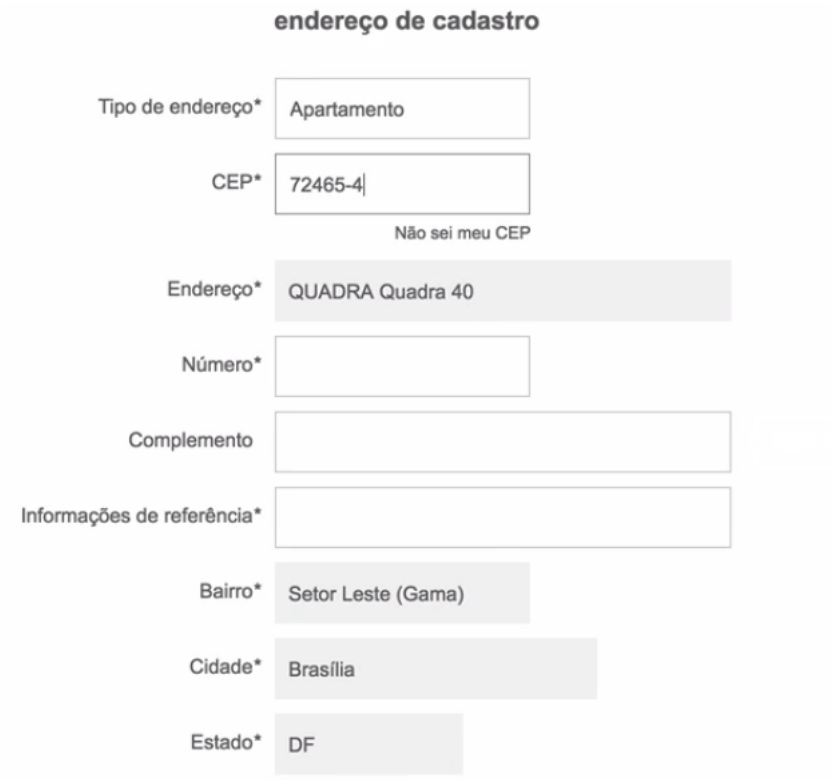
01

Integrando com o Via Cep

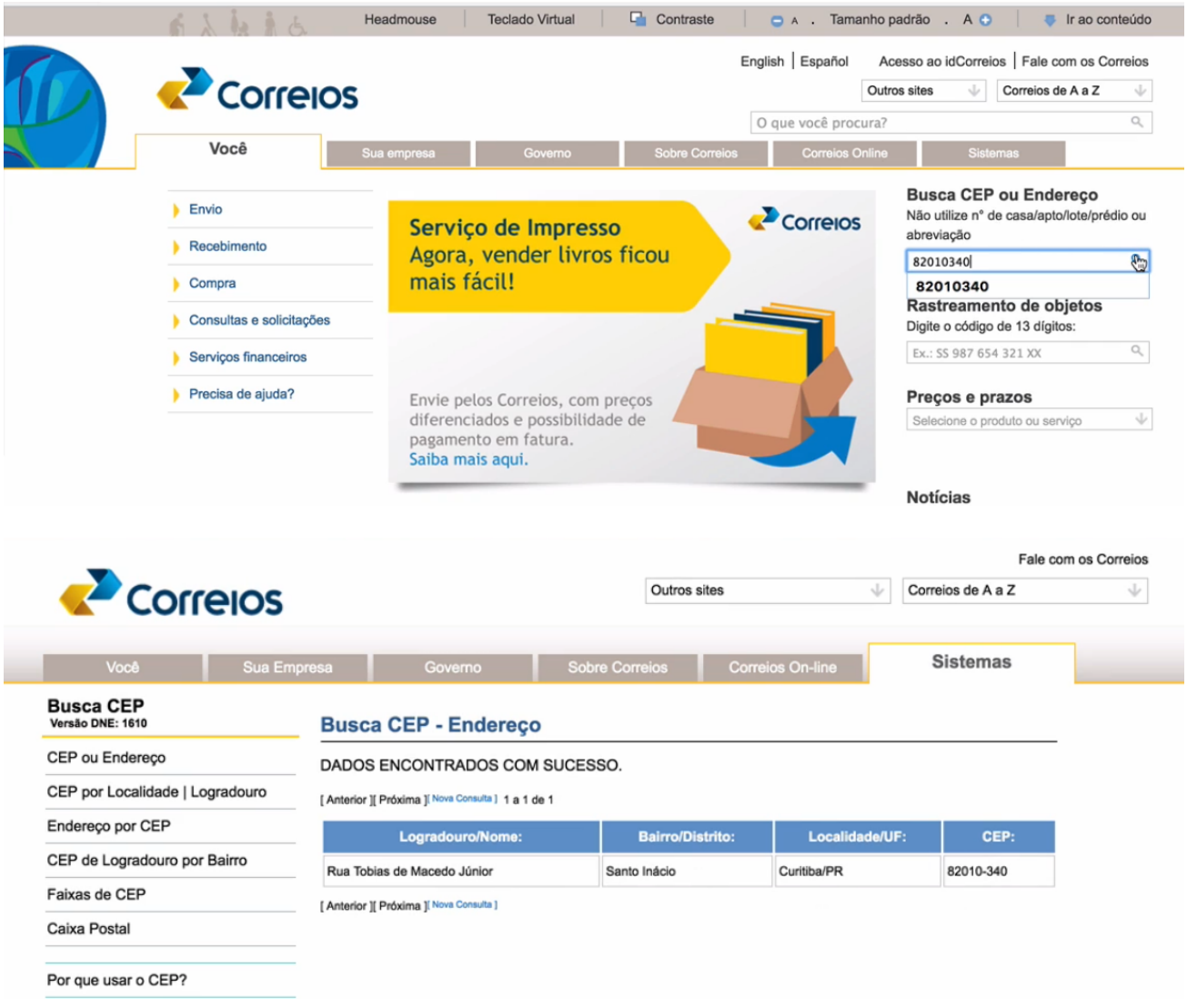
Transcrição

Mais uma das peculiaridades que temos nos documentos brasileiros é o trabalho com o CEP no Java. Tratando esse CEP, podemos trabalhar em uma base de dados