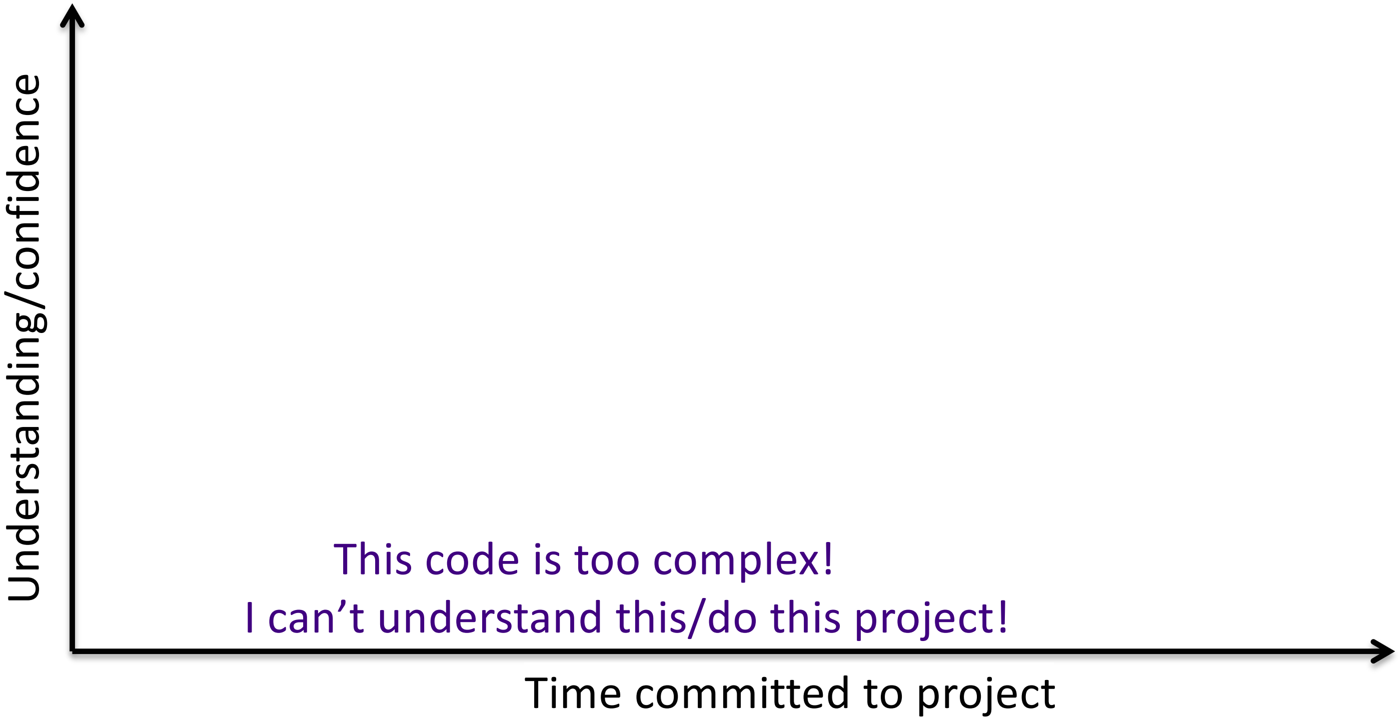
Objectives

- Picasso Design
  - Design patterns
- GUIs in Java
  - Anonymous inner classes
- Reflection

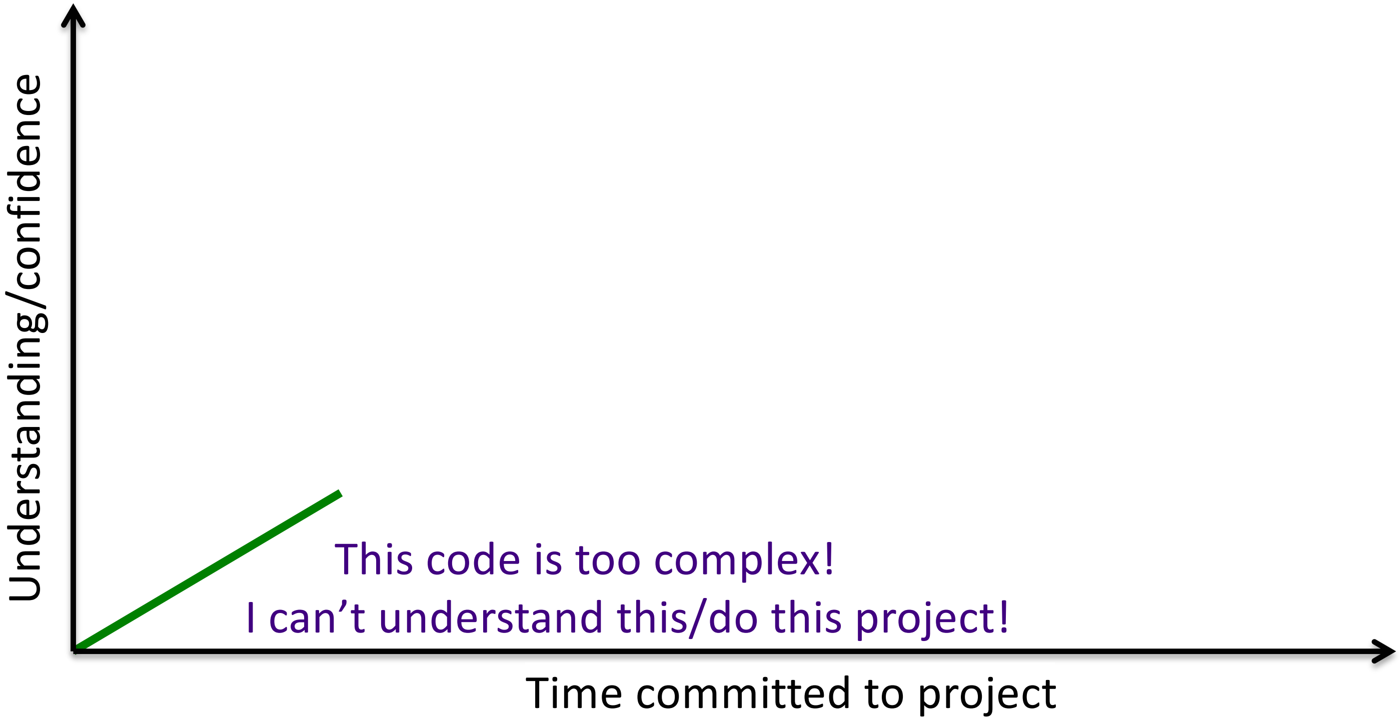
Typical Trajectory of Projects



This code is too complex!

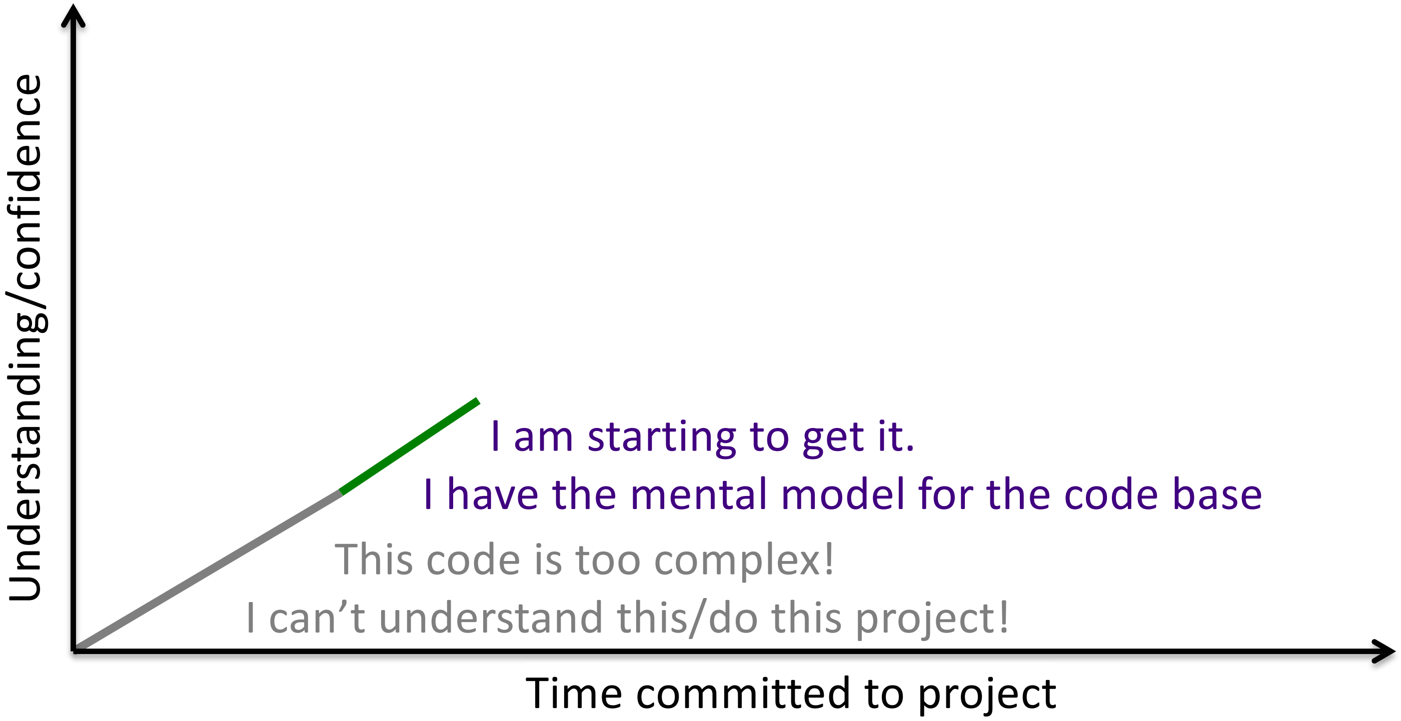
I can’t understand this/do this project!

