Imperative Programming 2:
Implicits (Scala magic)

Hongseok Yang
University of Oxford

Implicits

Program fragments that are automatically inserted by the Scala compiler.

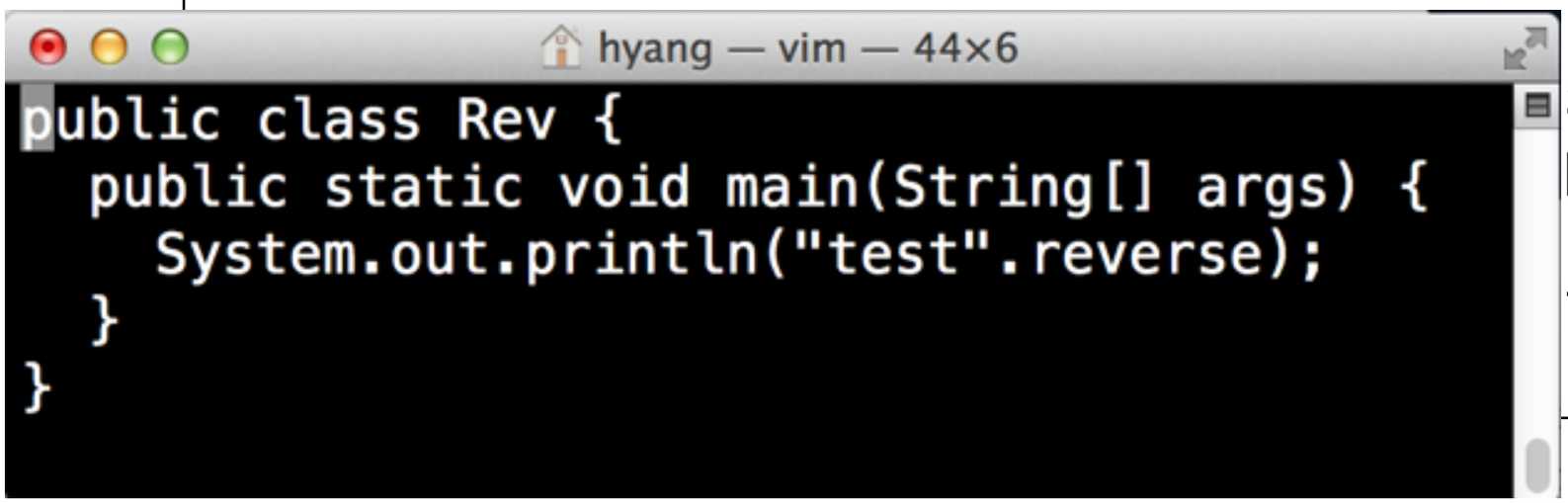
```
scala> val cs = “Computer“
cs: java.lang.String = Computer

scala> cs.reverse
res0: String = retupmoC

scala> scala.util.Sorting.stableSort(cs)
res1: Array[Char] = Array(C, e, m, o, p, r, t, u)

scala> scala.util.Sorting.stableSort(List(3,2,1))
res2: Array[Int] = Array(1, 2, 3)
```

The String class in Java does not have a reverse method.

```
scala> val cs = “Computer“
cs: java.lang.String = Computer

scala> cs.reverse
res0: String = retupmoC

scala> scala.util.Sorting.stableSort(cs)
res1: Array[Char] = Array(C, e, m, o, p, r, t, u)

scala> scala.util.Sorting.stableSort(List(3,2,1))
res2: Array[Int] = Array(1, 2, 3)
```

The String class in Java does not have a reverse method.

```
scala> val cs = “Computer“
cs: java.lang.String = Computer

scala> cs.reverse
res0: String = retupmoC
```




```
s)
p, r, t, u)

ist(3,2,1))
```