ou passar para o front-end. No segundo caso, quando preenchemos um formulário com CEP, os campos de estado, cidade, bairro e endereço são preenchidos automaticamente.



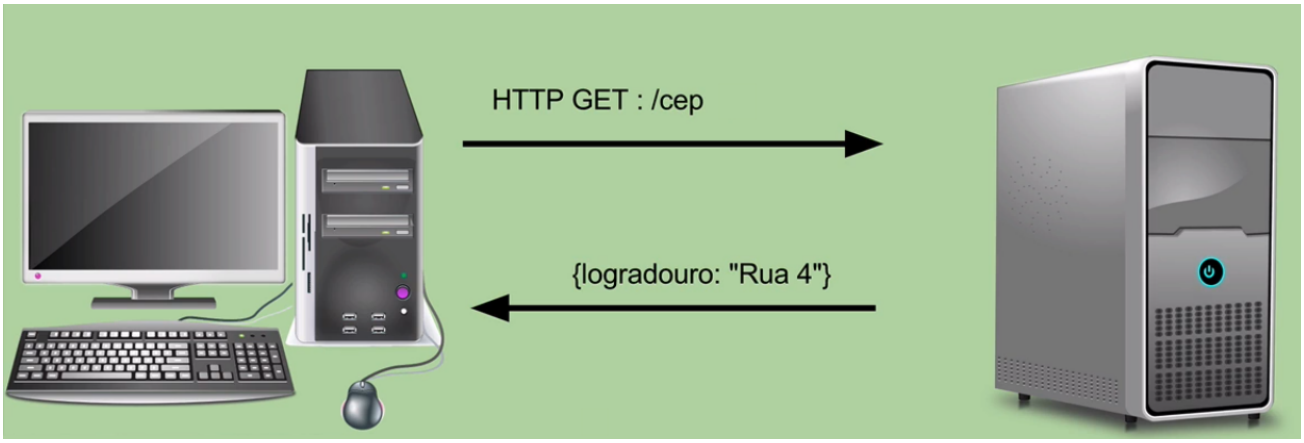
Se mudarmos o CEP, ele adequará os outros dados automaticamente.