Typical Trajectory of Projects



This code is too complex!
I can't understand this/do this project!

Typical Trajectory of Projects



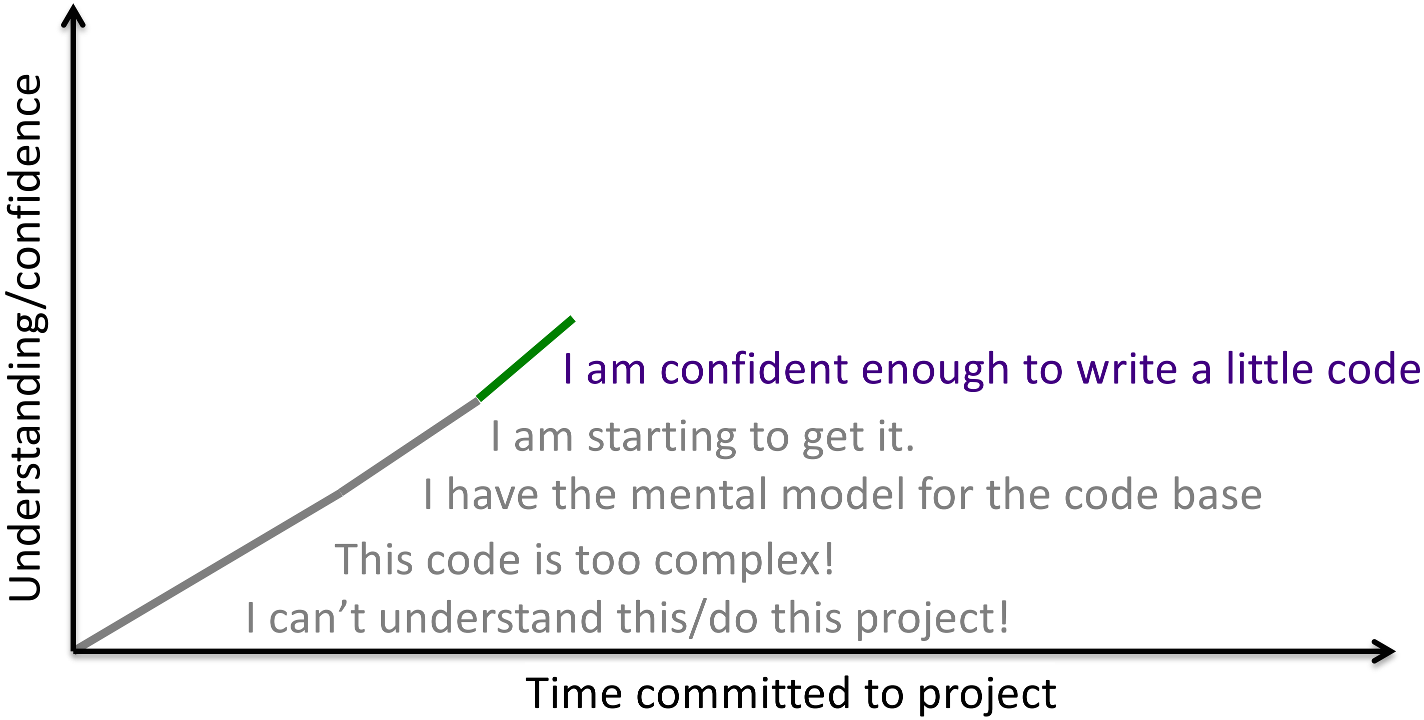
I am starting to get it.

I have the mental model for the code base

This code is too complex!

I can’t understand this/do this project!

Typical Trajectory of Projects



I am confident enough to write a little code

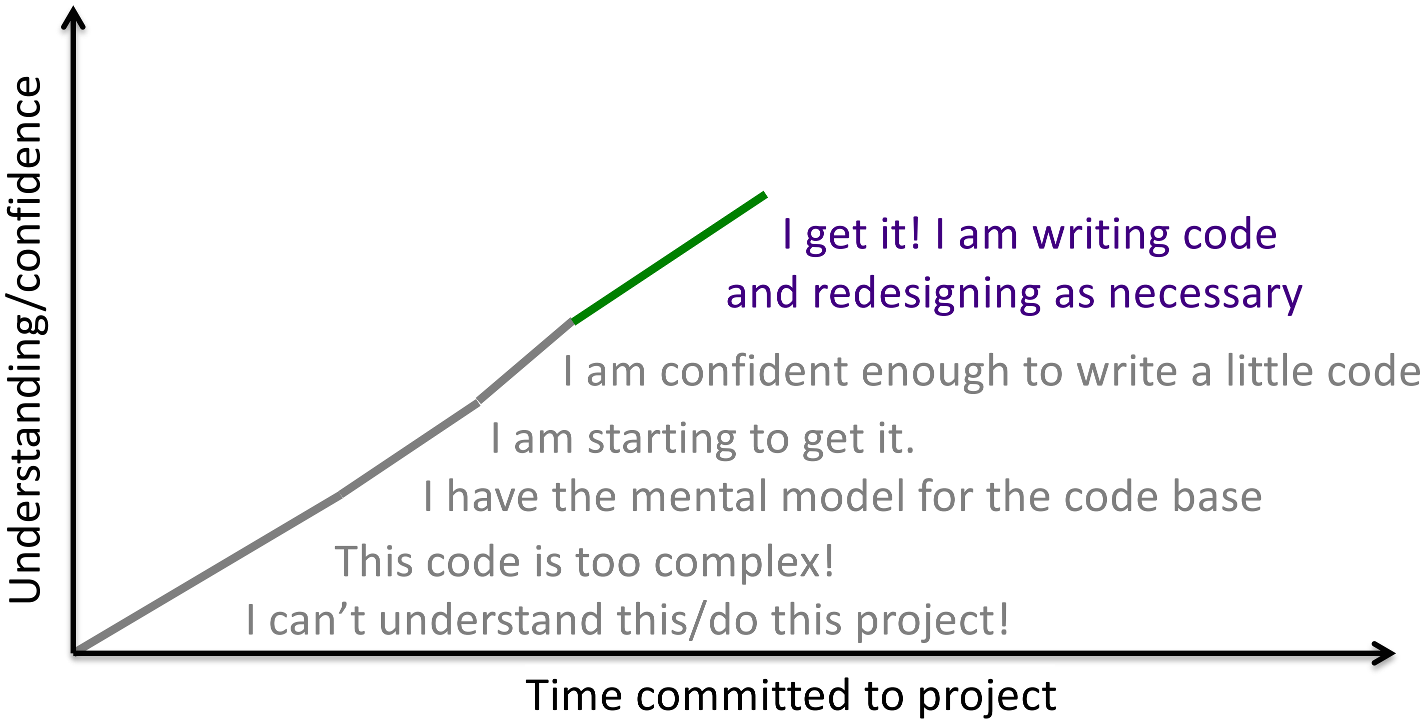
I am starting to get it.

I have the mental model for the code base

This code is too complex!

I can’t understand this/do this project!

Typical Trajectory of Projects



I get it! I am writing code
and redesigning as necessary

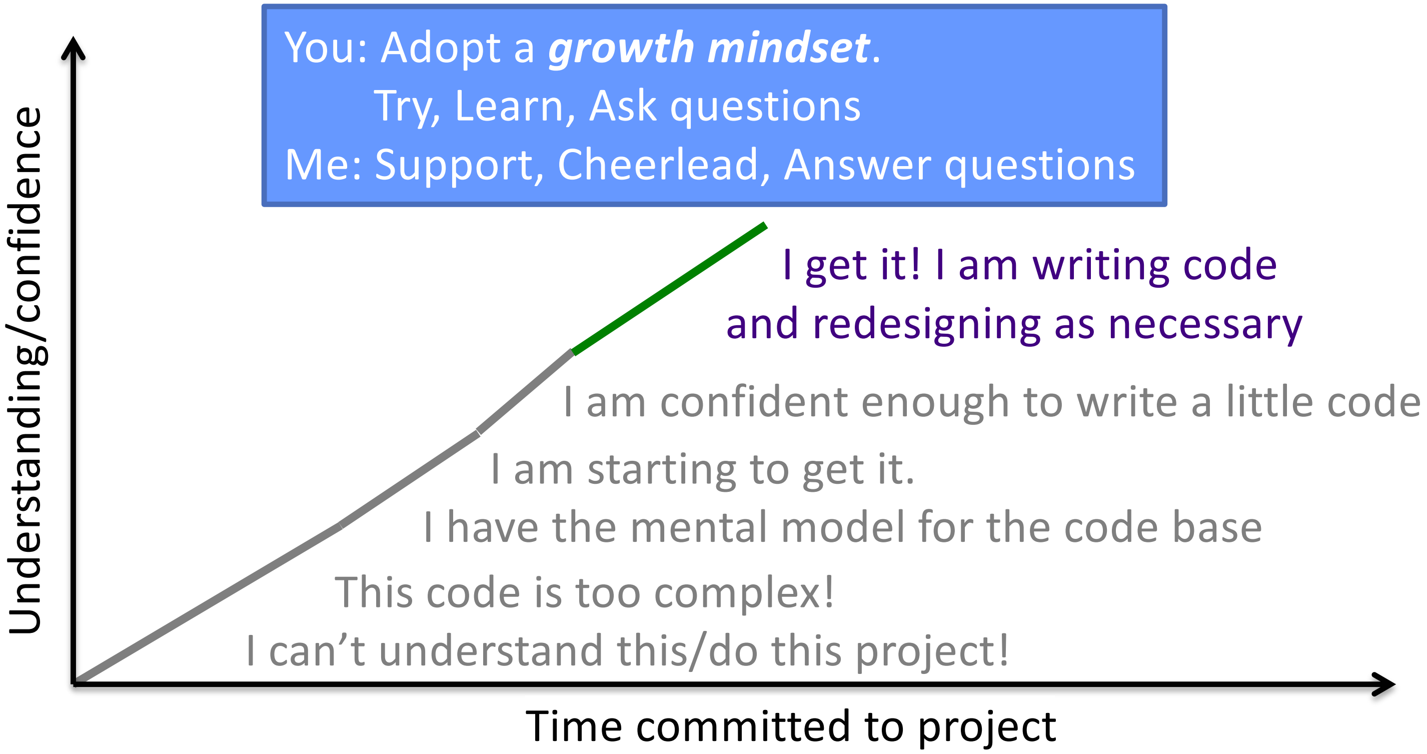
I am confident enough to write a little code

I am starting to get it.
I have the mental model for the code base

This code is too complex!
I can't understand this/do this project!