The String class in Java does not have a reverse method.

```
scala> val cs = “Computer“
cs: java.lang.String = Computer

scala> cs.reverse
res0: String = retupmoC
```

```
hyang — vim — 44×6
public class Rev {
  public static void main(String[] args) {
    System.out.println("test".reverse);
  }
}
```

```
hyang — bash — 46×8
Hongseoks-MacBook-Pro:~ hyang$ javac Rev.java
Rev.java:3: cannot find symbol
symbol  : variable reverse
location: class java.lang.String
    System.out.println("test".reverse);
                            ^
1 error
Hongseoks-MacBook-Pro:~ hyang$
```

```
scala> val cs = “Computer“
cs: java.lang.String = Computer

scala> cs.reverse
res0: String = retupmoC

scala> scala.util.Sorting.stableSort(cs)
res1: Array[Char] = Array(C, e, m, o, p, r, t, u)

scala> scala.util.Sorting.stableSort(List(3,2,1))
res2: Array[Int] = Array(1, 2, 3)
```

The same implementation of stableSort is invoked in both method calls.

It works because the Scala compiler inserts functions and provides missing parameters.

```
scala> val cs = “Computer“
cs: java.lang.String = Computer

scala> cs.reverse
res0: String = retupmoC

scala> scala.util.Sorting.stableSort(cs)
res1: Array[Char] = Array(C, e, m, o, p, r, t, u)

scala> scala.util.Sorting.stableSort(List(3,2,1))
res2: Array[Int] = Array(1, 2, 3)
```

It works because the Scala compiler inserts functions and provides missing parameters.

```
scala> val cs = “Computer“
cs: java.lang.String = Computer

scala> augmentString(cs).reverse
res0: String = retupmoC

scala> scala.util.Sorting.stableSort(cs))
res1: Array[Char] = Array(C, e, m, o, p, r, t, u)

scala> scala.util.Sorting.stableSort(List(3,2,1))
res2: Array[Int] = Array(1, 2, 3)
```

StringOp contains the reverse method.
Conversion from String to StringOp.

Footnote: Try scala -Xprint:typer

It works because the Scala compiler inserts functions and provides missing parameters.

```
scala> val cs = “Computer“
cs: java.lang.String = Computer

scala> augmentString(cs).reverse
res0: String = retupmoC

scala> scala.util.Sorting.stableSort(wrapString(cs))
res1: Array[Char] = Array(C, e, m, o, p, r, t, u)

scala> scala.util.Sorting.stableSort(List(3,2,1))
res2: Array[Int] = Array(1, 2, 3)
```

stableSort expects a parameter of Seq[T] type.
Conversion from String to WrappedString (<: Seq[Char]).

Footnote: Try scala -Xprint:typer

It works because the Scala compiler inserts functions and provides missing parameters.

```
scala> val cs = “Computer“
cs: java.lang.String = Computer

scala> augmentString(cs).reverse
res0: String = retupmoC

scala> scala.util.Sorting.stableSort(wrapString(cs))(
         reflect.this.Manifest.Char, math.this.Ordering.Char)
res1: Array[Char] = Array(C, e, m, o, p, r, t, u)

scala> scala.util.Sorting.stableSort(List(3,2,1))(
         reflect.this.Manifest.Int, math.this.Ordering.Int)
res2: Array[Int] = Array(1, 2, 3)
```

Ordering.* and Manifest.* param. say how to compare chars/ints, and create appropriate arrays for them.

Footnote: try scala -Xprint:typer

Implicit conversions.
Extension of subtyping.

```
scala> val cs = “Computer“
cs: java.lang.String = Computer

scala> augmentString(cs).reverse
res0: String = retupmoC

scala> scala.util.Sorting.stableSort(wrapString(cs))(
         reflect.this.Manifest.Char, math.this.Ordering.Char)
res1: Array[Char] = Array(C, e, m, o, p, r, t, u)

scala> scala.util.Sorting.stableSort(List(3,2,1))(
         reflect.this.Manifest.Int, math.this.Ordering.Int)
res2: Array[Int] = Array(1, 2, 3)
```

Implicit parameters.
Extension of operator overloading.

