# Mumbles and crumbles of Scala

D. Renault

The Tuesday's Geekeries
ENSEIRB-MATMECA

February 2014, v. 1.0

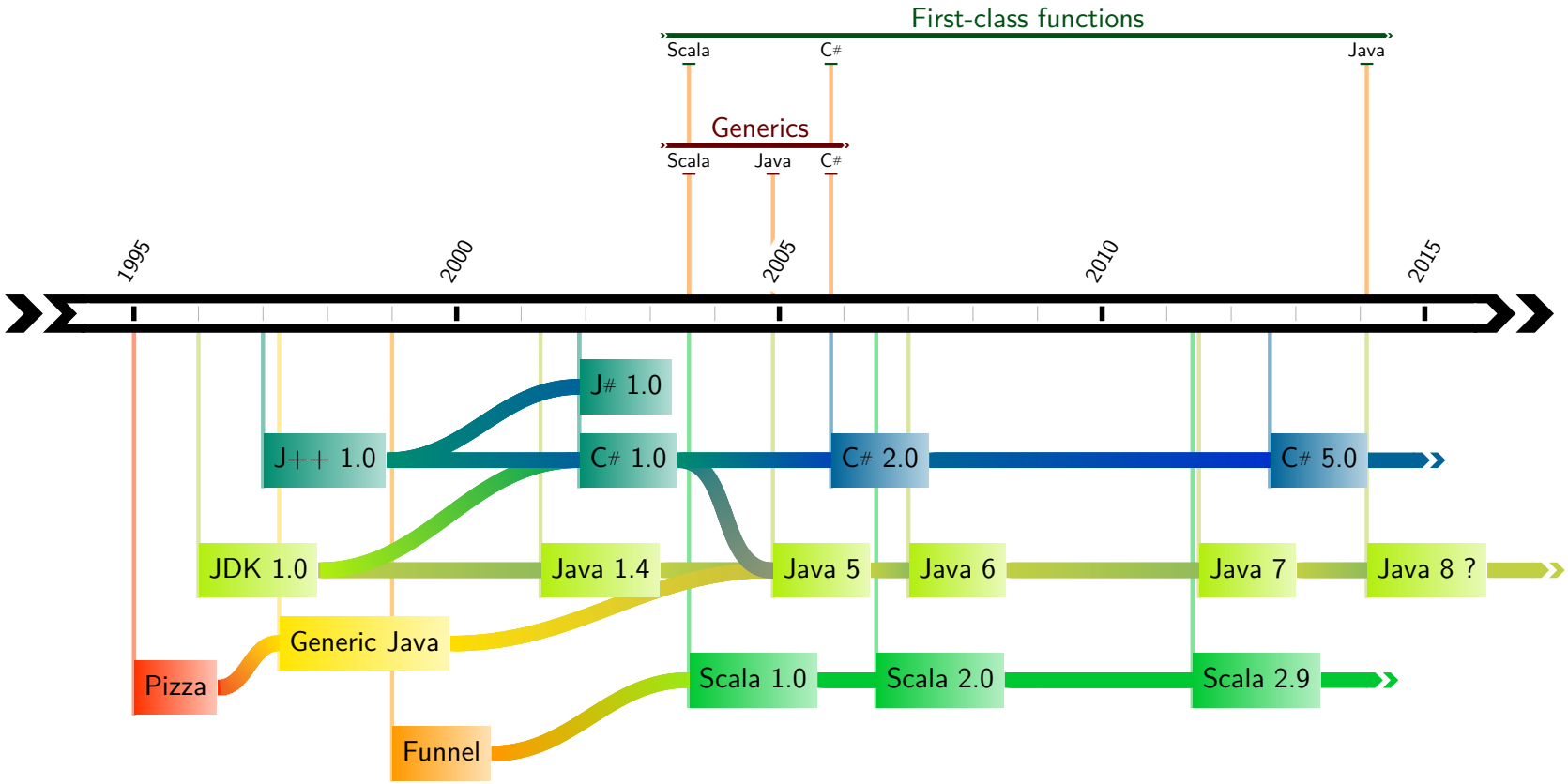
1996 - James **Gosling** invents **Java**. **Java** is a relatively verbose, garbage collected, class based, statically typed, single dispatch, object oriented language with single implementation inheritance and multiple interface inheritance. Sun loudly heralds **Java**'s novelty.

2001 - Anders **Hejlsberg** invents **C#**. **C#** is a relatively verbose, garbage collected, class based, statically typed, single dispatch, object oriented language with single implementation inheritance and multiple interface inheritance. Microsoft loudly heralds **C#**'s novelty.

2003 - A drunken Martin **Odersky** (...) has an idea. He creates **Scala**, a language that unifies constructs from both object oriented and functional languages. This pisses off both groups and each promptly declares jihad.

J.Iry, *"A Brief, Incomplete, and Mostly Wrong History of Programming Languages"*

# A brief history of Scala



# A brief history of Scala

- Statically typed, **pure object-oriented** language :
  every value is an object, every operation is a method call.
- Blended with **functional programming** features :
  first class functions, pattern matching, case classes ...
- Claims that the combination of the styles grants it **scalability** :
  **Scala** is adapted for writing small scripts as well as large programs.
- Compatibility with the JVM, but modifies/improves on **Java**'s capabilities and many aspects of the type system (class hierarchy, generics ...) and is not hindered by ascendant compatibility.
- Must stand the comparison with the upcoming **Java** 8 new features.

# For those with a computer at hand . . .

- Go to the language home page :

  `http://www.scala-lang.org/`
- Download the latest archive for the language.
- Run the interactive shell `scala`.
- Retrieve the code on the slides :

  `http://www.labri.fr/~renault/working/code.scala`
- Dampen the sound on your keyboards.

# Training wheels

The inevitable snippet about universality :

In **Scala** :

```
object HelloWorld {
  def main(args: Array[String]): Unit =
  { println("Scrmflghk") }
}
```

In **Java** :

```
class HelloWorld {
  public static void main(String[] args)
  { System.out.println("Scrmflghk"); }
}
```

Notice the difference at the toplevel : `object` vs. `class`

# Singleton objects

**Singleton objects** are objects that are the unique instance of their class.

```
object JapanMap extends HashMap[String,String] {}
JapanMap += (("Tokyo","Hotel"))      // -> res:JapanMap.type = Map(Tokyo -> Hotel)
JapanMap("Tokyo")                    // -> res: String = Hotel
```

⇒ Allows to do OOP without classes (fun !).

