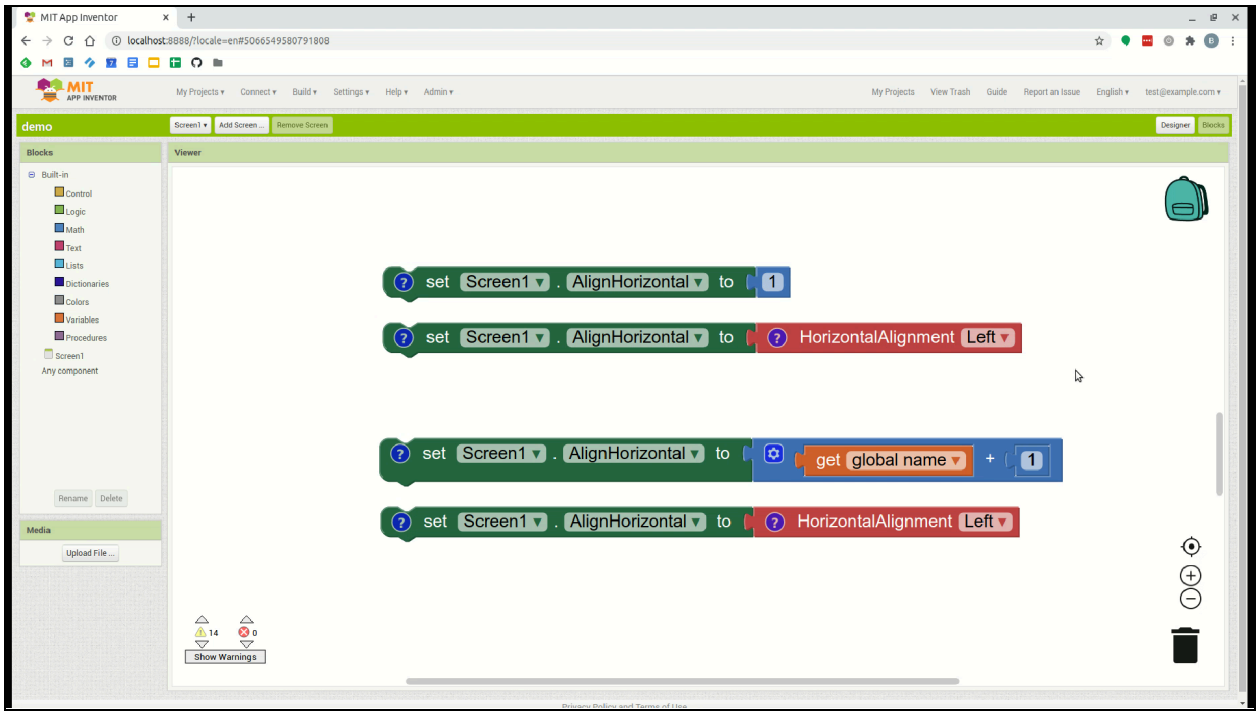
Operation Options: A Story of Language Design

Details some challenges of language design through the lens of Beka Westberg's 2020 GSoC project.

Video

A short video describing problems of language design & backwards compatibility solely through the lens of the blocks-based editor.



Languages in App Inventor

App Inventor is a system of components. These components interact with three different languages:

- Java.
- Yail, a Scheme variant.
- A blocks-based language.

Java is used for two things: (1) to declare the components' functionality, (2) to declare the blocks in the blocks editor. To have blocks associated with it a component must include special

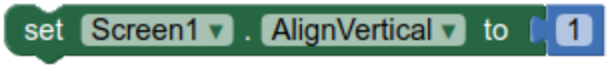
annotations like `@SimpleProperty`, `@SimpleFunction`, and `@SimpleEvent` in its Java definition. These annotations are read by an AnnotationProcessor which creates a special JSON file to tell the blocks editor which blocks to make.

The App Inventor user programs their app using these blocks. Once they are done, the blocks are used to generate Yail code, which is what is actually run on the user's phone. The Yail code calls functions in the Java component definitions, to make the components do things.

Operation Options

There were two goals to the Operation Options project.

1. **Add dropdown blocks to the App Inventor blocks editor.**
   These blocks would replace existing constants inside App Inventor, which usually represented states such as VerticalAlignment(1:top, 2:center, 3:bottom).

   Dropdown blocks are advantageous because they are internationalizable, they reduce invalid value errors, and they make working with the blocks editor more efficient. They also add abstraction to the blocks editor by hiding the concrete values (eg 1, 2, 3) behind abstract names (eg top, center, bottom).

2. **Allow enums to be used as parameters to Java functions.**
   These enums would then get translated into dropdown blocks by the AnnotationProcessor. Using an enum to define the dropdown block is advantageous because it increases the type safety of the component definitions. They also add abstraction to the Java code by hiding the concrete values (eg 1, 2, 3) behind abstract names (eg top, center, bottom).

