Generics

Jacek Wasilewski

Stack

```
abstract class IntStack {
    def push(x: Int): IntStack = new IntNonEmptyStack(x, this)
    def isEmpty: Boolean
    def top: Int
    def pop: IntStack
}
class IntEmptyStack extends IntStack {
    def isEmpty = true
    def top = sys.error("EmptyStack.top")
    def pop = sys.error("EmptyStack.pop")
}
class IntNonEmptyStack(elem: Int, rest: IntStack) extends IntStack {
    def isEmpty = false
    def top = elem
    def pop = rest
}
```

Generics

- Need for stack of strings
  - copy everything, rename it and change `Int` to `String`
  - code duplication
- Better would be to parameterize with the element type
  - like with passing values of functions
- Possible through the mechanism of generics
  - another way of getting polymorphic behavior
- Class is parametrized by providing type in [T]
  - T can be used instead of explicit types, like Int or String

Stack – Using Generics

```
abstract class Stack[T] {
    def push(x: T): Stack[T] = new NonEmptyStack[T](x, this)
    def isEmpty: Boolean
    def top: T
    def pop: Stack[T]
}

class EmptyStack[T] extends Stack[T] {
    def isEmpty = true
    def top = sys.error("EmptyStack.top")
    def pop = sys.error("EmptyStack.pop")
}
class NonEmptyStack[T](elem: T, rest: Stack[T]) extends Stack[T] {
    def isEmpty = false
    def top = elem
    def pop = rest
}
```

Box

- Implement a class that represents a box
- Box can hold anything
- Comparisons of object held in the box with one outside are possible
- Design similar to Stack
  - box class that defines abstract methods
  - empty box – `EmptyBox` class
  - box with an object – `NonEmptyBox` class
  - value of generic type
  - comparisons of values using >, <, etc.

Box

```
abstract class Box[T] {
    def element: T
    def put(x: T): Box[T] = new NonEmptyBox[T](x)
    def check(x: T): Boolean
}
class EmptyBox[T] extends Box[T] {
    def element = sys.error("Empty")
    def check(x: T): Boolean = false
}
class NonEmptyBox[T](e: T) extends Box[T] {
    def element = e
    def check(x: T): Boolean = element > x
}
```

Box

- `val intBox = new EmptyBox[Int]().put(5)`
  `intBox.check(6)`
- But
  `error: value > is not a member of type parameter T`
- `check` method
  `def check(x: T): Boolean = `**`element > x`**
- Method assumes that `T` is of `Nothing` type when compiling
  - it does not support comparisons
- It is possible to tell the compiler more about the type
  - upper and lower bounds of generic types

Upper Bound

- Without a bound
  `class Box[T] ...`
- **Upper bound** means that we can put a constraint on `T` that `T` is a subtype of other type, like `P`

```
T <: P
```

- With a bound

```
class Box[T <: String] ...
class Box[T <: Int] ...
```

Box – Upper Bound

- We need a bound **P** that would enable type comparisons
- Built in

```
trait Ordered[A] {
    def compare(that: A): Int
    def < (that: A): Boolean = (this compare that) < 0
    def > (that: A): Boolean = (this compare that) > 0
    def <= (that: A): Boolean = (this compare that) <= 0
    def >= (that: A): Boolean = (this compare that) >= 0
    def compareTo(that: A): Int = compare(that)
}
```

- Different types are implementing this trait

Box – Upper Bound

```
abstract class Box[T <: Ordered[T]] {
    def element: T
    def put(x: T): Box[T] = new NonEmptyBox[T](x)
    def check(x: T): Boolean
}
class EmptyBox[T <: Ordered[T]] extends Box[T] {
    def element = sys.error("Empty")
    def check(x: T): Boolean = false
}
class NonEmptyBox[T <: Ordered[T]](e: T) extends Box[T] {
    def element = e
    def check(x: T): Boolean = element > x
}
```

Lower Bound

- **Lower bound** means that we can put a constraint on `T` that we accept `T` that is a super type to other type `S`

```
T >: S
```

- Example:
  `[T >: Numeric]`
- That means `T` can be one of these:
  `Numeric, Ordering, PartialOrdering, Equiv, Serializable, Comparator, AnyRef, Any`
- These classes are parent classes to `Numeric`

Mixed Bounds

- Upper and lower bounds can be mixed

```
[T >: NonEmptyBox <: Box]
```

- `T` has to be on the interval between `Box` and `NonEmptyBox`
  - `[T <: Box]` means `T` is `Box`, `EmptyBox` or `NonEmptyBox`
  - `[T >: NonEmptyBox]` means `T` is `NonEmptyBox`, `Box`, `AnyRef`, `Any`
- All together we get that `T` is `NonEmptyBox` or `Box`

View Bounds

- If we have `[T <: S]`, our class expects type `T` to be a subtype of `S`
- `T` has to implicitly or explicitly inherit from `S`
- We use `java.lang.Integer` and want to make comparisons
  - in Scala comparisons are done through `Ordered[T]` trait – this is our `S`
  - we know that `java.lang.Integer` supports comparisons
-
  ```
  class EmptyBox[T <: Ordered[T]] ...
  val box = new EmptyBox[java.lang.Integer]()
  ```
  - does not work – why?
  - `java.lang.Integer` does not implement `Ordered`
  - possible to make compiler more generous

View Bounds

- We can use `<%` instead of `<:`
  - we accept types that implicitly converts to –in this case – `Ordered[T]`
- Code:

```
abstract class Box[T <% Ordered[T]] ...
```

  - if something converts to `Ordered[T]` we can use it as our type parameter

Co-variant Subtyping

- Hierarchy of classes
```
class A extends Ordered[A] { ... }
class B extends A { ... }
class C extends B { ... }
```
- Following boxes
```
val box1 = new EmptyBox[A]()
val box2 = new EmptyBox[B]()
val box3 = new EmptyBox[C]()
```
- Question: as `C` is a subtype of `B` and `A`, does it mean `EmptyBox[C]` is a subtype of `EmptyBox[B]` and `EmptyBox[A]`?

Co-variant Subtyping

- By default – no (non-variant subtyping)
- However it is possible to change that – co-variant subtyping
```
class Box[+T <: Ordered[T]] ...
class EmptyBox[+T <: Ordered[T]] extends Box[T] ...
class NonEmptyBox[+T <: Ordered[T]] extends Box[T] ...
```
- Now, `EmptyBox[C]` is a subtype of `EmptyBox[C]`
- It is also possible to change the direction
```
class Box[-T ...] ...
```
  - it would mean that if `C` is a subclass of `A`, `Box[A]` would be a subclass of `Box[C]` (which does not make much sense)