Learning outcome

- Can explain how implicit conversions and parameters work in Scala.
- Can describe the relationships among implicits, subtyping and overloading.
- Can use three programming idioms based on implicits.

Syntax

- Put "`implicit`" to the front:

```
implicit def f(...)...
implicit val x ...
implicit object C ...
def g(...)(implicit y1: C1, y2: C2, y3:C3)
```

- Only parameters in the last parenthesis can be implicit. They are all implicit, or none of them are so.

Syntax

- Put "`implicit`" to the front:

```
implicit def f(...)...
implicit val x ...
implicit object C ...
```

We provide the compiler with implicit values & methods.

```
def g(...)(implicit y1: C1, y2: C2, y3:C3)
```

- Only parameters in the last parenthesis can be implicit. They are all implicit, or none of them are so.

Syntax

- Put "`implicit`" to the front:

```
implicit def f(...)...
implicit val x ...
implicit object C ...
def g(...)(implicit y1: C1, y2: C2, y3:C3)
```

- Only parameters in the last parenthesis can be implicit. They are all implicit, or none of them are so.

The compiler use available implicit values and supply the actual parameters for `y1,y2,y3` if needed.

Syntax

- Put "`implicit`" to the front:

```
implicit def f(...)...
implicit val x ...
implicit object C ...
```

y1,y2,y3 themselves become implicit values in the body of g.

```
def g(...)(implicit y1: C1, y2: C2, y3:C3)
```

- Only parameters in the last parenthesis can be implicit. They are all implicit, or none of them are so.

How does it work?

- Compile given code without implicits.
- If it fails, insert implicits to fix compilation errors.
  1. Type mismatch : insert implicit methods.
  2. Missing implicit param. : supply implicit values.
- Always use the best-matching implicits in terms of types and scopes.

How does it work?

- Compile given code without implicits.
- If it fails, insert implicits to fix compilation errors.

1. Type mismatch : insert implicit methods.
2. Missing implicit param. : supply implicit values.

- Always use the best-matching implicits in terms of types and scopes.

Similar to how overloading and overriding work.

How does it work?

Custom subtyping

```
implicit f(x:A):B
```

- Compile given code without implicits.
- If it fails, insert implicits to fix compilation errors.

1. Type mismatch : insert implicit methods.
2. Missing implicit param. : supply implicit values.

- Always use the best-matching implicits in terms of types and scopes.

How does it work?

- Compile given code without implicits.
- If it fails, insert implicits to fix compilation errors.

1. Type mismatch : insert implicit methods.
2. Missing implicit param. : supply implicit values.

- Always use the best-matching implicits in terms of types and scopes.

Custom overloading, without code duplication

```
def sort[U](x:List[U])(implicit y:U=>Ordered[U])
```

Idioms of using implicits

1. Automatic conversion to a new type.
2. Enriching an existing type with new methods.
3. Overloading without code duplication.

Complex number

- Develop a class for complex numbers, with a + method.
- Ensure that the + method works between all combinations of complex numbers and doubles.

Example client program

```
val c1 = new Complex(1.0, 2.5)
val c2 = new Complex(-1.0, -2.0)
println(c1 + c2)
println(c1 + 3.0)
println(3.0 + c1)
```

Output

```
0.0 + 0.5i
4.0 + 2.5i
4.0 + 2.5i
```

Complex number

```
class Complex(val real: Double, val img: Double) {



}
```

Example client program

```
val c1 = new Complex(1.0, 2.5)
val c2 = new Complex(-1.0, -2.0)
println(c1 + c2)
println(c1 + 3.0)
println(3.0 + c1)
```

Output

```
0.0 + 0.5i
4.0 + 2.5i
4.0 + 2.5i
```

Complex number

```
class Complex(val real: Double, val img: Double) {
  override def toString = real +“ + “+ img +“i“




}
```

