

# Easier

- **Enhanced configuration**
  - **YAML**
  - **Typesafe configuration**
  - **Resolving configuration**

YAML™

---

A data serialization standard made for configuration files!



- **Pronounced YAM-EL, rhymes with Camel**
- **Data serialization language / format**
- **Since 2001**
- **Ruby, Python, ElasticSearch, MongoDB**

# YAML .properties

- **Defined spec:** <http://yaml.org/spec/>
- Human readable
- **key/value (Map), Lists, and Scalar types**
- Used in many languages
- **Hierarchical**
- Doesn't work with @PropertySource
- **Multiple Spring Profiles in default config**

- **java.util.Properties Javadoc is spec**
- Human readable
- **key/value (Map) and String types**
- Used primarily in Java
- **Non-hierarchical**
- Works with @PropertySource
- **One Spring Profile per config**

.properties

```
some_key=value  
some_number=9  
some_bool=true
```

.yaml

```
some_key: value  
some_number: 9
```

```
# could use values yes or on  
some_bool: true
```



## YAML Basics: Key/Value Scalars

- .properties **keys** and **values** are **Strings**
- .yaml **keys** are **Strings** and **values** are their **respective type**

.properties

# A map

**somemap.key**=value

**somemap.number**=9

# Another map

**map2.bool**=true

**map2.date**=2016-01-01

.yaml

# A map

**somemap:**

**key:** value

**number:** 9

# Inline map

**map2:** {**bool**=true, **date**=2016-01-01}



## YAML Basics: Maps

- .properties **uses dots to denote hierarchy** (a Spring convention)
- .yaml **uses hierarchy (consistent spaces) to create maps**

.properties

# A list

**numbers[0]**=one

**numbers[1]**=two

# Inline list

**numbers**=one,two

.yaml

# A list

**numbers:**

- one

- two

# Inline list

**numbers:** [one,two]



## YAML Basics: Lists

- .properties **uses prop[index] or commas for a List (a Spring convention)**
- .yaml **uses '- value' or commas surrounded with brackets for a List**

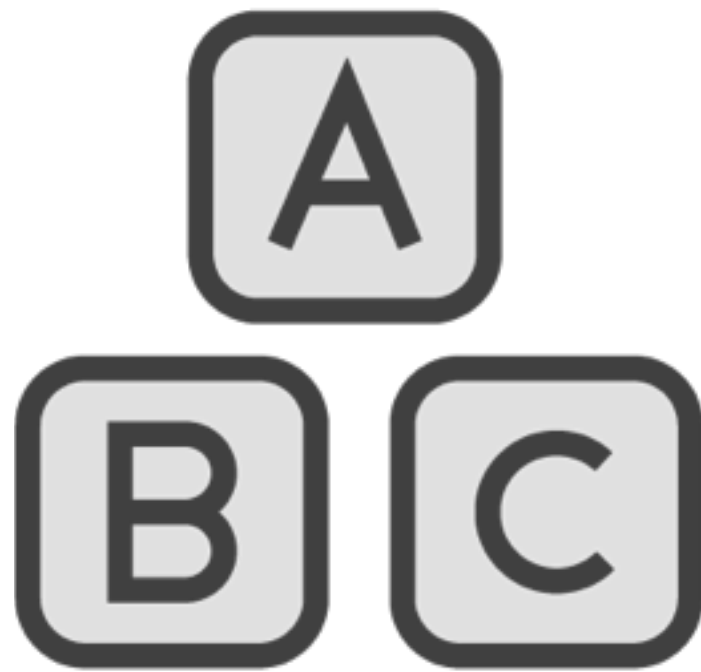
What should I use,  
Properties or YAML?



# Typesafe Configuration

---

# Getting Started with `@ConfigurationProperties`



- a. Annotate with `@ConfigurationProperties`**
- b. Define getters & setters (JavaBean Spec)**
- c. Annotate with `@Component`**
  - i. Can also use `@EnableConfigurationProperties`**

**@ConfigurationProperties**  
turns all of your application  
configuration into typesafe  
POJOs

.properties

.yaml

```
my.feature-enabled=true
```

```
my:  
  feature-enabled: true
```



Using the Above Configuration...



