Domain Specific Languages

Gal Lalouche
Oath Inc.

Computer Science
Technion

Spring 2018
Introduction

**Domain Specific Language (DSL)**

A configuration API to an underlying semantic model

Why do we need DSLs?

- The world doesn’t speak “code” (or XML), but its own domain-oriented terminology
- Translating repetitive business logic to code is therefore harder due to the language barrier
- DSLs bridge the gap by using terms closer to the business side
- They are geared towards developer productivity and minimizing boiler-plate and mistakes

A DSL can be either external or internal
Consumption and implementation

We won’t cover consumption (of the semantic model), or implementation (for the most part) in this lecture

- Consumption of the model is up to the clients
  - DSLs are generally about building a model
  - But some DSLs also expose nicer API for consumption as well
    - E.g., verification with Mockito (as opposed to mock object configuration and creation), or object creation with Guice (as opposed to module configuration)

- Implementation is usually simple, boring, and sometimes tedious
  - But on the on the plus side, pretty easy, especially with an IDE or other tools
  - Usually obvious from the method names and invocations
External DSL

A DSL defined outside of a host language

An external DSL uses a custom syntax or an existing serialization format

- **Custom syntax:**
  - Make (project life-cycle)
  - Regular expressions (string matching)
  - CSS (style definitions)
  - \TeX\ (typesetting)

- **Serialization format:**
  - Maven (XML)
  - HTML (XMLish)
  - JSON Schema (JSON)
  - NPM (JSON)
External DSL: Custom syntax

- By defining our own syntax, we have complete control
- Configuration can be more declarative and less verbose
- But we have to write our own parsers...
- And the outside world doesn’t know about our DSL
For example, a valid variable name starts with a letter and may include letters or digits

- Using an external custom-syntax DSL (regex):
  \[A-Za-z][A-Za-z0-9]*\]
- Imagine having to do that in XML:

```xml
<reg-exp>
  <concat>
    <either>
      <range from="A" to="Z" />
      <range from="a" to="z" />
    </either>
    <at-least-one>
      <either>
        <range from="A" to="Z" />
        <range from="a" to="z" />
        <range from="0" to="9" />
      </either>
    </at-least-one>
  </concat>
</reg-exp>
```
Or without a DSL at all

```java
public static boolean matches(String s) {
    if (!Character.isAlphabetic(s.charAt(0)))
        return false;
    for (int i = 1; i < s.length(); i++) {
        char c = s.charAt(i);
        if (!(Character.isAlphabetic(c) || Character.isDigit(c)))
            return false;
    }
    return true;
}
```
If custom DSLs are so great, why would we ever want to use something like XML?

- Don’t have to write our own parsers
  - Even a simple format like CSV can get complicated when you start dealing with real world data, e.g., data with commas, linebreaks
- External tools can help us write our configuration
  - XML/JSON syntax validation, XML namespaces for code completions, semantic validation via schemas
  - Plenty of other tools and viewers around
- No need to learn and teach new syntax
It is common to embed DSLs within one another

- This allows for DSL reuse and increase DSL cohesion
  - We parse until we read some marker
  - Switch to another parser until we read another marker
  - Go back to the original DSL
- For example, embedding a general programming language (GPL) in HTML gives us access to logic operations
  - Usually Javascript, but other variants exist (JSP, Razor, Play’s templates, etc.)
- We might embed a “calculator” DSL to write arithmetic expressions
  - For example, calc() in CSS
Benefits of embedding DSLs:

- **Cohesion.** We can keep our DSLs short and concise
  - And simpler to implement as well

- **Reuse.** We can define our embedded DSLs just once (or use existing ones)

We don’t want our DSLs to be *too* powerful (should be much less powerful than a GPL)
Internal DSLs

**Internal DSL**

A DSL defined within the syntax of a general purpose language

A stylized use of the **host** language for building the semantic model

- No need to write our own parsers
- We can get help from our IDE
  - Both for implementing the DSL and for clients using it
- But we are bound by the host language’s syntax
We’re building an app for a bank, and want to separate incomes into a category