Quem é do ramo da programação sabe que essa chamada para capturar os dados não é tão automática assim. O site dos Correios (http://www.correios.com.br/para-voce) disponibiliza uma pesquisa por CEP. Quando preenchemos, com um CEP, ela nos retorna o endereço correspondente, com bairro e localidade.

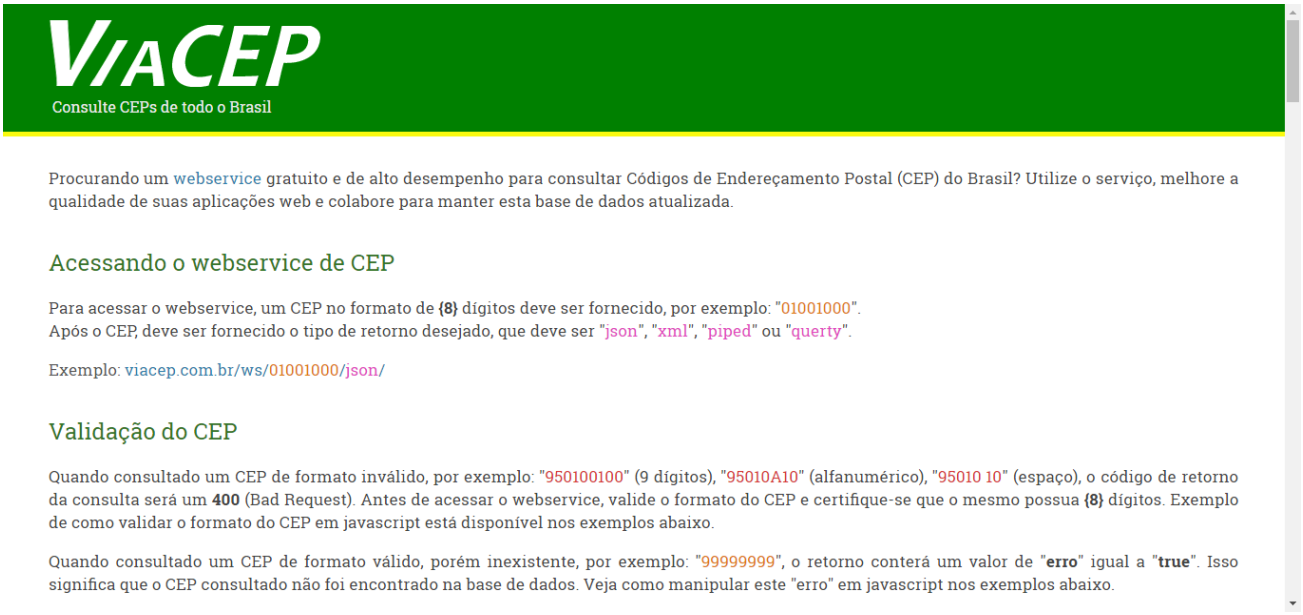


E como pegar os dados daqui, que é um HTML, e passo para um Java? Teríamos que fazer um `parse`, procurar onde está o `table` do logradouro (a `<div>logradouro</div>`), a do Bairro e assim por diante, picotando o HTML. Seria um pouco demorado fazer dessa maneira, por isso costumamos usar REST, fazendo a chamada em algum webservice que nos retorne um `.json` ou um `.xml`, que nos trazem dados mais fáceis de tratarmos. Se usarmos o HTML, virão junto com os dados que nos interessam o cabeçalho, o menu lateral... Temos aqui na Alura um curso (https://cursos.alura.com.br/course/webservices-rest-com-jaxrs-e-jersey) que ensina a consumir esse tipo de dado via REST, usando o JAX-RS e o Jersey. Vou explicar aqui o básico do REST, como ele busca um dado e como o servidor nos retorna.