Our Responsibilities

You: Adopt a ***growth mindset***.
Try, Learn, Ask questions
Me: Support, Cheerlead, Answer questions



I get it! I am writing code
and redesigning as necessary

I am confident enough to write a little code

I am starting to get it.

I have the mental model for the code base

This code is too complex!

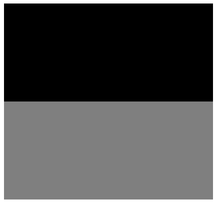
I can’t understand this/do this project!

Understanding/confidence

Time committed to project

Review

1. What is the goal of the Picasso project?
2. When you click the Evaluate button in the given version of Picasso, it generates the image for `floor(y)`
   - Explain why the generated image looks like this:
     - Include the constraints/rules of Picasso

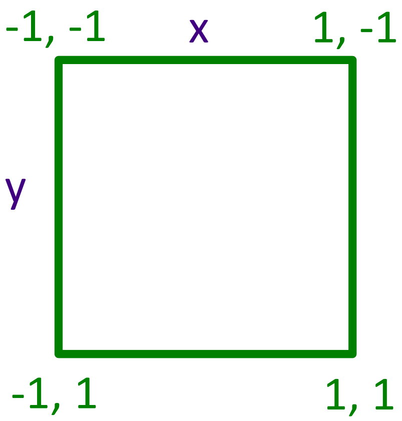


3. What should we think about during design and analysis of a project?
   - What are best practices?
4. How should we learn a code base?
5. How does an interpreter interpret a programming language?
   - What are the (important) Picasso classes that relate to each of those steps?

Review: Picasso Project Overview

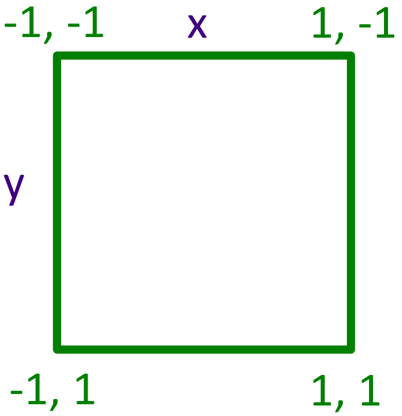
- Goal: Generate images from expressions
- Every pixel at position (x,y) gets assigned a color, computed from its x and y coordinate and the given expression
  - Range for x and y is [-1, 1]
- Colors are represented as RGB [red, green, blue] values
  - Component's range [-1, 1]
  - Black is [-1,-1,-1]
  - Red is [1,-1,-1]
  - Yellow is [1, 1,-1]

Points are (x,y)

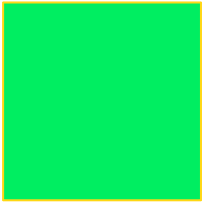


Review: Generating Images from Expressions

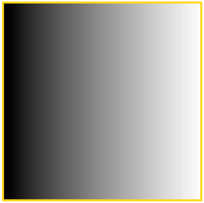
- ***Expressions*** at a specific (x,y) point/pixel evaluate to *RGB colors* [r,g,b]
  - `pixels[x][y] = expression.evaluate(x, y)`
- **x** evaluates to RGB color [x, x, x]
- In top right corner,
  - x evaluates to [1, 1, 1]
  - y evaluates to [-1, -1, -1]



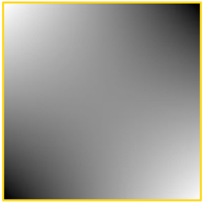
Review: Generated Expressions



[-1, 1, -1]



x



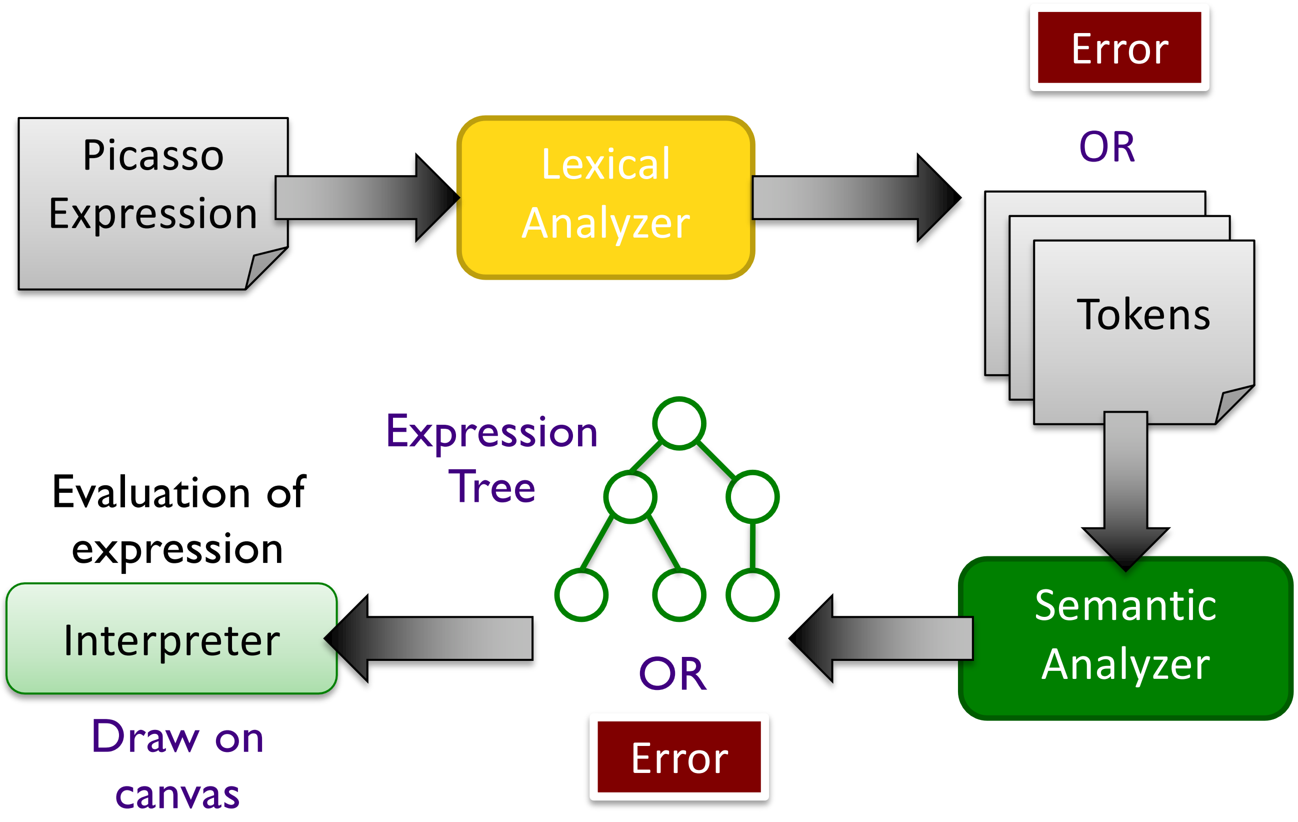
x*y

```
For all x:
    For all y:
        pixels[x][y] = expression.evaluate(x, y)
```

Review: Programming Language Design

- Must be unambiguous
  - Programming Language defines a ***syntax*** and ***semantics***
- Interpreting programming languages
  1. Parse program into tokens
  2. Verify that tokens are in a valid form
  3. Generate executable code
  4. Execute code