# Annotate class with `@ConfigurationProperties`

```
1
2 @ConfigurationProperties(prefix = "my")
3 public class MyConfig
4 {
5
6
7
8
9
10
11
12
13
14 }
```

# Create an Instance Variable for Your Property

```
1
2 @ConfigurationProperties(prefix = "my")
3 public class MyConfig
4 {
5     private Boolean featureEnabled;
6
7
8
9
10
11
12
13
14 }
```



# Define a Getter and a Setter for Your Property

```
1
2 @ConfigurationProperties(prefix = "my")
3 public class MyConfig
4 {
5     private Boolean featureEnabled;
6
7     public Boolean getFeatureEnabled() {
8         return featureEnabled;
9     }
10
11     public void setFeatureEnabled(Boolean featureEnabled) {
12         this.featureEnabled = featureEnabled;
13     }
14 }
```

# Annotate class with **@Component**

```
1 @Component
2 @ConfigurationProperties(prefix = "my")
3 public class MyConfig
4 {
5     private Boolean featureEnabled;
6
7     public Boolean getFeatureEnabled() {
8         return featureEnabled;
9     }
10
11     public void setFeatureEnabled(Boolean featureEnabled) {
12         this.featureEnabled = featureEnabled;
13     }
14 }
```



Or ... Use **@EnableConfigurationProperties**

```
1 @SpringBootApplication
2 @EnableConfigurationProperties(MyConfig.class)
3 public class MyApplication {
4     ...
5 }
```

# Autowire It into Any Class

```
1 @Service
2 public class MyService
3 {
4     @Inject
5     private MyConfig config;
6
7     ...
8 }
```



“Maps and Collections can be expanded with only a getter, whereas arrays require a setter.”

**Spring Boot Reference Documentation**

# Configuring Your @ConfigurationProperties

Attributes		
✓	boolean	exceptionIfInvalid = true
✓	boolean	ignoreInvalidFields = false
✓	boolean	ignoreNestedProperties = false
✓	boolean	ignoreUnknownFields = true
✓	[]	locations = [ "..."]
✓	boolean	merge = true
✓	String	prefix value="some.namespace"

# Validating Your Configuration



- **Simply annotate your instance variables with JSR-303 Annotations**
  - @NotNull
  - @Pattern
  - @Max
  - @Min
  - @Digits
  - **And more**

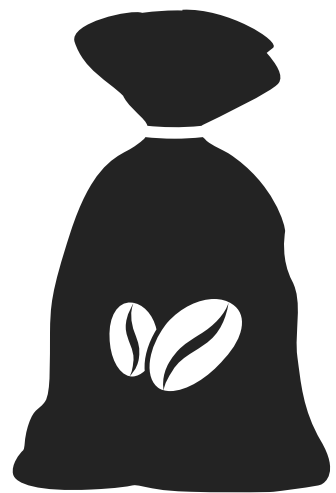
@ConfigurationProperties  
aren't only limited to beans  
you create. You can use them  
to configure third party beans  
too!

```
1 @Configuration
2 public class MyConfig
3 {
4     @Bean
5     @ConfigurationProperties(
6         prefix = "config.some-bean")
7     public SomeBean someBean()
8     {
9         // Has getters & setters
10        return new SomeBean()
11    }
12 }
```

**application.properties**

# someBean has setFirstName method  
**config.some-bean.first-name=Dustin**

# someBean has setLastName method  
**config.some-bean.last-name=Schultz**



## Configuring Third Party Beans

# Resolving Configuration

---





# Relaxed Configuration Names

## Camel Case

`featureEnabled`

## Dash Notation

`feature-enabled`

## Underscore

`PREFIX_FEATURE_ENABLED`

Spring Boot provides a  
standard cascading  
resolution of configuration.