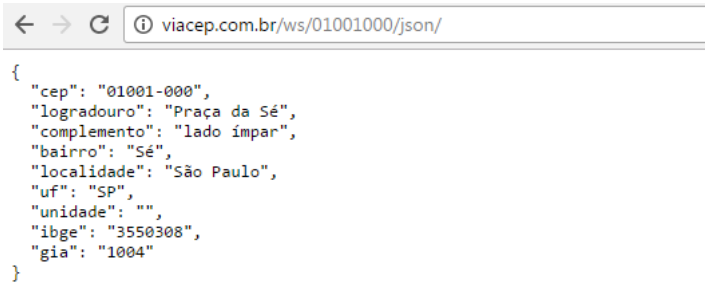
Estando em uma aplicação, em um celular, desktop ou navegador qualquer, faz-se uma chamada. A chamada é feita para o servidor em cima do protocolo HTTP, usando os verbos `get`, `post`, `delete` ou `update`. O servidor nos retorna só os dados solicitados no formato que especificamos, por exemplo um `.json`. O mais interessante é que o dado já vem tratado, diferente de todo um HTML, tornando muito mais fácil usá-lo no Java.



Tendo em mente que queremos o dado via `.json`, não podemos usar o site dos Correios, que só disponibiliza o HTML. Usaremos o ViaCEP (http://viacep.com.br/), um webservice gratuito que disponibiliza os CEPs do correio via `.json`.



A pesquisa do CEP tem o seguinte padrão `viacep.com.br/ws/01001000/json/` e deve ser feita na barra do navegador. O número do CEP consta antes do formato desejado. O resultado vem em texto. Só precisamos fazer um `parse` do que QUEREMOS.

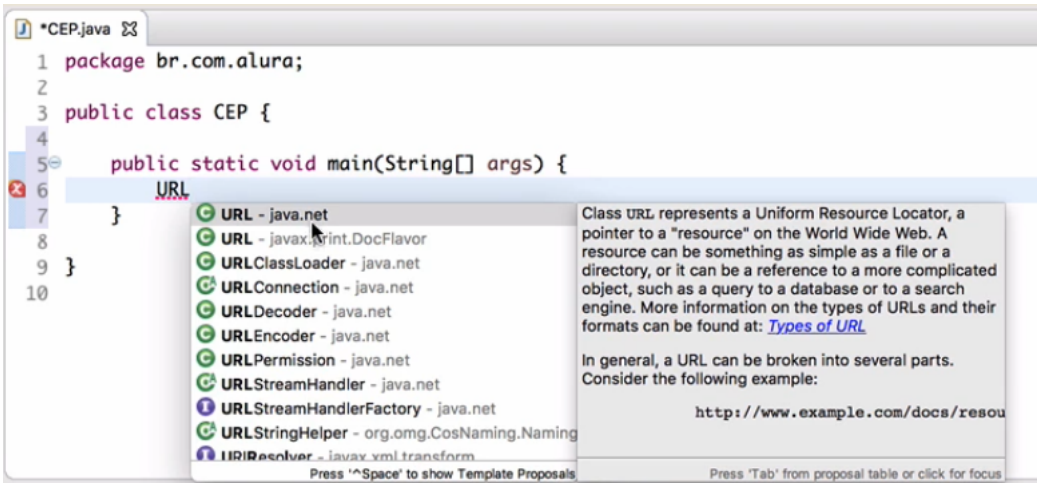


Caso desejemos os dados no formato `.xml`, basta alterar o final da URL.

Hoje em dia é mais comum trabalhar com `.json`, e é o que usaremos. Faremos uma chamada, passando a URL, no Java. Criaremos uma nova classe no nosso pacote (com o botão direito sobre ela `New > Class`). chamada `CEP`. Dentro dela, criaremos o método `main` para fazer a chamada dentro dele.

```
package br.com.alura

public class CEP {

  public static void main (String[] args) {
    URL

  }

}
```

Como a URL está no `java.net`, optaremos por ele.



A seguir, colocaremos a URL que queremos chamar, criando uma variável e a colocando em forma de string.

```
package br.com.alura

import java.net.URL;

public class CEP {

  public static void main (String[] args) {
    URL url = new URL("http://viacep.com.br/ws/01001000/xml/");

  }

}
```

O Eclipse nos avisa que há um problema de exceção e nos sugere duas soluções rápidas.