- Below 100000: very poor
- 100000-200000: poor
- 200000-300000: middle income
- 300000-400000: rich
- 400000-500000: very rich
- 500000-600000: super rich
- 600000+: filthy rich
Range without DSL

This is a very simple problem, so why not just code it?

```java
// Can you spot all the bugs?
IncomeLevel getIncomeLevel(int income) {
    if (income < 100000) return IncomeLevel.VERY_POOR;
    else if (income < 20000) return IncomeLevel.POOR;
    else if (income > 30000) return IncomeLevel.MIDDLE;
    else if (income < 50000) return IncomeLevel.SUPER_RICH;
    else return IncomeLevel.FILTHY_RICH;
}
```
Range with DSL

```java
RangeBuilder rb = Range.builder();
rb.add(100000, IncomeLevel.VERY_POOR)
rb.add(20000, IncomeLevel.POOR)
rb.add(30000, IncomeLevel.MIDDLE)
rb.add(50000, IncomeLevel.VERY_RICH)
rb.add(60000, IncomeLevel.SUPER_RICH)
rb.finishWith(IncomeLevel.FILTHY_RICH)
Range range = rb.build();

assert range.size() == IncomeLevel.values().length;
```
Takeaways:

1. We can throw if we get a number that is smaller than the previous one.
2. No <> symbols, so less room for bugs.
3. We can verify that we haven’t forgotten anything at the end.
4. Range can compute in $O(\log n)$ instead of $O(n)$. 
Our application allows users to configure virtual computing environments

- Each environment has many possible parameters
  - Some mandatory (CPU, memory, disk)
  - Some optional (GPU, disk speed, internet speed)
- We may want to reuse environments with little or no modifications
  - So we need some kind of a data type
What do these numbers mean?

Easy to mix-up parameter order

How do we handle optional parameters?
  * Can’t use overloads since all parameters have the same type
  * Defining a single method with dozens of parameters isn’t maintainable in the least
Computer resources without a DSL: maps

Map<String, Integer> map = Map.of(
    "CPU", 4,
    "Memory", 100,
    "Disk", 5,
    "GPU", 6);

- Not safe, easy to make typos, etc.
- Not discoverable
Resources.newBuilder()
  .cpu(4)
  .memory(100)
  .disk(5)
  .gpu(6)
  .build()

- Includes all the pros of the previous solutions
- Discoverable, since all options are methods and are completatable in any IDE
- Uses Fluent API
Varargs alternative

An alternative to the above builder is to define a super `Resource` type, and accepts a `varargs` argument

```java
public interface Resource {}
public class CPU extends Resource {
    public int count();
    public double ghz();
}
// Etc.

public class Resources() {
    public Resources(Resource... resources) { ... }
}

// Usage
new Resources(
    new CPU(4), new Memory(100), new Disk(5), new GPU(6));
```
Varargs considerations

Things to consider in the above solution

- Do we want to enable clients to **extend** resource?
  - If not, make the interface package-private, or an **abstract class** with a package-private constructor

- Should we use **enums**?
  - Only if all resources have the same **arity** (number of parameters)

- We can **enforce required resources** by using a mix of varargs and regular parameters

```java
public class Resources() {
    public Resources(Cpu cpu, Memory memory,
                     Resource... resources) { ... }
}
```
We’ll look at several kinds of fluent APIs

1. Convenience
2. Transformative
3. Alternative APIs
4. Progressive interfaces
Convenience fluent APIs

The simplest fluent APIs are simply those the replace methods returning `void` with methods returning `this`

```java
// FreeBuilder example; both pieces of code are equivalent
Table t = Table.Builder()
    .setRows(10)
    .setCols(5)
    .build();

Table.Builder builder = new Table.builder();
builder.setRows(10);
builder.setCols(5);
Table p = builder.build();
```
Convenience fluent APIs (cont.)

What do we gain by the above API transformation?

