xs1)生成 State 作为 change 的参数
    //更简洁的 endState 写法：
//      def endState: State = (history foldRight initialState)(_ change _)//sta
      def extend(move: Move) = new Path(move :: history, move change endState)
      override def toString = (history.reverse mkString " ") + "-->" + endState
    }
   val initialPath = new Path(Nil, initialState)
   def from(paths: Set[Path], explored: Set[State]): Stream[Set[Path]] = {
     if (paths.isEmpty) Stream.empty
     else {
       val more = for{
         path <- paths
         next <- moves map path.extend//extend 接受单一 Move,这是把所有可能的 move 都加给
path
       } yield next
       paths #:: from(more, explored ++ (more map (_.endState)))//??
     }
   }

   val pathSets = from(Set(initialPath), Set(initialState))

   def solutions(target: Int):Stream[Path] = {
     for {
       pathSet <- pathSets
       path <- pathSet
       if path.endState contains target
     } yield path
   }
 }
//  object test extends App{
    val problem = new Pouring(Vector(4, 7))
    println(problem.moves)//所有的 move 操作集合
    problem.solutions(5)//take a long time to go
```

week3

class 1

substitution law: things
function f with n parameters x1 to xn and body b.
you have a call of the same function f with actual values v1 to vn. Then the program can be: keeping the application and all the other program elements. But replacing the call by the body of the function b.

$$\begin{aligned} & \text{def } f(x_1, \ldots, x_n) = B; \ \ldots \ f(v_1, \ldots, v_n) \\ \rightarrow \\ & \text{def } f(x_1, \ldots, x_n) = B; \ \ldots \ [v_1/x_1, \ldots, v_n/x_n]\, B \end{aligned}$$

Lambda calculus

Lambda calculus (also written as λ-calculus) is a formal system in mathematical logic for expressing computation based on function abstraction and application using variable binding and substitution.

Computable functions are a fundamental concept within computer science and mathematics. The λ-calculus provides a simple semantics for computation, enabling properties of computation to be studied formally. The λ-calculus incorporates two simplifications that make this semantics simple. The first simplification is that the λ-calculus treats functions "anonymously", without giving them explicit names. For example, the function

$$\mathbf{square\_sum}(x, y) = x^2 + y^2$$

can be rewritten in *anonymous form* as

$$(x, y) \mapsto x^2 + y^2$$

(read as "the pair of x and y is mapped to $x^2 + y^2$"). Similarly,

$$\mathrm{id}(x) = x$$

can be rewritten in anonymous form as $x \mapsto x$, where the input is simply mapped to itself.

The second simplification is that the λ-calculus only uses functions of a single input. An ordinary function that requires two inputs, for instance the **square_sum** function, can be reworked into an equivalent function that accepts a single input, and as output returns *another* function, that in turn accepts a single input. For example,

$$(x, y) \mapsto x^2 + y^2$$

can be reworked into

$$x \mapsto (y \mapsto x^2 + y^2)$$

This method, known as currying, transforms a function that takes multiple arguments into a chain of functions each with a single argument.

Function application of the **square_sum** function to the arguments (5, 2), yields at once

$$\begin{aligned}&((x, y) \mapsto x^2 + y^2)(5, 2)\\&= 5^2 + 2^2\\&= 29,\end{aligned}$$

whereas evaluation of the curried version requires one more step

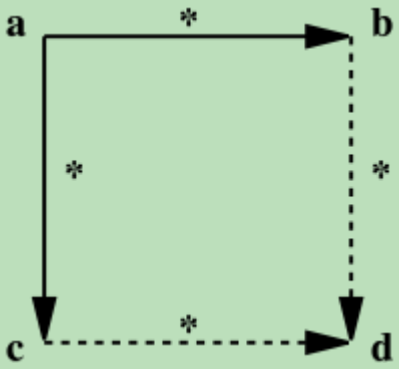
$$\begin{aligned}&((x \mapsto (y \mapsto x^2 + y^2))(5))(2)\\&= (y \mapsto 5^2 + y^2)(2)\\&= 5^2 + 2^2\\&= 29\end{aligned}$$

to arrive at the same result.

Church-rosser theorem

In mathematics and theoretical computer science, the Church–Rosser theorem states that, when applying reduction rules to terms in the lambda calculus, the ordering in which the reductions are chosen does not make a difference to the eventual result.

The theorem is symbolized by the diagram at right: if term a can be reduced to both b and c, then there must be a further term d (possibly equal to either b or c) to which both b and c can be reduced.



Viewing the lambda calculus as an abstract rewriting system, the Church–Rosser theorem states that the reduction rules of the lambda calculus are confluent.

Example:

```
def iterate(n: Int, f: Int => Int, x: Int) =
if (n == 0) x else iterate(n-1, f, f(x))
def square(x: Int) = x * x
```

Then the call iterate(1, square, 3) gets rewritten as follows:

```
// if (1 == 0) 3 else iterate(1-1, square, square(3))

// iterate(0, square, square(3))

// iterate(0, square, 3 * 3)

// iterate(0, square, 9)

// if (0 == 0) 9 else iterate(0-1, square, square(9)) // 9
```

Or in this way

```
if (1 == 0) 3 else iterate(1 - 1, square, 3 * 3)
```

Stateful object

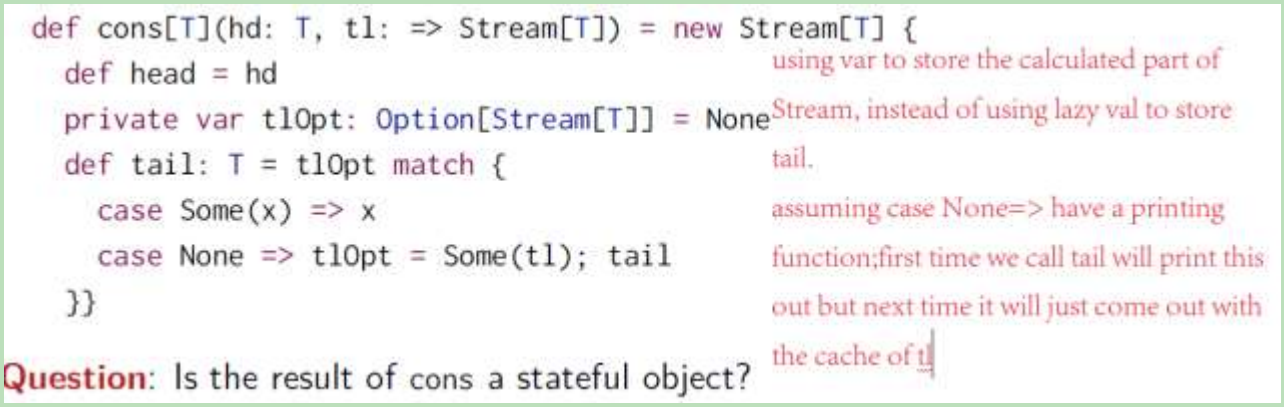
An object has a state if its behavior is influenced by its history.

Example:

```
class BankAccount {
private var balance = 0
def deposit(amount: Int): Unit = {
if (amount > 0) balance = balance + amount
}
def withdraw(amount: Int): Int =
```

```
if (0 < amount && amount <= balance) {
balance = balance - amount
balance
} else throw new Error("insufficient funds")
}
val account = new BankAccount // account: BankAccount = BankAccount
account deposit 50 //
account withdraw 20 // res1: Int = 30
account withdraw 20 // res2: Int = 10
account withdraw 15 // java.lang.Error: insufficient funds
```

QUIZ



If you hesitated in your answer, I don't blame you. Because in fact, in a sense, both the yes and the no are valid responses, depending on what assumptions you make on the rest of your system. One common assumption is that streams should only be defined over purely functional computations. So, the tail operation here should not have a side effect**(functional programming using immutable variable has no side effect, one important consequence of this is that we will see that the concept of time wasn't important, C-R theorem make this sure in Lambda calculus substitution law!!!)**. In that case, the **optimization to cache the first value of tlOPt** and reuse it on all previous calls to tail is purely a optimization that avoids computations, but that *does not have an observable effect outside the class of streams*. So the answer would be that, no streams are not stateful objects.

On the other hand, if you allow side effect in computations for tail, **let's say tail could have a printing statement**, then you would see that the second time tail is caught in this string. It would **come straight out of the cache**, so there would be no side effect performed. Whereas, **the first time**, it would **be called the operation would be performed, including the printing statement**. So that means clearly the operation tail depends on the previous history of the object. It would be different depending on whether a previous tail was performed or not. So in that sense, the answer would be cons is a stateful object, provided that you also allow imperative side effect in computations for tail.

Rig

```
class BankAccountProxy(ba: BankAccount) {
  def deposit(amount: Int): Unit = ba.deposit(amount)
  def withdraw(amount: Int): Int = ba.withdraw(amount)
}
```

Question: Are instances of BankAccountProxy stateful objects?

0 Yes

0 No

yes, even if it does not use var, its output and content depend on the history

Class 2

operational equivalence.

val x = new BankAccount
val y = new BankAccount
Question: Are x and y the same?
The precise meaning of "being the same" is defined by the property of ***operational equivalence***.
Informal statement:

Suppose we have two definitions x and y. x and y are operationally equivalent if *no possible test* can distinguish between them.

Testing :
To test if x and y are the same, we must
- Execute the definitions followed by an arbitrary sequence of operations that involves x and y, observing the possible outcomes.

val x = new BankAccount	val x = new BankAccount
val y = new BankAccount	val y = new BankAccount
f(x, y)	f(x, x)

- Then, execute the definitions with another sequence S' obtained by renaming all occurrences of y by x in S
- If the results are different, then the expressions x and y are certainly different.
- On the other hand, if all possible pairs of sequences (S, S') produce the same result, then x and y are the same.

Result:

```
val x = new BankAccount
val y = new BankAccount
x deposit 30 // val res1: Int = 30
y withdraw 20 // java.lang.Error: insufficient funds

val x = new BankAccount
val y = new BankAccount
x deposit 30 // val res1: Int = 30
x withdraw 20 // val res2: Int = 10
```

conclusion

So once we allow for assignments, the two formulations above, `val x = new BankAccount` and `val y = new BankAccount` , don't mean the same thing anymore.(no substitution)

So, clearly going from here to here is not a valid step, and that means that the Substitution Model, as a whole, stops being valid once we add assignment to a language.

CODE bankAccount

```
class BankAccount{
   private var balance = 0
   def deposit(amount: Int): Unit = {
      if (amount >0) balance = balance + amount
   }
   def withdraw(amount: Int):Int =
      if (0 < amount && amount <= balance){
         balance = balance - amount
         balance
      } else throw new Error("insufficient funds")
}

val x = new BankAccount
val y = x
val z = new BankAccount
def f(a: BankAccount, b : BankAccount): (Int, Int) = {
   a.deposit(50)
   b.deposit(50)
```

Rig