# Resolving Configuration in Spring Boot



# 1.) Command Line Arguments



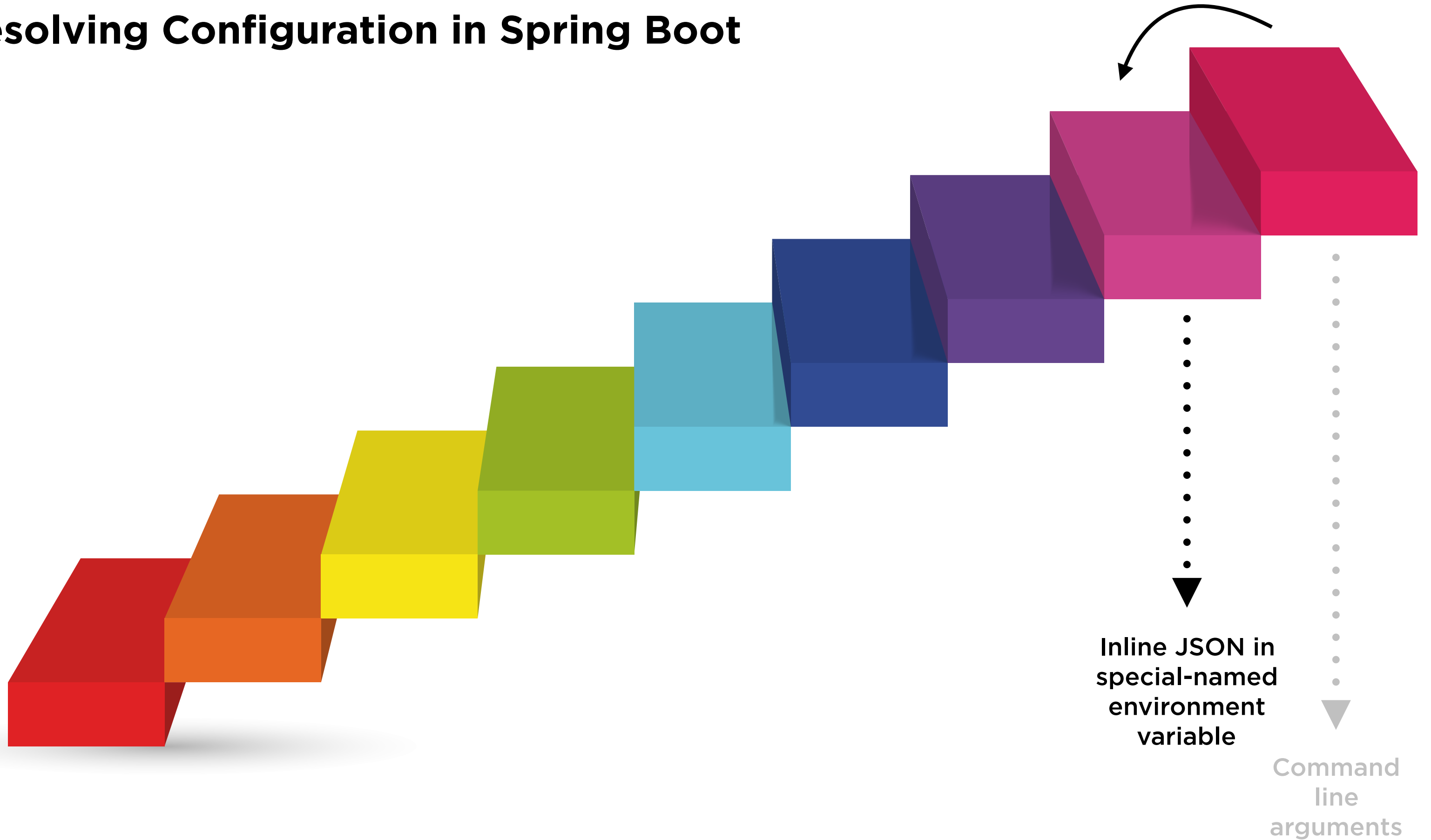
- **Prefix any property with a double dash**