Review: Interpreting the Picasso Language

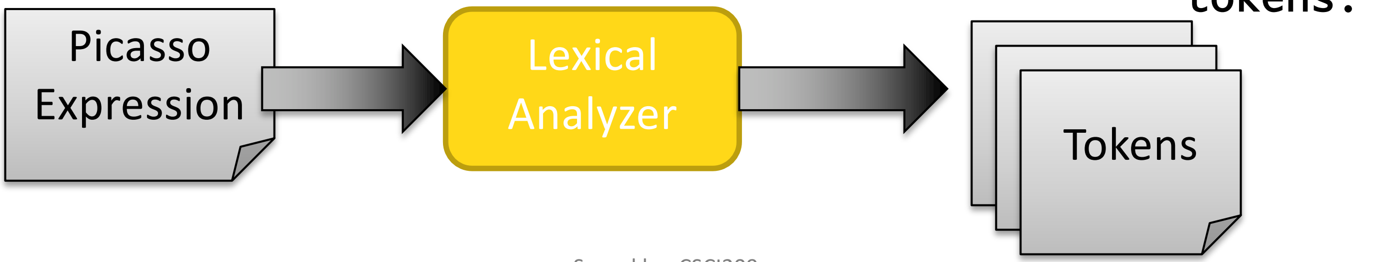


Understanding the Code

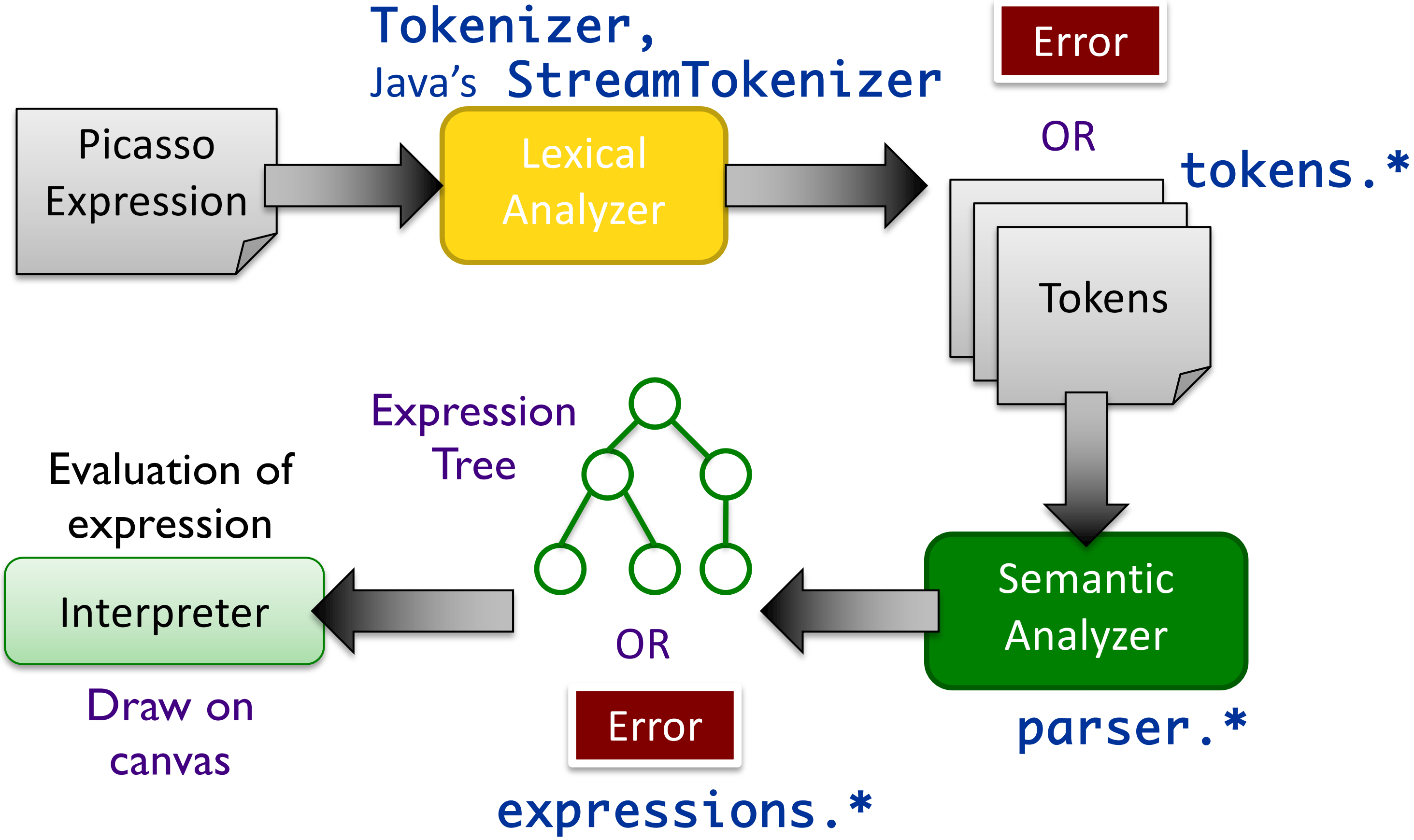
- How does the given code map to lexical analysis, semantic analysis, and evaluation components?
  - Look for packages, classes that map to these steps
- Suggestions:
  - Look for important words/terms from problem domain
  - Look for terms from design patterns
  - Put code in black boxes or group code together
- Task: Label the process picture with the associated packages/classes

Process of Understanding Code: Building Your Mental Model

- Look for important words/terms from problem domain
- Look for terms from design patterns
- Put code in black boxes or group code together
- Example:



Interpreting the Picasso Language



Tokenizer,

Java's StreamTokenizer

Error

OR

tokens.*

Picasso Expression

Lexical Analyzer

Tokens

Semantic Analyzer

parser.*

Expression Tree

OR

Error

expressions.*

Evaluation of expression

Interpreter

Draw on canvas

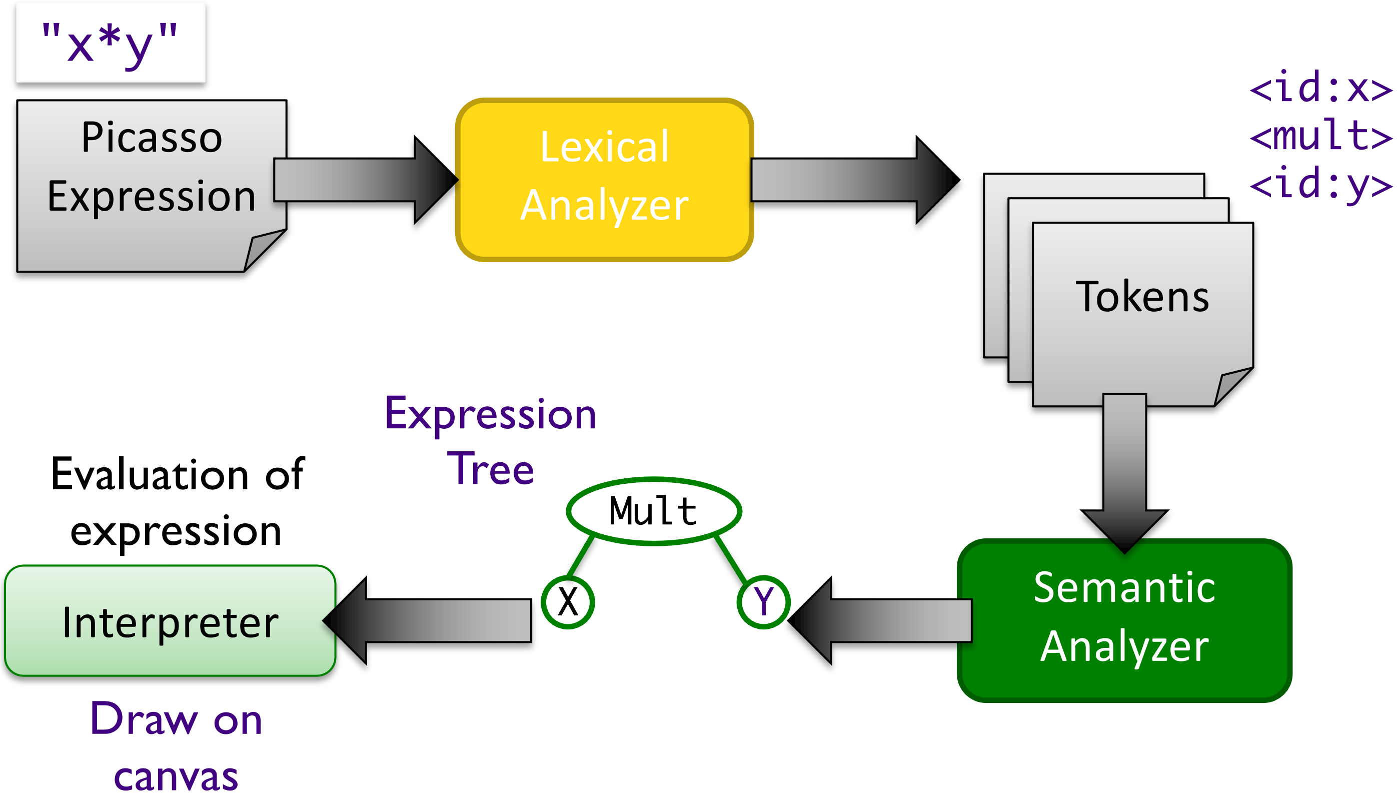
Process of Understanding Code: Building Your Mental Model

- Apply spiral model to understanding code
- Review problem specification (low-cost effort)
- Explore code at the top-level (low-cost effort)
  - Look at packages, class names
  - Don’t take a deep-dive until you have the bigger picture