   For example a concrete function definition like this:

```
@SimpleProperty
public void AlignVertical(int align) {
  if (align > 0 && align <= 3) {
    this.alignVertical = align;
  } else {
    this.dispatchErrorOccurred();
  }
}
```

Turns into a type-safe function definition like this:

```
@SimpleProperty
public void AlignVertical(VerticalAlignment align) {
  this.alignVertical = align;
}
```

Ideal Solution

If we were building App Inventor from scratch we could create an ideal world in which these enum values are very abstract and type safe. The following is a description of this ideal world.

- Dropdown blocks would never act like concrete values, only enum values.
- Functions which expect enum values would only accept enum values of the type they expect.
- Comparisons containing enum values would only ever possibly be true if the values are of the same type.
- Dictionary lookups would only ever possibly return values if there was a key in the dictionary matching the type of the lookup.
- Where applicable, generated YAIL code would only contain enum values.
- Where applicable, Java would only work with enum values.

Backwards Compatibility

The problem we ran into was that we were not building App Inventor from scratch. We were extending an existing set of languages to support enum values.

This meant that we had to make sure that our language extensions were **backwards-compatible** with programs people had written in the existing versions of our languages. This was especially important because App Inventor has many novice users, and it is important that they don't have to do any confusing upgrades of their code.

The following is a description of the specific backwards compatibility problems that we had to design around.

Java

App Inventor allows outside-developers to create things called "extensions". Extensions are just like built-in components, except that they must be installed separately. App Inventor allows extension developers to create extensions that subclass built-in components, or otherwise leverage their functionality. This put severe constraints on Operation Options because it meant

that if we wanted to be backwards compatible with extensions, **we could not change any public or protected Java function headers**.

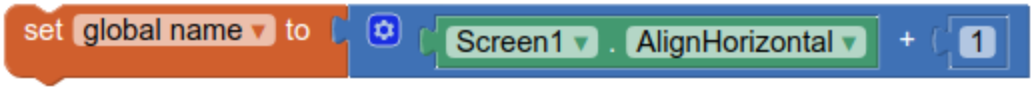
This meant that our ideal version of Java could not exist. We would have to allow Java to work on concrete values into perpetuity, just in case those values were being passed from extensions.

Blocks

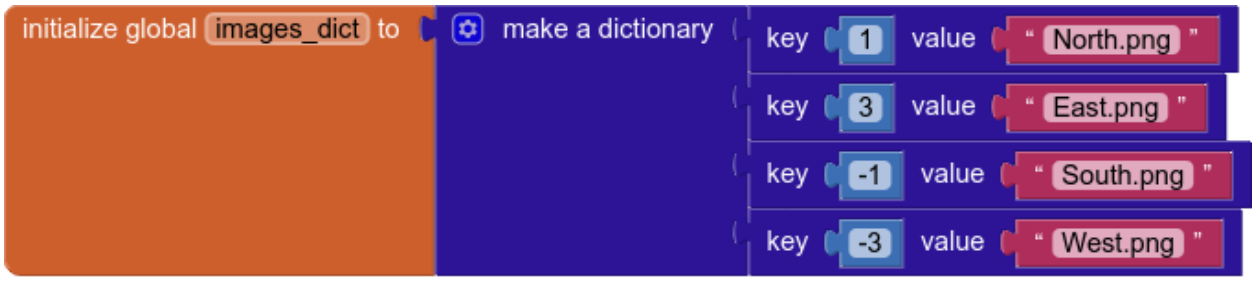
The blocks editor also caused us problems. As stated earlier these enum values were meant to replace concrete constants which already existed within the blocks editor, like this for example:

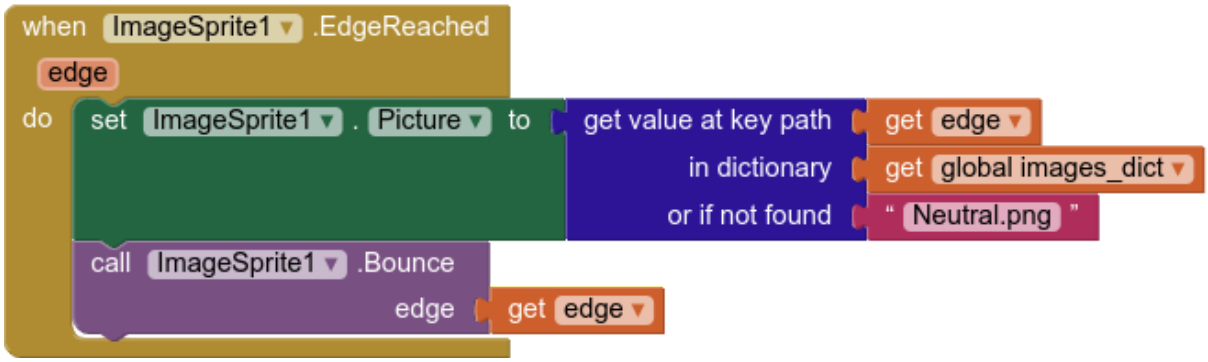
Most of these instances could be upgraded to use dropdown blocks automatically (above) but some (below) could not.