In **Scala**, singleton objects may be used to store the static methods of a class. In this case, they are called **companion objects**.

```
import java.math.BigInteger

class BigInt(val bigIntgr: BigInteger)
extends ScalaNumber ... {
  def equals(that: BigInt):Boolean = ...
  def compare(that: BigInt):Int = ...
  def + (that: BigInt):BigInt = ...
}
```

```
object BigInt {
  def apply(i: Int):BigInt  = ...
  def apply(l: Long):BigInt = ...
}
new BigInt(new BigInteger("5"))
// -> scala.math.BigInt = 5
BigInt(43)
// -> scala.math.BigInt = 43
```

Otherwise, **Scala** has the same look-and-feel than **Java** :

```scala
class A {}
trait I { def moo() : Unit }

class C(x : Int) extends A with I {



  def moo() : Unit =
    println("Grou")
}

val x : A = new C(1)
```

Scala

```java
class A {}
interface I { void moo(); }

class C extends A implements I {
  public final int x;
  public C(int x) { this.x = x };

  public void moo() {
    System.out.println("Grou"); }
}

A x = new C(1);
```

Java

- **Java** interfaces are replaced by **Scala** traits.
- Integration with **Java** code is particularly simple

```scala
import java.util.GregorianCalendar
val c = new GregorianCalendar()    // -> c: java.util.GregorianCalendar
c.get(java.util.Calendar.YEAR)     // -> res: Int = 2014
```

Scala

# Functional programming in Scala

- Functions are first-class values :

```
val max = (x:Int, y:Int) => {
  if (x > y) x else y }       // -> max: (Int, Int) => Int = <function2>
max(63,49)                    // -> res: Int = 63
```

- Case objects and **case classes** represent algebraic data types :

```
abstract class List
final case object Nil                      extends List
final case class  Cons(hd:Int, tl:List) extends List
val l : List = Cons(1, Cons(2, Nil))      // -> l: List = Cons(1,Cons(2,Nil))
```

- **Pattern matching** (on case classes, simple values and classes) :

```
def last(l:List) : Int = l match {
     case Nil          => throw new Exception("last")
     case Cons(x,Nil) => x
     case Cons(x,v)   => last(v) }
last(l)  // -> res:Int = 2
```

# Type inference

- In many cases, **Scala** is able to infer the types of values :

```
val l = List("This","is","a","list")                    // -> List[String]
l.sortWith (_ > _)                                  // -> (list, is, a, This)
```

- Nevertheless, its "flow-based" inference algorithm is not all that powerful. For instance, it cannot generalize the types of functions :

```
val max = (x,y) => { if (x > y) x else y }  // Error : Missing parameter type
```

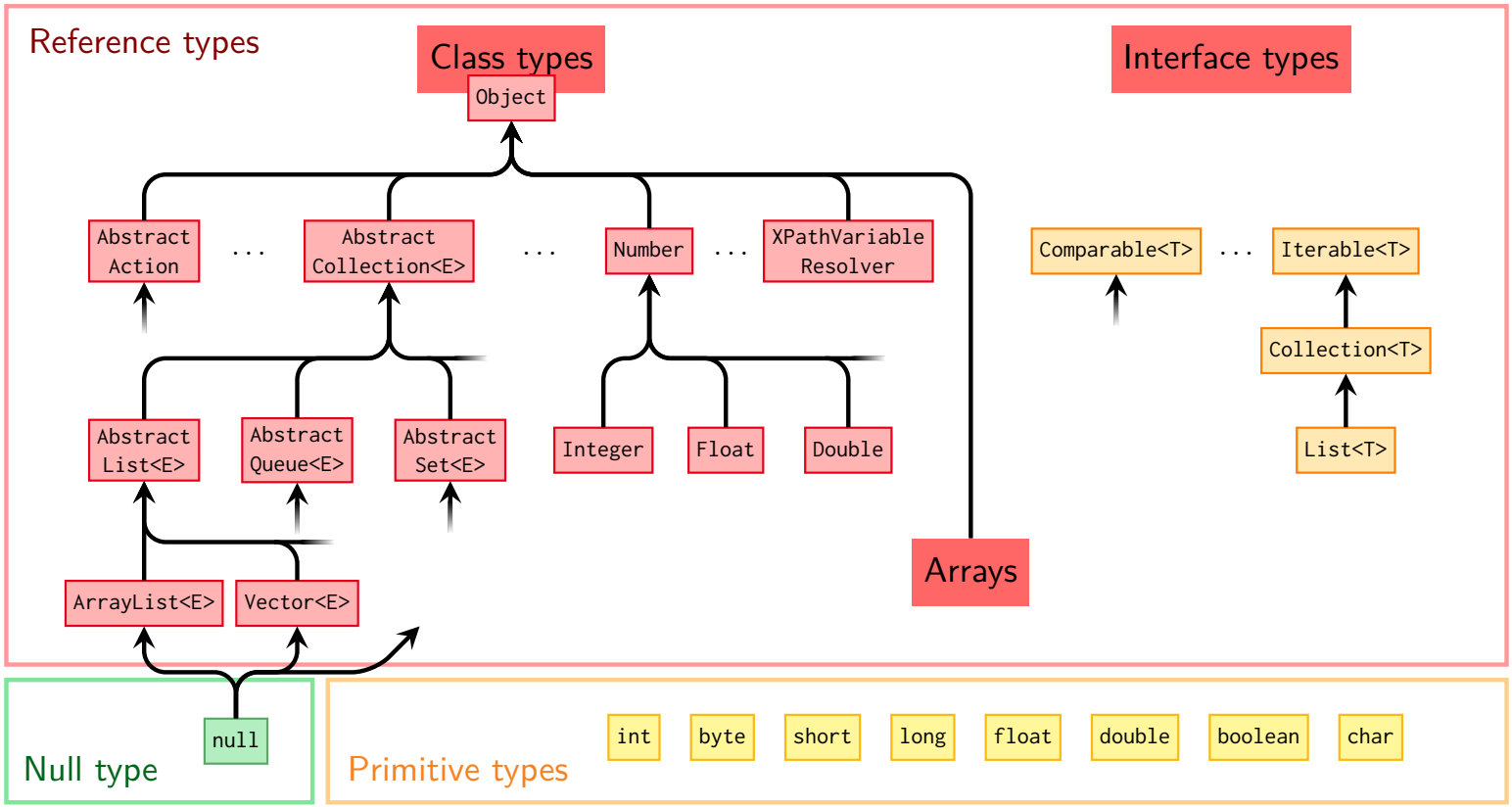
- Granted, getting a correct type is not that obvious :

```
def max[T<:Ordered[T]](x:T, y:T) = { if (x < y) x else y }
```