- Fewer elements
  - The code is slightly shorter
  - We aren’t creating a useless builder variable
- But more importantly, reveals intent
  - In the first example, we have a single expression for building, signifying the cohesion of all related statements
  - In the latter, we have a bunch of seemingly unrelated statements
  - By using fluent APIs to chain commands we are asserting all commands part of a cohesive whole
Transformative fluent APIs

Methods that modify the element and return a new element (e.g., for immutable objects, but not just)

```java
// Immutable
"Foo"
    .toUpperCase()
    .replaceAll("F", "B")
    .replaceAll("oo", "ar");

// Monadic / Generic
Streams.range(1, n)
    .map(x -> gcd(x, n))
    .filter(x -> x == 1)
    .count();
```
In a way, these kinds of APIs are **fluent by accident** rather than by design

- A side-effect of using methods instead of global functions
- In other languages, e.g., Python, Haskell, the order of method invocation is reversed

```python
len(
    filter(lambda x: x == 1,
            map(lambda x: gcd(x, n),
                 range(1, n)))
)
```

But a fluent API could **mistakenly** give the appearance of an **immutable** class

```java
// Is this a new resource or the same one? No way to tell
// from looking at this code...
Resource r2 = resource.updateCpu(10);

stream.forEach(consumer1);
// This throws, but nothing in the signature indicates it
stream.forEach(consumer2);
```
Fluent APIs as a design choice

Of course, the most interesting type of fluent API (for this lecture anyway), is fluent APIs as an alternative to method parameters

```java
// Spark
dataFrame
    .write() // Returns a DataFrameWriter
    .saveMode(SaveMode.Overwrite) // Convenience fluent
    .option("header", "true")
    .csv("filepath")

// Theoretical API which is even more fluent and declarative
dataFrame
    .write()
    .overwrite()
    .withHeader()
    .csv("filepath")
```
Fluent APIs as a design choice (cont.)

What do we gain?

- Discoverability
- Readability
- Safety

How would this look like without a fluent API?

dataFrame
  .write(SaveMode.Overwrite, Format.CSV, "filepath",
   true,  // What's true?
   /* a bunch of other variables since there are no
      * defaults in Java; alternative is combinatorial
      * explosion */);}
A special (and rare) kind of fluent API is a progressive interface

```java
// Non-progressive API
Table.builder()
    .setRows(10)
    .setRows(10)
    // Oh-oh, you forgot to set columns!
    .build()
```

In a progressive API, the above code will fail at compile time

- This is done by progressing to a different interface after each setter
- A lot of work and details, but completely compile-time safe
```java
public class Table {
    private final int columns;
    private final int rows;
    public Table(int rows, int columns) {
        this.rows = rows;
        this.columns = columns;
    }
    public static RowsBuilder builder() {
        return new RowsBuilder();
    }
}

public class RowsBuilder {
    // package-private constructor
    RowsBuilder() {}
    public ColsBuilder setRows(int rows) {
        return new ColsBuilder(rows);
    }
}
```
Progressive APIs implementation (cont.)

```java
public class ColsBuilder {
    private final int rows;
    // package-private constructor
    ColsBuilder(int rows) {
        this.rows = rows;
    }

    public Table buildWithCols(int cols) {
        return new Table(rows, cols);
    }
}

Usage:

// Anything else won't compile!
Table t = Table.builder().setRows(10).buildWithCols(5);
```
Building a GUI menu

Of course, DSLs shine when building more complicated models (picture taken from https://docs.oracle.com/javase/tutorial/uiswing/components/menu.html)
public void createMenu(JFrame frame) {
    JMenuBar mb = new JMenuBar();
    frame.setJMenuBar(mb);

    JMenu file = new JMenu("File");
    file.setMnemonic('F');
    mb.add(file);

    JMenuItem open = new JMenuItem("Open");
    open.setMnemonic('O');
    open.addActionListener(new OpenCommand());
    file.add(open);

    JMenuItem save = new JMenuItem("Save");
    save.setMnemonic('S');
    save.addActionListener(new SaveCommand());
    file.add(save);
}

Code is very repetitive

Low signal to noise ration

Not discoverable

In other words, the usual problems of trying to build stuff without a DSL!
public void createMenu(JFrame frame) {
    MenuBuilder builder = new MenuBuilder();
    builder.child("File")
        .child("Open")
        .shortcut('O')
        .action(new OpenCommand())
        .up()
    .child("Save")
        .shortcut('S')
        .action(new SaveCommand())
        .up()
    .up()
    .child("Edit")
    .child("Copy")
    .shortcut('C')
    .action(new CopyCommand())
    .up()
    .up();
    frame.setJMenuBar(builder.build());
}
The above solution works, but we cheated a little bit...
The above solution works, but we cheated a little bit...

