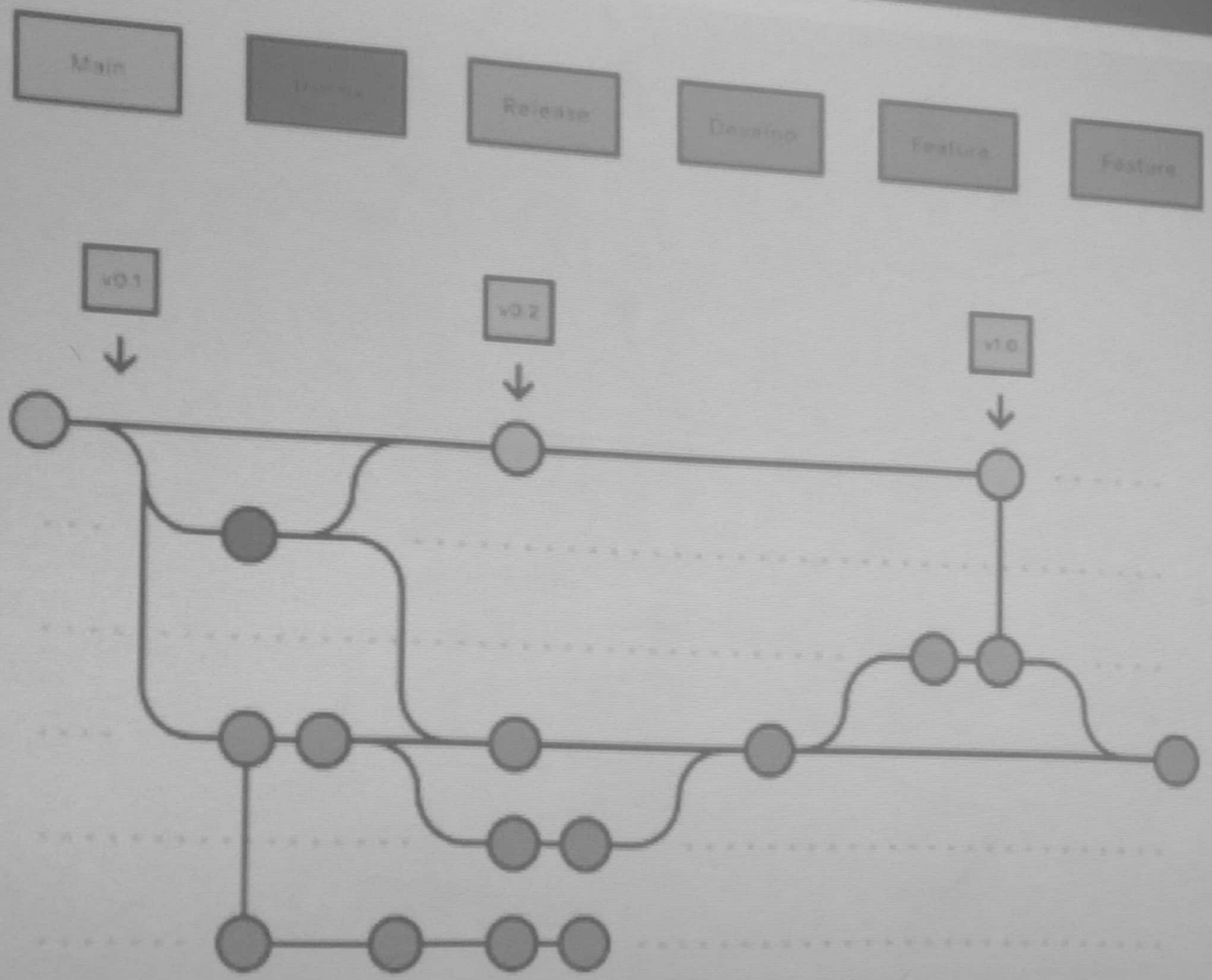
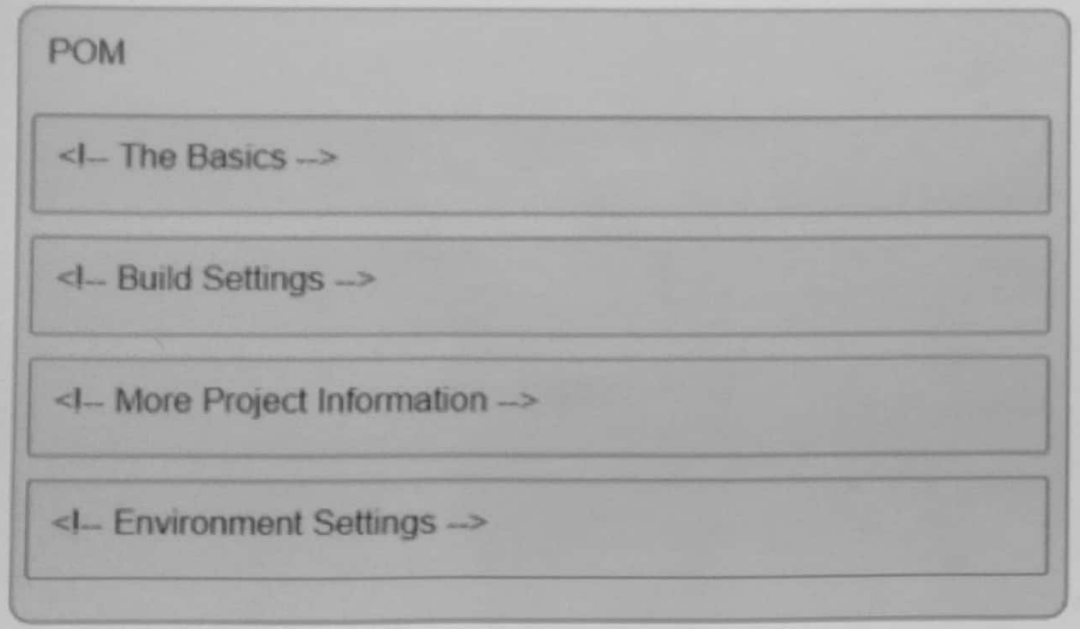


Gitflow is a legacy **Git workflow that was originally a disruptive and novel strategy for managing Git branches**. Gitflow is an alternative Git branching model that **involves the use of feature branches and multiple primary branches**. Compared to trunk-based development, Gitflow has numerous, longer-lived branches and larger commits. **Under this model, developers create a feature branch and delay merging it to the main** trunk branch until the feature is complete. These long-lived feature branches require more collaboration to merge and have a higher risk of deviating from the trunk branch.