Example client program

```
val c1 = new Complex(1.0, 2.5)
val c2 = new Complex(-1.0, -2.0)
println(c1 + c2)
println(c1 + 3.0)
println(3.0 + c1)
```

Output

```
0.0 + 0.5i
4.0 + 2.5i
4.0 + 2.5i
```

Complex number

```
class Complex(val real: Double, val img: Double) {
  override def toString = real +“ + “+ img +“i“
  def +(that: Complex): Complex =
    new Complex(real + that.real, img + that.img)


}
```

Example client program

```
val c1 = new Complex(1.0, 2.5)
val c2 = new Complex(-1.0, -2.0)
println(c1 + c2)
println(c1 + 3.0)
println(3.0 + c1)
```

Output

```
0.0 + 0.5i
4.0 + 2.5i
4.0 + 2.5i
```

Complex number

```
class Complex(val real: Double, val img: Double) {
  override def toString = real +“ + “+ img +“i“
  def +(that: Complex): Complex =
    new Complex(real + that.real, img + that.img)
  def +(that: Double): Complex =
    new Complex(real + that, img)

}
```

Example client program

```
val c1 = new Complex(1.0, 2.5)
val c2 = new Complex(-1.0, -2.0)
println(c1 + c2)
println(c1 + 3.0)
println(3.0 + c1)
```

Output

```
0.0 + 0.5i
4.0 + 2.5i
4.0 + 2.5i
```

Complex number

But `3.0` is not a Complex object.
[Q] How should we handle this?

```
class Complex(val real: Double, val img: Double) {
  override def toString = real +“ + “+ img +“i“
  def +(that: Complex): Complex =
    new Complex(real + that.real, img + that.img)
  def +(that: Double): Complex =
    new Complex(real + that, img)
  ...
}
```

Example client program

```
val c1 = new Complex(1.0, 2.5)
val c2 = new Complex(-1.0, -2.0)
println(c1 + c2)
println(c1 + 3.0)
println(3.0 + c1)
```

Output

```
0.0 + 0.5i
4.0 + 2.5i
4.0 + 2.5i
```

Complex nu

But `3.0` is not a Complex object.

[Q] How should we handle this?

[A] Implicit method. Custom extension of subtyping.

```
class Complex(val real: Double, val img: Double) {
  override def toString = real +“ + “+ img +“i“
  def +(that: Complex): Complex =
    new Complex(real + that.real, img + that.img)
  def +(that: Double): Complex =
    new Complex(real + that, img)
}
implicit def Double2Complex(d: Double): Complex =
  new Complex(d, 0.0)
```