```
    (a.withdraw(10), b.withdraw(10))
}
f(x,z)
f(x,y)//this suggests that y is another name for x
```

class 3

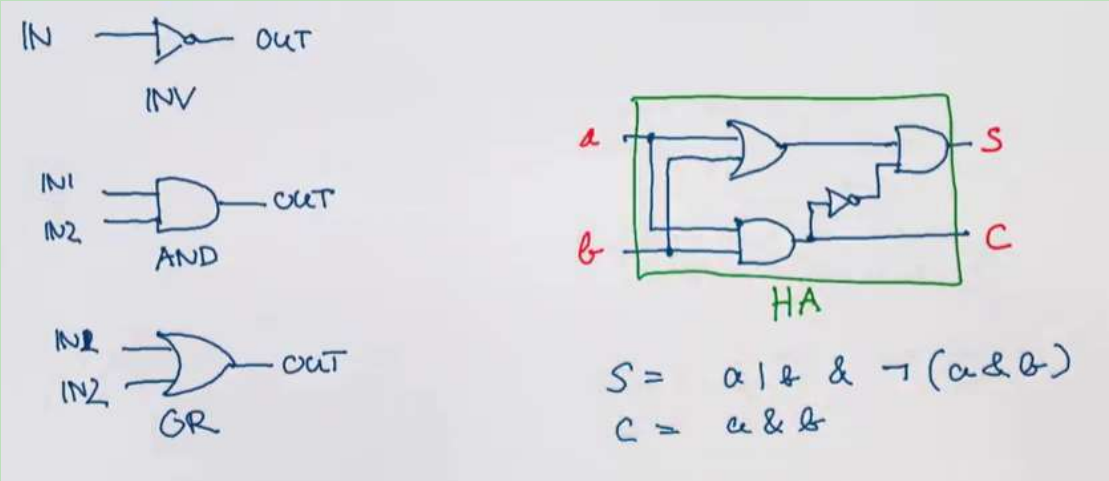
while-loop

```
def power (x: Double, exp: Int): Double = {
var r = 1.0
var i = exp
while (i > 0) { r = r * x; i = i - 1 }
r
}
```

While function can be defined as followd:

```
def WHILE(condition: => Boolean)(command: => Unit): Unit =
if (condition) {
command
WHILE(condition)(command)
}
else ()
```

class 4

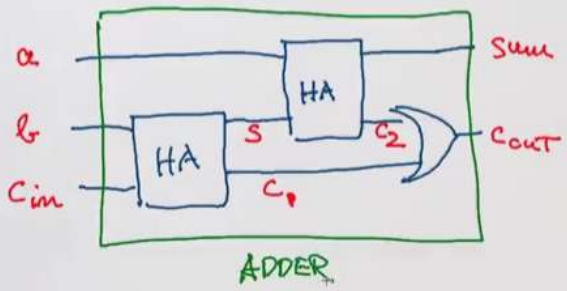


Rig

More Components

This half-adder can in turn be used to define a full adder:

```
def fullAdder(a: Wire, b: Wire, cin: Wire, sum: Wire, cout: Wire): Unit = {
  val s = new Wire
  val c1 = new Wire
  val c2 = new Wire
  halfAdder(b, cin, s, c1)
  halfAdder(a, s, sum, c2)
  orGate(c1, c2, cout)
}
```



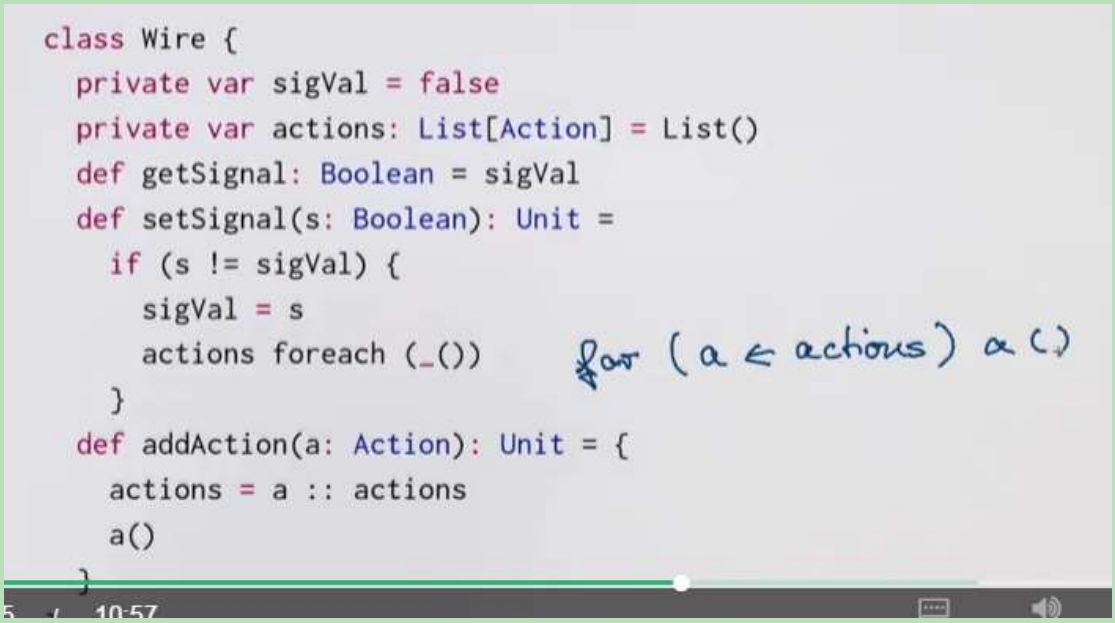
CODE 实现门电路

```
val a = new Wire; val b = new Wire; val c = new Wire
or, equivalently:
val a, b, c = new Wire

def inverter(input: Wire, output: Wire): Unit
def andGate(a1: Wire, a2: Wire, output: Wire): Unit
def orGate(o1: Wire, o2: Wire, output: Wire): Unit

def halfAdder(a: Wire, b: Wire, s: Wire, c: Wire): Unit = {
val d = new Wire
val e = new Wire
orGate(a, b, d)
andGate(a, b, c)
inverter(c, e)
andGate(d, e, s)
}

def fullAdder(a: Wire, b: Wire, cin: Wire, sum: Wire, cout: Wire): Unit = {
val s = new Wire
val c1 = new Wire
val c2 = new Wire
halfAdder(b, cin, s, c1)
halfAdder(a, s, sum, c2)
orGate(c1, c2, cout)
}
```

Rig

Class 5

So everything in action do, it does by its side side effect.



What happens if we compute `in1Sig` and `in2Sig` inline inside `afterDelay` instead of computing them as values?

```
def orGate2(in1: Wire, in2: Wire, output: Wire): Unit = {
  def orAction(): Unit = {
    afterDelay(OrGateDelay) {
      output setSignal (in1.getSignal | in2.getSignal) }
    }
  in1 addAction orAction
  in2 addAction orAction
}
```

orGate will wait for the completeness of getSignal and then go to orGateDelay, while orGate2 getSignal take places after the orGateDelay

0 'orGate' and 'orGate2' have the same behavior.

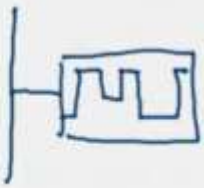
0 'orGate2' does not model OR gates faithfully.

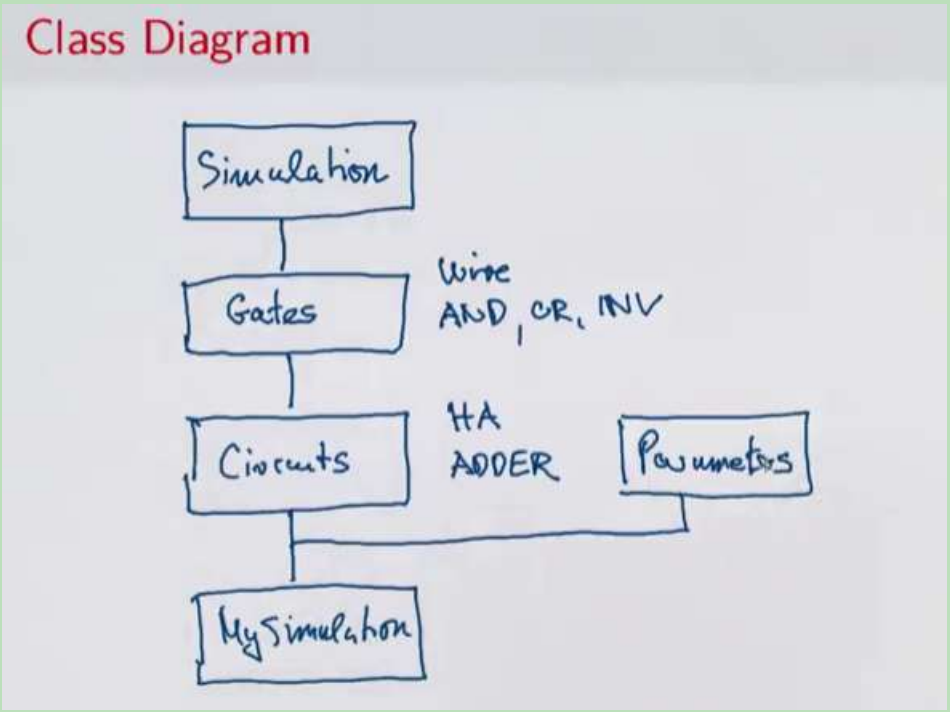
Rig

Class 6

```
def probe(name: String, wire: Wire): Unit = {
  def probeAction(): Unit = {
    println(s"$name $currentTime value = ${wire.getSignal}")
  }
  wire addAction probeAction
}
```

name + " " + currentTime + " value = " + wire.getSignal





```
package week2

abstract class Simulation {

  type Action = () => Unit

  case class Event(time: Int, action: Action)

  private var curtime = 0
  def currentTime: Int = curtime

  private var agenda: List[Event] = List()

  private def insert(ag: List[Event], item: Event): List[Event] = ag match {
    case first :: rest if first.time <= item.time => first :: insert(rest, item)
    case _ => item :: ag
  }

  def afterDelay(delay: Int)(block: => Unit): Unit = {
    val item = Event(currentTime + delay, () => block)
    agenda = insert(agenda, item)
  }
```

Block 是什么

```
abstract class Gates extends Simulation {

  def InverterDelay: Int
  def AndGateDelay: Int
  def OrGateDelay: Int

  class Wire {

    private var sigVal = false
    private var actions: List[Action] = List()

    def getSignal = sigVal

    def setSignal(s: Boolean) =
      if (s != sigVal) {
        sigVal = s
        actions foreach (_())
      }

    def addAction(a: Action) = {
```

Rig

```
    def addAction(a: Action) = {
      actions = a :: actions
      a()
    }
  }

  def inverter(input: Wire, output: Wire): Unit = {
    def invertAction(): Unit = {
      val inputSig = input.getSignal
      afterDelay(InverterDelay) {
        output setSignal !inputSig
      }
    }
    input addAction invertAction
  }

  def andGate(in1: Wire, in2: Wire, output: Wire) = {
    def andAction() = {
      val in1Sig = in1.getSignal
```

```
def andGate(in1: Wire, in2: Wire, output: Wire) = {
  def andAction() = {
    val in1Sig = in1.getSignal
    val in2Sig = in2.getSignal
    afterDelay(AndGateDelay) {
      output setSignal (in1Sig & in2Sig)
    }
  }
  in1 addAction andAction
  in2 addAction andAction
}

/** Design orGate analogously to andGate */
def orGate(in1: Wire, in2: Wire, output: Wire): Unit = {
  def orAction() = {
    val in1Sig = in1.getSignal
    val in2Sig = in2.getSignal
    afterDelay(OrGateDelay) {
      output setSignal (in1Sig | in2Sig)
    }
  }
```

```
  in1 addAction orAction
  in2 addAction orAction
}

def probe(name: String, wire: Wire): Unit = {
  def probeAction(): Unit = {
    println(name + " " + currentTime + " new-value = " + wire.getSignal)
  }
  wire addAction probeAction
}
```

Rig

