1 Conversions

With all the literate programming utilities in place, we will want to access different stages without always setting up a `MarkupGenerator`, reading files etc. The following conversion functions will prove useful:

⟨*⟩ ≡

```
package scalit.util

object conversions {
  <to line format>

  <to block format>
}
```

1.1 Conversions to line format

The line format will usally be the first step. It is usually either generated from a file or from standard input:

⟨to line format⟩ ≡

```
import java.io.{BufferedReader,FileReader,InputStreamReader}
import scala.util.parsing.input.StreamReader

import markup.{Line,MarkupGenerator}

def linesFromLiterateFile(filename: String): Stream[Line] = {
  val input = StreamReader(
                            new BufferedReader(
                            new FileReader(filename)))
  (new MarkupGenerator(input,filename)).lines
}

def linesFromLiterateInput(in: java.io.InputStream): Stream[Line] = {
  val input = StreamReader(new InputStreamReader(in))
  (new MarkupGenerator(input,"")).lines
}
```

We could, of course also get input in markup format. This is treated in the class `MarkupReader`:

⟨*to line format*⟩ + ≡

```
import markup.MarkupParser
def linesFromMarkupFile(filename: String): Stream[Line] = {
  val input = StreamReader(
                      new BufferedReader(
                      new FileReader(filename)))
  (new MarkupParser(input)).lines
}
def linesFromMarkupInput(in: java.io.InputStream): Stream[Line] = {
  val input = StreamReader(new InputStreamReader(in))
  (new MarkupParser(input)).lines
}
```

1.2 Conversions to block format

The block format takes a stream of lines as input, so we will have four similiar functions that just call the corresponding line generating functions.

⟨*to block format*⟩ ≡

```
import markup.{BlockBuilder,Block}
def blocksFromLiterateFile(filename: String): Stream[Block] =
  BlockBuilder(linesFromLiterateFile(filename)).blocks
def blocksFromLiterateInput(in: java.io.InputStream): Stream[Block] =
  BlockBuilder(linesFromLiterateInput(in)).blocks
def blocksFromMarkupFile(filename: String): Stream[Block] =
  BlockBuilder(linesFromMarkupFile(filename)).blocks
def blocksFromMarkupInput(in: java.io.InputStream): Stream[Block] =
  BlockBuilder(linesFromMarkupInput(in)).blocks
```

Another demand will be to just get the code blocks (for tangle, for example). We'll also have to make a (safe) downcast, unfortunately.

⟨*to block format*⟩ + ≡

```
import markup.{CodeBlock,DocuBlock}
def codeblocks(blocks: Stream[Block]): Stream[CodeBlock] =
  (blocks filter {
    case c: CodeBlock ⇒ true
    case d: DocuBlock ⇒ false
  }).asInstanceOf[Stream[CodeBlock]]
```