Example client program

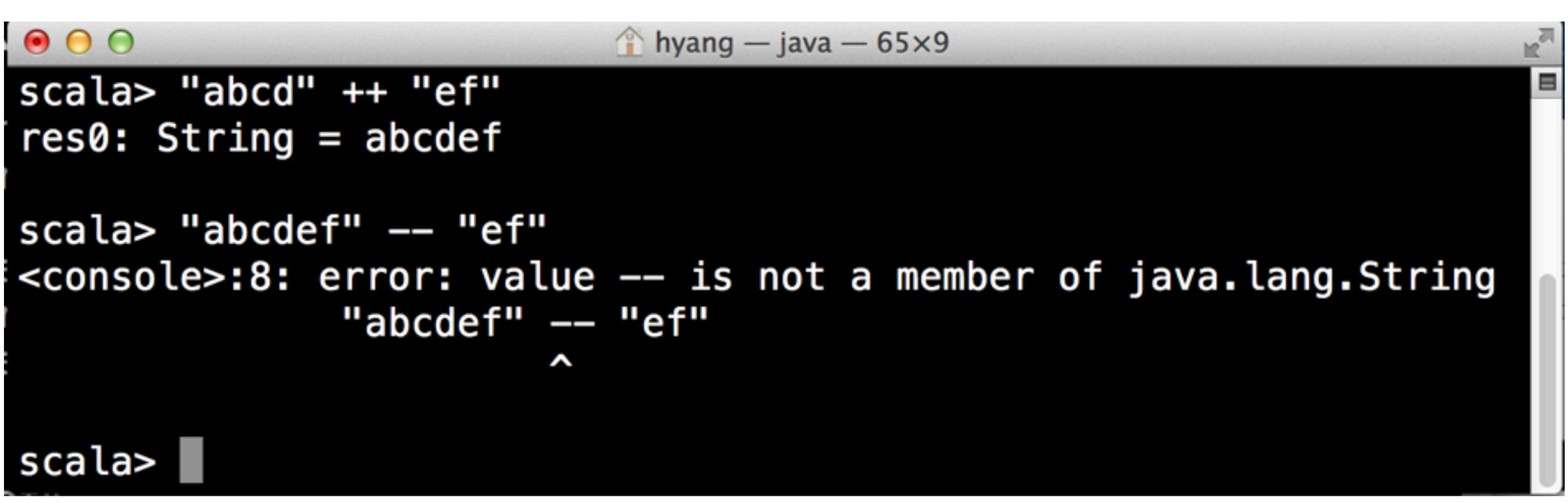
```
val c1 = new Complex(1.0, 2.5)
val c2 = new Complex(-1.0, -2.0)
println(c1 + c2)
println(c1 + 3.0)
println(3.0 + c1)
```

Output

```
0.0 + 0.5i
4.0 + 2.5i
4.0 + 2.5i
```

Adding a -- method to String

- The ++ method of a string does concatenation.
- But strings don't have a corresponding -- method.



Adding a -- method to String

- The ++ method of a string does concatenation.
- But strings don't have a corresponding -- method.

```
println("abcd" ++ "ef")
println(                    "abcdef"   -- "ef")
```

Adding a -- method to String

- The ++ method of a string does concatenation.
- But strings don't have a corresponding -- method.

```
class MyRichString(private val contents: String) {
  def --(that: String): String = contents.stripSuffix(that)
}

println("abcd" ++ "ef")
println((new MyRichString("abcdef")) -- "ef")
```

Adding a -- method to String

- The ++ method of a string does concatenation.
- But strings don't have a corresponding -- method.

```
class MyRichString(private val contents: String) {
  def --(that: String): String = contents.stripSuffix(that)
}

println("abcd" ++ "ef")
println((new MyRichString("abcdef")) -- "ef")
```

Ugly solution.
[Q] Can you do it better?

Adding a -- method to String

- The ++ method of a string does concatenation.
- But strings don't have a corresponding -- method.

```
class MyRichString(private val contents: String) {
  def --(that: String): String = contents.stripSuffix(that)
}
implicit def String2MyRichString(s: String) =
  new MyRichString(s)

println("abcd" ++ "ef")
println("abcdef" -- "ef")
```

Ugly solution.
[Q] Can you do it better?

[A] Implicit method.
Enrichment with methods.

Sum over a list of elements

- Define a `sum` function for summing elements in a list.

Example client program

```
println(sum(List(“a”, “bc”, “def”)))
println(sum(List(1,2,3)))
```

Output

```
abcdef
6
```

Sum over a list of elements

- Define a `sum` function for summing elements in a list.
- Use operator overloading.

```
object S {
  def sum(xs: List[String]): String =
    if (xs.isEmpty) "" else (xs.head ++ sum(xs.tail))
  def sum(xs: List[Int]): Int =
    if (xs.isEmpty) 0 else (xs.head + sum(xs.tail))
}
import S._
```

Example client program

```
println(sum(List("a", "bc", "def")))
println(sum(List(1,2,3)))
```

Output

```
abcdef
6
```

Sum over a list of elements

- Define a `sum` function for summing elements in a list.
- Use operator overloading. But code duplication.

```
object S {
  def sum(xs: List[String]): String =
    if (xs.isEmpty) "" else (xs.head ++ sum(xs.tail))
  def sum(xs: List[Int]): Int =
    if (xs.isEmpty) 0 else (xs.head + sum(xs.tail))
}
import S._
```

Example client program

```
println(sum(List("a", "bc", "def")))
println(sum(List(1,2,3)))
```

Output

```
abcdef
6
```

Overloading without duplication.