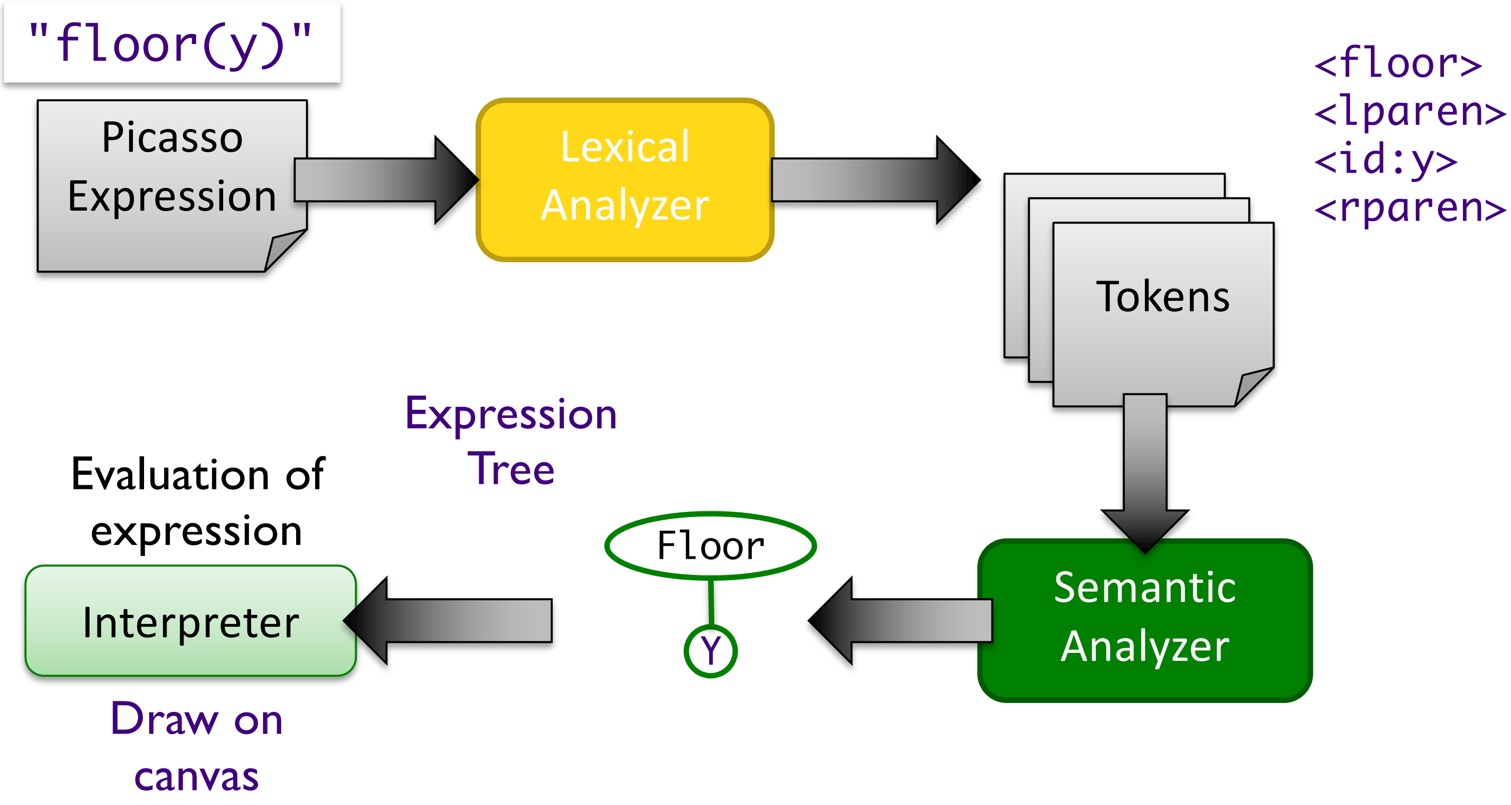
Process of Understanding Code: Building Your Mental Model

- After you have the big picture, look at most important classes
- Decide: Does this class merit a closer look? Or do I just need the big picture of what it does?
    - Lean towards the latter towards the beginning
    - Look for class hierarchy and focus on parent classes
- Iterate!
    - Grow your mental model
    - What a "closer look" means changes over time
        - Early: what public methods does the class have? What does the documentation say they do? What do they return?
        - Later: what do these methods do? How does this class interact with other objects?

Interpreting the Picasso Language



Interpreting the Picasso Language



Understanding the Code: Lexical Analysis

- Process
  - `picasso.parser.Tokenizer`
  - `picasso.parser.tokens.TokenFactory`
- Output:
  - `picasso.parser.tokens.*`

`FloorToken`

Understanding the Code: Semantic Analysis

- Process
  - `picasso.parser.ExpressionTreeGenerator`
  - `picasso.parser.SemanticAnalyzer`
  - `picasso.parser.*Analyzer`
- Output
  - `picasso.parser.language.expressions.*`

`FloorAnalyzer`

Understanding the Code: Evaluation

- Process
  - `picasso.parser.language.ExpressionTreeNode`
- Output:
  - `picasso.parser.language.expressions.RGBColor`

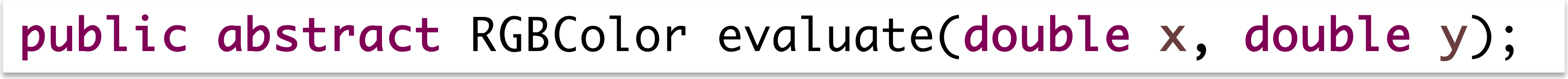
- Displayed in `PixMap` on `Canvas`

`Floor`

Understanding the Code: Evaluation

- Key Parent class:

`picasso.parser.language.ExpressionTreeNode`

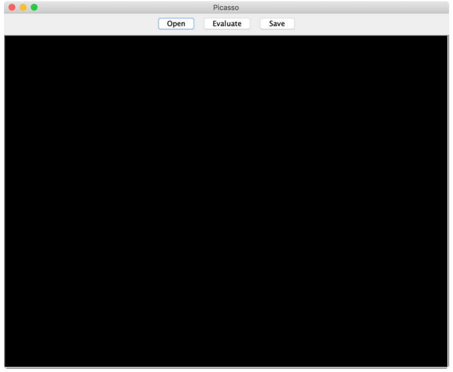


```
public abstract RGBColor evaluate(double x, double y);
```

- “Old” version of expressions:
  - ReferenceForExpressionEvaluations

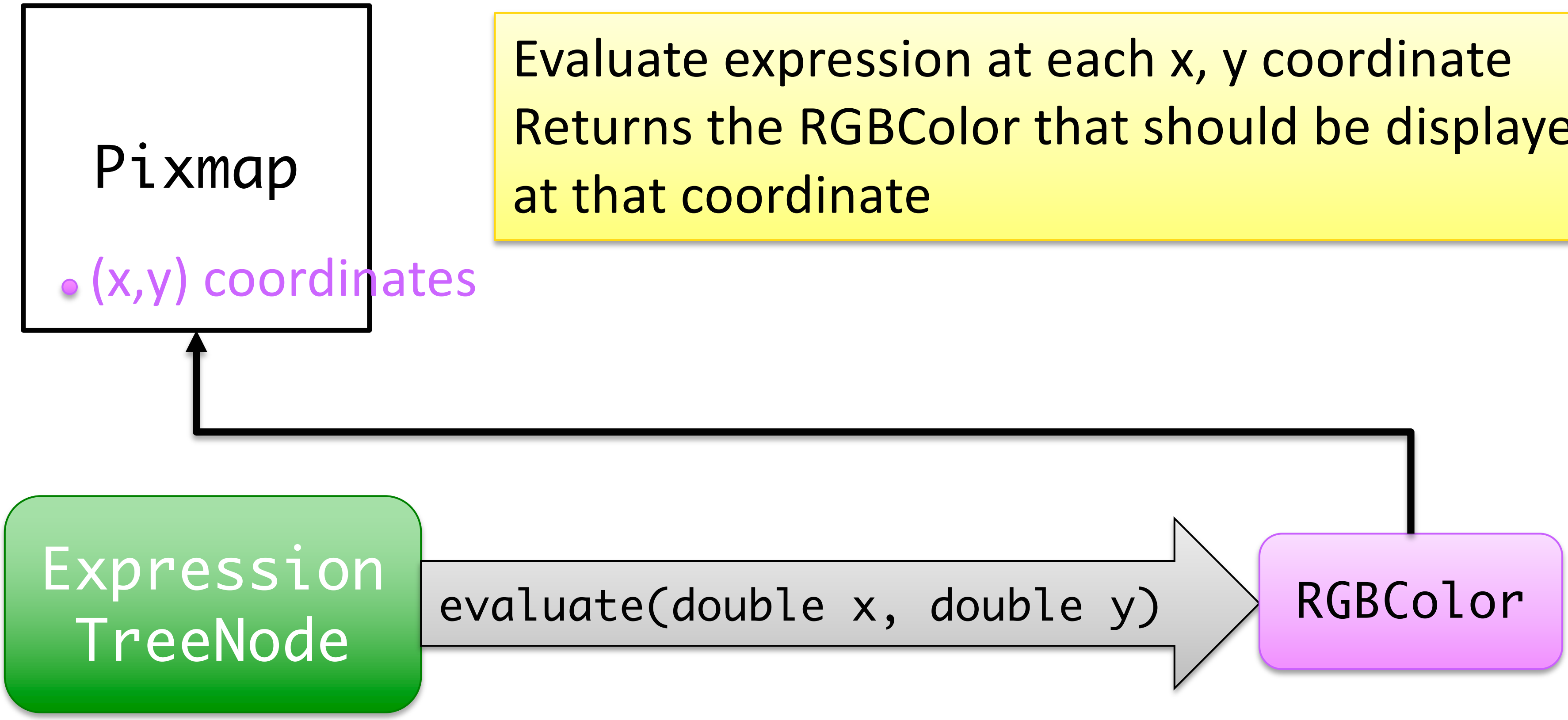
Understanding Code: A Top-Down Approach

- Run program
- Start at `Main.java`
  - Follow calls to see how GUI is created
    - Breadth- or depth-first search
  - What classes make up the GUI?
- GUIs often follow the MVC design pattern
  - Identify the model, view-controller in Picasso



Evaluator: Expression Evaluation

Evaluate expression at each x, y coordinate
Returns the RGBColor that should be displayed at that coordinate



Pixmap

- (x,y) coordinates

Expression TreeNode

evaluate(double x, double y)

RGBColor

How is the floor function parsed?

(in given code)

- What classes are needed?
- How would you add another function to the language?
  - For example, consider how you would add the cosine function

How is the floor function parsed?