MAVEN - POM



POM stands for "**Project Object Model**". It is an XML representation of a Maven project held in a file named pom.xml. When in the presence of Maven folks, **speaking of a project is speaking in the philosophical sense, beyond a mere collection of files containing code**. A project contains configuration files, as well as the developers involved and the roles they play, the defect tracking system, the organization and licenses, the URL of where the project lives, the project's dependencies, and all of the other little pieces that come into play to give code life. It is a one-stop-shop for all things concerning the project. In fact, **in the Maven world, a project does not need to contain any code at all, merely a pom.xml**.

MAVEN - POM - The Basics

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
https://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>org.codehaus.mojo</groupId>
  <artifactId>my-project</artifactId>
  <version>1.0</version>
...
  <packaging>jar</packaging>
...
</project>
```

groupId: This is generally unique amongst an organization or a project. For example, all core Maven artifacts do (well, should) live under the groupId org.apache.maven. Group ID's do not necessarily use the dot notation, for example, the junit project. Note that the dot-notated groupId does not have to correspond to the package structure that the project contains. It is, however, a good practice to follow.

artifactId: The artifactId is generally the name that the project is known by. Although the groupId is important, people within the group will rarely mention the groupId in discussion. Along with the groupId, the artifactId fully defines the artifact's living quarters within the repository.

version: This is the last piece of the naming puzzle.In short: code changes, those changes should be versioned, and this element keeps those versions in line. It is also used within an artifact's repository to separate versions from each other.

packaging: Now that we have our address structure of groupId:artifactId:version, there is one more standard label to give us a really complete what: that is the project's packaging.

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
https://maven.apache.org/xsd/maven-4.0.0.xsd">
  ...
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.12</version>
      <type>jar</type>
      <scope>test</scope>
      <optional>true</optional>
    </dependency>
    ...
  </dependencies>
  ...
</project>
```

groupId, artifactId, version: You will see these elements often. This trinity is used to compute the Maven coordinate of a specific project in time, demarcating it as a dependency of this project. The purpose of this computation is to select a version that matches all the dependency declarations.

groupId, artifactId: directly the corresponding coordinates of the dependency.

version: a dependency version requirement specification, that is used to compute the dependency's effective version.

type: Corresponds to the chosen dependency type. This defaults to jar. While it usually represents the extension on the filename of the dependency, that is not always the case: a type can be mapped to a different extension and a classifier.

MVN - POM - Dependencies - Exclusions

TUP UTN

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
https://maven.apache.org/xsd/maven-4.0.0.xsd">
  ...
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.12</version>
      <exclusions>
        <exclusion>
          <groupId>org.apache.maven</groupId>
          <artifactId>maven-core</artifactId>
        </exclusion>
      </exclusions>
    </dependency>
    ...
  </dependencies>
  ...
</project>
```

It is sometimes useful to limit a dependency's transitive dependencies. A dependency may have incorrectly specified scopes, or dependencies that conflict with other dependencies in your project. Exclusions tell Maven not to include a specified artifact in the classpath even if it is a dependency of one or more of this project's dependencies (a transitive dependency).

exclusions: Exclusions contain one or more exclusion elements, each containing a groupId and artifactId denoting a dependency to exclude. Unlike optional, which may or may not be installed and used, exclusions actively remove artifacts from the dependency tree.

MAVEN - POM - Inheritance

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
https://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <parent>
    <groupId>org.codehaus.mojo</groupId>
    <artifactId>my-parent</artifactId>
    <version>2.0</version>
  </parent>

  <artifactId>my-project</artifactId>
</project>
```

One powerful addition that Maven brings to build management is the concept of project inheritance. Although in build systems such as Ant inheritance can be simulated, Maven makes project inheritance explicit in the project object model.

MVN - POM - Properties

```
<project>
  ...
  <properties>
    <maven.compiler.source>1.7</maven.compiler.source>
    <maven.compiler.target>1.7</maven.compiler.target>
    <!-- Following project.-properties are reserved for Maven in will become
elements in a future POM definition. -->
    <!-- Don't start your own properties properties with project. -->
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

<project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
  </properties>
  ...
</project>
```

Properties are the last required piece to understand POM basics. Maven properties are value placeholders. Their values are accessible anywhere within a POM by using the notation ${X}, where X is the property.
They come in five different styles:

env.X: Prefixing a variable with "env." will return the shell's environment variable. For example, ${env.PATH} contains the PATH environment variable.

project.x: A dot (.) notated path in the POM will contain the corresponding element's value. For example: <project><version>1.0</version></project> is accessible via ${project.version}.

settings.x: A dot (.) notated path in the settings.xml will contain the corresponding element's value. For example: <settings><offline>false</offline></settings> is accessible via ${settings.offline}.

Java System Properties: All properties accessible via java.lang.System.getProperties() are available as POM properties, such as ${java.home}.

x: Set within a <properties /> element in the POM. The value of <properties><someVar>value</someVar></properties> may be used as ${someVar}.