- `--server.port=9000`

- `--spring.config.name=config`

- `--debug`

# Resolving Configuration in Spring Boot

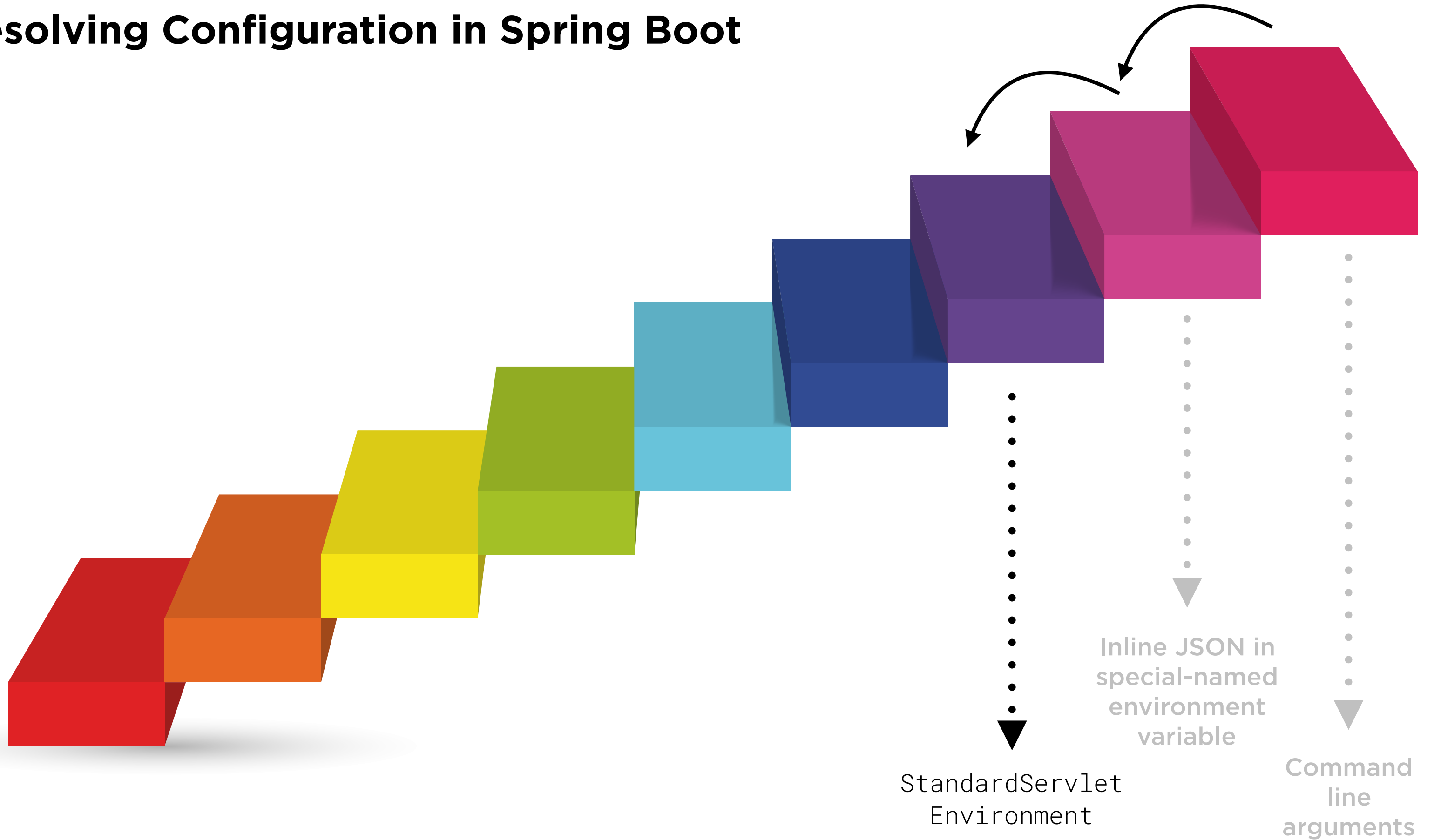


## 2.) Embedded JSON in SPRING\_APPLICATION\_JSON



- `SPRING_APPLICATION_JSON=<JSON_STRING>`
  - e.g. `SPRING_APPLICATION_JSON=`  
`' {"server": {"port": "9000" }} '`