```scala
package week2

abstract class Circuits extends Gates {

  def halfAdder(a: Wire, b: Wire, s: Wire, c: Wire) {
    val d, e = new Wire
    orGate(a, b, d)
    andGate(a, b, c)
    inverter(c, e)
    andGate(d, e, s)
  }

  def fullAdder(a: Wire, b: Wire, cin: Wire, sum: Wire, cout: Wire) {
    val s, c1, c2 = new Wire
    halfAdder(a, cin, s, c1)
    halfAdder(b, s, sum, c2)
    orGate(c1, c2, cout)
  }
}
```

```scala
package week2

trait Parameters {
  def InverterDelay = 2
  def AndGateDelay = 3
  def OrGateDelay = 5
}
```

```scala
package week2

object test {
  println("Welcome to the Scala worksheet")          > Welcome to the Scala worksheet
  object sim extends Circuits with Parameters
  import sim._
  val in1, in2, sum, carry = new Wire                > in1  : week2.test.sim.Wire = week2.Gc
                                                     | in2  : week2.test.sim.Wire = week2.Gc
                                                     | sum  : week2.test.sim.Wire = week2.Gc
                                                     | carry  : week2.test.sim.Wire = week2.

  halfAdder(in1, in2, sum, carry)
  probe("sum", sum)                                  > sum 0 new-value = false
  probe("carry", carry)                              > carry 0 new-value = false

  in1 setSignal true
  run()                                              > *** simulation started, time = 0 ***
                                                     | sum 8 new-value = true
}
```

this is a new created worksheet, carry is not changed so that is no printing about carry

```scala
in1 setSignal true
run()                                                > *** simulation started, time = 0 ***
                                                     | sum 8 new-value = true

in2 setSignal true
run()                                                > *** simulation started, time = 8 ***
                                                     | carry 11 new-value = true
                                                     | sum 16 new-value = false

in1 setSignal false
run()                                                > *** simulation started, time = 16 ***
                                                     | carry 19 new-value = false
                                                     | sum 24 new-value = true
```

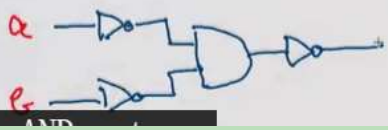
Rig

A Variant

An alternative version of the OR-gate can be defined in terms of AND and INV.

```
def orGateAlt(in1: Wire, in2: Wire, output: Wire): Unit = {
  val notIn1, notIn2, notOut = new Wire
  inverter(in1, notIn1); inverter(in2, notIn2)
  andGate(notIn1, notIn2, notOut)
  inverter(notOut, output)
}
```

$a \mid b = \neg(\neg a \,\&\, \neg b)$




```
halfAdder(in1, in2, sum, carry)
probe("sum", sum)                    > sum 0 new-value = false
probe("carry", carry)                > carry 0 new-value = false

in1 setSignal true
run()                                > *** simulation started, time = 0 ***
                                     | sum 5 new-value = true
                                     | sum 10 new-value = false
                                     | sum 10 new-value = true
in2 setSignal true
run()                                > *** simulation started, time = 10 ***
                                     | carry 13 new-value = true
                                     | sum 18 new-value = false
in1 setSignal false
run()                                > *** simulation started, time = 18 ***
                                     | carry 21 new-value = false
                                     | sum 26 new-value = true
}
```

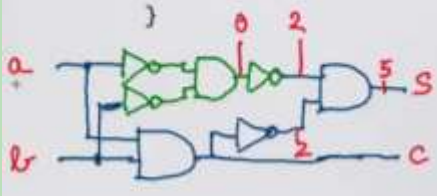
after changing the definition of OrGate we rerun the simulation

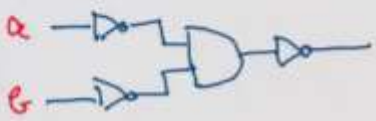
A Variant

An alternative version of the OR-gate can be defined in terms of AND and INV.

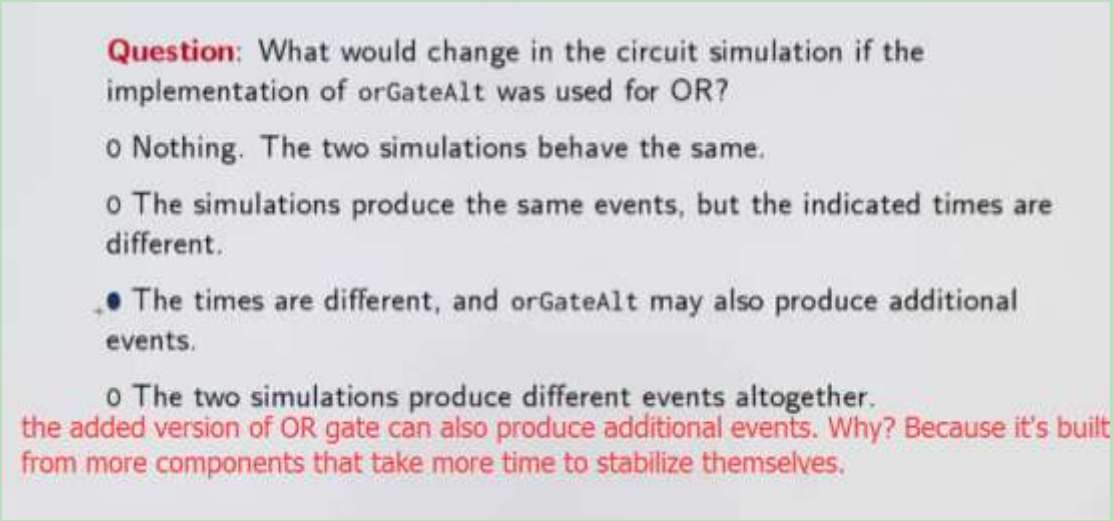
```
def orGateAlt(in1: Wire, in2: Wire, output: Wire): Unit = {
  val notIn1, notIn2, notOut = new Wire
  inverter(in1, notIn1); inverter(in2, notIn2)
  andGate(notIn1, notIn2, notOut)
  inverter(notOut, output)
}
```

$a \mid b = \neg(\neg a \,\&\, \neg b)$





Rig



Summary：

So, to summarize, we've seen that adding state and assignments makes our mental model of computation more complicated. In particular, we lose the property of referential transparency which says that it doesn't matter whether we use a named or the thing it refers to. We've seen with the bank account example that it matters quite a lot whether we refer to it an existing bank account, or we create a new one. The other thing that we lose is the substitution model so we do not have anymore an easy way to trace computations by rewriting. On the other hand, assignments allow us to formulate some programs in an elegant and concise way. We've seen that with the example of discrete events simulation where a system was represented by a list of action. And that list was a mutable variable. It changed during the time of simulations. The effect of the actions when they're called would in turn change the state of objects, and they could also install other actions to be executed in the future.

17:26

You've seen that in this way combining higher functions and assignments in state led to some very, very powerful techniques that let you express fundamentally complex computations in a concise and understandable way. So in the end it's a trade off. You get more expressiveness that helps you tackle certain problems in a simpler way, but on the other hand you loose tools for reasoning about your program, referential transparency, and the substitution model. So I guess the moral would be that you should stick to the purely functional model whenever you can, and you should use state responsibly when you must.

函数分析

```
def afterDelay(delay: Int)(block : =>Unit) : Unit ={
   val item = Event(currentTime + delay, () => block)
   agenda = insert(agenda, item)
}
```

Rig

Referential transparency

As you might know, the term "referentially transparent" means that the value of expression can depend only on the values of its parts, and not on any other facts about them.
For example, it cannot depend on the following:

- Whether some part of expression is already evaluated or not (in a lazy language)
- Whether two equal values are shared (are pointers to the same location in memory) or not
- Whether a data structure is cyclic (i.e. its pointers create a cycle) or not

All those facts about the current state of the program are either true or false, but no expression can change its value depending on them. So those things are called non-observable.
This webcomic and its discussion on reddit might enlighten you as well.

下面我们谈谈函数副作用的问题，也称边界问题，降低甚至消灭副作用已经成为我们编程的一个重要原则，不要让一个方法或函数执行主要功能时，会产生其他意想不到的次要功能，这些额外功能不是我们要求的，称为副作用。

蒯因又称奎因，美国哲学家，逻辑学家，逻辑实用主义的代表，与罗素齐名，强调系统的、结构式的哲学分析，主张把一般哲学问题置于一个系统的语言框架内进行研究。

蒯因从逻辑的观点出发，把语言分析作为哲学研究的核心内容，这就为哲学的语言转向划上了一个圆满的句号：自蒯因哲学起，当哲学家开始讨论哲学问题的时候，都需要首先考察所要讨论的命题的意义，而不会像传统哲学家那样依然专注于对世界存在的思考。

蒯因强调以严格精确的形式语言表达我们的思想，蒯因这一逻辑应用在我们编程语言中，演化成了各种面向对象或面向函数编程的原则。

下面我们谈谈函数副作用的问题，也称边界问题，降低甚至消灭副作用已经成为我们编程的一个重要原则，不要让一个方法或函数执行主要功能时，会产生其他意想不到的次要功能，这些额外功能不是我们要求的，称为副作用。

更严格地消灭副作用方式是引用透明(referentially transparent)，这是面向函数范式 FP 中的一个术语，也是 FP 语言如 Scala 等相比 Java 特色所在。

为什么他们称为引用透明 Referentially transparent 一文中谈到：

“引用透明”(指称透明)这个概念来自于蒯因(Quine)的关于自然语言哲学。看看下面语句：BanQ 比老鼠大， 如果把"BanQ"去除，留个空白在那里，显然变成：“??? 比老鼠大”。哲学家认为这是一个 Context 场景，通过填补场景中的空白，你得到了这个语句真实含义。

Rig

再次想一下北京这个城市，它是一个 thing，有许多方式可以指称或引用(refer)它，“中国首都”或“在经纬 XXX 度的城市”。

因此，“引用透明”(referentially transparent)中的“引用 referentially”这个词语的含义是在谈论这样一个事实：北京这个事物(thing)有很多名称，有很多途径来表达指称这个事物。

“引用透明”(referentially transparent)中“透明 transparent”含义是“没有什么不同”，“都是一样”。

蒯因认为： 如果我们能够用不同的名称替代上面语句中的空白，但是没有改变任何意思，总是得到同样的结果，那么这个场景 Context 就是“引用透明”的。

如：
[北京]比老鼠大 是真；
[中国首都]比老鼠大 是真
[在经纬 XXX 度的城市]比老鼠大 是真

这些命题的真正含义不会因为我们选择不同的引用或指称名称而受到影响。我们就可以认为场景"??? 比老鼠大"是引用透明的。

与引用透明相反，蒯因认为存在引用不透明(REFERENTIALLY OPAQUE)，名称的改变对于整个语句意思非常重要。

比如：??? 有 9 个字符。那么：[北京]有 9 个字符 是伪的。

这样场景“??? 有 9 个字符” 的含义因为名称指称(name/refernce)不同而不同。

罗素 蒯因等代表的形式逻辑是数学或编程语言的元语言，在数学中，所有函数都是引用透明的，比如 sin(x)，无论如何调用，结果只依赖 x 值，或者说：对于一个特定的 x 值，无论怎么调用这个函数得到都是同一个结果。

因为国内很多人只知道数学，而不知道数学后面的哲学与形式逻辑，就把我们编程语言中的这些约束和数学划上关系，实际上真正有联系的是数学背后的形式逻辑，或称符号逻辑。

对于编程语言，引用透明的含义可以表达为：函数的输出只依赖于其输入，引用透明就没有副作用，也无需指定前后运行顺序，比如：

```
int getResult(x){
    return x;
}
```

是一个引用透明无副作用的方法函数，而：

```
int getResult(x){
    return x + g;
}
```

则是引用不透明，因为 getResult 结果不但依赖 x，还依赖 g 值。

那么 g 值是什么呢？一般是这个方法所在类的属性字段，如完整是如下：

