Part I

basics

Chapter 1

Overview

- syntax very similar to java, and some fatures like in python
- optionally typed

Chapter 2

Syntax

2.1 def

Tutorialspoint introduces it in a very messy way. Find another source explaining exactly what is this.
A keyword used to define an identifier.
Can be also used as a method return type. But a method can be also void and type-parametarised. So, what the difference????

2.2 default parameters

They provide a default value. **Must be listed after non-default parameters**:

```
method(a, b, c - 3.0, d = "wortd"){
    ...
}
```

2.3 ranges

Inclusive and exclusive, using nambers or characters:

```
a..b
a..<b   \\ exclusive
'd'..'l'
```

There are builtin methods to operate on ranges: contains(), get(), getFrom, getTo(), isReverse(), size(), subList().

2.4 string literals - triple quotes

Triple quotes can span multiple lines.

2.5 lists

`[a, b, c]`

- indexed
- standard list methods (add, get, size, contains, isEmpty, etc)

2.6 regex

`~'^a.b?c?d{2}.[eaouyi]e*[c-m]*f+[0-9]+g$'`

- a wildcard - **.**
- quantifiers:
  - types:
    - **{x}** - exactly x times
    - **?** - 0 or 1
    - ***** - 0 or many
    - **+** - 1 or many
  - usage:
    - applies to preceding character.
    - applies to preceding set
- sets
  - explicit

    `[efh47]`

  - using ranges

    `[d-k3-8]`

  - a quantifier can be applied after the set
  - default - exactly 1 element when no quantifier applied.
- $ and ^ denotes beginning and end of the line respectively

2.7 traits

Not sure, but it looks like an interface, but with concrete methods and global variables. Then a trait can be implemented in the same way like a regular interface:

```
trait abc{
   var = 15
   traitMethod(){
    doSomething();
   }

class MyClass implements abc{}
// and now var and the method doSomething are the parts of MyClass
```

2.8 closures

The anonymous methods, a lambdaexpression. Can be used as parameters or assigned to variables. Mind the syntax for parameter injected to a string and being standalone and that the closure is called with **.call()**:

```
def closure = {a -> 10 + a};
def b = closure.call(2);
println(b);                 // prints '12'

// it's just a lambda but with 0 parameters
def closure2 = {println "Hi"};
closure2.call();            // prints 'Hi'

// no brackets after println
// a parameter marked with $
def closure3 = {s -> println "Hi ${s}" };
closure3.call("Joe");        // prints 'Hi Joe'
```

2.8.1 usage with collections

The method **.each** returns a stream of the collection items, similar to foreach in java. They are passed to a closure as a parameter (multiple calls).

```
def list = [1, 2];
// this is 'x -> println(x)' under the hood
// x's are suplied by the each() method
list.each(println x);
```

This prints:
1
2

Part II

XML

Chapter 3

(

Generating XML) MarkupBuilder library provides a support for XML:

```
import groovy.xml.MarkupBuilder

class Example {
   static void main(String[] args) {
      def mp = [1 : ['Enemy Behind', 'War, Thriller','DVD','2003',
         'PG', '10','Talk about a US-Japan war'],
         2 : ['Transformers','Anime, Science Fiction','DVD','1989',
         'R', '8','A scientific fiction'],
         3 : ['Trigun','Anime, Action','DVD','1986',
         'PG', '10','Vash the Stam pede'],
         4 : ['Ishtar','Comedy','VHS','1987', 'PG',
         '2','Viewable boredom ']]

      def mB = new MarkupBuilder()

      // Compose the builder
      def MOVIEDB = mB.collection('shelf': 'New Arrivals') {
         mp.each {
            sd ->
            mB.movie('title': sd.value[0]) {
               type(sd.value[1])
               format(sd.value[2])
               year(sd.value[3])
               rating(sd.value[4])
               stars(sd.value[4])
               description(sd.value[5])
            }
         }
      }
```

```
   }
}
```

outputs:

```
<collection shelf = 'New Arrivals'>
   <movie title = 'Enemy Behind'>
      <type>War, Thriller</type>
      <format>DVD</format>
      <year>2003</year>
      <rating>PG</rating>
      <stars>PG</stars>
      <description>10</description>
   </movie>
   <movie title = 'Transformers'>
      <type>Anime, Science Fiction</type>
      <format>DVD</format>
      <year>1989</year>
  <rating>R</rating>
      <stars>R</stars>
      <description>8</description>
   </movie>
   <movie title = 'Trigun'>
      <type>Anime, Action</type>
      <format>DVD</format>
      <year>1986</year>
      <rating>PG</rating>
      <stars>PG</stars>
      <description>10</description>
   </movie>
   <movie title = 'Ishtar'>
      <type>Comedy</type>
      <format>VHS</format>
      <year>1987</year>
      <rating>PG</rating>
      <stars>PG</stars>
      <description>2</description>
   </movie>
</collection>
```

Chapter 4

Parsing XML

Supported by builtin **XmlParser** class (in groovy.util library) and its **parse()** method. Using XML files generated in previous chapter:

```
import groovy.xml.MarkupBuilder
import groovy.util.*

class Example {

   static void main(String[] args) {

      def parser = new XmlParser()
      def doc = parser.parse("D:\\Movies.xml");

      doc.movie.each{
         bk->
         print("Movie Name:")
         println "${bk['@title']}"

         print("Movie Type:")
         println "${bk.type[0].text()}"

         print("Movie Format:")
         println "${bk.format[0].text()}"

         print("Movie year:")
         println "${bk.year[0].text()}"

         print("Movie rating:")
         println "${bk.rating[0].text()}"

         print("Movie stars:")
```

```
         println "${bk.stars[0].text()}"

         print("Movie description:")
         println "${bk.description[0].text()}"
         println("*******************************")
      }
   }
}
```

This produces:

```
Movie Name:Enemy Behind
Movie Type:War, Thriller
Movie Format:DVD
Movie year:2003
Movie rating:PG
Movie stars:10
Movie description:Talk about a US-Japan war
*******************************
Movie Name:Transformers
Movie Type:Anime, Science Fiction
Movie Format:DVD
Movie year:1989
Movie rating:R
Movie stars:8
Movie description:A schientific fiction
*******************************
Movie Name:Trigun
Movie Type:Anime, Action
Movie Format:DVD
Movie year:1986
Movie rating:PG
Movie stars:10
Movie description:Vash the Stam pede!
*******************************
Movie Name:Ishtar
Movie Type:Comedy
Movie Format:VHS
Movie year:1987
Movie rating:PG
Movie stars:2
Movie description:Viewable boredom
```

Part III

JSON

Groove contains builtin libraries supporting JASON parsing and creation:

- **JsonSlurper** class supports parsing. Methods:
  - **parseText()**. We need JsonSLurper instance to use it.
- **JsonOutput** class supports generating. Methods:
  - static JsonOutput.**toJson**()

Part IV

resurces

This one is terrible poor, try to find something else:
`https://www.tutorialspoint.com/groovy/index.htm`