But **OCaml** and **Haskell** are able to infer it :

```
let max x y = if (x#gt y) then x else y;;
val max : (<gt : 'a -> bool; .. > as 'a) -> 'a -> 'a = <fun>
```

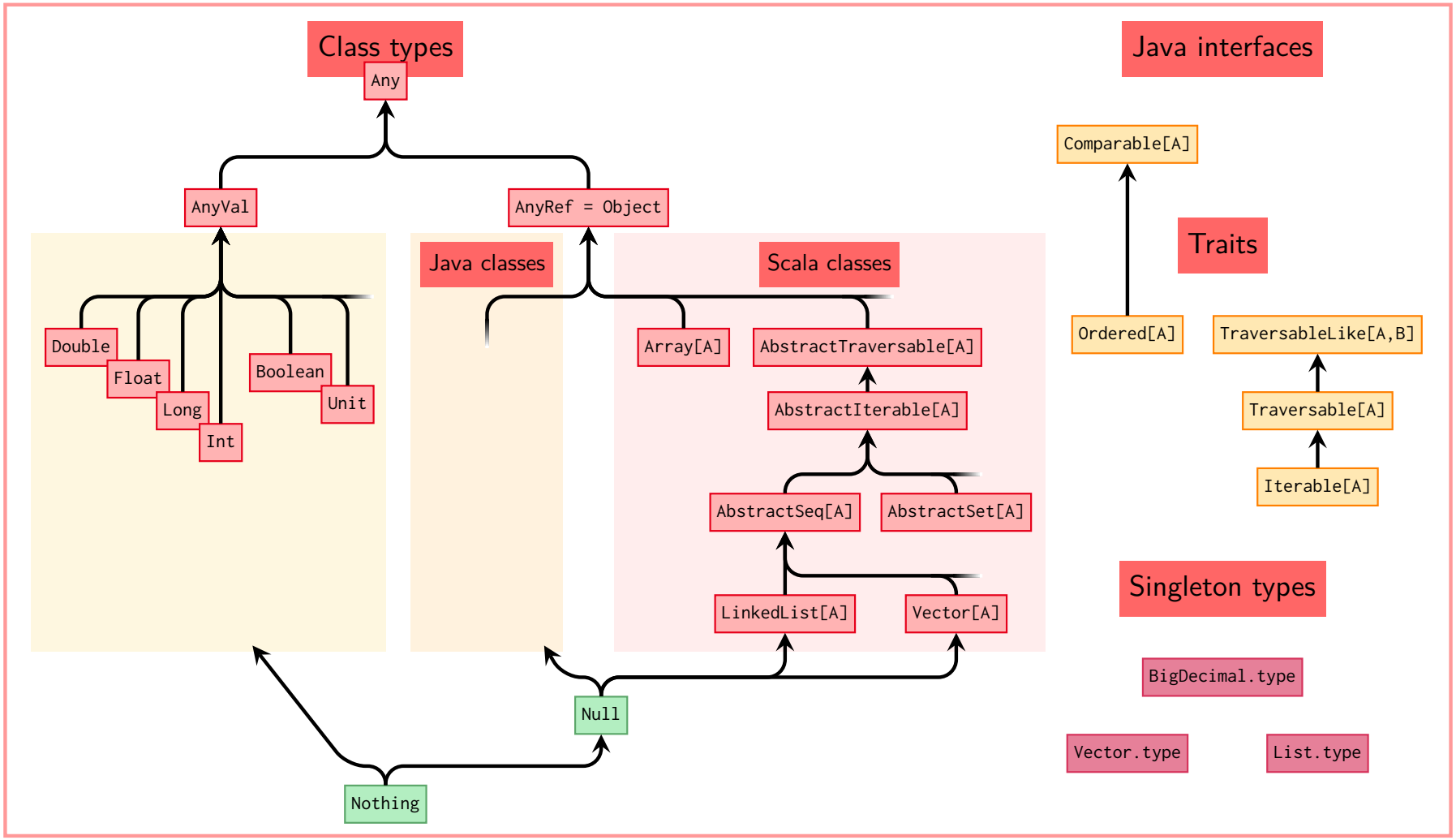
# Java types and values




Shortcomings of **Java** type systems :

- Primitive types (duplication, performance issues, compatibility w/generics)
- Arrays (duplication, compatibility w/generics, type safety)
- `void` does not belong to the hierarchy (functional types not exhaustive)

# Overview of Scala types and values



# Java and inheritance

The **Java** language allows two forms of inheritance :

- Single implementation inheritance : a class extends a unique other class
- Multiple specification inheritance : a class may implement several interfaces

$\Rightarrow$ Single inheritance limits code reuse (i.e "you only get one shot")

**Scala** allows multiple code inheritance with the help of **traits**.

# Scala traits

## Trait

A trait is an abstract class which is meant to be included (as a unit of code) into another class.

```
trait Ordered[A] extends Any {
  def compare(that: A): Int                          // abstract function
  def < (that: A): Boolean = (this.compare that) <  0
  def > (that: A): Boolean = (this.compare that) >  0
}
```

```
class Point(val x:Int,val y:Int) extends Ordered[Point] {
  def compare(that: Point) = this.x - that.x }
val p = new Point(2,3)
val q = new Point(3,2)
p < q // -> Boolean = true
```

- A trait is not meant to be instantiated by itself
- It can contain methods, but also values (and maintain a state)

# Mixin composition

## Mixin composition

A **Scala** class can extend only one class but implement multiple traits.
In this case, it is called a **mixin composition** (or simply mixin).