```
public Class A{
   int g;

   int getResult(x){
       return x + g;
   }
}
```

很显然，含有属性状态 g 的类 A，我们称为有状态类，也就是说：有状态的类都不是引用透明，都是引用不透明的。

引用透明概念由于本源起源于形式逻辑，而数学也建立在形式逻辑基础上，很显然，关系代数中的幂等函数含义与引用透明概念是一致的。

引用透明或幂等函数都是多次调用都会得到同样的结果，而且这个结果只依赖输入，不依赖其他值，这就为我们进行并行或异步计算提供了可能，也会分布式架构的粒度切分提供了可能，我们如果把一个引用不透明的有状态服务切分为引用透明无状态服务，那么无疑性能和可扩展性 Scalable 大大提高了。

引用透明虽然是编程范式中一个原则，实际也是性能设计可扩展性中的一个原则，这就完美体现了良好的性能和良好的编程习惯是密切不可分的。

Pure function

In computer programming, a function may be considered a **pure function** if both of the following statements about the function hold:

1. The function always evaluates the same result value given the same argument value(s). The function result value cannot depend on any hidden information or state that may change while program execution proceeds or between different executions of the program, nor can it depend on any external input from I/O devices (usually—see below).
2. Evaluation of the result does not cause any semantically observable side effect or output,

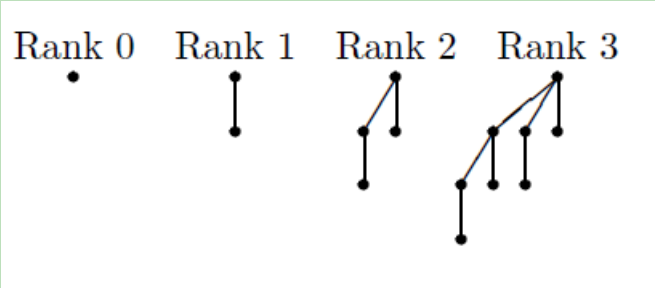
such as mutation of mutable objects or output to I/O devices (usually—see below).
满足下面两点的为纯函数：
第一点即纯函数不能有状态属性，
第二点即不能改变可变变量和不能在 io 有输出（non-observable）

Wiki
In functional programming only referentially transparent functions are considered.!!!

HW

Binomial queue 二项队列

http://www.cnblogs.com/hapjin/p/5468817.html



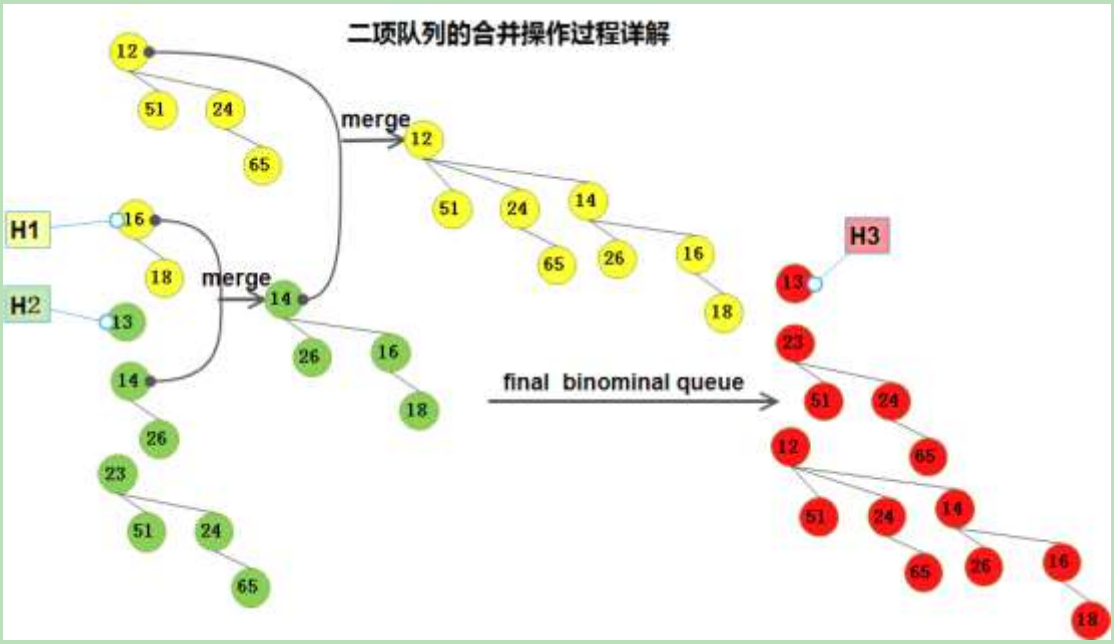
1）一个二项队列是若干棵树的集合（如下面图中 H2 有两棵树）。也就是说，二项队列不仅仅是一棵树，而是多棵树，并且每一棵树都遵守堆序的性质，所谓堆序的性质，就是指每个结点都比它的左右子树中结点要小（小顶堆）。这棵树称为“二项树”

2）二项队列中的树高度不同，一个高度至多存在一棵二项树。将高度为 0 的二项树记为 B(0)，高度为 k 的二项树记为 B(k)

譬如 21 is 10101, and the binomial queue contains trees of ranks 0, 2, and 4 (of sizes 1, 4, and 16, respectively

①merge 操作
merge 操作是合并二个二项队列，合并二项队列过程中需要合并这两个二项队列中 高度相同的二项树(后面的 combineTrees()方法)
假设需要合并二项队列 H(1)和 H(2)，合并后的结果为 H(3)。合并两个二项队列的过程如下：
a)寻找 H(1)和 H(2)中高度相同的二项树，调用 combineTrees()合并这两颗二项树
b)重复 a) 直至树中不再有高度相同的二项树为止
应该要从 B（rank）rank 小的树开始合并

Rig



②insert 操作可以看作是特殊的合并操作。新的元素作为一棵高度为 0 的 rank0 二项树插入，即 rhs 二项队列中只有一棵高度为 0 的二项树。

③We can find this minimum element in O(log n) time by scanning through the roots.

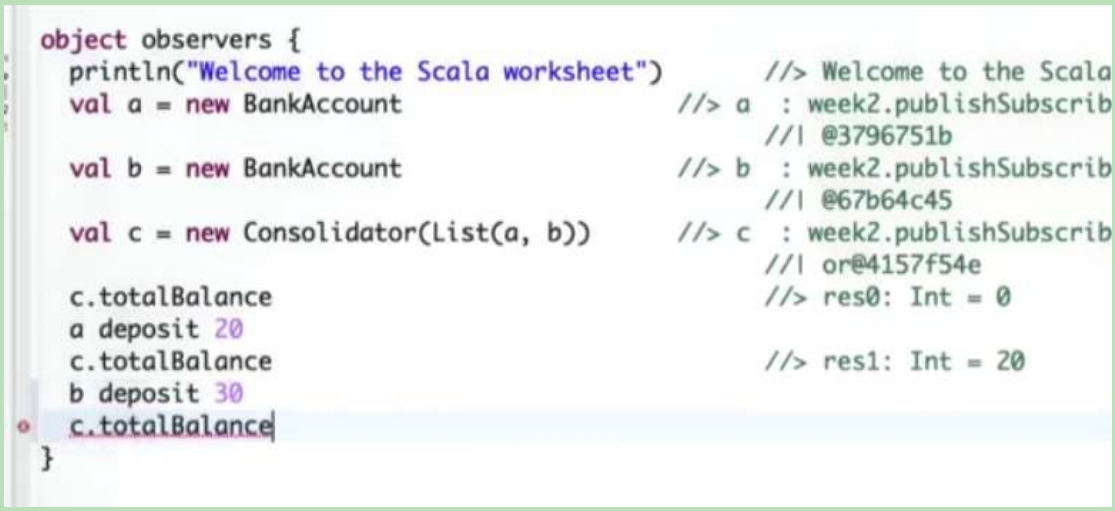
④删除最小值： 寻找一颗具有最小权值的二项树，设为 B(i), 删除 B（i）的根结点后，得到若干的 B（0），B（1）， 。。。B(i-1)这 I 个新二项树，把这新的二项树集合作为 H，执行 H 与原来剩下的二项树集合的合并

三，二项队列与二叉堆的比较

基本操作：	insert(平均情况下)	deleteMin	merge
二项队列：	O(1)	O(logN)	O(logN)
二叉堆：	O(1)	O(logN)	O(N)

C2Week4

Class 1



Observer pattern

The Observer Pattern is widely used when views need to react to changes in a model.
Some variants are also called publish/subscribe, model/view/(controller).

http://blog.csdn.net/sundacheng1989/article/details/50285369

这里通过一个生活例子来讲观察者模式。这个场景就是，有一个小孩在睡觉，然后他老爸在旁边，小孩醒了他老爸就要喂他吃东西。从面相对象的角度来讲，我们通过描述就可以抽象出实体类，小孩，老爸是两个对象，然后睡觉是小孩的状态，喂东西是老爸的方法。

第一种方法，我们用最直接的方式去模拟这个过程。当小孩在睡觉的时候，老爸会一直的去查看小孩是否醒了，1 秒钟查看一次，模拟到程序里边，就是一种死循环，以轮循的方式去判断小孩的属性。这里把小孩设计成一个线程，最初是睡着状态，10 秒后醒来。那么 10 秒

后老爸就要调用其 feed 方法。小孩的 toString()方法输出这个对象的唯一标识，表示这个特定的小孩。

这种方式存在一定的弊端，即老爸得时时刻刻盯着小孩，这个过程中不能干别的事情，即使客厅里有足球赛，这时候老爸也不能把小孩放在房间里自己去客厅。反应到程序中来讲，这种死循环的方式，对于 CPU 是一种无端的消耗。

接下来我们用第二种方式进行模拟。我们可以这样，我们让小孩持有老爸的引用，当小孩醒了之后，主动调用老爸的 feed 方法。就相当于在小孩与老爸之间绑了一根绳子，小孩在里屋睡觉，老爸在客厅看球，当小孩醒了之后，拽动这个绳子，那么老爸得知消息后再进里屋来喂东西。代码可以如下。
其实，到这种实现方法，我们已经引入了观察者模式。观察者模式的核心就是让行为的行使着变主动为被动。行为的行使着就是观察者。但是从面向对象的角度来讲，有些地方我们还可以接着完善。比如说，小孩什么时候醒的，这个属性不是小孩的，也不是老爸的，而是发生的这件事情本身的，所以醒来这件事应该也是一个类。所以接下来，我们完善一下代码，用第三种方式实现。

这里如果我们要求小孩的老妈也要对小孩的醒来进行一个些反应，我们就要修改小孩内部的代码，让小孩同时也持有老妈的引用。面向对象的一个重要原则就是添加而不是修改，就是说可以向小孩添加监听者，而不应该修改小孩的内部代码。所以，现在我们可以让所有的监听者实现同一个接口。而小孩内部持有这个接口的集合，那么就可以添加无数的监听者了

CODE bankaccount 加入观察者模式