Example client program

```
println(sum(List(“a”, “bc”, “def”)))
println(sum(List(1,2,3)))
```

Output

```
abcdef
6
```

```
abstract class Monoid[T] {
  def unit: T
  def add(x: T, y: T): T
}
```

Overloading without duplication.

Example client program

```
println(sum(List(“a”, “bc”, “def”)))
println(sum(List(1,2,3)))
```

Output

```
abcdef
6
```

```
abstract class Monoid[T] {
  def unit: T
  def add(x: T, y: T): T
}

def sum[T](xs: List[T])(implicit m: Monoid[T]): T =
  if (xs.isEmpty) m.unit
  else m.add(xs.head, sum(xs.tail)(m))
```

Overloading without duplication.

Example client program

```
println(sum(List(“a”, “bc”, “def”)))
println(sum(List(1,2,3)))
```

Output

```
abcdef
6
```

```
abstract class Monoid[T] {
  def unit: T
  def add(x: T, y: T): T
}

def sum[T](xs: List[T])(implicit m: Monoid[T]): T =
  if (xs.isEmpty) m.unit
  else m.add(xs.head, sum(xs.tail)(m))

implicit object stringMonoid extends Monoid[String] {
  def add(x: String, y: String): String = x ++ y
  def unit: String = ""
}
```

Overloading without duplication.

Example client program

```
println(sum(List("a", "bc", "def")))
println(sum(List(1,2,3)))
```

Output

```
abcdef
6
```

```
abstract class Monoid[T] {
  def unit: T
  def add(x: T, y: T): T
}

def sum[T](xs: List[T])(implicit m: Monoid[T]): T =
  if (xs.isEmpty) m.unit
  else m.add(xs.head, sum(xs.tail)(m))

implicit object stringMonoid extends Monoid[String] {
  def add(x: String, y: String): String = x ++ y
  def unit: String = ""
}
```

Overloading without duplication.

Example client program

```
println(sum(List("a", "bc", "def")))
println(sum(List(1,2,3)))
```

```
abcdef
6
```

[Q] Do something for integer lists.

```
abstract class Monoid[T] {
  def unit: T
  def add(x: T, y: T): T
}

def sum[T](xs: List[T])(implicit m: Monoid[T]): T =
  if (xs.isEmpty) m.unit
  else m.add(xs.head, sum(xs.tail)(m))

implicit object stringMonoid extends Monoid[String] {
  def add(x: String, y: String): String = x ++ y
  def unit: String = ""
}

implicit object intMonoid extends Monoid[Int] {
  def add(x: Int, y: Int): Int = x + y
  def unit: Int = 0
}
```

Overloading without duplication.

Example client program

```
println(sum(List("a", "bc", "def")))
println(sum(List(1,2,3)))
```

```
abcdef
6
```

[Q] Do something for integer lists.

Idioms of using implicits

1. Automatic conversion to a new type.
    - Complex number.
    - WrappedString (<: Seq[Char]) class.
2. Enriching an existing type with new methods.
    - -- method. StringOp class.
3. Overloading without code duplication.
    - list-sum example. Scala collection library.

Summary

- Implicits are powerful and allow concise clean code.
- But don't forget:

  With great power, comes great responsibility.

- Read Chapter 21.

Optional reading

Section 9 of the Scala design paper:

http://www.scala-lang.org/docu/files/ScalaOverview.pdf