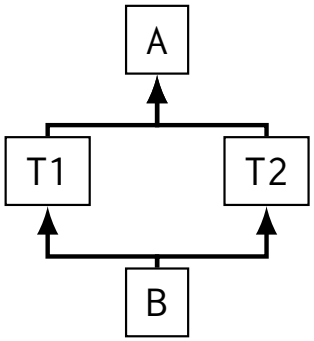
- Allows to create "rich" classes by mixing "thin" traits, each trait possibly representing an aspect of the behavior of the class.

```
class Point extends Ordered[Point] with Serializable with Cloneable
```

- Promotes modularity and the reuse of existing code.

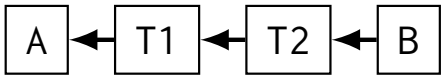
# Multiple inheritance

Multiple inheritance is usually problematic, as shown in the "dreaded" **diamond problem** :




```
class A { def f = "A"}
trait T1 extends A { override def f = "T1-" + super.f }
trait T2 extends A { override def f = "T2-" + super.f }
class B extends A with T1 with T2
(new B()).f // -> T2-T1-A or T1-T2-A ?
```

**C++** disambiguates by specifying explicitly the parent implementation.
**Scala** performs a linearization of the inheritance graph :



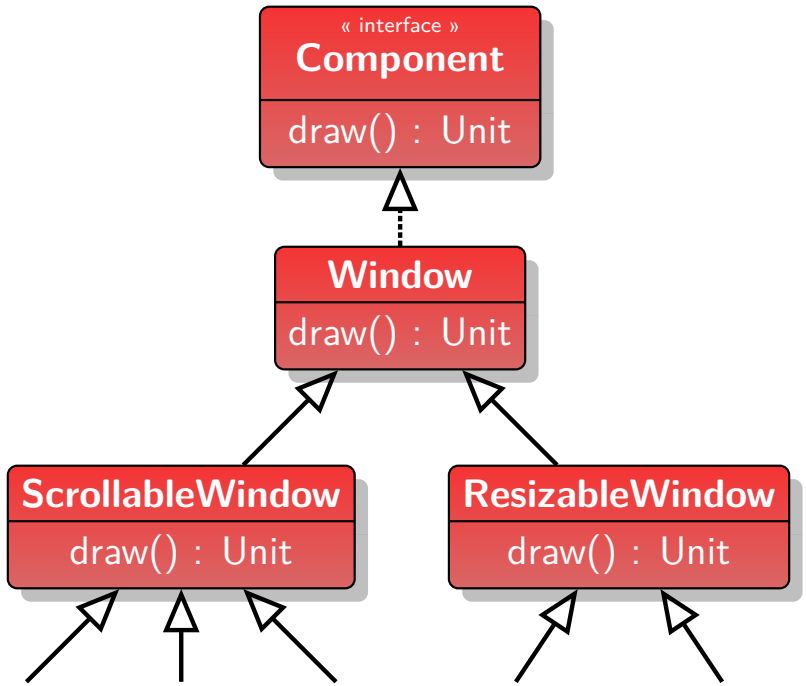

```
(new B()).f // -> T2-T1-A
```

- Defines an order on the inclusion of the classes;
- At dispatch time, the first class implementing the method is selected.

# Multiple overriding problem

## Multiple overriding problem

Intent : to add additional multiple orthogonal responsibilities to an object, while preserving its type.




(Disclaimer : this is not the classical Decorator pattern)

# In Java : overriding with inheritance

Intent : to add multiple orthogonal responsibilities to an object.

```
interface Component {
  String draw(); }

class Window implements Component {
  public String draw() {
    return "|__|"; }}
```

```
class ScrollableWindow extends Window {
  public String draw() {
    return "|" + super.draw() + "|";
}}

class ResizableWindow extends Window {
  public String draw() {
    return super.draw() + "+/-|";
}}
```

Client-side use of the overriden methods :

```
Window w1 = new ScrollableWindow();
w1.draw()    // -> ||__||
Window w2 = new ResizableWindow();
w2.draw();   // -> |__|+/-|
```

- Problem : combinatorial explosion of the children classes

# In C++ : overriding with inheritance

Intent : to add multiple orthogonal responsibilities to an object.

```
class Component {
public: virtual string draw() = 0;
};

class Window : public Component {
public: string draw() { return "|__|"; }
};
```

```
template<class T>
class ScrollableWindow : public T {
  static_assert(
    std::is_base_of<Window, T>::value);

  public: string draw() {
    return "|" + T::draw() + "|"; }
};
```

Client-side use of the overriden methods :

```
Window* w1 = new Window();
cout << w1->draw() << endl;     // -> "|__|"
Window* w2 = new ResizableWindow<ScrollableWindow<Window>>();
cout << w2->draw() << endl;     // -> "||__||+/-|"
```

- More flexible : the composite classes need not be constructed;
- Allowed in C++, not in Java because of the style of the generics.

# In Scala : Stackable trait pattern

Intent : to add multiple orthogonal responsibilities to an object.

```
trait Component {
  def draw () : String
}

class Window extends Component {
  def draw () = { "|__|" }
}
```

```
trait Scrollable extends Window {
  override def draw () = {
    "|" + super.draw () + "|"
}}

trait Resizable extends Window {
  override def draw () = {
    super.draw () + "+/-|"
}}
```

Client-side use of the overriden methods :

```
val w1 = new Window
println(w1.draw())     // -> |__|
val w2 = new Window with Scrollable with Resizable
println(w2.draw())     // -> ||__||+/-|
```

- Called the “stackable trait” pattern in Scala.

# In Java : the classical Decorator pattern

Intent : to add multiple orthogonal responsibilities to an object.

```
interface Component { String draw(); }

class Window implements Component {
  public String draw() { return "|__|";
}}
```

```
class ScrollableWindow extends Window {
  private Window m = null;

  public ScrollableWindow(Window m) {
    this.m = m;  }

  public String draw() {
    return "|" + m.draw() + "|";
}}
```

Decorations are dynamic : they can be added or removed at runtime.