A setter acting concretely.



A getter acting concretely.





An event parameter acting concretely.

This meant that our ideal block language could not exist. Blocks would have to be allowed to act as if they were concrete under certain circumstances.

Yail

There was also a problem with backwards compatibility wrt the Yail language, but that will be discussed later. When the below systems were being designed, only the above issues were known.

Possible Designs

After recognising the constraints of backwards compatibility we designed several systems that compromised on aspects of our ideal system while still being backwards compatible.

Concrete System

The Concrete System was backwards compatible, but gave us none of the benefits of our ideal system.

In this system **all values** which could be enums **were instead concrete**. This included getters, setters, methods, event parameters, and dropdown blocks. For example the below dropdown block would generate the concrete Yail code 1, and it could be used as a concrete value.

The advantages of this system were:

1) No Java code needed to be modified, which meant less work for us as language developers.

The disadvantages of this system were:

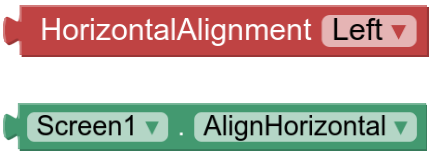
1) We had no type safety at any of the language levels, which means it did not fulfill goal 2 of Operation Options.

An improvement to this system came from the observation that dropdown blocks were new additions, which meant that they did not need to be backwards compatible. As such they could be more abstract and type safe than the rest of the system.

Enums are Abstract System

The Enums are Abstract System was again backwards compatible, but it gave us some of the benefits of our ideal system via the new dropdown blocks.

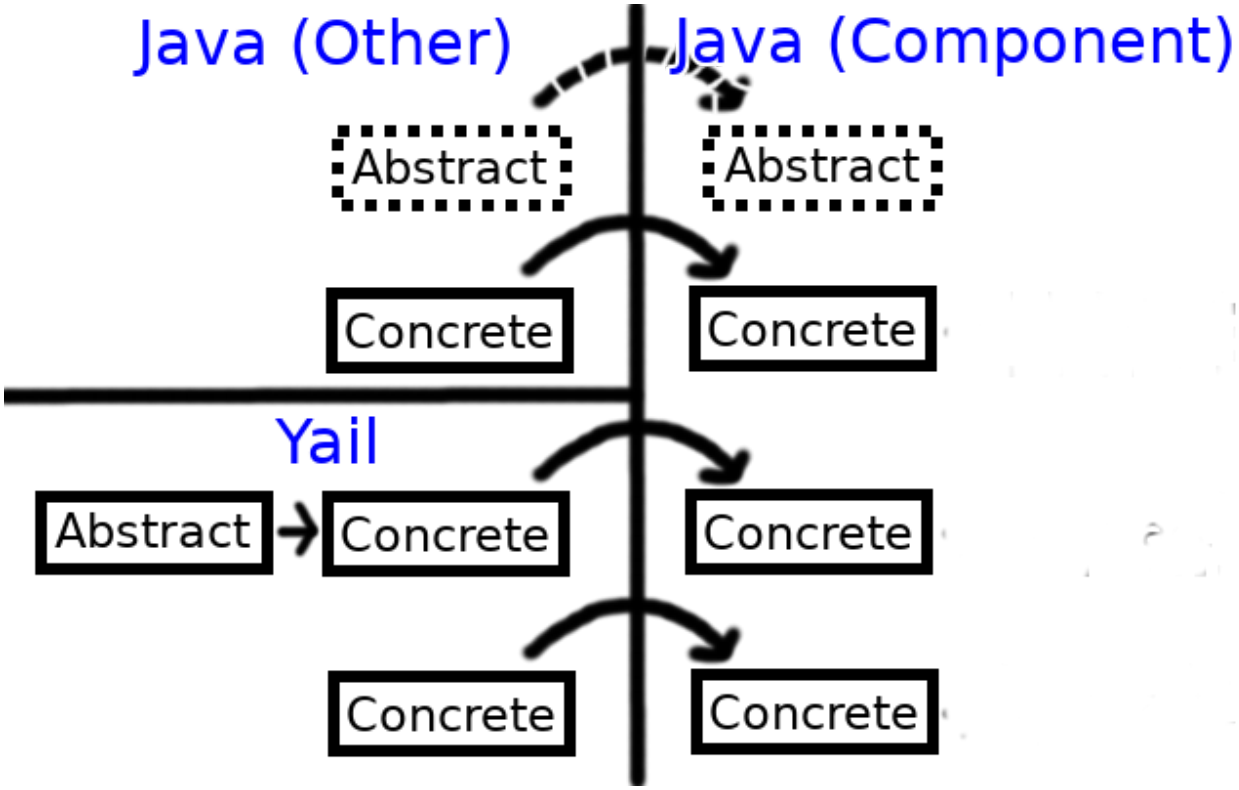
In this system the new **dropdown blocks returned enum values**, while everything else continued to be concrete. This included getters, setters, methods, and event parameters.



In this system, dropdown blocks could not be used as concrete values, they were only allowed to connect to blocks that accepted their specific type of enum.