# Resolving Configuration in Spring Boot



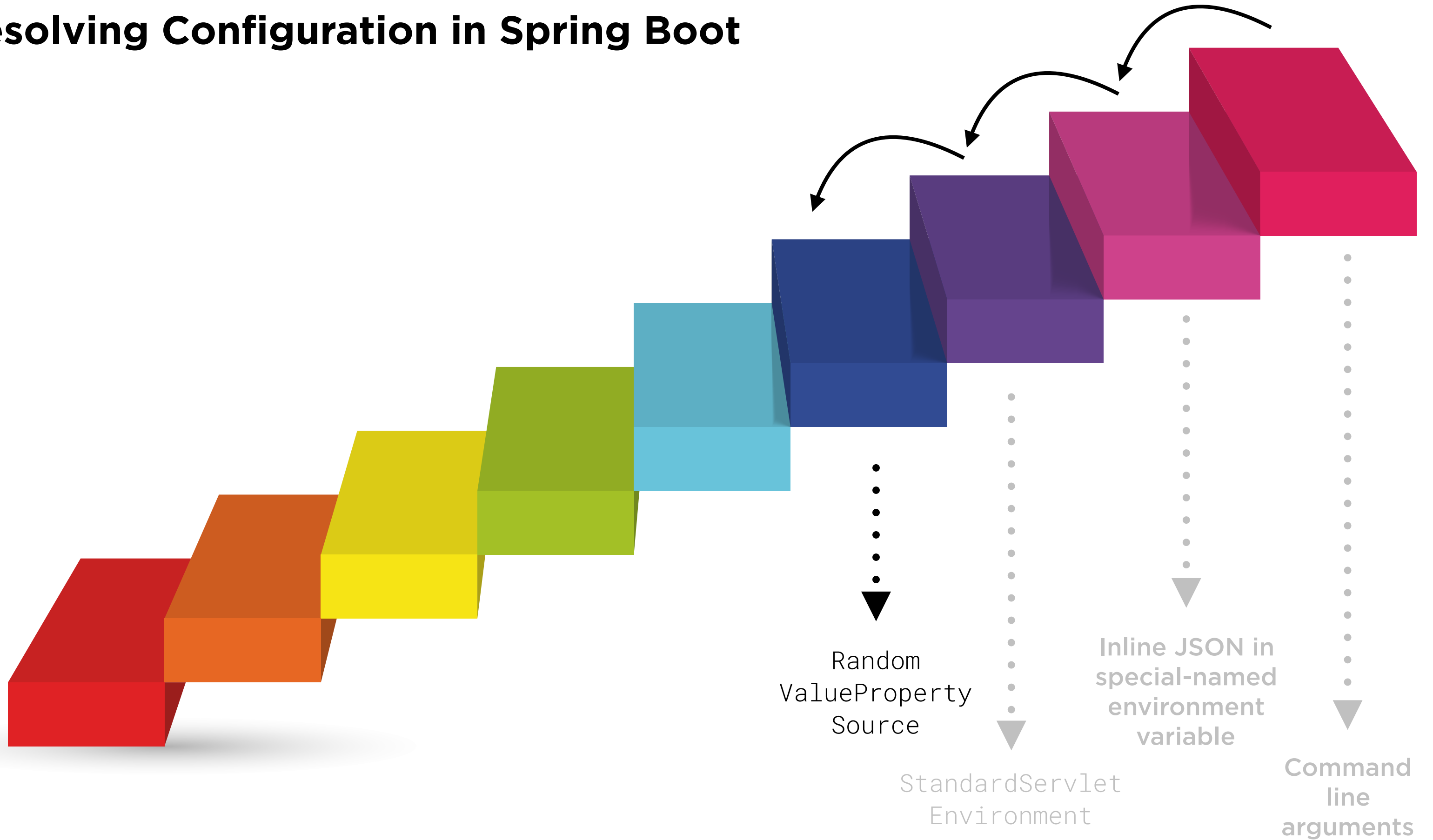
### 3.) StandardServletEnvironment



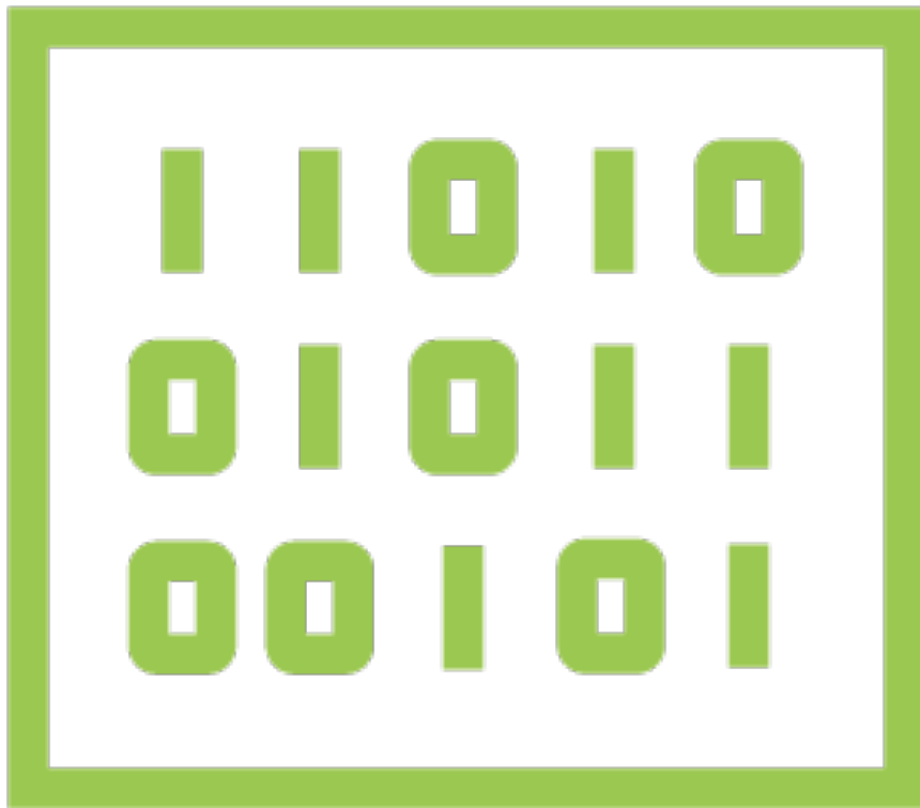
- **A hierarchy within itself**
  - a) ServletConfig init parameters**
  - b) ServletContext init parameters**
  - c) JNDI attributes**
  - d) System.getProperties()**
  - e) OS environment vars**



