Implicit Conversions

```
case class Complex(real: Double, imag: Double) {
        def +(that: Complex) =
                new Complex(this.real + that.real, this.imag + that.imag)

        def -(that: Complex) =
                new Complex(this.real - that.real, this.imag - that.imag)

        override def toString = real + " + " + imag + "i"
}

implicit def Double2Complex(value: Double) = new Complex(value, 0.0)

val res = 2.0 + Complex(2, 10)   // res == Complex(4.0, 10.0)
```

Implicit Function Parameters

```
def multiplier(i: Int)(implicit factor: Int) {
        println(i * factor)
}

implicit val factor = 2

multiplier(2)                    // res: 4
multiplier(2)(3)                 // res: 6
```

Rules of Implicit Conversion

Implicit conversion are considered in:

1. If the type of an expression differs from the expected type.
2. If an object accesses a nonexistent member.
3. If an object invokes a method whose parameters don't match the given arguments.

Implicit conversion is NOT attempted:

1. No implicit conversion is used if the code compiles without it.
2. The compiler will never attempt multiple conversions.
3. Ambiguous conversions are an error.

Importing Implicits:

1. Implicit functions in the companion object of the source or target type
2. Implicit functions that are in scope as a single identifier

View Bound

View: Foo[A <% B]

```
def f[A <% B](a: A) = a.bMethod
def f[A](a: A)(implicit ev: A => B) = a.bMethod
```

Question:

```
class Container[A <% Int](val value: A) {
        def inc = value + 1
}

implicit def strToInt(x: String) = x.toInt

val c = new Container("10")
println(c.inc)                          // ???
println(c.value + 1)                    // ???
```

Context Bound

View: Foo[A : B]

```
def g[A : B](a: A) = h(a)
def g[A](a: A)(implicit ev: B[A]) = h(a)
```

View Bound by Context Bound:

```
type IntView[T] = T => Int
def f[T: IntView](x: T): Int = x
```

Implicit Evidence

View:

T =:= U - T must be equal to U
T <:< U - T must be a subtype of U
T <%< U - T must be viewable as U (2.10 depricated)

```
class Container[A](val value: A) {
        def inc(implicit ev: A =:= Int) = value + 1
        def dec(implicit ev: A => Int) = value + 1
}

val c = new Container("10")
implicit def strToInt(x: String) = x.toInt

println(c.dec)
println(c.inc)                    // compile time error: Cannot prove that String =:= Int
```

“Pimp my Lib” principle in Scala

```
val x = Array(1, 2, 3)
val y = Array(4, 5, 6)
val z = x append y

class RichArray[T](value: Array[T]) {
        def append(other: Array[T]): Array[T] = {
                val result = new Array[T](value.length + other.length)
                Array.copy(value, 0, result, 0, value.length)
                Array.copy(other, 0, result, value.length, other.length)
                result
        }
}

object RichArray {
        implicit def enrichArray[T](xs: Array[T]) = new RichArray[T](xs)
}
```

Type Classes

```
trait Plus[A] {
        def plus(x: A, y: A): A
}

def plus2[A : Plus](x: A, y: A): A = implicitly[Plus[A]].plus(x, y)


implicit object IntPlus extends Plus[Int] {
        def plus(x: Int, y: Int): Int = x + y
}

implicit object StringPlus extends Plus[Int] {
        def plus(x: String, y: String): Int = new StringBuilder(x).append(y).toString
}
```

Syntax Extension

```
trait PlusSyntax[A] {
        def plus(y: A): A
}

implicit def toPlusSyntax[A : Plus](x : A)= new PlusSyntax[A] {
        def plus(y: A) = implicitly[Plus[A]].plus(x, y)
}
```