This lead to the question of how we should deal with a situation like the above, where an enum value is passed to a setter which expects a concrete value.

The solution to this came from the fact that a setter block represents a Java function, which is called by Yail. Yail knows certain information about the Java function, such as which types it expects. This allowed us to **coerce** the enum value returned by the dropdown block to a concrete value before calling the Java function.



The advantages of this system were:

1) No Java code needed to be modified.
2) Dropdown blocks conformed to the ideal system.

The disadvantages of this system were:

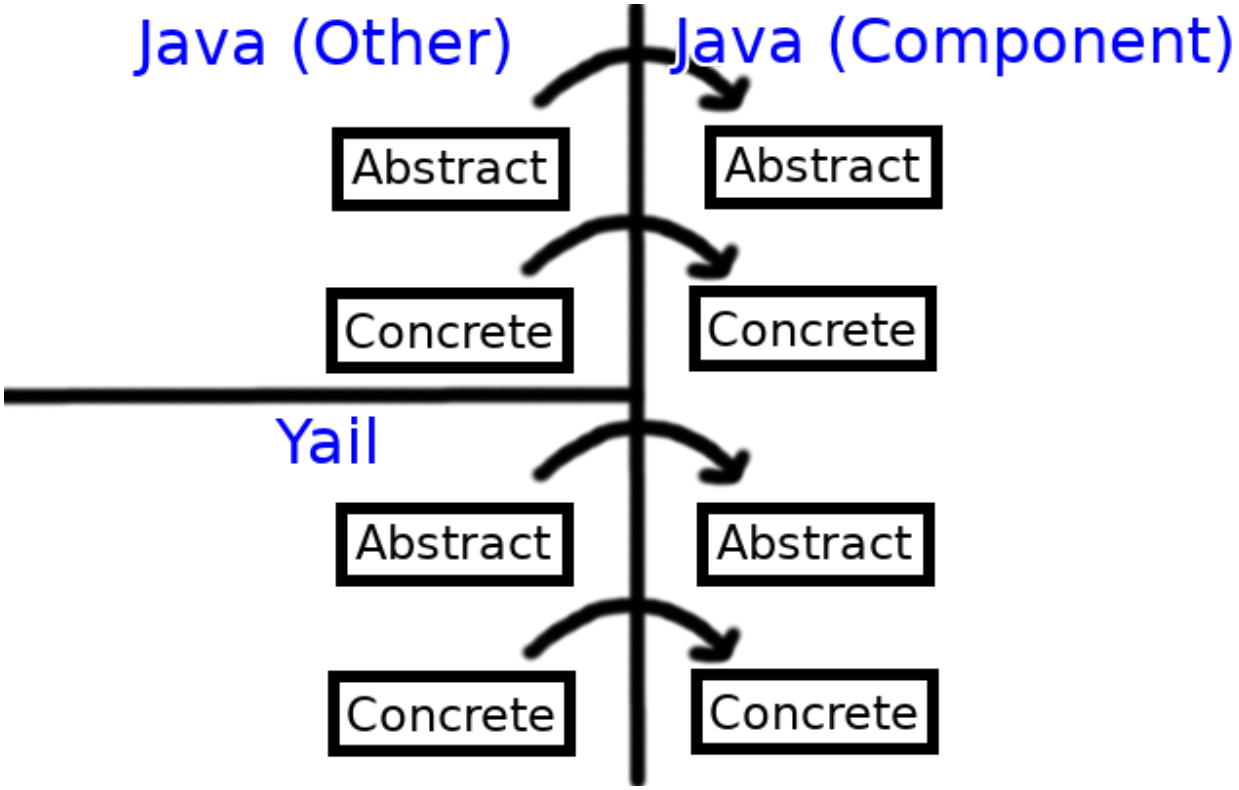
1) Java was always receiving concrete values from Yail.

There were two systems we designed that fixed the above disadvantage, both of which involved adding overloads to the Java code.

Two Systems System

The Two Systems System was again backwards compatible, and made no changes to the blocks language presented in the previous system. But this system added a concept of function overloads so that communication from Yail to Java could be in terms of abstract values.

In this system if an enum value was passed to a method or setter block that **enum value was passed directly into Java, instead of being coerced** to a concrete value. Concrete values were likewise passed directly to Java without being coerced. Getters, method return values, and event parameters (ie values returned from Java) continued to be concrete.



This required method overloads within Java, meaning we needed functions with a single name that could take in either a concrete value or an enum value. As stated earlier Yail has information about the types a Java function expects, but currently it has no concept of overloading. If we wanted to support this Two Systems System Yail would need to be heavily modified to have an understanding of overloads.

The advantages of this system were:

1) Dropdown blocks still conformed to the ideal system
2) Abstract values are passed to Java, if available.