```
trait Subscriber{
  def handler(pub: Publisher)
}

trait Publisher{
  private  var subscribers: Set[Subscriber] = Set()

  def subscribe(subscriber: Subscriber): Unit =
    subscribers += subscriber

  def unsubscribe(subscriber: Subscriber): Unit =
  subscribers -= subscriber

  def publish():Unit =
    subscribers.foreach(_.handler((this)))
}
```

```
class BankAccount extends Publisher{
  private var balance = 0
  def currentBalance: Int = balance // <---make BankAccount a Publisher
  def deposit(amount: Int):Unit = {
    if (amount > 0){
      balance = balance + amount
//      publish() // <---make BankAccount a Publisher
    }
  }

  def withdraw(amount: Int):Unit = {
    if (0<amount && amount <= balance){
      balance = balance - amount
      publish() // <---make BankAccount a Publisher
    }
    else throw new Error("insufficient funds")
  }
}

class Consolidator(observed: List[BankAccount]) extends Subscriber{
  // total 的值是 sum()返回
  def sum():Int = observed.map(_.currentBalance).sum
  def handler(pub: Publisher) = sum()
//  def totalBalance = total
//  private var total : Int = this.sum
  def totalBalance = sum //修正做法
}

//object observers extends App{
  println("Welcome to the scala worksheet")
  val a = new BankAccount
  val b = new BankAccount
  val c = new Consolidator(List(a,b))
  c.totalBalance
  a deposit 20
  a.currentBalance
  b.currentBalance

  c.sum//可以计算 a.currBalance + b.currBalance 的结果，所以 a 和 b 对自己改变后，因为他们都在 C 的观察之下，(把自己的引用传给了 C)因此只需调用 C 就可以知道整体的变化
  c.totalBalance//仍然为 0？？？为什么会这样,只有把 totalBalance 直接改为 = sum 才可以修正，懒惰计算？
  b deposit 30
  a.currentBalance
```

```
b. currentBalance
c. totalBalance
```

good and bad

good of observer pattern

- Decouples views from state
- Allows to have a varying number of views of a given state
- Simple to set up

Bad

- Forces imperative style, since handlers are Unit-typed
- Many moving parts that need to be co-ordinated
- Concurrency makes things more complicated
- Views are still tightly bound to one state; view update happens immediately.

To quantify (Adobe presentation from 2008):

- 1/3*rd* of the code in Adobe’s desktop applications is devoted to event handling.
- 1/2 of the bugs are found in this code.

More about publisher-subscriber(beobserved---observer)

http://blog.csdn.net/zhangming1013/article/details/25248269

二、示例代码
商品价格打折后，所有关注、收藏该商品的用户都收到相关的信息提醒。
角色：
1）商品：被观察者；
2）用户：观察者

1.商品（发布者）
[java] view plain copy
print?

```
1. package com.csdnproject.observer.model1;
2.
3. import java.util.ArrayList;
4. import java.util.Iterator;
5. import java.util.List;
6.
7. /**
8.  * 商品-发布者
```

```
 * @author Administrator
 *
 */
public class Product {
    private String name;
    private double price;
    private List<Observer> focusUsers;//观察者集合

    /**
     * 价格折扣
     * @param off
     */
    public synchronized void payOff(double off) {
        this.price = getPrice() * (1 - off);
        StringBuffer msg = null;

        if (focusUsers != null && !focusUsers.isEmpty()) {
            Iterator<Observer> it = focusUsers.iterator();
            while (it.hasNext()) {
                Observer user = (Observer) it.next();

                String msgPart = ", " + this.getName() + "的价格 "
                        + this.getPrice() + ", 价格折扣 " + off * 100 + "%!";
                msg = new StringBuffer();
                msg.append("~~~~ 您好 " + user.getName());
                msg.append(msgPart);

                user.notify(msg.toString());//发送提醒   //一旦降价，就读取该商品的观察者列表 focusUsers，以被观察者触发观察者函数，为观察者添加更新信息，因此观察者便不需要实时盯着被观察者
            }
        }
    }

    /**
     * 添加关注用户
     * @param user
     */
    public void addFocusUsers(User user) {
        this.getFocusUsers().add(user);  //增加观察者，商品是被观察
    }

    /**
     * 删除关注用户
```

```
51.     * @param user
52.     */
53.    public void delFocusUser(User user) {
54.        this.getFocusUsers().remove(user);
55.    }
56.
57.    public Product() {
58.        focusUsers = new ArrayList<Observer>(); //观察者列表，可变数组
59.    }
60.
61.    public String getName() {
62.        return name;
63.    }
64.
65.    public void setName(String name) {
66.        this.name = name;
67.    }
68.
69.    public double getPrice() {
70.        return price;
71.    }
72.
73.    public void setPrice(double price) {
74.        this.price = price;
75.    }
76.
77.    public List<Observer> getFocusUsers() {
78.        return focusUsers;
79.    }
80.
81.    public void setFocusUsers(List<Observer> focusUsers) {
82.        this.focusUsers = focusUsers;
83.    }
84.
85. }
```

2.观察者(订阅者)接口

[java] view plain copy print?

```
1.  package com.csdnproject.observer.model1;
2.
3.  /**
4.   * 观察者(订阅者)接口
5.   * @author Administrator
```

Rig

```
6.  *
7.  */
8.  public interface Observer {
9.
10.     public void notify(String msg);
11.
12.     public String getName();
13.
14. }
```

3.观察者(订阅者)

[java] view plain copy

print?

```
1.  package com.csdnproject.observer.model1;
2.
3.  import java.util.HashSet;
4.  import java.util.Set;
5.
6.  /**
7.   * 观察者(订阅者)
8.   * @author Administrator
9.   *
10.  */
11. public class User implements Observer {
12.     private String name;
13.     private Set<Product> focusPdts;
14.
15.     /**
16.      * 通知方法
17.      */
18.     public void notify(String msg) {
19.         System.out.println(msg);
20.     }
21.
22.     public User() {
23.         focusPdts = new HashSet<Product>();
24.     }
25.
26.     public String getName() {
27.         return name;
28.     }
29.
30.     public void setName(String name) {
31.         this.name = name;
```

```
32.     }
33.
34.     public Set<Product> getFocusPdts() {
35.         return focusPdts;
36.     }
37.
38.     public void setFocusPdts(Set<Product> focusPdts) {
39.         this.focusPdts = focusPdts;
40.     }
41.
42. }
```

4.client 端调用

[java] view plain copy

print?

```
1.  package com.csdnproject.observer.model1;
2.
3.  public class Client {
4.
5.      /**
6.       * @param args
7.       */
8.      public static void main(String[] args) {
9.          //产品
10.         Product mobile = new Product();
11.         mobile.setName("SAMSUNG 手机");
12.         mobile.setPrice(2000);
13.
14.         Product book = new Product();
15.         book.setName("JAVA 设计模式");
16.         book.setPrice(80);
17.
18.         //用户
19.         User user1 = new User();
20.         user1.setName("张三");
21.         user1.getFocusPdts().add(mobile);//关注某一款三星手机
22.         //user1.getFocusPdts().add(book);//关注 JAVA 设计模式
23.
24.         User user2 = new User();
25.         user2.setName("李四");
26.         user2.getFocusPdts().add(mobile);//关注某一款三星手机
27.         user2.getFocusPdts().add(book);//关注 JAVA 设计模式
28.
29.         //建立商品和订阅者关联
```

```
30.         mobile.getFocusUsers().add(user1);
31.         book.getFocusUsers().add(user1);
32.         book.getFocusUsers().add(user2);
33.
34.         //产品打折,发送站内信提醒
35.         mobile.payOff(0.1);
36.         book.payOff(0.2);
37.     }
38.
39. }
```

三、功能设计
常用的处理方式：
将数据库作为数据存储的介质，消息提醒数据保存在数据库表中，采用定时任务的方式来汇总和发送。具体流程：
1.存储用户-关注关联数据
将用户和所关注的数据存储到一张“用户-关注商品关联表”；

2.执行汇总任务
商品打折时，触发汇总任务，遍历“用户-关注商品“关联表，将符合发送条件的记录汇总到”提醒消息表“；数据量巨大的情况下，可采用在“用户-关注商品关联表”冗余字段的方式，不再创建”提醒消息表“减小数据量。

3.发送折扣提醒消息
遍历”提醒消息表“并发送，发送完成后，将记录标示为已发送。

class2

The theme is : Functional Reactive Animation.

Scale embedded Fundamental Signal Operations

1. obtain the value of the signal at the current time. In our library this is expressed by () application.
mousePosition()//the current mouse position(I suggest that return type is Signla)

2. define a signal in terms of other Signals.

```
def inReactangle(LL: Position, UR: Position): Signal[Boolean] =
Signal {
val pos = mousePosition()
LL <= pos && pos <= UR
}
```

Rig

define a signal that varies in time

- We can use externally defined signals, such as mousePosition and map over them.
- Or we can use a Var.

Subclass of Signal: Var

Values of type Signal are immutable.
But our library also defines a subclass Var of Signal for signals that can be changed.
Var provides an “update” operation, which allows to redefine the value of a signal from the current time on.