Optaremos por resolver com um `try/catch` .

```
public class CEP {

  public static void main (String[] args) {
    try {
      URL url = new URL("http://viacep.com.br/ws/01001000/xml/");
    } catch (MalformedURLException e){
      e.printStackTrace();

    }

  }

}
```

Agora é preciso abrir a conexão, com o `openConnection()` , que nos retorna um `HttpURLConnection` .

```
public class CEP {

  public static void main (String[] args) {
    try {
      URL url = new URL("http://viacep.com.br/ws/01001000/xml/");
      HttpURLConnection conexao = url.openConnection();
    } catch (MalformedURLException e){
```

```
        e.printStackTrace();

      }

    }

  }
```

Precisaremos fazer um `cast`, pois a `conexao` na verdade retorna uma `URLConnection`. E aqui também há uma exceção, então precisaremos acrescentar mais um `catch`.

```
public class CEP {

  public static void main (String[] args) {
    try {
      URL url = new URL("http://viacep.com.br/ws/01001000/xml/");
      HttpURLConnection conexao = (HttpURLConnection) url.openConnection();
    } catch (MalformedURLException e){
      e.printStackTrace();
    } catch (IOException e) {
      e.printStackTrace();
    }

  }

}
```

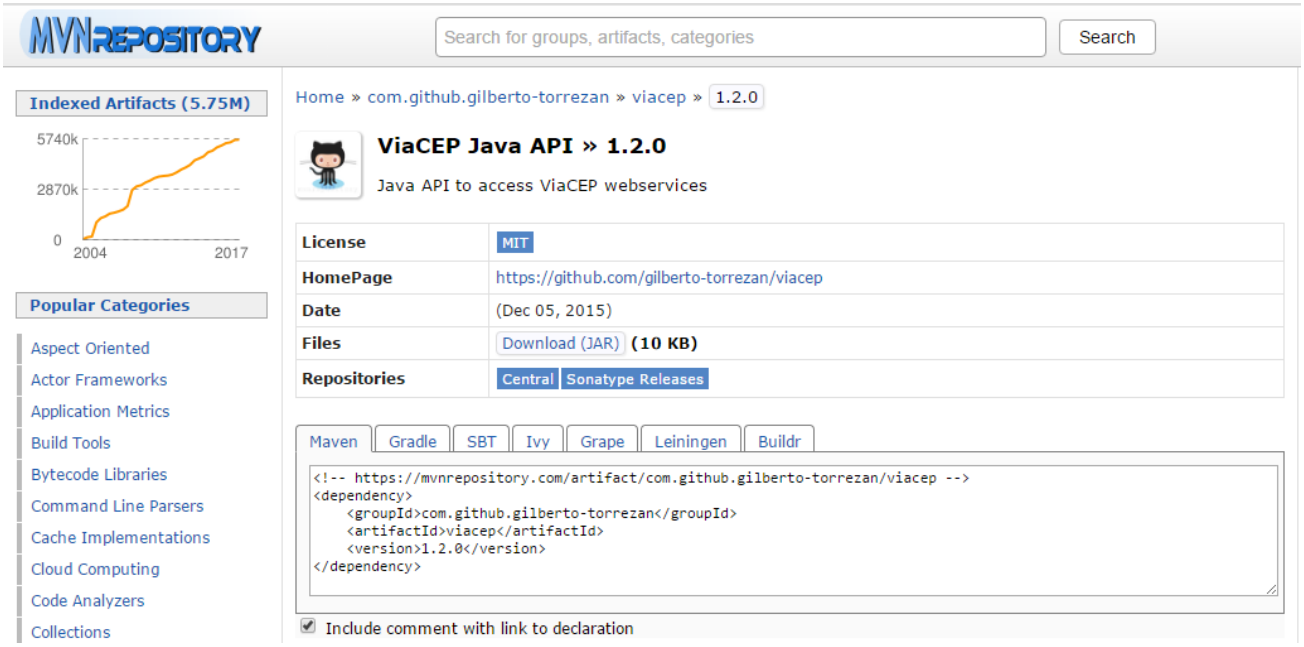
Então, poderemos configurar nossa conexão, usando `setRequestMethod()`, que recebe uma string. O método que usaremos é o `"GET"`.

```
public class CEP {

  public static void main (String[] args) {
    try {
      URL url = new URL("http://viacep.com.br/ws/01001000/xml/");
      HttpURLConnection conexao = (HttpURLConnection) url.openConnection();
      conexao.setRequestMethod("GET");
    } catch (MalformedURLException e){
      e.printStackTrace();
    } catch (IOException e) {
      e.printStackTrace();
    }

  }

}
```