The disadvantages of this system were:

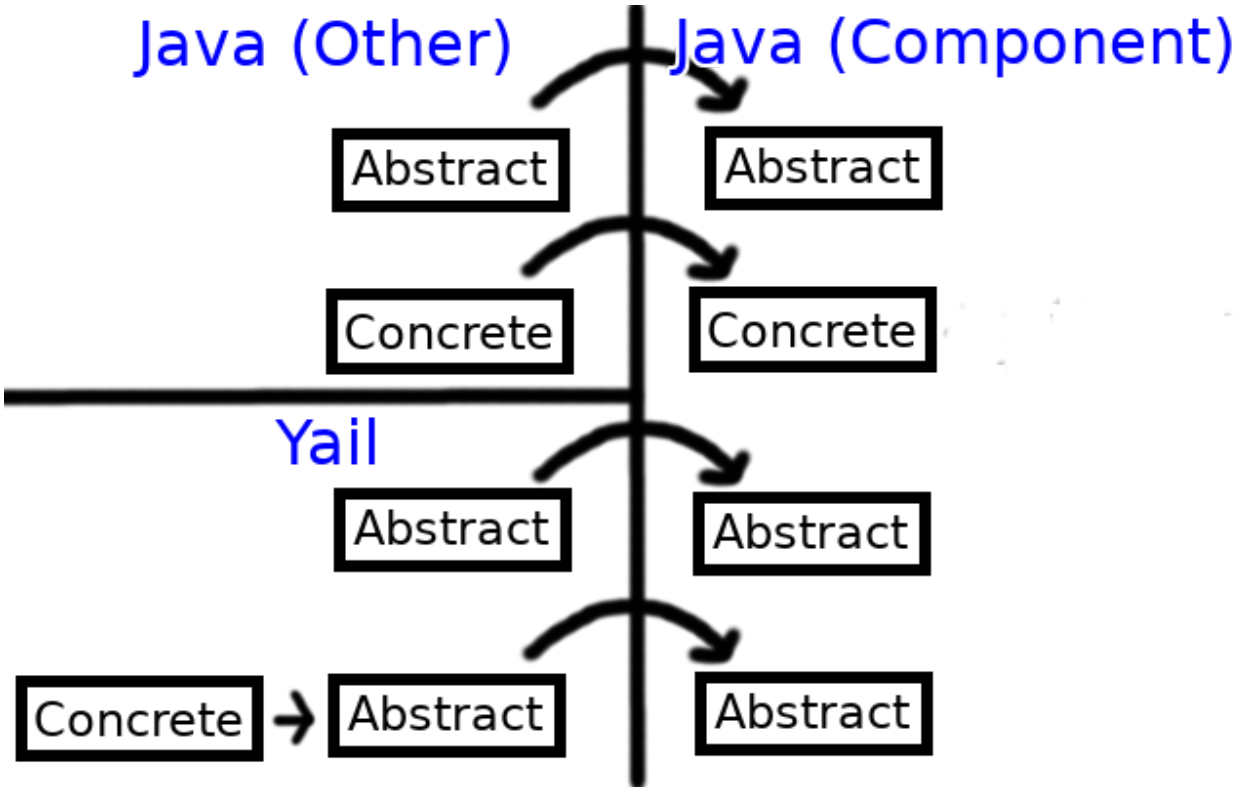
1) The Yail language needed to be modified to support overloads. Block code generation also needed to be modified to reflect this.
2) The Java code needed to be modified with overloads for every setter and method which should accept enums.
3) The communication from Yail to Java was still sometimes concrete.

The last system presents a solution to disadvantage 3, which makes all communication from Yail to Java abstract.

Java is Abstract System

The Java is Abstract System is as close as we got to our ideal system, while still being backwards compatible. Again the blocks language worked identically to the one originally presented by the Enums are Abstract System, but instead of all communication from Yail to Java being concrete, it was abstract.

In this system if a concrete value was passed to a method or setter, that **concrete value would be coerced to an enum** before being passed to Java. Enum values were passed in directly, because they were already abstract. Getters, method return values, and event parameters continued to be concrete.



This system also required method overloads within Java, as we needed functions to accept enum values, but could not remove the existing concrete functions because of backwards compatibility with extensions. It did not require Yail to have any concept of overloads. Yail only needed to know about the abstract version of a function, so we could use the existing system to accomplish that.

The advantages of this system were:

1) Dropdown blocks still conformed to the ideal system
2) Enum values passed to Java were always abstract.

The disadvantages of this system were:

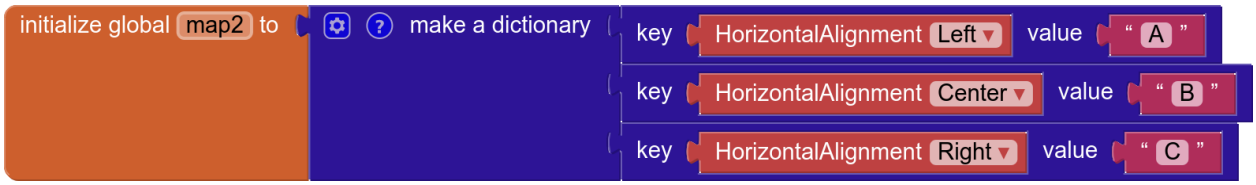
1) The Java code needed to be modified with overloads for every setter and method which should accept enums.
2) Enum definitions would need to provide a static method to go from a concrete value to an abstract value.