- The code has to be **manually indented!**
- Any IDE will flatten it
  - Often without us intending it or noticing it;
- Many linters won’t accept it
- Harder to know relations between nested commands
- All-in-all, a nightmare to maintain

This might seem like a minor quibble, but part of the power DSL is being
caller to use
public void createMenu(JFrame frame) {
    MenuBuilder builder = new MenuBuilder();
    builder.child("File")
        .child("Open")
        .shortcut('O')
        .action(new OpenCommand())
        .up()
    .child("Save")
        .shortcut('S')
        .action(new SaveCommand())
        .up()
    .up()
    .child("Edit")
    .child("Copy")
        .shortcut('C')
        .action(new CopyCommand())
        .up()
    .up();
    frame.setJMenuBar(builder.build());
}
import MenuBuilderFunction.*;

public void createMenu(JFrame frame) {
    MenuBuilder builder = new MenuBuilder(
        child("Open",
            shortcut('O'),
            action(new OpenCommand()))
        ,
        child("Save",
            shortcut('S'),
            action(new SaveCommand()))
        ,
        child("Edit",
            child("Copy",
                action(new CopyCommand)
            )
        )
    );
    frame.setJMenuBar(builder.build());
}
Nested functions

- child function implemented using varargs

```java
Child child(String name, Widget... widgets)
class Shortcut extends Widget { ... }
class Action extends Widget { ... }
class Child extends Widget { ... }
```

- IDE can take of formatting for us
- Easy to see nested relationships
- No need to call up()
- Fluent APIs aren’t always the right choice
- But we do lose a bit on discoverability
- Can use new with types instead of global functions (style preference)
Final example: state machine

We’ll implement a simple state machine for a 2D fighting game AI

- If the AI’s health is higher than the player’s, get close and punch
- If the AI’s health is lower, get back and shoot

We’ll look at different kinds of implementations styles

1. External custom DSL
2. External serialization based DSL
3. Non-DSL code
4. Internal DSL
State machine: custom external

```
start advance:
    action: move_forward
    close_to_player -> punch_player
    far_from_player -> advance

punch_player:
    action: punch
    health_lower -> retreat
    next_to_player -> punch
    far_from_player -> advance

retreat:
    action: move_backward
    close_to_player -> retreat
    far_from_player -> shoot_player

shoot_player:
    action: shoot
    health_higher -> advance
    close_to_player -> retreat
    far_from_player -> shoot_player
```
<state-machine>
  <state start="true" name="advance">
    <action>move-forward</action>
    <event name="close-to-player" goto="punch-player" />
    <event name="far-from-player" goto="advance" />
  </state>
  <state name="punch-player">
    <action>punch</action>
    <event name="health-lower" goto="retreat" />
    <event name="close-to-player" goto="punch-player" />
    <event name="far-from-player" goto="shoot-player" />
  </state>
  <state name="run-away">
    <action>move-backward</action>
    <event name="close-to-player" goto="run-away" />
    <event name="far-from-player" goto="recover-health" />
  </state>
</state-machine>
External DSL comparison

- Custom DSL
  - Good signal to noise ratio
  - Simple enough to parse (for now)
  - Easiest to use by non-programmers

- XML
  - Noisier and more verbose
  - But we can use XML schema for documentation + validation + auto completion
  - Easier to expand upon
  - Still not too hard to use by non-programmers
Map<String, String> actions = new HashMap<>();
Map<Pair<String, String>, String> transitions =
    new HashMap<>();

actions.put("advance", "move-forward");
actions.put("punch", "punch");
actions.put("retreat", "move-backward");
actions.put("shoot", "shoot");

transitions.put(new Pair<>("advance", "far-away"), "advance");
transitions.put(new Pair<>("advance", "close"), "punch");
// Etc.
Event playerFar = new Event("player far");
Event playerClose = new Event("player close");
Event lowerHealth = new Event("lower health");
State advance = new State("advance");
State punch = new State("attack player");
State run = new State("run away");

advance.addAction(new MoveForward());
advance.addTransition(playerFar, advance);
advance.addTransition(playerClose, punch);

// Etc.