Precisamos pegar os dados do servidor com o `getInputStream()`, que é mais um trabalho. No começo do curso, mencionamos que esse é o dia a dia do programador brasileiro. Será que não tem algo mais fácil?

Como muitas pessoas passam por isso, foi criada uma API para facilitar o trabalho, também chamada ViaCEP. Está disponível no GitHub (https://github.com/gilberto-torrezan/viacep). Vamos baixá-la via <a href= https://mvnrepository.com/artifact/com.github.gilberto-torrezan/viacep/1.2.0

(https://mvnrepository.com/artifact/com.github.gilberto-torrezan/viacep/1.2.0)" target="blank">Maven, na versão mais atual. Basta copiar o código correspondente da dependência.



Ele será colado no `pom.xml`, depois da dependência do Moneta.

```
<project xmlns="http://maven.apache.org/POM/4.0.0""xmlns:xsi="http://www.w3.org/2001/XMLSchema-
  <modelVersion>4.0.0</modelversion>
  <groupId>br.com.alura.brasileirice</groupId>
  <artifactId>Brasileirice</artifactId>
  <version>0.0.1-SNAPSHOT</version>

  <dependencies>
    <dependency>
        <groupId>br.com.caelum.stella</groupId>
        <artifactId>caelum-stella-core</artifactId>
        <version>2.1.2</version>
      </dependency>
    <dependency>
      <groupId>org.javamoney</groupId>
      <artifactId>moneta</artifactId>
      <version>1.1</version>
    </dependency>
    <dependency>
        <groupId>com.github.gilberto-torrezan</groupId>
        <artifactId>viacep</artifactId>
        <version>1.2.0</version>
    </dependency>
  </dependencies>

</project>
```

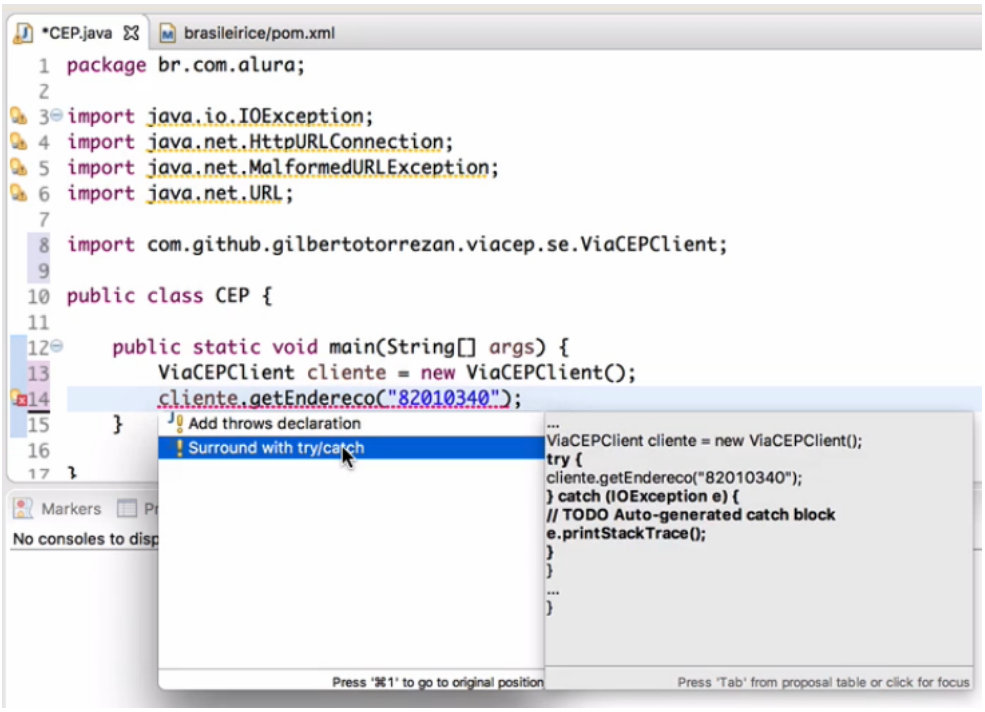
Assim que o ViaCEP aparecer dentre as dependências do Maven, poderemos substituir todo aquele código que digitamos. Usaremos `ViaCEPClient`, pois somos os consumidores de um servidor.

```
  public static void main(String[] args) {
    ViaCEPClient cliente = new ViaCEPClient();
}
```