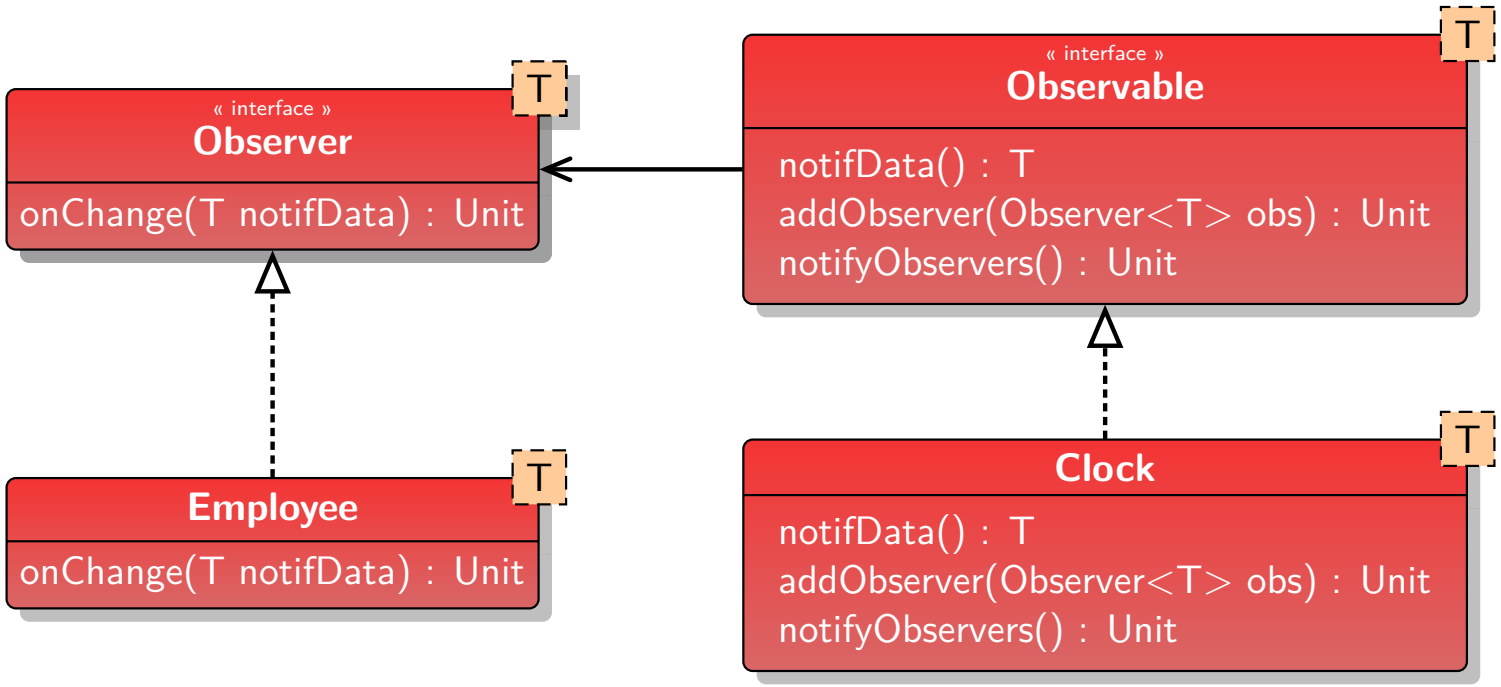
```
Window w = new ScrollableWindow(new Window());
w.draw();         // -> ||__||
w = new ResizableWindow(w);
w.draw();         // -> ||__||+/-|
```

- Flexible solution with a bit of code duplication.

# Design pattern : Observer

## Observer pattern

Intent : establish a relation between objects such that when one object changes state, all its dependents are notified and updated automatically.



# Example : the Observer pattern in Java

Intent : establish a relation between objects such that when one object changes state, all its dependents are notified and updated automatically.

```
interface Observer<T> {
  void onChange(T notifData); }

interface Observable<T> {
  T notifData();
  void addObserver(Observer<T> obs);
  void notifyObservers(); }

abstract class ObservableImpl<T>
  implements Observable<T> {
  private ArrayList<Observer<T>>
    observers = new ArrayList<>();
  public void addObserver(Observer<T> obs) {
    observers.add(obs);  }
  public void notifyObservers() {
    for (Observer<T> obs : observers)
      obs.onChange(notifData()); }}
```

```
class Clock extends ObservableImpl<Integer> {
  public Integer notifData() { /* ... */ }
};

class Employee implements Observer<Integer> {
  public void onChange(Integer notifData) {
    System.out.println(this.toString() +
      " time = " + notifData);  }
};
```

```
Clock c = new Clock();
Employee e1 = new Employee();
Employee e2 = new Employee();
c.addObserver(e1); c.addObserver(e2);
c.notifyObservers(); // Empl@1 time = 19
                     // Empl@2 time = 19
```

- Problem : the `Clock` class can no longer extend another class.

# Example : the Observer pattern in Scala

Intent : establish a relation between objects such that when one object changes state, all its dependents are notified and updated automatically.

```
trait Observer[T] {
  def onChange(notifData: T) : Unit
}

trait Observable[T] {
  var observers:List[Observer[T]] = Nil
  def notifData() : T

  def addObserver(obs: Observer[T]) = {
    observers = obs::observers
  }
  def notifyObservers() : Unit =
    observers.foreach((obs:Observer[T])
      => obs.onChange(notifData()))
}
```

```
class Clock {
  def notifData() = // .. Get time
}

class Employee extends Observer[Int] {
  def onChange(x : Int) : Unit = {
    println(this + " time = " + x) }
}
```

```
val c = new Clock with Observable[Int]
val e1 = new Employee
val e2 = new Employee
c.addObserver(e1); c.addObserver(e2)
c.notifyObservers() // Empl@1 time = 19
                    // Empl@2 time = 19
```

- The `Clock` class is now free to implement (or not) other traits.

# Generic programming

## Polymorphism

Property of the type system where some elements of the language are compatible with a set of types.