More Compatibility Problems

The systems presented above conform to our ideal system to varying degrees, but they all work within the constraints set out in the Backwards Compatibility section. Sadly, there were issues with compatibility that were not discovered until after these alternative systems were designed.

Forwards Compatibility: Dictionaries

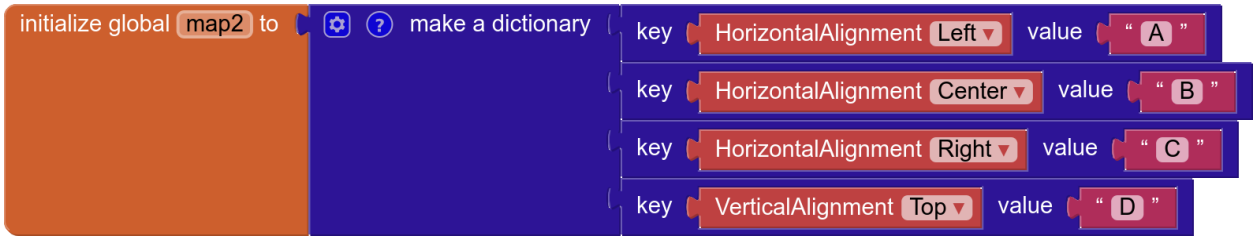
None of the above systems supported dropdown blocks as dictionary keys in an intuitive way.



In the Concrete System dropdown blocks returned concrete values. So in the above example we would get a dictionary that looks like the following:

```
{
  1: "A",
  3: "B",
  2: "C"
}
```

This is fine, unless we add another key to the dictionary which has the same concrete value as a different key. For example this set of blocks:



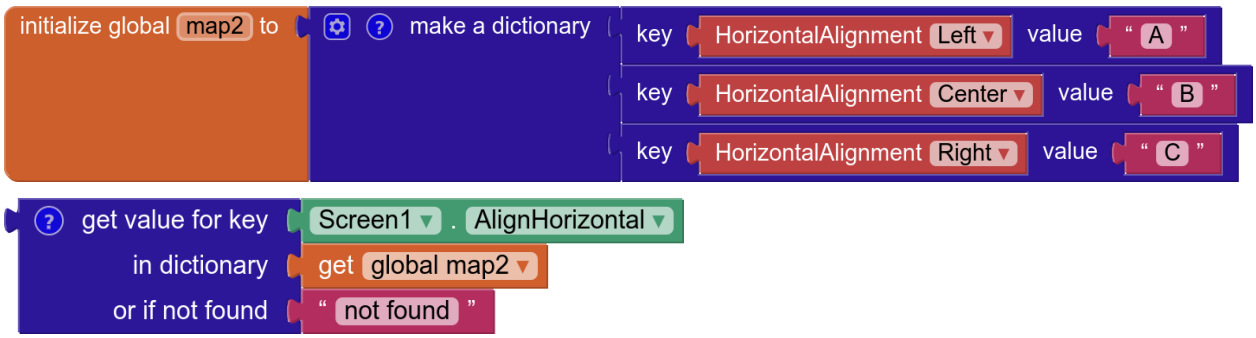
Which generates a dictionary that looks like the following:

```
{
  1: "A",
  3: "B",
  2: "C",
  1: "B"
}
```

The HorizontalAlignment.Left key is overwritten by the VerticalAlignment.Top key, even though from the perspective of a developer working in the blocks language, it looks like they should not conflict.

The other systems (Enums are Abstract, Two Systems, and Java is Abstract) worked differently by having dropdown blocks return enum values. Sadly this causes problems when we want to access values in an enum-keyed dictionary using a concrete lookup.

Consider the following situation where a developer using the blocks language wants to access values tied to abstract keys using a getter which returns a concrete value:

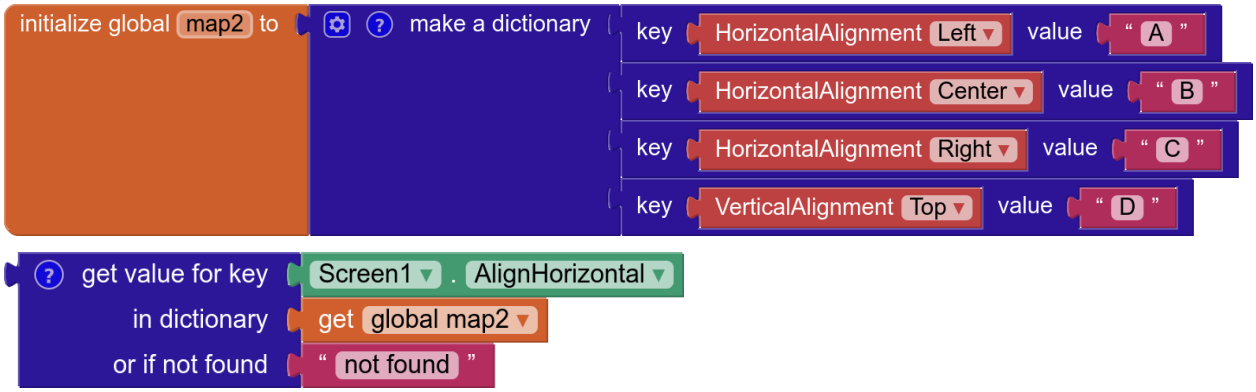


This looks like it should be possible from the perspective of said developer, yet it is not possible because our lookup is `1`, `3`, or `2` but our dictionary looks like the following:

```
{
  HorizontalAlignment.Left: "A",
  HorizontalAlignment.Center: "B",
  HorizontalAlignment.Right: "C",
}
```

In the case that our system is the Java is Abstract System we could coerce the concrete value to an enum before doing the lookup, because Java is Abstract requires enums to declare a

static method for going from concrete values to abstract values. But this still causes problems if we have two keys in the dictionary with the same concrete value:



If Screen1.AlignHorizontal returns 1, should we coerce it to a value of HorizontalAlignment.Left? Or a value of VerticalAlignment.Top? This is undecidable.

A simple solution to these problems is to not allow dropdown blocks to be dictionary keys. But this is disappointing as it would be handy for the user to be able to store state-based information using those values.

Yail Backwards Compatibility

The second unknown compatibility problem we ran into was Yail. Put in simple terms, we could not guarantee that an App Inventor user's version of Yail would be up to date. This meant that we could not include any new classes in Yail (including our enum definitions) and we could not change any Yail function headers.

This meant that the only system that would work backwards compatibly with Yail was the Concrete System, which provided us none of the advantages of our ideal system.

Final Plan

After discovering these other compatibility problems, we came up with a new 3-stage plan for completing Operation Options.

1: Totally Concrete

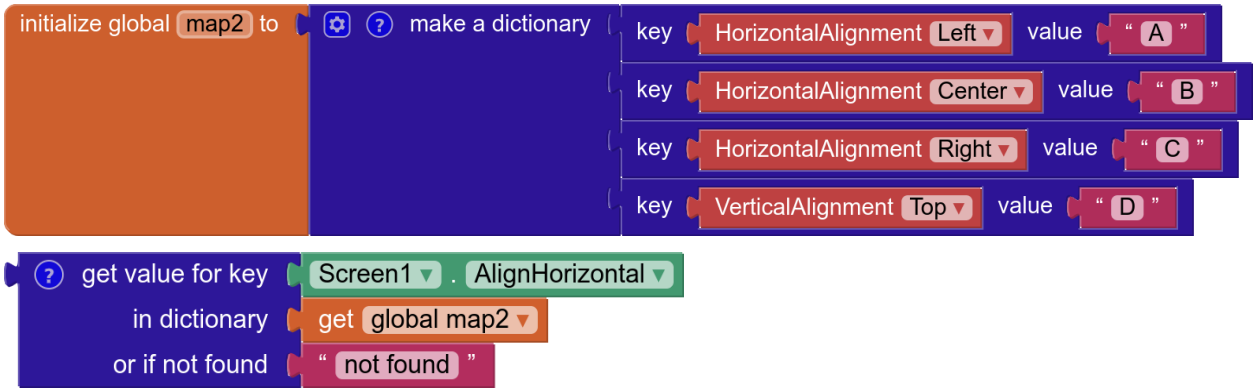
The first stage of the plan was to implement the Concrete System because it was the only system that was backwards compatible with the blocks language, Yail, and Java.

2: Sanitization & Abstraction

The second stage was meant to add support for dropdown blocks as dictionary keys. The system was changed to act like the Enums are Abstract System. Dropdown blocks returned enum values, but all communication from Yail to Java was concrete. Note that dropdown blocks only returned enum values when we detected that the user had an updated version of Yail (which supported our new enum definitions) otherwise the blocks still returned concrete values.

At this time we also added sanitization, which meant that when concrete values were passed from Java to Yail we converted them to an enum value, if the method returning the value supported doing so. Getters, method returns, and event parameters could now act abstractly under some circumstances. This allowed us to support dropdown blocks as dictionary keys.

Once again take for example this set of blocks:



In this case our generated dictionary would look like the following:

```
{
  HorizontalAlignment.Left: "A",
  HorizontalAlignment.Center: "B",
  HorizontalAlignment.Right: "C",
  VerticalAlignment.Top: "D"
}
```

But after stage 2 the value returned by Screen1.AlignHorizontal would be one of `HorizontalAlignment.Left`, `HorizontalAlignment.Center`, or `HorizontalAlignment.Right` (after sanitization), rather than a value of `1`, `3`, or `2`. This meant that we would no longer have conflicts.

3: Overloads

The final stage of the project was adding support for overloads in Yail as in the Two Systems System. This would allow abstract enum values to be passed from Yail to Java in the case where the user's version of Yail supported enum values.