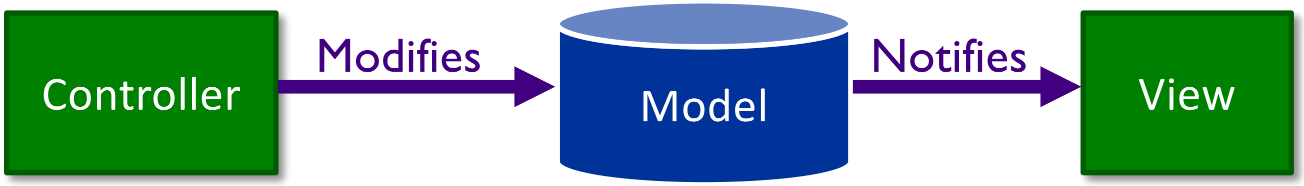
(in given code)

- Has a *token* to represent the *floor* function
  - Same prefix as function, e.g., `FloorToken.java`
  - `floor` is listed in `functions.conf`
- `FloorAnalyzer` is the *semantic analyzer* for the function
  - Note has same prefix as function: `FloorAnalyzer.java`
  - `Analyzer` class implements `SemanticAnalyzerInterface`, returns an instance of `ExpressionTreeNode`
    - Specifically: `Floor` object

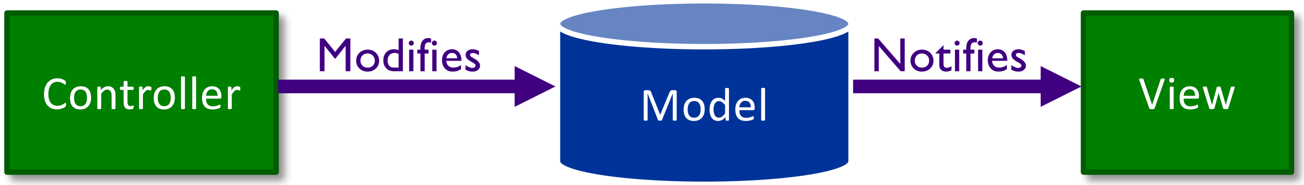
Why is the naming important for the token and analyzer?

Model - Viewer - Controller (MVC)

- A common **design pattern** for GUIs
- Loosely coupled
  - Model: application data
  - View: graphical representation
  - Controller: input processing

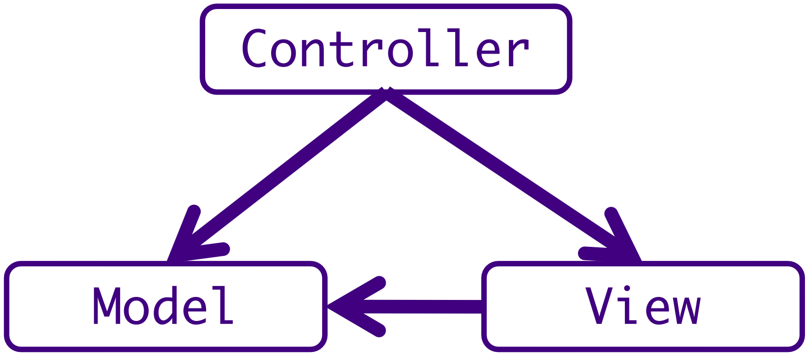


Model-Viewer-Controller

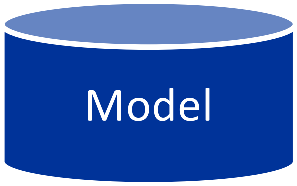


- Can have multiple viewers and controllers
- Goal: modify one component without affecting others

Direct associations

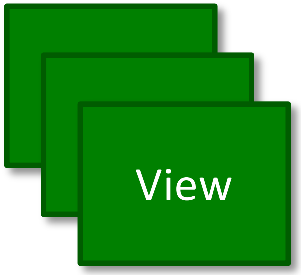


Model

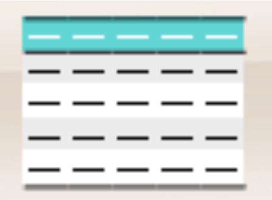

Model

- Represents application state
- Responsible for managing application state
- Purely **functional**
  - Nothing about how view presented to user

Multiple Views

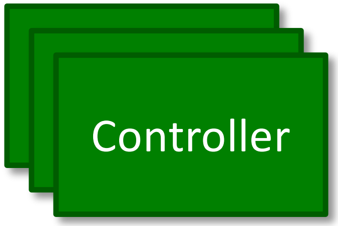


- Provides graphical components for model
  - Look & Feel of the application
- User manipulates view
  - Informs **controller** of change



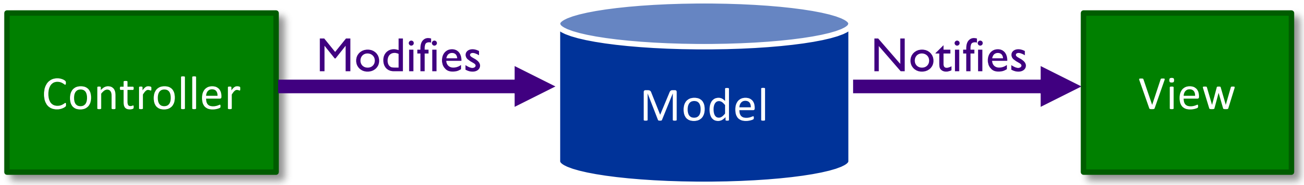
- Example of multiple views: spreadsheet data
  - Rows/columns in spreadsheet
  - Pie chart, bar chart, …

Controller(s)



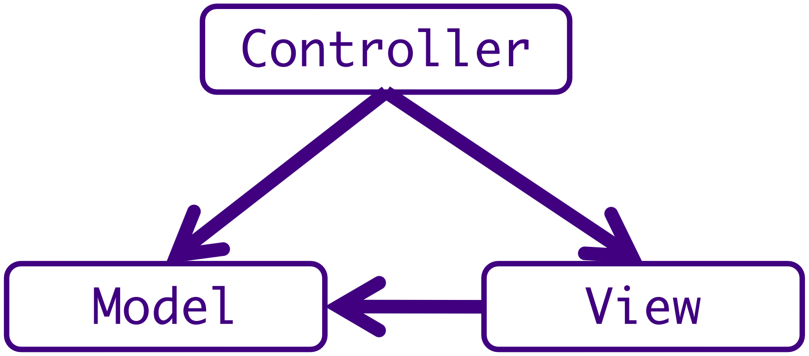
- Handles user input
- Update **model** as user interacts with **view**
  - Call model's methods (often mutators)
  - Makes decisions about behavior of model based on UI
- Views are associated with controllers

Discussion: Map MVC to Goblin Game

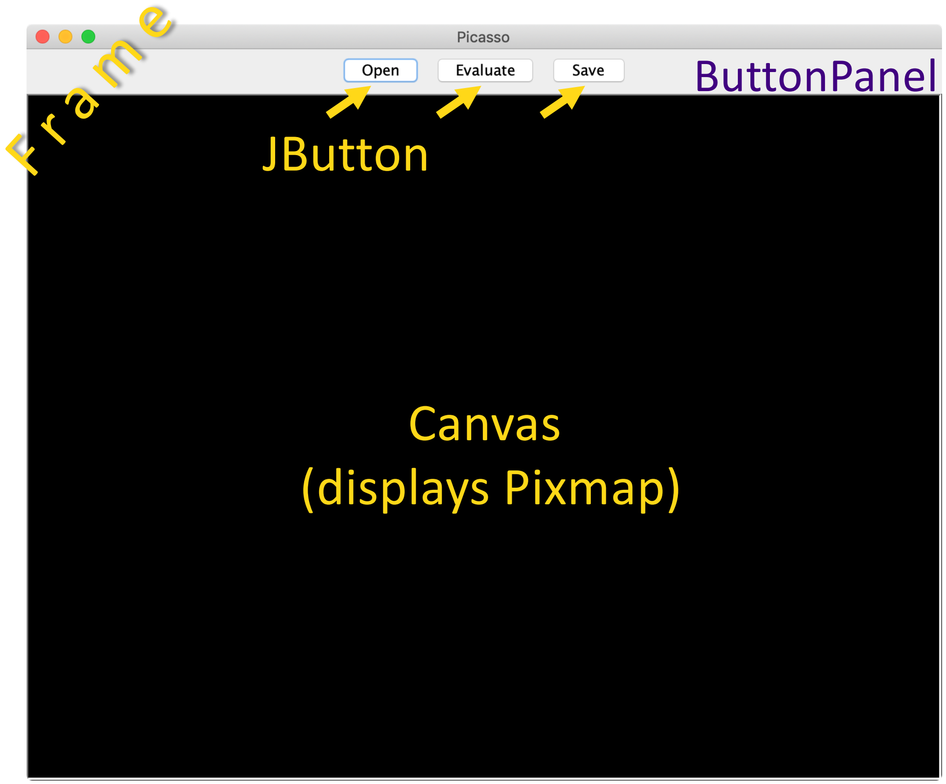


- Can have multiple viewers and controllers
- Goal: modify one component without affecting others