Universal polymorphism corresponds to the case where the set of compatible types is defined by a particular structure.

- **Inclusion polymorphism** : the structure is defined by set inclusion
  Ex. : class inheritance (**Java** and **Scala**), subtyping ...

```
p1 = new JPanel();                          // Swing container (of JComponents)
l1 = new JLabel("Chat Log"); p1.add(l1);    //    JLabel <: JComponent
t1 = new JTextArea(10,10);   p1.add(t1);    // JTextArea <: JComponent
```

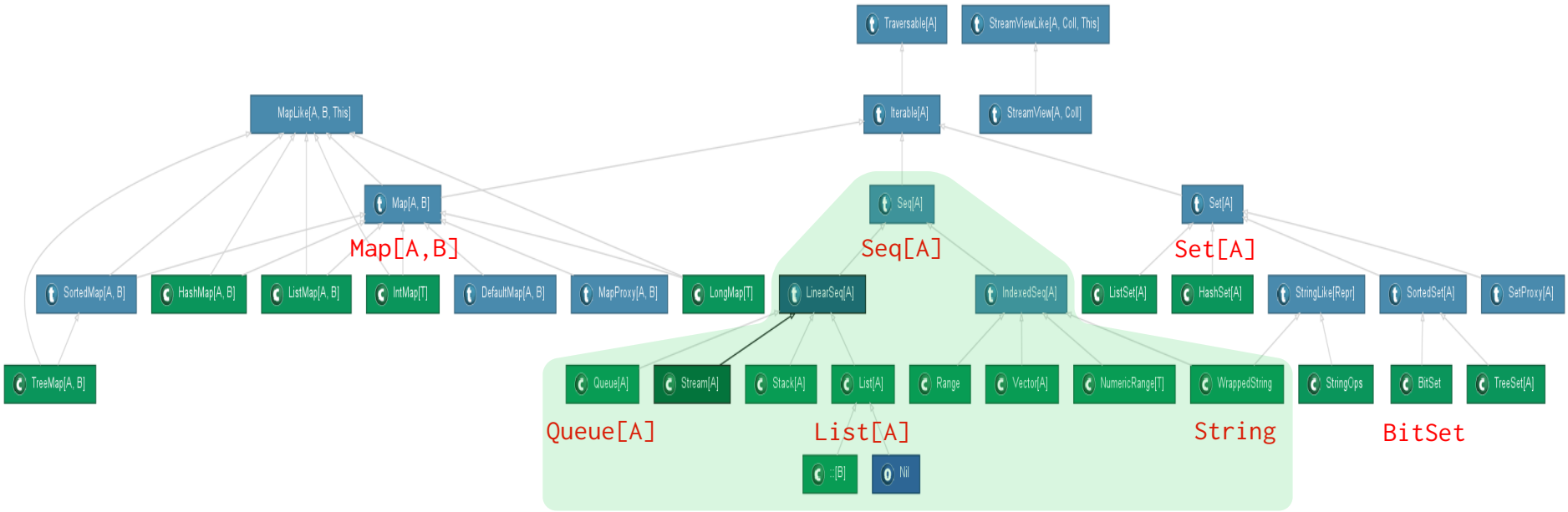
- **Parametric polymorphism** : the structure is defined by parameters
  Ex. : generics (**Java** and **Scala**), templates (**C++**), **OCaml** ...

```
abstract class List[+T]
final case object Nil                          extends List[Nothing]
final case class  Cons[T](hd:T, tl:List[T]) extends List[T]
val l = Cons(1, Cons(2, Nil))                  // -> Cons[Int] = Cons(1,Cons(2,Nil))
```

Construction naturally using parametric polymorphism : **collections**.

$$\underbrace{\texttt{Container}}_{\text{Higher-order polymorphism}}\texttt{[}\underbrace{\texttt{Containee}}_{\text{First-order polymorphism}}\texttt{]}$$

For example in **Scala** (showing only the *immutable* package) :



# Basic use of collections

- Instantiation :

```
val l = List(1,2,3,4)          // List[Int] = List(1, 2, 3, 4)
```

- Iteration :

```
for(x <- l) { println(x) }    // Prints 1 2 3 4
l.foreach((x) => println(x))  // Prints 1 2 3 4
```

- Mapping :

```
l.map((x) => x*2)             // List[Int] = List(2, 4, 6, 8)
```

- Folding :

```
l.fold(0)((x,y) => x+y)       // Int = 10, the sum of the elements of l
```

And each of these operations remains valid when replacing `List` by `Array`, (or anything inheriting from `Traversable`).

Why so many different collections ? Primarily for **efficiency**.

- Different structures, different complexities :

| Data Structure | Time Complexity | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | Average | | | | Worst | | | |
| | Index | Search | Insert | Delete | Index | Search | Insert | Delete |
| Array | 1 | n | n | n | 1 | n | n | n |
| Linked List | n | n | 1 | 1 | n | n | 1 | 1 |
| Hash Table | - | 1 | 1 | 1 | - | n | n | n |
| Binary Search Tree | log(n) | log(n) | log(n) | log(n) | n | n | n | n |
| Red-Black Tree | log(n) | log(n) | log(n) | log(n) | log(n) | log(n) | log(n) | log(n) |

Excerpt from http://bigocheatsheet.com/

- Different structures, different properties, mostly based on mutability :
  - Mutable structures $\rightarrow$ spatial efficiency, easier sharing
  - Immutable structures $\rightarrow$ no shared state, easier parallelization
- Different structures handle different data types :
  bitsets (integers), strings (characters) ...

# Generics trivia

Consider the **map** function, that transforms elements of a list.
For example in **OCaml** :

```
List.map                              (* ('a -> 'b) -> 'a list -> 'b list *)
List.map (fun x -> x+1)      [1;2;3] (* ->  [2;3;4] *)
List.map (fun x -> x mod 2) [1;2;3] (* ->  [1;0;1] *)
```

**Scala** possesses an equivalent **map** function that has the same behavior, but acts on a much richer set of collections than in **OCaml**.