Stages 1 and 2 were completed by me (Beka Westberg) during my 2020 GSoC project, but Stage 3 was not completed.

Advice for Stage 3

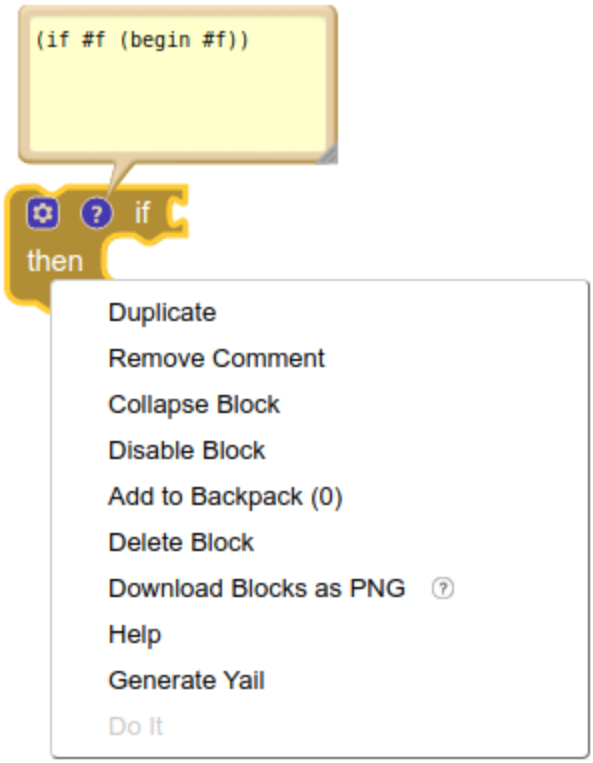
This section of the document is meant for a future person attempting to implement stage 3 of this project. It includes some guidance to (hopefully) help you get started on completing this.

Preparation / Learning

If you have never written Scheme or Lisp it may be helpful to complete the first two SICP lectures (1A and 1B). This is because you will be modifying the runtime.scm file, which is written in Yail, a variant of Scheme.

You should also learn how Yail handles the coercion of values to different types. This logic lives in the coerce-args procedure, which relies heavily on the coerce-arg function. These procedures make sure that a value is of the desired type, so that it can be safely passed into Java. For example the coerce-arg procedure will convert a value of "1" (which is a string) to a value of 1 (a number) if a number is desired.

You should know the basics of how Blockly code generators work, because you will have to modify the generators for the component blocks. You should familiarize yourself with those specific generators in particular. All of those weird constants like YAIL_OPEN_COMBINATION can be found in the yail.js file. Play around with it so you understand how the expected types get passed to yail. The "Generate Yail" option in the right-click dropdown menu of blocks (when you're running the site in admin mode) will be helpful.



You should also know the basics of how Blockly connection checks work, and you should familiarize yourself with the Blockly.Blocks.Utilities.YailTypeToBlocklyTypeMap which is where App Inventors “type hierarchy” is defined. You will likely need to modify the component blocks connection checks to support overloads.

You should check out the ComponentProcessor and get a grasp of how this reads the Java component files and compiles data into a more usable structure. Then you should check out the ComponentDescriptorGenerator which creates a simple_components.json file. This file tells the Blockly system which component blocks it needs to make, and you will need to modify this to contain information about overloads. The simple_components.json file is read by the ComponentDatabase (js side) so that the information is made available to Blockly, so you should also familiarize yourself with how this works.

Design Work

You should have a discussion with the dev team about whether you want to support all overloads, or just overloads for enums vs their underlying types.

If you decide to support overloads in general, here are some edge cases you should consider

Two different types. This is the simple case.

```
@SimpleFunction
public void Foo(String st) { }

@SimpleFunction
public void Foo(boolean bl) { }
```

Different numbers of types. This would be hard to make work in the blocks. I recommend not allowing it.

```
@SimpleFunction
public void Foo(String st) { }

@SimpleFunction
public void Foo(String st, boolean bl) { }
```

Multiple different types. You don't want to allow a combination of int & bool in the blocks. This would be hard to get working correctly with the type checking system. I recommend not allowing this either.

```
@SimpleFunction
public void Foo(String st, boolean bl) { }

@SimpleFunction
public void Foo(int in, String st) { }
```

All limitations you want to introduce should be handled via the ComponentProcessor. These cases should throw clear errors to help extension developers (or built-in component developers for that matter) know what to fix.

You should also be aware that *overrides* (not necessarily overloads) have special behavior in terms of component definitions. See this comment for more information. You should make sure that your solution is backwards compatible with this behavior.

Implementation

Implementation will be highly dependent on the choices you make in the design phase. But you will probably need to modify the following things:

- Modify ComponentDescriptorGenerator so that it exports information about parameters having multiple available types.
- Modify block checks so that they support accepting multiple types.
- Modify code generators so they export information about what types their inputs can accept.
- Modify coerce-args so that it can work with parameters of multiple types.

Best of luck! And I hope this document has helped you get started on your project =)