In Java a class can implement an arbitrary number of interfaces
- useful to declare that it exposes multiple abstractions and
- to implement a fictitious multiple inheritance

But . . .
- the same interface is implemented with the same code with little
  or none adaptation,
- part of that code could be unrelated to the main class and
- there isn’t an easy mechanism to reuse it

The terms mixin or concern are often used for such focused
and potentially reusable parts of an instance.

Scala provides a complete mixin solution called trait
- classes can “mix in” traits in scala as can implement interfaces in java
- traits can be mixed in as well as the instances are created

Traits preserve separation of concerns while allowing to com-
pose behaviors on demand.

As a java programmer you can see traits as
- interfaces with optional implementations or
- a “constrained” form of multiple inheritance.
### Stackable Traits

Several traits can be stacked on the same class. Let us reconsider our example focusing on such characteristics.

```scala
class Button(val label: String) {
  def click() = { /* Logic to give the appearance of clicking a button... */ }
}

trait Subject {
  type Observer = def receiveUpdate(subject: Any)
  private var observers = List[Observer]
  def addObserver(observer: Observer) = observers ::= observer
  def notifyObservers = observers foreach (...) receiveUpdate(this)
}

class ButtonCountObserver {
  var count = 0
  def receiveUpdate(subject: Any) = count += 1
}

class ObservableButton(name: String) extends Button(name) with Subject {
  override def click() = {
    super.click()
    notifyObservers
  }
}

object ButtonObserverTest {
  def main(args: Array[String]) = {
    val observableButton = new ObservableButton("Okay")
    val buttonObserver = new ButtonCountObserver
def addObserver(observer: Observer) = observers ::= observer
  }
  def notifyObservers = observers foreach (...) receiveUpdate(this)
}

val observableButton = new ObservableButton("Okay")
observableButton.addObserver(buttonObserver)
for (i <- 1 to 3) observableButton.click()
printf("The button has been clicked %d times\n", buttonObserver.count)
```

### Observer Pattern: an Example! (Cont’d)

The new trait will add:
- the possibility of putting a veto on a change (a click)

```scala
trait VetoableClicks extends Clickable {
  val maxAllowed = 1 // default
  private var count = 0
  abstract override def click() = {
    if (count < maxAllowed) { count += 1; super.click() }
  }
}

object ButtonClickableObserverVetoableTest {
  def main(args: Array[String]) = {
    val observableButton = new Button("Okay") with Subject {
      override def click() = {
        super.click()
        notifyObservers
      }
    } with VetoableClicks
    val buttonObserver = new ButtonCountObserver
    observableButton.addObserver(buttonObserver)
    for (i <- 1 to 3) observableButton.click()
    printf("The button has been clicked %d times\n", buttonObserver.count)
  }
}
```

- `super` and `abstract` again;
- it only calls the `super.click()` method when `count <= maxAllowed`.

When the mixed class is necessary just once:
- the `ObservableButton` class can be omitted
- the trait can be directly mixed into the instance.

```scala
object ButtonObserverTest {
  def main(args: Array[String]) = {
    val observableButton = new Button("Okay") with Subject {
      override def click() = {
        super.click()
        notifyObservers
      }
    } with VetoableClicks
    val buttonObserver = new ButtonCountObserver
    observableButton.addObserver(buttonObserver)
    for (i <- 1 to 3) observableButton.click()
    printf("The button has been clicked %d times\n", buttonObserver.count)
  }
}
```
Traits

- don’t support auxiliary constructors nor do they accept an argument list for the primary constructor;
- can extend classes or other traits but they can’t pass arguments to them (so they can extend only classes/traits with a no argument constructor);
- are executed every time an instance is created that uses the trait.

trait T1 { println("in T1: x = "+x); val x=1; println("in T1: x = "+x) }
trait T2 { println("in T2: y = "+y); val y="T2"; println("in T2: y = "+y) }
class Base12 { println(" in Base12: b = "+b); val b="Base12"; println(" in Base12: b = "+b) }
class C12 extends Base12 with T1 with T2 { println(" in C12: c = "+c); val c="C12"; println(" in C12: c = "+c) }

created C12:  
```scala
Creating (C12): 
in Base12: b = null  
in Base12: b = Base12  
in T1: x = 0  
in T1: x = 1  
in T2: y = null  
in T2: y = T2  
in C12: c = null  
in C12: c = C12  
After Creating C12
```