What should the following calls return ?

```
val s : Set[Int] = Set(1,2,3)
s.map((x) => x % 2) // % is modulo
```

```
val s : String = "abc"
s.map ((x) => (x.toInt+1))
```

# Generics trivia

Consider the **map** function, that transforms elements of a list.
For example in **OCaml** :

```
List.map                              (* ('a -> 'b) -> 'a list -> 'b list *)
List.map (fun x -> x+1)      [1;2;3] (* ->  [2;3;4] *)
List.map (fun x -> x mod 2) [1;2;3] (* ->  [1;0;1] *)
```

**Scala** possesses an equivalent **map** function that has the same behavior, but acts on a much richer set of collections than in **OCaml**.

What should the following calls return ?

```
val s : Set[Int] = Set(1,2,3)
s.map((x) => x % 2) // % is modulo
```

```
Set[Int] = Set(1, 0)
```

```
val s : String = "abc"
s.map ((x) => (x.toInt+1))
```

# Generics trivia

Consider the **map** function, that transforms elements of a list.
For example in **OCaml** :

```
List.map                              (* ('a -> 'b) -> 'a list -> 'b list *)
List.map (fun x -> x+1)     [1;2;3] (* ->  [2;3;4] *)
List.map (fun x -> x mod 2) [1;2;3] (* ->  [1;0;1] *)
```

**Scala** possesses an equivalent **map** function that has the same behavior, but acts on a much richer set of collections than in **OCaml**.

What should the following calls return ?

```
val s : Set[Int] = Set(1,2,3)
s.map((x) => x % 2) // % is modulo
```

```
Set[Int] = Set(1, 0)
```

```
val s : String = "abc"
s.map ((x) => (x.toInt+1))
```

```
IndexedSeq[Int] = Vector(66, 67, 68)
```

# Building collections

Each **Scala** class inheriting from `Traversable` contains a `Builder` :

```
trait HasNewBuilder[+A, +Repr] extends Any {
  /* The builder that builds instances of Repr */
  protected[this] def newBuilder: Builder[A, Repr]                    }
```

... where { `Repr` is the type of the container — `Set[Int]`; `A` is the type of the contained — `Int`

```
trait Builder[-A, +Repr] extends Growable[A] {
  def +=(elem: A): this.type          /* Adds a single element to the builder. */
  def result(): Repr          /* Produces a collection from the added elements. */}
```

... which is accessible in the collection companion object :

```
val s = Set(1,2,3)         // -> Set(1,2,3)
s.companion.apply(4,2,4)   // -> Set(4,2)
```

`Builder` instances allow to write generic code, for instance filters :

```
private def filterImpl(p: A => Boolean, isFlipped: Boolean): Repr = {
  val b = newBuilder
  for (x <- this)
    if (p(x) != isFlipped) b += x
  b.result
}
def filter(p: A => Boolean): Repr    = filterImpl(p, isFlipped = false)
def filterNot(p: A => Boolean): Repr = filterImpl(p, isFlipped = true)
```

This code is inherited from `Traversable`, at the top of the hierarchy, along with the following (non exhaustive) list of functions :

- partition, take, drop, splitAt (for `Traversable`)
- reverse, intersect, distinct (for `Seq`)

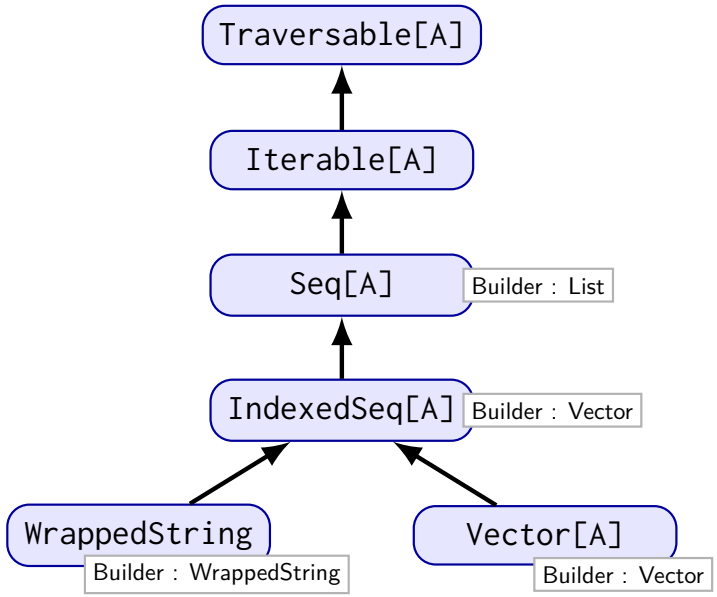
## What about the concatenation (++) of two collections ?

```
val s : WrappedString = "abc"
```

```
s ++ "de"       // -> WrappedString("abcdef")
```

```
s ++ IndexedSeq(1,2) // -> Vector(a,b,c,1,2)
```

```
s ++ Seq(1,2)        // -> Vector(a,b,c,1,2)
```




## Question : what is the result type of the concatenation ?

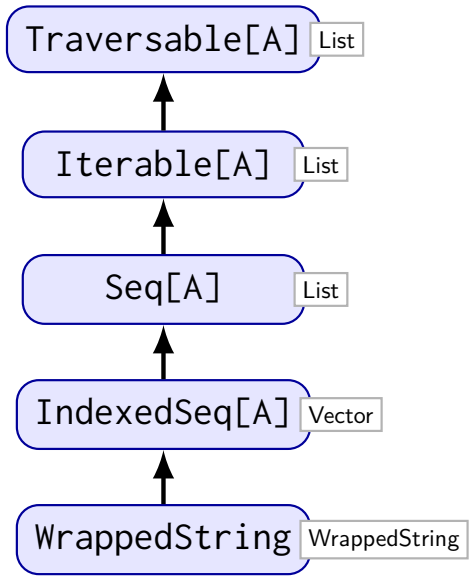
```
def ++[B >: A, That](that: GenTraversableOnce[B])
                    (implicit bf: CanBuildFrom[Repr, B, That]): That
```

[†] Promise not to ask questions about implicits for the moment.