Direct associations



Picasso GUI

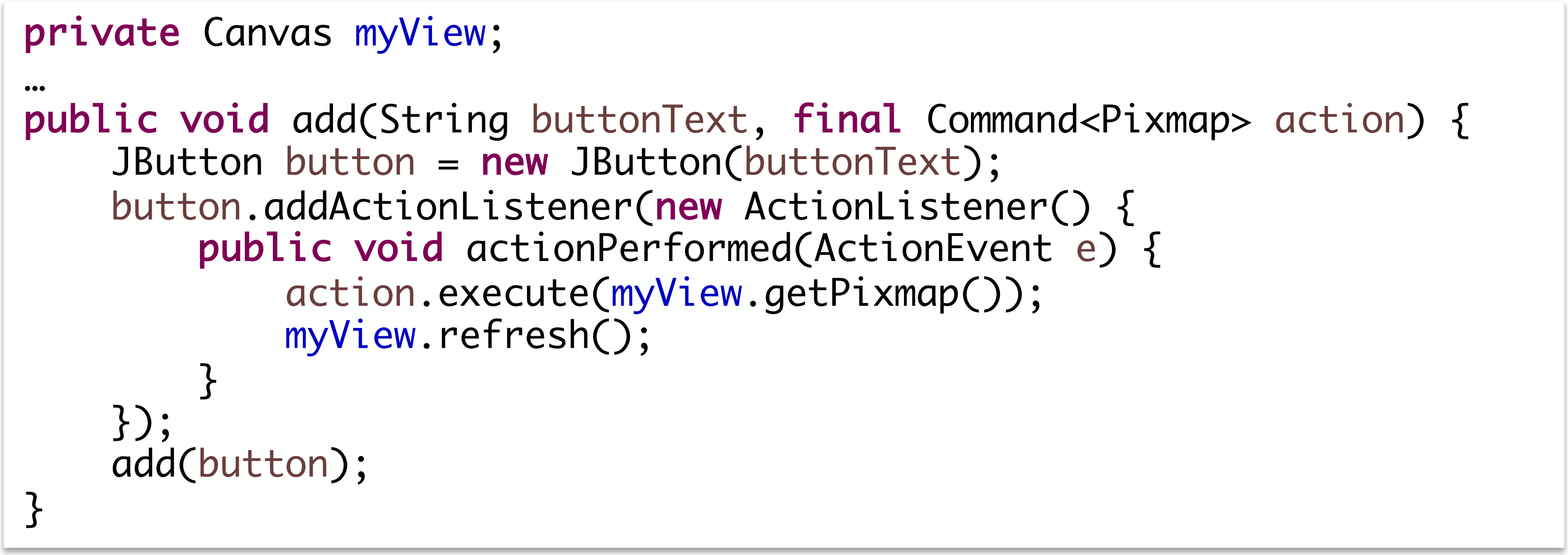


Picasso's GUI uses classes from two main Java packages:

- Abstract Windowing Toolkit: `java.awt`
- Swing: `javax.swing`

Understanding GUI Code

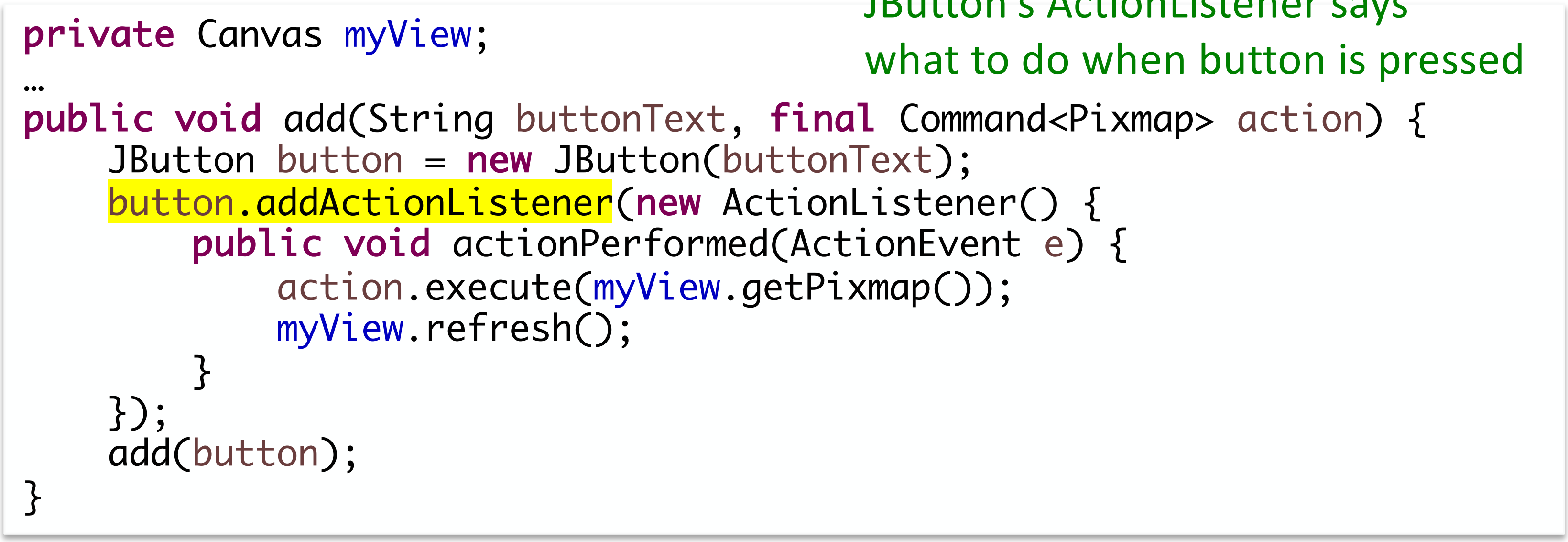
- In `ButtonPanel.java`, buttons are associated with a command or action



```
private Canvas myView;
…
public void add(String buttonText, final Command<Pixmap> action) {
    JButton button = new JButton(buttonText);
    button.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            action.execute(myView.getPixmap());
            myView.refresh();
        }
    });
    add(button);
}
```

Understanding GUI Code

- In `ButtonPanel.java`, buttons are associated with a command or action

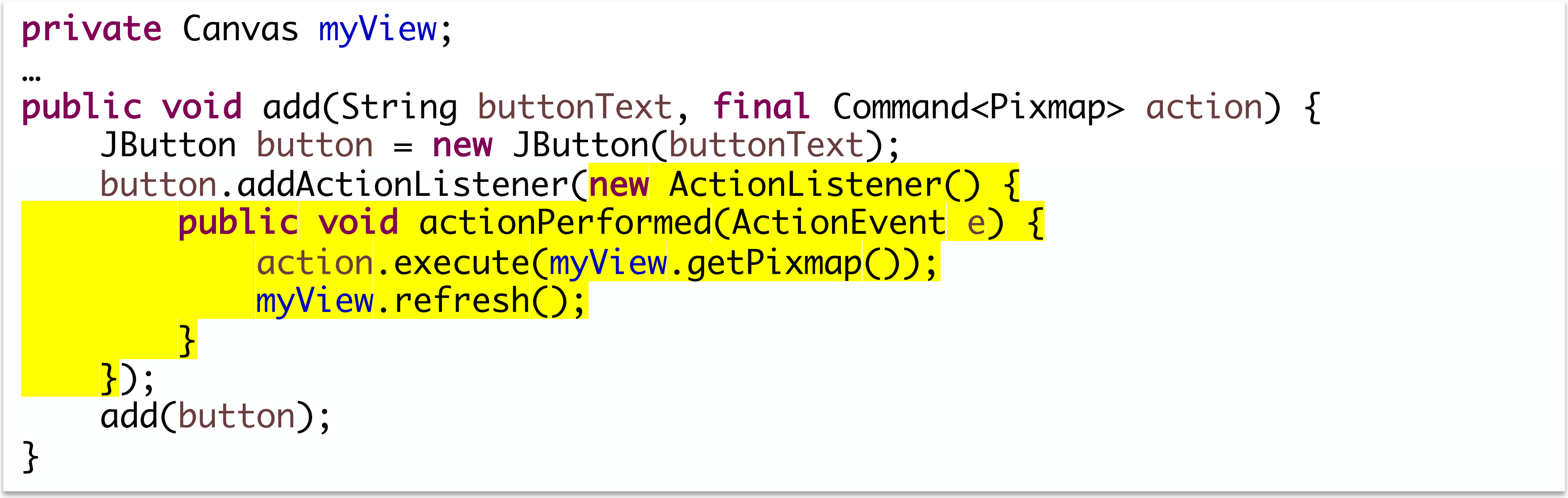


```
private Canvas myView;
…
public void add(String buttonText, final Command<Pixmap> action) {
    JButton button = new JButton(buttonText);
    button.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            action.execute(myView.getPixmap());
            myView.refresh();
        }
    });
    add(button);
}
```

JButton’s ActionListener says what to do when button is pressed

Understanding GUI Code

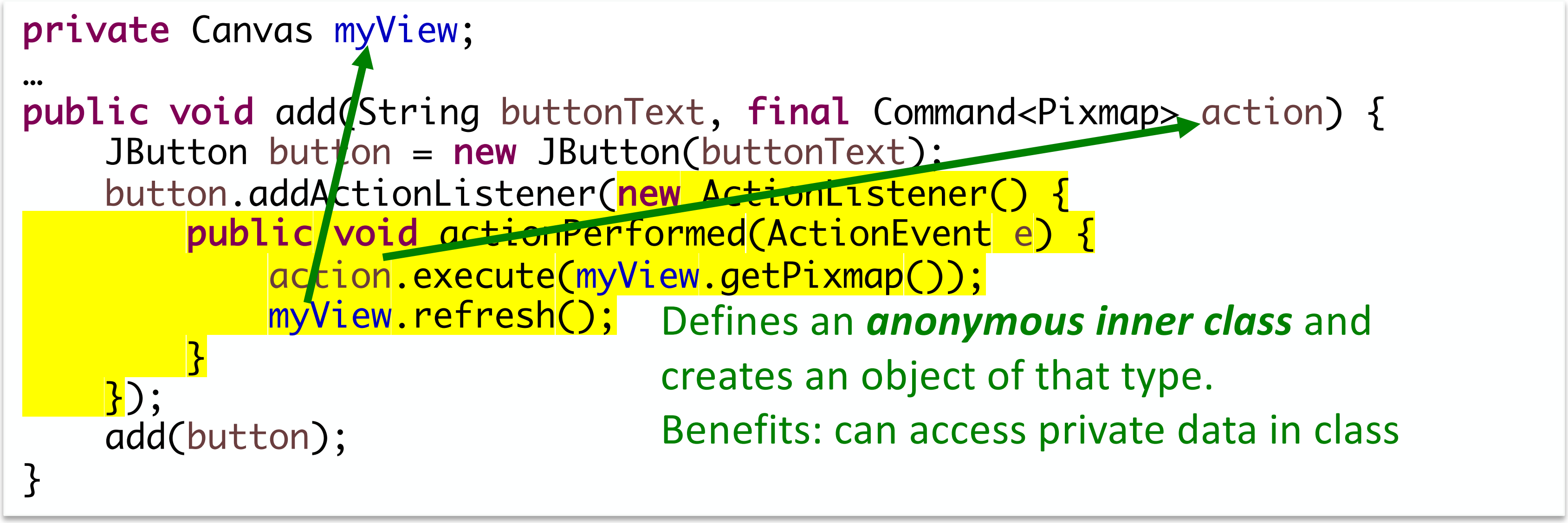
- In `ButtonPanel.java`, buttons are associated with a command or action



```
private Canvas myView;
…
public void add(String buttonText, final Command<Pixmap> action) {
    JButton button = new JButton(buttonText);
    button.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            action.execute(myView.getPixmap());
            myView.refresh();
        }
    });
    add(button);
}
```