```
val sig = Var(3)
sig.update(5) // From now on, sig returns 5 instead of 3.
```

he update operation uses the name update for a reason because in fact in Scala update calls can be rewritten as assignments using some syntactic sugar.

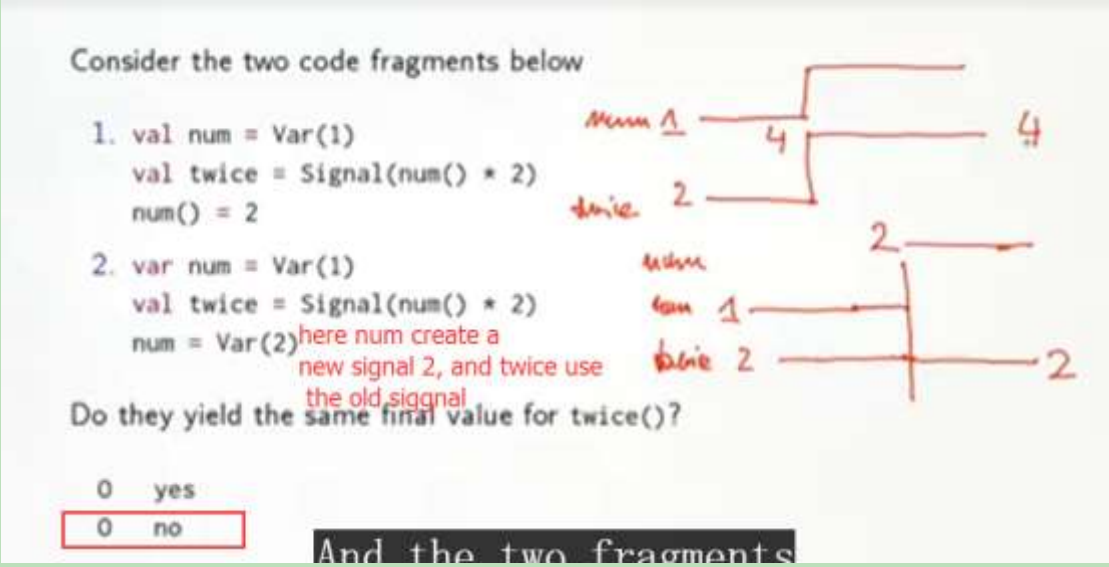
Automatically update

Sig() // this is dereferencing. (https://en.wikipedia.org/wiki/Dereference_operator

Dereferencing is something that we use a pointer to get the value of specific variable

Sig() = newValue // this is the update, equal to : sig.update(newValue)

We can map over signals, which define the relationship between signals.



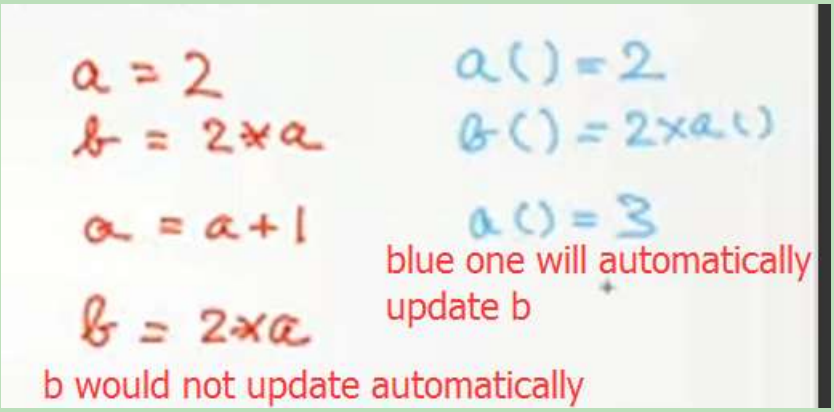
Num is a val of Signal Type(Var is sublass of Signal), once update num() = 2, （I guess, once implement updating in num instead of reassignment, this update function will automatically change the value of twice by focusing updating of observer）twice will be automatically updated to 4.

Therefore the followed sentence is illegal:

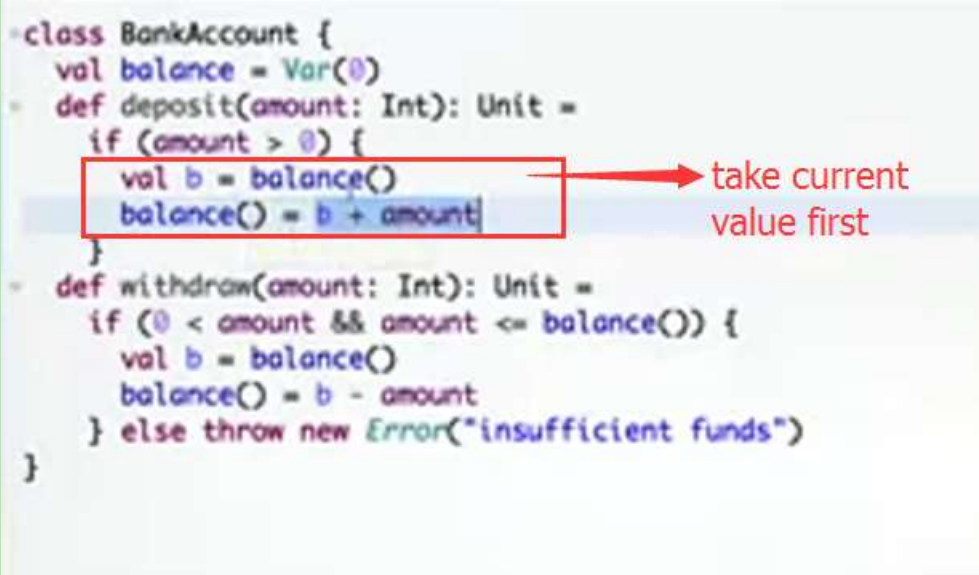
Sig() = sig() + 1 because this sentence describe the relationship of sig itself, we need to change it in order to realize update function

Val s = sig()

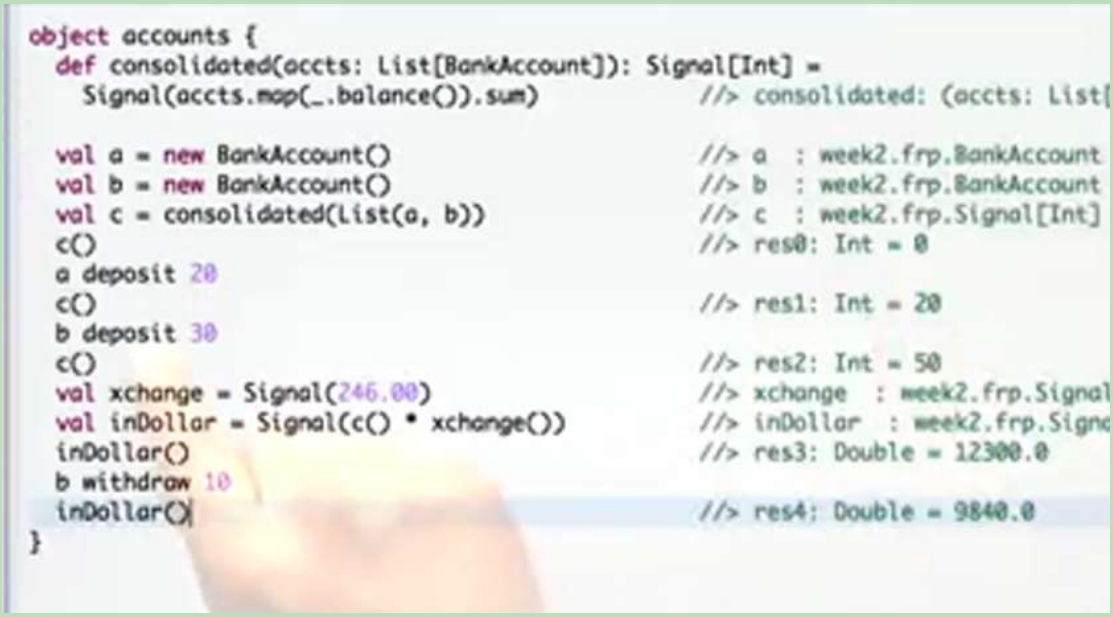
Sig() = s + 1



Rig




```
class BankAccount {
  val balance = Var(0)
  def deposit(amount: Int): Unit =
    if (amount > 0) {
      val b = balance()
      balance() = b + amount
    }
  def withdraw(amount: Int): Unit =
    if (0 < amount && amount <= balance()) {
      val b = balance()
      balance() = b - amount
    } else throw new Error("insufficient funds")
}
```



```
object accounts {
  def consolidated(accts: List[BankAccount]): Signal[Int] =
    Signal(accts.map(_.balance()).sum)          //> consolidated: (accts: List[

  val a = new BankAccount()                     //> a  : week2.frp.BankAccount
  val b = new BankAccount()                     //> b  : week2.frp.BankAccount
  val c = consolidated(List(a, b))              //> c  : week2.frp.Signal[Int]
  c()                                           //> res0: Int = 0
  a deposit 20
  c()                                           //> res1: Int = 20
  b deposit 30
  c()                                           //> res2: Int = 50
  val xchange = Signal(246.00)                  //> xchange  : week2.frp.Signal
  val inDollar = Signal(c() * xchange())        //> inDollar  : week2.frp.Signa
  inDollar()                                    //> res3: Double = 12300.0
  b withdraw 10
  inDollar()                                    //> res4: Double = 9840.0
}
```

Class 3

Class apply

gives you the current value of the signal for the moment we have left out its implementation.

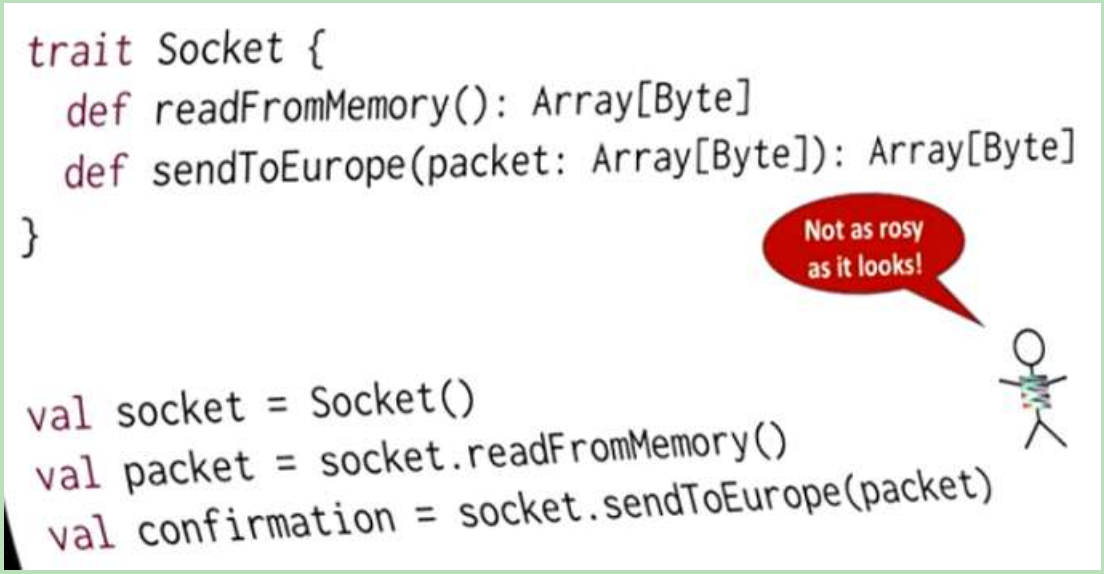
Object apply

would be the form that maps all the signals or creates constant signals.

thread-local state is an improvement of unprotected global state. But it has its own set of problems, it's fragile plays well only with some approaches to concurrency. And it has still the problem that it is fundamentally state that is shared by a large part of the application.

Rig

Class4



Is there a monad that we can use to
7:24
express the fact that computations take time.

Call back 回调函数

回调函数就是一个通过函数指针调用的函数。如果你把函数的指针（地址）作为参数传递给另一个函数，当这个指针被用来调用其所指向的函数时，我们就说这个被指向的函数是回调函数
运行机制：
⑴定义一个回调函数；
⑵提供函数实现的一方在初始化的时候，将回调函数的函数指针注册给调用者；
⑶当特定的事件或条件发生的时候，调用者使用函数指针调用回调函数对事件进行处理。

不关心被调用者在何处，也不关心被调用者用它传递的处理程序做了什么，它只关心返回值，因为基于返回值，它将继续执行或退出。

用途

回调的用途十分广泛。例如，假设有一个函数，其功能为读取配置文件并由文件内容设置对应的选项。若这些选项由散列值所标记，则让这个函数接受一个回调会使得程序设计更加灵活：函数的调用者可以使用所希望的散列算法，该算法由一个将选项名（函数参数）转变为散列值（函数返回值）的回调函数实现；因此，回调允许函数调用者在运行时调整原始函数的行为。
回调的另一种用途在于处理信号或者类似物。例如一个 POSIX 程序可能在收到 SIGTERM 信

号时不愿立即终止；为了保证一切运行良好，该程序可以将清理函数注册为 SIGTERM 信号对应的回调。
回调亦可以用于控制一个函数是否作为：Xlib 允许自定义的谓词用于决定程序是否希望处理特定的事件。

你到一个商店买东西，刚好你要的东西没有货，于是你在店员那里留下了你的电话，过了几天店里有货了，店员就打了你的电话，然后你接到电话后就到店里去取了货。在这个例子里，你的电话号码就叫回调函数，你把电话留给店员就叫登记回调函数，店里后来有货了叫做触发了回调关联的事件，店员给你打电话叫做调用回调函数，你到店里去取货叫做响应回调事件。回答完毕。

作者：常溪玲
链接：https://www.zhihu.com/question/19801131/answer/13005983
来源：知乎

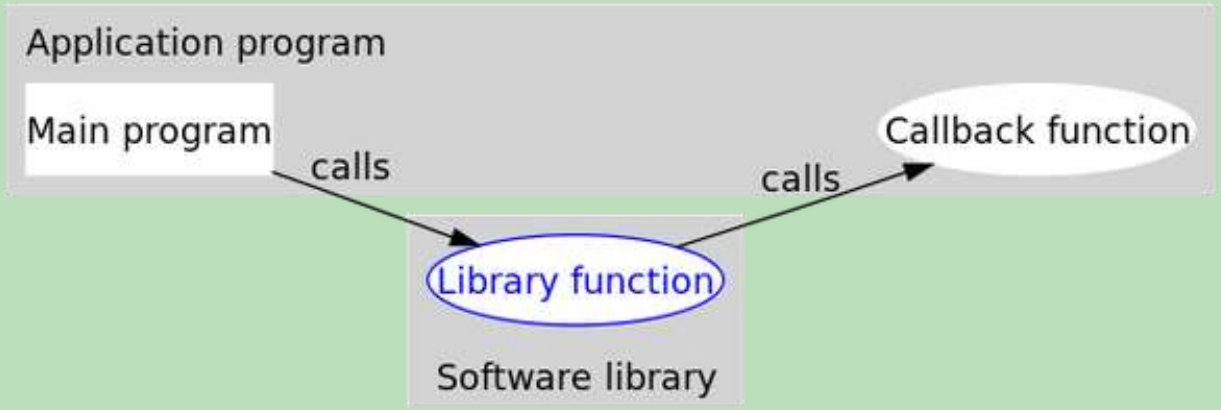
callback 详细具体讲解

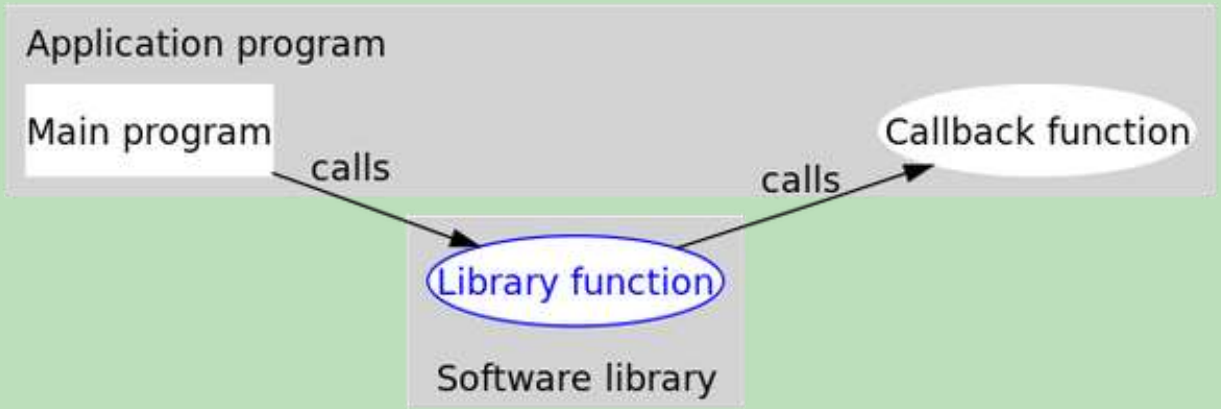
作者：桥头堡
链接：https://www.zhihu.com/question/19801131/answer/27459821
来源：知乎

编程分为两类：系统编程（system programming）和应用编程（application programming）。所谓系统编程，简单来说，就是编写**库**；而应用编程就是利用写好的各种库来编写具某种功用的程序，也就是**应用**。系统程序员会给自己写的库留下一些接口，即 API（application programming interface，应用编程接口），以供应用程序员使用。所以在抽象层的图示里，库位于应用的底下。

当程序跑起来时，一般情况下，应用程序（application program）会时常通过 API 调用库里所预先备好的函数。但是有些库函数（library function）却要求应用先传给它一个函数，好在合适的时候调用，以完成目标任务。这个被传入的、后又被调用的函数就称为**回调函数**（callback function）。

打个比方，有一家旅馆提供叫醒服务，但是要求旅客自己决定叫醒的方法。可以是打客房电话，也可以是派服务员去敲门，睡得死怕耽误事的，还可以要求往自己头上浇盆水。这里，“叫醒”这个行为是旅馆提供的，相当于库函数，但是叫醒的方式是由旅客决定并告诉旅馆的，也就是回调函数。而旅客告诉旅馆怎么叫醒自己的动作，也就是把回调函数传入库函数的动作，称为**登记回调函数**（to register a callback function）。如下图所示（图片来源：维基百科）：





可以看到，回调函数通常和应用处于同一抽象层（因为传入什么样的回调函数是在应用级别决定的）。而回调就成了一个高层调用底层，底层再**回**过头来**调**用高层的过程。（我认为）这应该是回调最早的应用之处，也是其得名如此的原因。

回调机制的优势

从上面的例子可以看出，回调机制提供了非常大的灵活性。请注意，从现在开始，我们把图中的库函数改称为**中间函数**了，这是因为回调并不仅仅用在应用和库之间。任何时候，只要想获得类似于上面情况的灵活性，都可以利用回调。

这种灵活性是怎么实现的呢？乍看起来，回调似乎只是函数间的调用，但仔细一琢磨，可以发现两者之间的一个关键的不同：在回调中，我们利用某种方式，把回调函数像参数一样传入中间函数。可以这么理解，在传入一个回调函数之前，中间函数是不完整的。换句话说，程序可以在运行时，通过登记不同的回调函数，来决定、改变中间函数的行为。这就比简单的函数调用要灵活太多了。请看下面这段 Python 写成的回调的简单示例：

`even.py`
#回调函数 1
#生成一个 2k 形式的偶数