Agora usaremos o `getEndereco(cep)` , para que possamos pegar o endereço com base no CEP.

```
  public static void main(String[] args) {
    ViaCEPClient cliente = new ViaCEPClient();
    cliente.getEndereco(82010340);
}
```

O Eclipse novamente nos avisa de um erro. Colocaremos a solicitação dentro de um `try/catch` . O importante é que ele retorna um `ViaCEPEndereco` , que é efetivamente o endereço da chamada, e a partir dele conseguimos pegar o logradouro, o bairro, e as demais informações.




```
  public static void main(String[] args) {
    ViaCEPClient cliente = new ViaCEPClient();
    try {
      ViaCEPEndereco endereco = cliente.getEndereco(82010340);
    } catch (IOException E) {
      e.printStackTrace();
    }
}
```

Quando clicamos em `ViaCEPEndereco` com o `Ctrl` pressionado, vemos a classe como um todo.

```
public class ViaCEPEndereco implements Serializable {

  private static final long serialVersionUID = 1L;

  private String cep;
```

```
  private String logradouro;
  private String complemento;
  private String bairro;
  private String localidade;
  private String uf;
  private String ibge;

  public String getCep() {
    return cep;
  }
}
```

Todos os dados ali listados podem ser trazidos para o método `main`. Vamos imprimir em nosso código o bairro.

```
  public static void main(String[] args) {
    ViaCEPClient cliente = new ViaCEPClient();
    try {
      ViaCEPEndereco endereco = cliente.getEndereco(82010340);
      System.out.println(endereco.getBairro());
    } catch (IOException E) {
      e.printStackTrace();
    }
}
```

Agora basta salvar e rodar para obter a informação desejada. O console nos mostrará o seguinte:

```
Santo Inácio
```

Vamos testar também com o logradouro.

```
  public static void main(String[] args) {
    ViaCEPClient cliente = new ViaCEPClient();
    try {
      ViaCEPEndereco endereco = cliente.getEndereco("82010340");
      System.out.println(endereco.getLogradouro());
    } catch (IOException E) {
      e.printStackTrace();
    }
}
```

Ao rodar, o console nos mostrará:

```
Rua Tobias de Macedo Júnior
```

Todo o procedimento complexo é feito pelo simples `getEndereco`, facilitando o trabalho para nós. Quando clicamos em `ViaCEPClient` com o `Ctrl` pressionado, vemos que todo o código que estávamos criando está contido na classe.

```
...
String urlString = getHost() + cep + "/json/";
URL url = new URL(urlString);
```

```
HttpURLConnection URLConnection = (HttpURLConnection) url.openConnection;
try{
  InputStream in = new BufferedInputStream(URLConnection.getInputStream());
  ViaCEPEndereco obj = getService().benFrom(ViaCEPEndereco.class, in);
  if(obj == null || obj.getCep() == null){
    return null;
  }
  return obj;
}
```

A API serve justamente para que não tenhamos que criar todo aquele código. Ela cria o chamado para nós, e fica bem mais simples consumir a API que implementar tudo. Até a próxima!