The `CanBuildFrom` trait just contains `Builder` instances :

```
trait CanBuildFrom[-From, -Elem, +To] {
  def apply(from: From): Builder[Elem, To]
  def apply(): Builder[Elem, To] }
```

And each collection implements instances of this trait describing :

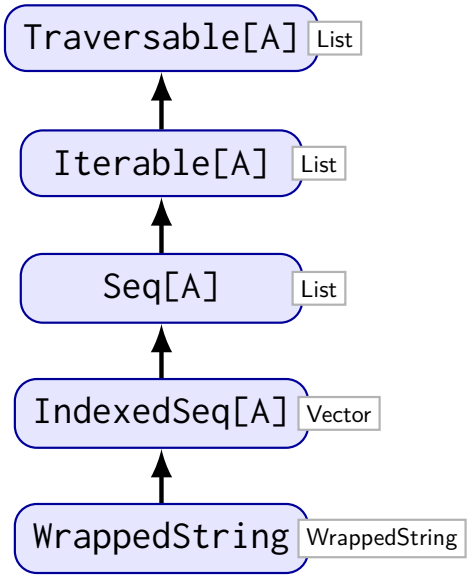



```
CBF[Traversable[_], A,    Traversable[A]]
CBF[Iterable[_],    A,    Iterable[A]]
CBF[Seq[_],         A,    Seq[A]]
CBF[IndexedSeq[_],  A,    IndexedSeq[A]]
CBF[WrappedString,  Char, WrappedString]
```

In effect, this describes a *function* on types :

(BaseCollection, ElementToAdd) $\rightarrow$ ResultCollection

```
def ++[B >: A, That](that: GenTraversableOnce[B])
                    (implicit bf: CanBuildFrom[Repr, B, That]): That
```




```
CBF[Traversable[_], A,    Traversable[A]]
CBF[Iterable[_],    A,    Iterable[A]]
CBF[Seq[_],         A,    Seq[A]]
CBF[IndexedSeq[_],  A,    IndexedSeq[A]]
CBF[WrappedString,  Char, WrappedString]
```

When concatenating two collections, the result type That is looked up :

| Repr | B | | That ? | built as : |
|---|---|---|---|---|
| WrappedString | Char | → | WrappedString | WrappedString |
| IndexedSeq[Char] | Int | → | IndexedSeq | Vector |

**Scala**'s generic implementation of **map** is particularly concise :

```
def map[B, That](f: A => B)(implicit bf: CanBuildFrom[Repr, B, That]): That = {
    def builder = {     // builder extracted for inlining
      val b = bf(repr)
      b.sizeHint(this)
      b
    }
    val b = builder
    for (x <- this) b += f(x)
    b.result
  }
```

Aside from that, this implementation has several qualities :

- it keeps the same behavior as the classical map function on lists;
- it preserves the properties of the collections such as uniqueness;
- it changes the type of the collections if necessary.

# Comparison with Java collections

- Java 7 contains in general two versions of its operations.
  For example for concatenation :
  - a standard version in the current class :

```
boolean addAll(Collection<? extends E> c)
```

  - a static version in the Collections class :

```
static <T> boolean addAll(Collection<? super T> c, T... elements)
```

- Java 8 collections support streams with generic operations, such as :
  - the concatenation function :

```
static <T> Stream<T> concat(Stream<? extends T> a, Stream<? extends T> b)
```

  - the map function :

```
<R> Stream<R> map(Function<? super T,? extends R> mapper)
```

  - conversion to another collection :

```
stream.collect(ArrayList::new, ArrayList::add, ArrayList::addAll);
```

- Keep in mind that in most cases, the Java collections are mutable.

# One word on Implicits

**Scala** possesses a mechanism for marking definitions and function parameters as **implicit**.

```
def greet(name : String)(implicit age : Int) {
  println("Captain " + name + ", you are " + age + " years old")
}
implicit def default_age : Int = 42
greet("Ahab")             // Captain Ahab, you are 42 years old
greet("Kirk")(-219)       // Captain Kirk, you are -219 years old
```

- When an **implicit** parameter is not filled in, the compiler looks up for an **implicit** definition of a compatible type in the current namespace.
- Tastes like *default parameters* (**Python**, **C++**, . . . ), but implicits are associated to a **type** and not to a parameter.

# Another word on Implicits

Implicits have a bunch of possible applications, among which :

- `CanBuildFrom` type functions are enforced seamlessly :

```
BitSet(1,2,3).map((x) => x.toString)              // Set[String] = Set(1, 2, 3)
BitSet(1,2,3).map((x) => x.toString)(Set.canBuildFrom)         // Same result
BitSet(1,2,3).map((x) => x.toString)(BitSet.canBuildFrom)      // Type error
```

- Implicit transformations for implicit type conversions :
  Ex : `Int` implicitely converted to `RichInt`

```
implicit def intWrapper(x : Int) = new runtime.RichInt(x)
1.until(5) // -> res : Range = Range(1, 2, 3, 4)
```

  Also called the "pimp my library" pattern.

- Emulation of **Haskell** type classes : a set of types linked to a function

```
trait SemiGroup[A] { def add (x : A, y : A) : A }
def double[A](x : A)(implicit m : SemiGroup[A]) : A = m.add(x, x)
implicit object ISG extends SemiGroup[Int] { def add(x:Int,y:Int):Int = x+y }
double(1) // double will only answer to Int parameters
```

# Why such an effort on the collections code ?

From a large-scale perspective :

- Factorization of code over a rich set of collections,
- Reasonable extensibility cost (in terms in lines of code),
- Very high flexibility of client code in a type-safe environment.

But on the other hand, the code itself is particularly complex, due to :

- Spaghetti code syndrome : 288 files, 412 class definitions, 292 traits, and everything scattered everywhere,
- Overly complex documentation, behavior not always obvious.

# Good reading

- M. Odersky, L. Spoon, B. Venners : *Programming in Scala*
  http://www.cs.ucsb.edu/~benh/260/Programming-in-Scala.pdf
  http://www.artima.com/pins1ed/index.html
- M. Odersky : *Scala by Example*
  http://www.scala-lang.org/docu/files/ScalaByExample.pdf
- M. Odersky : *The Scala language specification*
  http://www.scala-lang.org/files/archive/nightly/pdfs/ScalaReference.pdf