```
def double(x):
    return x * 2

#回调函数 2
#生成一个 4k 形式的偶数
def quadruple(x):
    return x * 4
```

`callback_demo.py`

```
from even import *

#中间函数
#接受一个生成偶数的函数作为参数
#返回一个奇数
def getOddNumber(k, getEvenNumber):
    return 1 + getEvenNumber(k)

#起始函数，这里是程序的主函数
def main():
    k = 1
    #当需要生成一个 2k+1 形式的奇数时
    i = getOddNumber(k, double)
    print(i)
    #当需要一个 4k+1 形式的奇数时
    i = getOddNumber(k, quadruple)
    print(i)
    #当需要一个 8k+1 形式的奇数时
    i = getOddNumber(k, lambda x: x * 8)
    print(i)

if __name__ == "__main__":
    main()
```

运行`callback_demp.py`，输出如下：

```
3
5
9
```

上面的代码里，给`getOddNumber`传入不同的回调函数，它的表现也不同，这就是回调机制的优势所在。值得一提的是，上面的第三个回调函数是一个匿名函数。

易被忽略的第三方

通过上面的论述可知，中间函数和回调函数是回调的两个必要部分，不过人们往往忽略了回

调里的第三位要角，就是中间函数的调用者。绝大多数情况下，这个调用者可以和程序的主函数等同起来，但为了表示区别，我这里把它称为**起始函数**（如上面的代码中注释所示）。

之所以特意强调这个第三方，是因为我在网上读相关文章时得到一种印象，很多人把它简单地理解为两个个体之间的来回调用。譬如，很多中文网页在解释“回调”（callback）时，都会提到这么一句话：“If you call me, I will call you back.”我没有查到这句英文的出处。我个人揣测，很多人把起始函数和回调函数看作为一体，大概有两个原因：第一，可能是“回调”这一名字的误导；第二，给中间函数传入什么样的回调函数，是在起始函数里决定的。实际上，回调并不是“你我”两方的互动，而是 ABC 的三方联动。有了这个清楚的概念，在自己的代码里实现回调时才不容易混淆出错。

另外，回调实际上有两种：阻塞式回调和延迟式回调。两者的区别在于：阻塞式回调里，回调函数的调用一定发生在起始函数返回之前；而延迟式回调里，回调函数的调用有可能是在起始函数返回之后。

Class 5

what we asked ourselves is can we invent a monad to make this visible to make the fact that there's latency visible in the type.

We are creating three email messages and then we're reading from memory and future array of bytes. And this is defined as follows, we're going to take the email from the queue, we serialize it and then we return the binary message. This is the read from memory, but now notice that whenever I put a callback to this read from memory, the code in that body will be executed only once. So it's not that when you register two callbacks, that two emails will be read, so that's important. And so here's a body with side effects, but for every callback, the side effect will happen only once. So this is a very important thing with a future.

Class 6

So, ZIP will always confirm to the shortest list. That you're zipping. And the same is true when you're zipping two futures. In that case, the error case is the empty list, and the success case is the non-empty list. So when I'm zipping two futures, and one of them fails, the whole zip will fail.

The thing here that I want to focus on is that recover and recover with are like the map and the flat map, but then for the error case of the future.

Class 7

you try this. If that fails, you try that. And then, you know? In the end, you say, okay. If both fails, then we just return this.

You should always, once you're asynchronous, you should never, ever block. Except when you're debugging or doing Little script,
promise it to me, never block when you have an asynchronous computation.

Okay so the reason that you introduce futures was because there was latency. As soon as you introduce blocking you're destroying all of that. because once you block somewhere in this whole pipeline, you kind of may not have used asynchrony

Class 9

We want to execute them one by one because every time we execute it might have side effects so we have to be very careful to delay the execution of the block.

now we're going to map the number of attempts over that, and so now we get a List of blocks, and remember this is super important To do this here because we don't want to accidentally execute the blocks. So we have to build this little funk that hides the block.

HW

calculator 载入错误解决

It looks like IntelliJ doesn't understand the layout of calculator by default, but I was able to get it to work from the "Project Structure" dialog:

- select on "Modules" and remove "calculator" from the list (with the red minus sign)
- click on the "webUI" module and select the "Dependencies" tab. Add a new dependency (with the green "+" sign on the far right) and select "root".

There should now be four modules listed: "root", "root-build", "webUI" and "webUI-build".
Once I did that, my project became navigable and I could run the tests from inside IntelliJ.
Hope that helps someone else.

Appendix

Warn

SBT project import

[warn] Multiple dependencies with the same organization/name but different versions. To avoid conflict, pick one version:

[warn] * org.scala-lang:scala-reflect:(2.11.2, 2.11.7)

[warn] * org.scala-lang.modules:scala-xml_2.11:(1.0.2, 1.0.4)

Resource

Java 自动装箱和拆箱

http://www.cnblogs.com/danne823/archive/2011/04/22/2025332.html

查询函数

http://www.scala-lang.org/api/current/scala/collection/TraversableLike%24WithFilter.html?search=withFilter

常见问题

同名的 class 和 object

近来一直在阅读 Spark 的源码，发现一个问题，一直不太理解为什么在同一个 .Scala 文件中可以同时存在名字相同的 class 和 object
经过一番研究，终于明白原来是在 scala 中没有定义 static 类型的变量，取而代之的是单例对象：singleton object
除了 object 关键字取代了 class，单例对象看着就像是类的定义。
一个单例对象与一个类共享同一个名称时，他被称作是这个类的**伴生对象：companion object**。他们必须在同一个源文件中。同理，这个类叫做这个对象的伴生类：**companion class**。 类和它的伴生对象之间都可以相互访问各自的私有成员。
类和单例对象时间的差别是：单例对象不带参数，而类可以。道理也很简单，单例对象是不需要 new 的，更谈不上参数的传递。单例对象会在第一次被访问的时候被初始化。
另外：不与伴生类共享名称的单例对象被称为是**孤立对象：standalone object**。

Class object trait

1 Class 中的 method 的参数 默认 是 val，及不可修改类型

```
def Add(b:Byte)
{
b = 1;    //will not compile
sum+=b;
}
```

2
2.1 Object 在 scala 里面的引入可能就是因为 scala 没有静态的方法和字段，没有办法 实现类似 C++的 singleton 模型所以引入 object
书上云 object 是 holder for static methods
2.2 Object 由于上述原因，同类的差别在于没有状态，只有方法。换句话说成员中仅有 val 没有 var，而且不能给 object 传递参数
2.3 Object 定义并不表示声明了一个新的类型，定义一个变量是 obj 类型是错误的。

3
Trait 可以带字段和状态同 class 一样，实际上 trait 同 class 完全一致，除了以下两点：
1 trait 不能带参数，即构造函数中不能带参数，需要内 部定义抽象的 method 表示状态