# Resolving Configuration in Spring Boot



## 4.) RandomValuePropertySource



- **`${random.*}` replacements**

- **`" * "` can be one of**

A. value

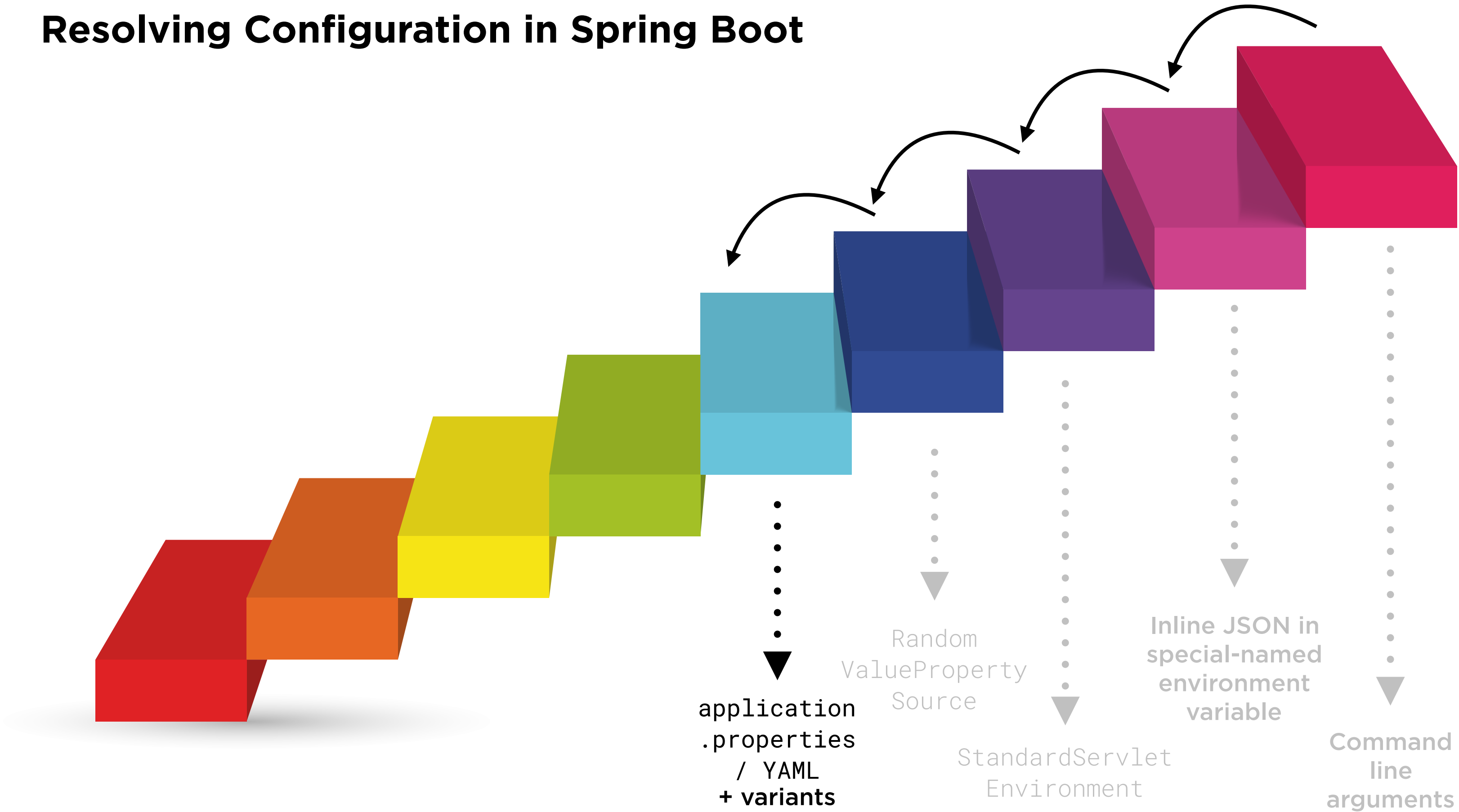
B. int

C. long

D. `int(<number>)`

E. `int[<num1>, <num2>]`

# Resolving Configuration in Spring Boot

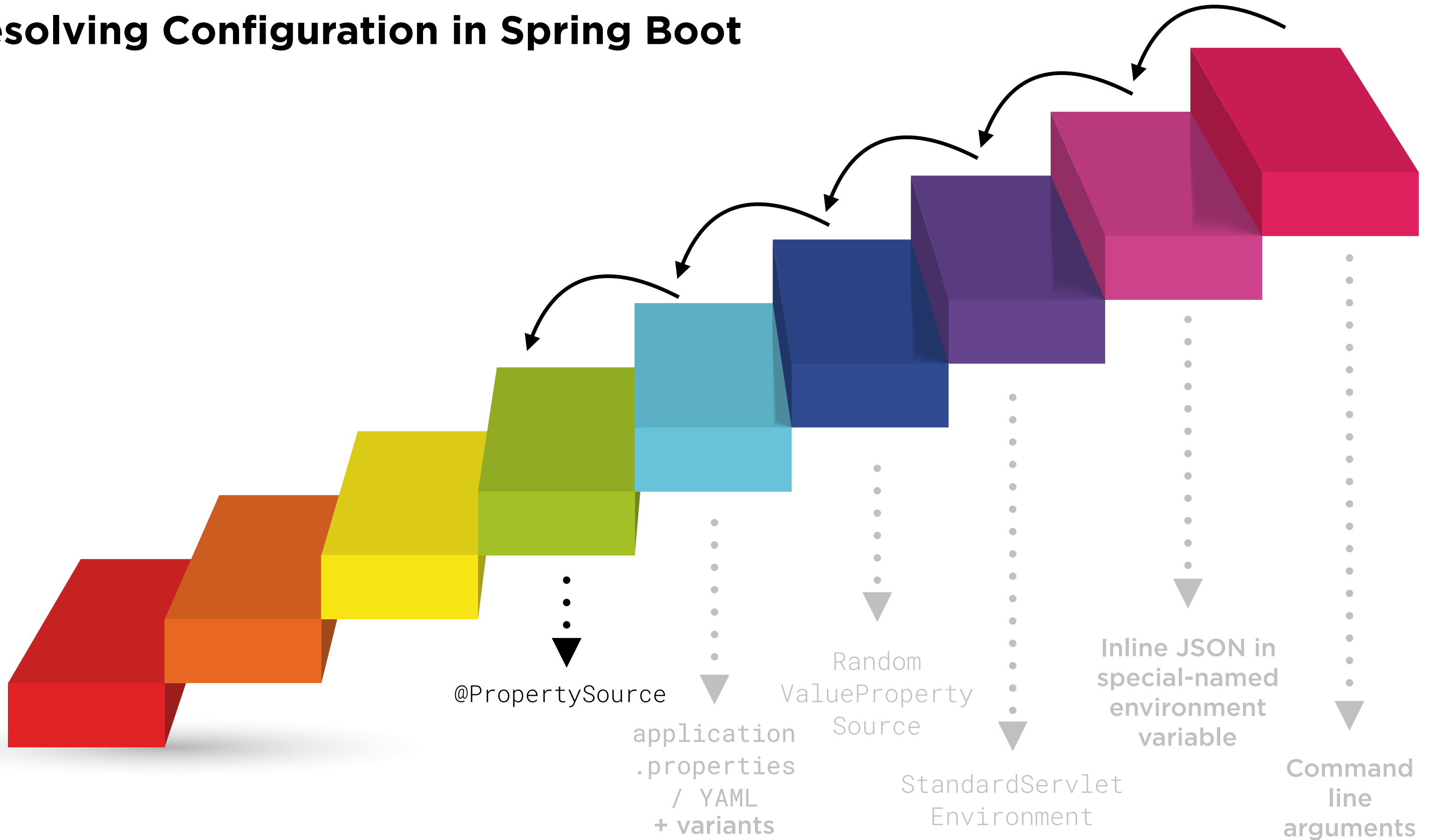


## 5.) `application.properties` / YAML + Variants



- **Look for profile-specific configuration 1st**
  - `application-{profile}.properties`
  - `application-{profile}.yaml`
- **Look for generic configuration 2nd**
  - `application.properties` / `application.yaml`
- **Check these locations**
  - `$CWD/config` **AND** `$CWD`
  - `classpath:/config` **AND** `classpath:`