StateMachine sm = new StateMachine();
sm.addState(advance);
sm.addState(punch);
sm.addState(run);
StateMachineBuilder builder = new StateMachineBuilder();
State advance = builder.addState("advance");
State punch = builder.addState("attack player");
State run = builder.addState("run away");
Event playerFar = builder.addEvent("player far");
Event playerClose = builder.addEvent("player close");
Event lowerHealth = builder.addEvent("lower health");

advance.action(new MoveForward())
    .on(playerFar).loop()
    .on(playerClose).goTo(punch);

punch.action(new PunchPlayer())
    .on(playerFar).goTo(advance);
    .on(playerClose).loop()
    .on(lowerHealth).goTo(run);

// Etc.
Internal code comparisons

- Non-domain code
  - Simplest to implement
  - But hardest to use: no safety, no validation, no discovery

- Command-oriented API
  - Better type-safety, but still possible to make mistakes (e.g., forgetting to add a state)
  - Validation is done at the end

- DSL
  - Easiest (and safest) to use, hardest to implement
Comparison

Summary
- External APIs
  - Give you more control over the language
  - But need to be parsed
  - Can be used by non-programmers
  - Can be consumed by multiple programming languages
  - Not as safe
- Internal APIs
  - Leverage the hosting language
    - We are limited by the host language (In some languages more-so than others)
    - But we don’t have to write a parser
  - Work great with an IDE
  - Can guide the client
- Both
  - Create a semantic model, but don’t **consume** it
  - Are always more up-front work for creating the DSL
  - But are always nicer to use by its clients (which may be the same people)
Language and terminology

DSLs often use more prose-like method names

- Code reads more like English than a set of instructions to a computer
- Lots of prepositions and postpositions
  - with, of, for, etc.
  - bind(x).to(y).in(z) instead of bind(x, y, z)
  - when, thenReturn, thenThrow
- Use of multiple grammatic tenses, gerunds, etc.
  - sort, sorted, sortingWith
- Extension methods are often used to this effect
  - 5.seconds is nicer than new Seconds(5)
For example, consider how **ScalaTest** uses DSL to construct tests

```scala
class ExampleSpec extends FlatSpec with Matchers {
  "A Stack" should "pop values in last-in-first-out order" in {
    val stack = new Stack[Int]
    stack.push(1)
    stack.push(2)
    stack.pop() should be (2)
    stack.pop() should be (1)
  }

  it should "throw if an empty stack is popped" in {
    val emptyStack = new Stack[Int]
    a [NoSuchElementException] should be thrownBy {
      emptyStack.pop()
    }
  }
}
```
Hybrid DSLs

Some DSLs reuse (purely or not) the GPL syntax to lower the learning curve

- Rake (Ruby’s make) reuses Ruby’s syntax
- SBT (Scala’s Maven) uses mostly Scala syntax
- Gradle (an alternative to Maven) reuses Groovy’s syntax

These DSLs aren’t a GPL, but it’s often hard to tell the difference
Auto-generating code

Programmatic code-generation is a common technique for creating DSLs

- i.e., meta-programming, creating code programmatically
- Useful for generating static types and classes from dynamic data
  - Not all that common in dynamic languages, since classes and data are interchangeable
  - But there are still some examples, e.g., protobuf parsers
- Or for creating repetitive boiler-platey code
  - Again, in dynamic languages we wouldn’t have this problem
  - But we wouldn’t enjoy any of its benefits
Ways to auto-generate code

There are three main sources code generation

1. Plain data
2. Schema
3. Annotations
We can generate code by consuming plain serialized data, gaining the usual DSL benefits

- Safety
- Discoverability

Code generation is usually done with a DSL
We’re building an API for course browsing

- We could just create a dynamic data structure to hold all the courses...
- But we want to do in a type-safe manner
- Every course will be a singleton constant (enum)
Data generation strategy

Our general strategy for generating code will usually be the same:

1. Parse the data
2. Generate a runtime/dynamic representation of the data
3. Iterate over the structured data to generate your class and method code
We will use the following classes:

```java
interface Course {
    String getNumber();
    String getName();
    List<String> getStaff();
}

// Parsed from the data and used to fill auto-generated code
class ParsedCourse implements Course {
    ...
}
```
And the following class will be generated:

```java
class Course {
    private final String name;
    private final String number;
    private final List<String> staff;

    private Course(String number, String name, List<String> staff) {
        this.number = number;
        this.name = name;
        this.staff = staff;
    }

    @Override public String getName() { return name; }
    @Override public String getNumber() { return number; }
    @Override public List<String> getStaff() { return staff; }
}
```
Generating the skeleton

First we generate the skeleton: fields, constructor, and methods

```java
public static TypeSpec.Builder generateSkeleton() {
    return TypeSpec.enumBuilder("course.CourseImpl")
        .addSuperinterface(TypeName.get(Course.class))
        .addField(String.class, "number", Modifier.PRIVATE, Modifier.FINAL)
        .addField(String.class, "name", Modifier.PRIVATE, Modifier.FINAL)
        .addField(ParameterizedTypeName.get(List.class, String.class), "staff", Modifier.PRIVATE, Modifier.FINAL)
        .addMethod(MethodSpec.constructorBuilder()
            .addModifiers(Modifier.PRIVATE)
            .addParameter(String.class, "number")
            .addParameter(String.class, "name")
            .addParameter(
                ParameterizedTypeName.get(List.class, String.class), "staff")
            .addStatement("this.number = number")
            .addStatement("this.name = name")
            .addStatement("this.staff = staff")
            .build());
}
```

(Continued on next slide)
We finish by generating the methods

```java
.addMethod(MethodSpec.methodBuilder("getNumber")
  .addAnnotation(Override.class)
  .addModifiers(Modifier.PUBLIC)
  .returns(String.class)
  .addStatement("return number")
  .build())
.addMethod(MethodSpec.methodBuilder("getName")
  .addAnnotation(Override.class)
  .addModifiers(Modifier.PUBLIC)
  .returns(String.class)
  .addStatement("return name")
  .build())
.addMethod(MethodSpec.methodBuilder("getStaff")
  .addAnnotation(Override.class)
  .addModifiers(Modifier.PUBLIC)
  .returns(ParameterizedTypeName.get(List.class, String.class))
  .addStatement("return staff")
  .build())
```
Skeleton takeaways

- Uses multiple techniques we learnt about
  - Static factory methods
  - Builders
  - Fluent APIs
  - Abstraction-level terminology (constructors, methods, fields, etc.)
- Saves us from a lot of the boilerplate
  - Braces, parenthesis, semicolons, indentation, etc.
  - Automatically handles imports (!)
- Pretty declarative, but could be even more by supporting common patterns
  - Not allowing a super class for enums
    - Although it (and many other properties) are checked at run time
    - But not every thing is checked
  - Constructors setting fields
  - Automatic getters/setters
  - .public() instead of Modifiers.PUBLIC, overrides() instead of addAnnotation(Override.class), etc.
// Create a List.of("x", "y") literal
private static String toListOfLiteral(List<String> list) {
    return "List.of" + list
    .stream()
    .map(s -> "" + s + "\""")
    .collect(Collectors.joining("", "", "(", ")"));
}

public static JavaFile buildAll(List<DynamicCourse> courses) {
    TypeSpec.Builder skeleton = generateSkeleton();
    for (Course c : courses)
        skeleton.addEnumConstant("NO_" + c.getNumber(),
            TypeSpec.anonymousClassBuilder(
                "$S, $S, " + toListOfLiteral(c.getStaff()),
                c.getNumber(), c.getName()).build());

    return JavaFile.builder("il.ac.technion.cs",
        skeleton.build()).build();
}
We want to enable type safe grading

- Can only grade students registered to course (hundreds of students)
- Can only grade test or assignments (variant number of assignments from course to course)
- Grade has to be 1 to 100

```java
Students.GAL.gradeAssignment1(Grade.GRADE_100)
gradeAssignment2(Grade.GRADE_90)
gradeAssignment3(Grade.GRADE_95)
gradeTest(Grade.GRADE_55);
```