```
class Point(x: Int, y: Int)   OK
trait NoPoint(x: Int, y: Int) // Does not compile
```

2 super 的调用时动态的，而不是静态的。

Trait 作用：

Rig

1 丰富接口

```
scala> trait myTrait{
     | def a:Int
     | def b:Int
     | def sub = {
     | a - b
     | }
     | }
defined trait myTrait


scala> class myClass(val a:Int,val b:Int) extends myTrait{
     | }
defined class myClass


scala> val test = new myClass(1,2)
test: myClass = myClass@82feb7



scala> test.sub
res2: Int = -1
```

2 作用 2 stackable 修改实体的 class

对具体的类可以堆砌修改，比如实例类有 A B C D 方法，来了个新需求仅仅修改 B 方法,这时可以定义一个 trait 单独实现 B 方法。

再让这个具体的类混入这个 trait 已达到新类功能。

Cpp 的 singleton

在我们日常的工作中经常需要在应用程序中保持一个唯一的实例，如：IO 处理，数据库操作等，由于这些对象都要占用重要的系统资源，所以我们必须限制这些实例的创建或始终使用一个公用的实例，这就是我们今天要介绍的——单例模式（Singleton）。

单件模式（Singleton）：保证一个类仅有一个实例，并提供一个访问它的全局访问点。

它拥有一个私有构造函数，这确保用户无法通过 new 直接实例它。

http://www.cnblogs.com/rush/archive/2011/10/30/2229565.html

Illegal start of simple expression in Scala

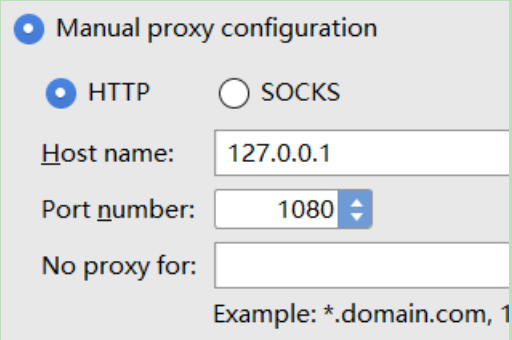

```
val charCode: Map[Char, Char] =
  for {
    (digit, str) <- mnem
    //use mnem1 will triger the problem of type mismatch
    // because "" is string '' is char
    ltr <- str
  } yield ltr -> digit
//这里不能将 for 循环用括号括起来，不然会提示这个错误，其实就是找不到返回值
```

使用 for 时注意

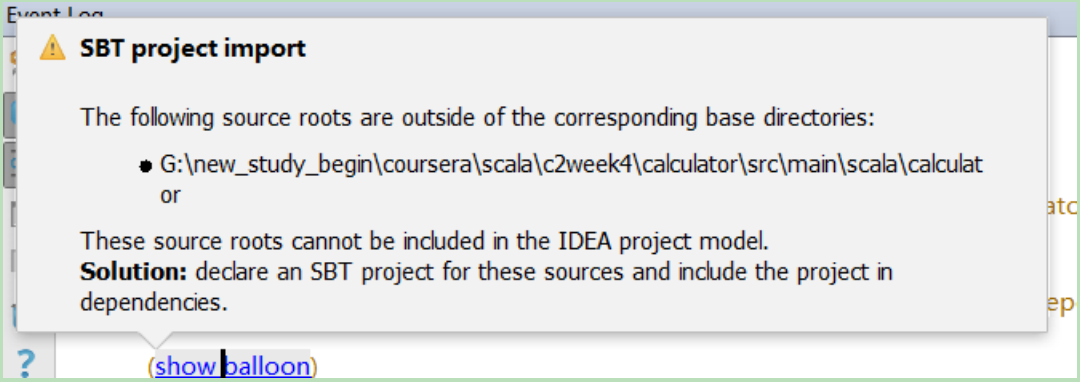
```
  for {
    (b, m) <- neighbors
    if (!explored.contains(b))
  } (b, m)
}
```

像这样忘记写 yield 会导致 type founded Unit
E 能用 filter map 等代替绝不用 for

设置代理



Project import 的问题



概念问题

静态语言和动态语言

通常我们所说的动态语言、静态语言指 动态类型语言（Dynamically Typed Language）和 静态类型语言 Statically Typed Language）。
还有一个 Dynamic Programming Language （动态编程语言），静态编程语言。

动态类型语言：在运行期间检查数据的类型的语言。用这类语言编程，不会给变量指定类型，而是在附值时得到数据类型。如：Python 和 ruby 就是典型动 态类型语言。很多脚本语言 vbscrīpt,javascrīpt 也是这类语言。看下面 javascrīpt 代码:

```
function add(a,b){
return a+b;
```

Rig

```
}
add(1,2);
add('1',2);
```

静态类型语言：相反静态类型语言是在运行前编译时检查类型。在写代码时，没声明一个变量必须指定类型。如：java,c#,c,c++等等。

```
public int add(int a,int b){
return a+b;
}
```

而 Dynamic Programming Language （动态编程语言）指在程序运行过程中可以改变数据类型的结构，对象的函数，变量可以被修改删除。比如：javascrīpt 就是这类语言，ruby, python 也属于这类语言。而 c++,java 不属于这类语言。看 javascrīpt 代码：

```
function Person(name){
this.name=name;
}
Person.prototype.getName=function(){
return this.name;
}
var person=new Person("okok");
alert(person.getName());
person.getName=function(){return "nono"};
alert(person.getName());
```

当然静态编程语言 是运行时不可改变结构了。

jVM

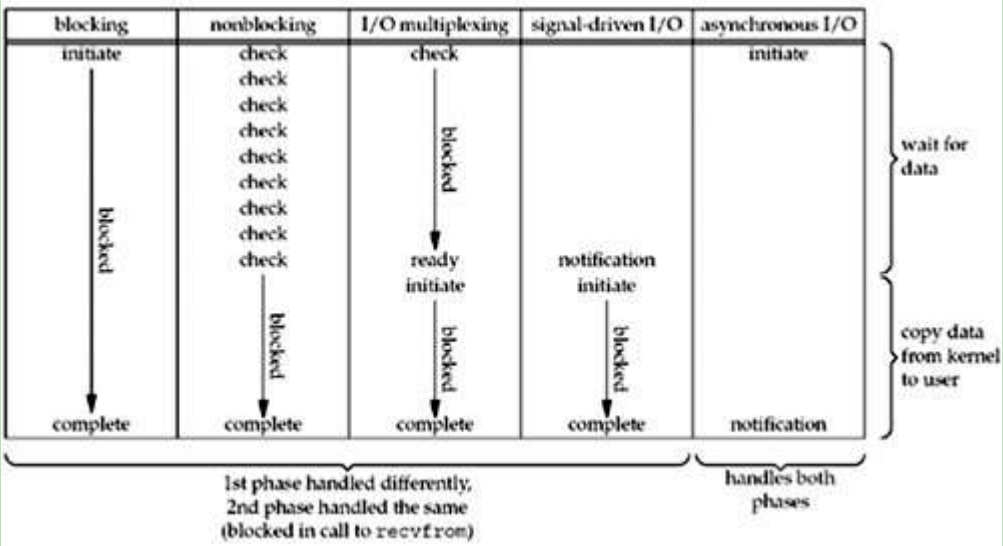
java virtual maching，一种用于计算机设备的规范，一个虚构出来的计算机，用在实际的计算机上仿真模拟各种计算机功能来实现
java 语言使得编程与平台无关。一般高级语言在不同的平台运行至少需要编译成不同的目标代码（object code，计算机科学中编译器或汇编器处理源代码后所声称的代码，它一般由机器代码或接近于机器语言的代码组成）。Java 编译程序只需要生产在 java 虚拟机上运行的目标代码（字节码），就能够在多种平台上不加修改的运行。Java 虚拟机在执行字节码时，再把字节码解释成具体平台上的机器码

jvm 定义了控制 java 代码解释执行和具体实现的五种规格
jvm 指令系统，jvm 寄存器，jvm 栈结构，jvm 碎片回收堆，jvm 存储区

blocking IO and Asynchronous

http://www.cnblogs.com/whyandinside/archive/2012/03/04/2379234.html
概括来说，一个 IO 操作可以分为两个部分：发出请求、结果完成。如果从发出请求到结果返回，一直 Block，那就是 Blocking IO；如果发出请求就可以返回（结果完成不考虑），就是 non-blocking IO；如果发出请求就返回，结果返回是 Block 在 select 或者 poll 上的，则其只能称为 IO multiplexing；如果发出请求就返回，结果返回通过 Call Back 的方式被处理，就是 AIO。

下面是以上五种模型的比较



可以看出，越往后，阻塞越少，理论上效率也是最优。

Non-Blocking IO 提交请求后只能通过提交的操作函数来查询操作是否完成，这是一个很大的限制。而 AIO 往往会提供多种通知或者查询机制，也就是说用 Non-Blocking IO 时只能轮询，而 AIO 有更多选择。所以是否支持轮询外的其他机制是 AIO 和 Non-Blocking IO 的区别。

集合 常用函数

http://blog.csdn.net/sbq63683210/article/details/51921389

mkString

把 List 转为字符串

```
val a = List(1, 2, 3) mkString
val aspace = List(1, 2) mkString "."
println(a, aspace)
```

结果

(123,1.2)

all possible combination of List

```
val t2 = (List(('a',1),('b',2)).toSet[(Char,Int)].subsets map (_.toList)).toList
```

indexWhere

indexWhere 返回满足条件的第一个 index，如果全部 index 不满足条件返回-1

```
def findChar(c: Char, levelVector: Vector[Vector[Char]]): Pos = {
    val x = levelVector.indexWhere(a => a.contains(c))
    val y = levelVector(x).indexOf(c)
    Pos(x, y)
  }
```

Copy 函数

Case class 的成员函数，复制出新的 case class 对象，并且修改

```
case class Employee(name:String, number: Int, index:Int){
  override def toString:String = name + number.toString
}
object Employee{
  def apply(name:String, number:Int):Employee = new Employee(name, number, 0)

}
val e1 = Employee("john", 123)
val e2 = e1.copy(name = "wzk", number = 123)
//val e3 = copy(e2.name, "cba")//这样定义会失败，因为 copy 是 case class 的成员方法
```

删除文件夹

```
import java.io._
```

Rig

```
def delete(file: File) {
  if (file.isDirectory)
    Option(file.listFiles).map(_.toList).getOrElse(Nil).foreach(delete(_))
  file.delete
}
```

读写文件

```
import java.io._

object Test {
   def main(args: Array[String]) {
      val writer = new PrintWriter(new File("test.txt" ))

      writer.write("Hello Scala")
      writer.close()
   }
}
```

HeadOption

```
scala> List(1).headOption
res35: Option[Int] = Some(1)

scala> List().headOption
res36: Option[Nothing] = None
List(1,2,3) headOption match{
  case Some(a) => a
  case _ => None
}
val COUNT: (Seq[Any]) => Int =
  (VALUES: Seq[Any]) => {
    VALUES.headOption match {
      case Some(_) => 1 + COUNT(VALUES.tail)
      case None => 0
    }
  }

COUNT(List(1,2,3))
```

res0: Any = 1

Rig

COUNT: Seq[Any] => Int = <function1>

res1: Int = 3

https://bradcollins.com/2015/10/24/scala-saturday-list-headoption/

scala 实验项目

socket 对话编程
http://blog.csdn.net/u012951554/article/details/45511563

https://github.com/Karasiq/proxychain

scala 爬虫

写爬虫先看下面这篇文章
http://duanhengbin.iteye.com/blog/2332020

end