# Resolving Configuration in Spring Boot

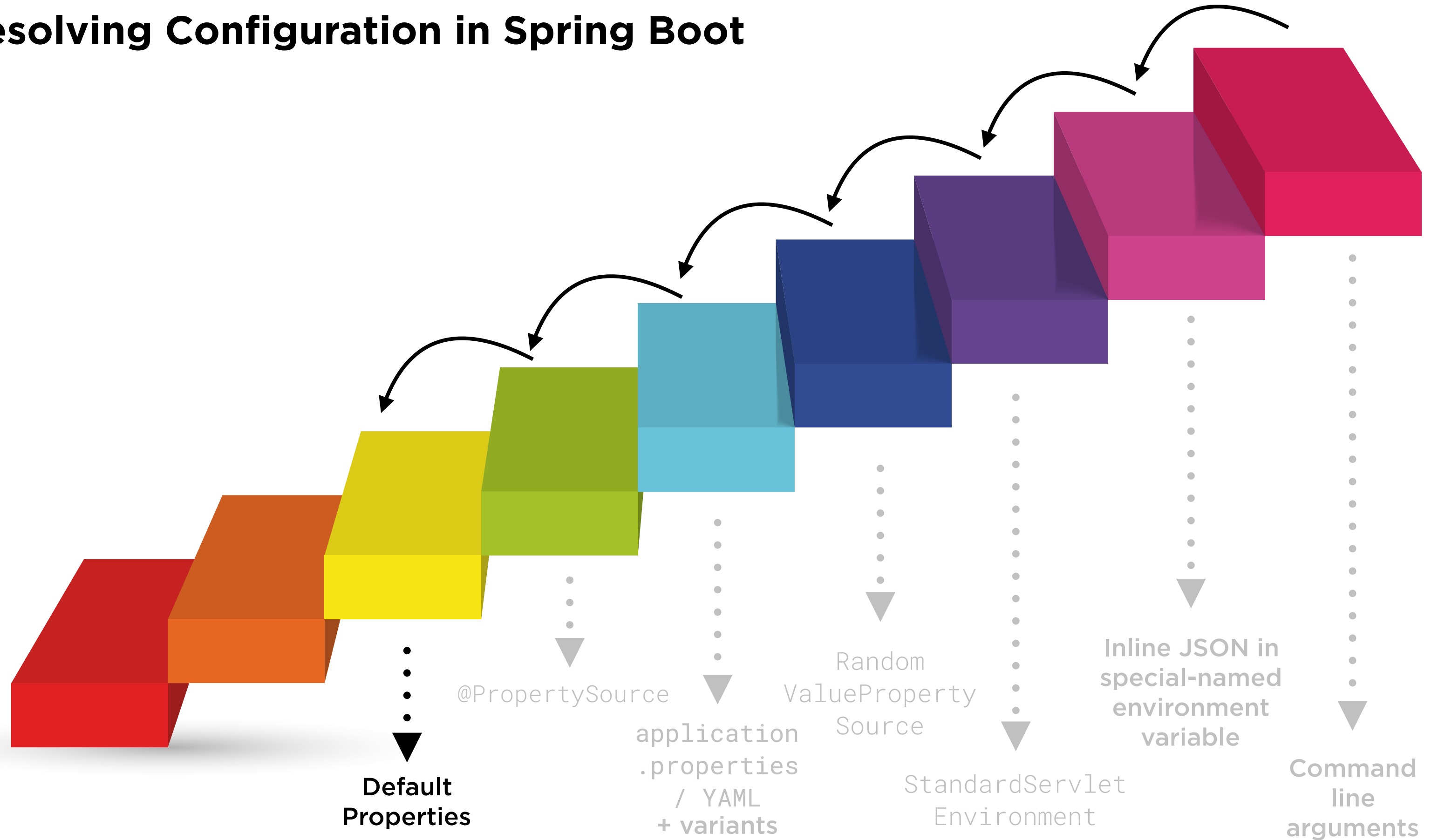


## 6.) @PropertySource



```
1 @SpringBootApplication
2 @PropertySource("/some/path/foo.properties")
3 public class MyApplication {
4     ...
5 }
```

# Resolving Configuration in Spring Boot



## 7.) Default Properties



```
1 @SpringBootApplication
2 public class MyApplication {
3     public static void main(String args[])
4     {
5         SpringApplication.setDefaultProperties(...)
6     }
7 }
```



# Recapping this Module

- **New configuration format: YAML**
- `@ConfigurationProperties`
- **Cascading resolution**