Almost trivial with code generation, but very hard without
We can do more than just generate data or repetitive code

- We can also structure our classes to model complex relations
- For example, type-safe graph route definitions
- Each vertex will be a `singleton` class
- Each edge will be a method from one class to another
- We can generate other utility methods, such as `getAllNeighbors()`, etc.
We want A.INSTANCE.B().D().G() to compile, but not A.INSTANCE.C().G()
Graph generation: template

We’ll use the following interface for all nodes

```java
public interface Node {
    String name();
    List<String> neighbourNames();
    default int outDegree() { return neighbourNames().size(); }
    int inDegree();
}
```

And the following template for a class:

```java
enum A implements Node {
    INSTANCE;

    Node b() { return B.INSTANCE; }
    Node c() { return C.INSTANCE; }

    @Override public int inDegree() { return 0; }
    @Override List<Node> neighbours() {
        return List.of("B", "C");
    }
}
```
TypeSpec.Builder builder = TypeSpec.enumBuilder(node.name()).
    .addSuperinterface(Node.class)
    .addEnumConstant("INSTANCE")
    .addMethod(MethodSpec.methodBuilder("name")
        .addAnnotation(Override.class).addModifiers(Modifier.PUBLIC)
        .returns(String.class)
        .addStatement("return $S", node.name())
        .build())
    .addMethod(MethodSpec.methodBuilder("inDegree")
        .addAnnotation(Override.class).addModifiers(Modifier.PUBLIC)
        .returns(TypeName.INT)
        .addStatement("return $L", node.inDegree())
        .build())
    .addMethod(MethodSpec.methodBuilder("neighbors")
        .addAnnotation(Override.class).addModifiers(Modifier.PUBLIC)
        .returns(ParameterizedTypeName.get(List.class, Node.class))
        .addStatement("return " + node.neighbourNames().
            .stream()
            .map(s -> "\"" + s + "\""
            .collect(Collectors.joining("","", "List.of("", ")")))
        .build());

(Continued on next slide)
for (String name : node.neighbourNames())
    builder
        .addMethod(MethodSpec.methodBuilder(name)
            .addModifiers(Modifier.PUBLIC)
            .returns(Node.class)
            .addStatement("return " + name + ".INSTANCE")
        .build());

return JavaFile.builder("graph.nodes", builder.build())
    .build();
Pros of code generation

- Can generate many more repetitive code than is manually possible
- We can replace dynamic, run-time checked code with compile time checked code
- We don’t have to maintain the generated code
  - Simply rerun the script and regenerate all classes
Code generation costs

Of course, like all good things in life and software, code generation isn’t free

- Metaprogramming is orders of magnitude harder than regular programming
  - Every change we want to make has to go through an extra layer of abstraction
  - Greatly complicates the process of building and compiling your application with additional, harder to debug, steps
- Auto-generated code is usually less readable than manually written code
  - And the code generator is much less readable

Remember: It’s a powerful hammer, but not everything is a nail
Another common use of code-generation is generate class definition from schema

- From example, generating class, field, and methods from a SQL table
  - A reverse-alternative to Object-relation mapping (ORM)
- Speeding up serialization
  - No need for reflection, parsers and formatters are generated at compile time
- Allowing multiple languages to consume the same types and services using language-idiomatic code
  - As in protobuffs and gRPCs, which you will learn about in the tutorial
Annotation processing

(Not to be confused with runtime consumption of annotations using reflection, e.g., JUnit, Guice)
Extending classes by parsing the abstract syntax tree as a pre-compilation stage

- Classic example: `@Override` is checked at compile time to ensure the method actually overrides and not just overloads
- Allows mixing manual and auto-generated code
  - For example, `FreeBuilder` and `AutoValue` generate data classes from interface definitions
  - `Dagger` is a compile-time-safe alternative to Guice
- While it can be used to generate any code we want, it’s a bit more complicated to use than plain-old code generation à la JavaPoet
  - But it gives us the ability to customize our code based on manual code
  - Custom processor support is more ingrained in most toolings, e.g., your IDE

For a complete example, see [ANNOTATION PROCESSING 101](#)
Schemas demo