Understanding GUI Code

- In `ButtonPanel.java`, buttons are associated with a command or action



```
private Canvas myView;
…
public void add(String buttonText, final Command<Pixmap> action) {
    JButton button = new JButton(buttonText);
    button.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            action.execute(myView.getPixmap());
            myView.refresh();
        }
    });
    add(button);
}
```

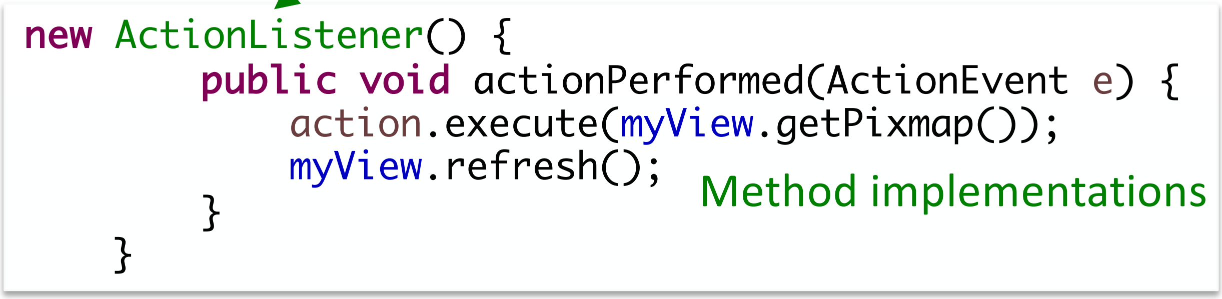
Defines an ***anonymous inner class*** and creates an object of that type.

Benefits: can access private data in class

Anonymous Inner Classes

- Common way to write (certain) code
- No classname
  - Class is *anonymous*
- Extends a parent class or implements an interface

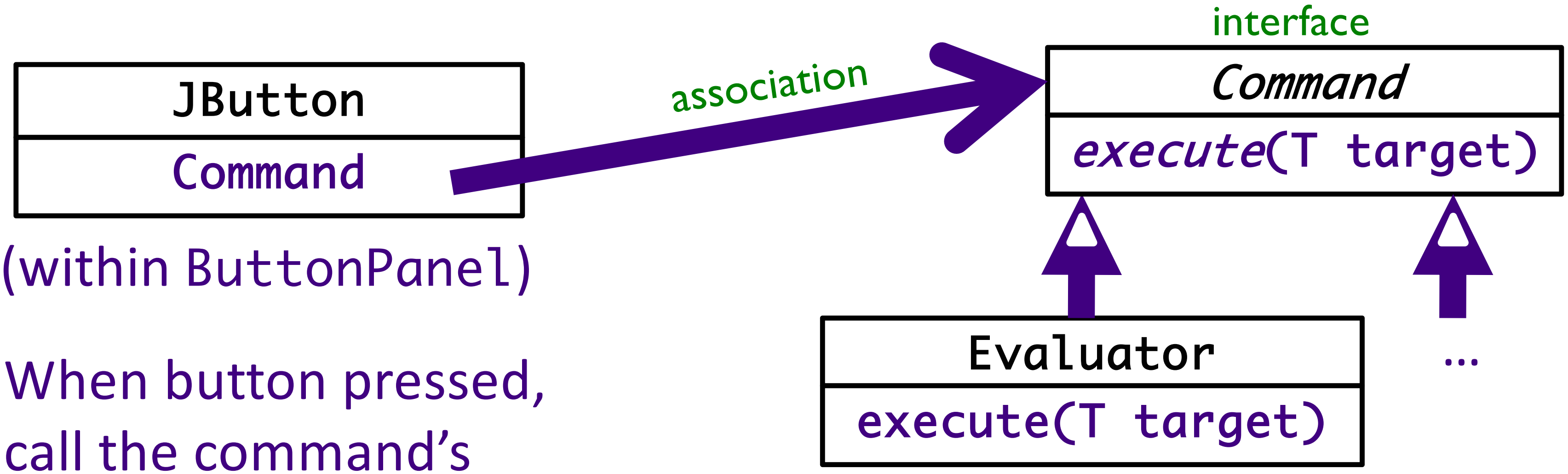
the parent class/interface



```
new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            action.execute(myView.getPixmap());
            myView.refresh();
        }
    }
```

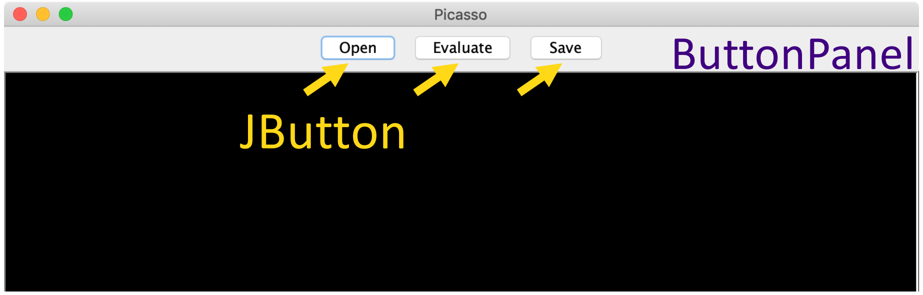
Method implementations

Picasso GUI: ButtonPanel



(within ButtonPanel)

When button pressed, call the command's execute method



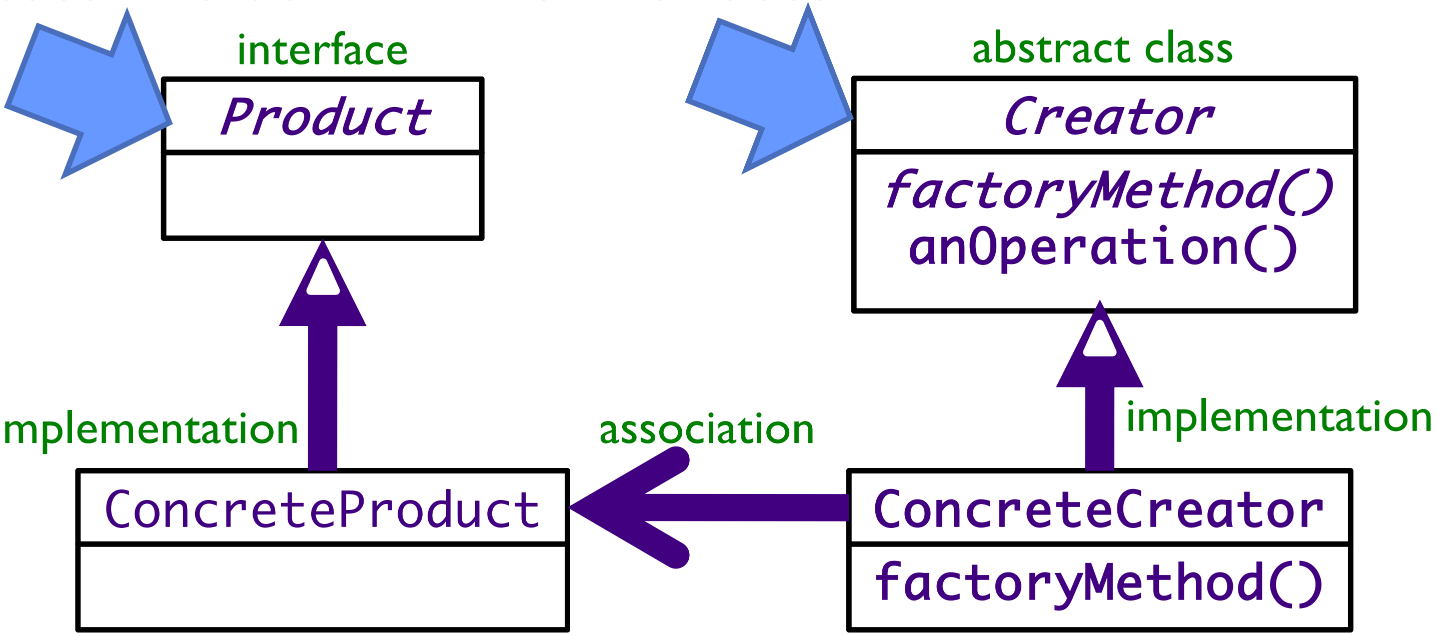
FACTORY DESIGN PATTERN

Design Pattern: **Factory Methods**

- Allows creating objects without specifying exact (concrete) class of created object
- Often used to refer to any method whose main purpose is creating objects
- How it works:
  1. Define a method for creating objects
  2. Child classes override method to specify the derived type of product that will be created

Factory Method Pattern

Client classes interact with the interfaces



interface

Product

abstract class

Creator

factoryMethod()
anOperation()

implementation

association

implementation

ConcreteProduct

ConcreteCreator

factoryMethod()

UML Class Diagram

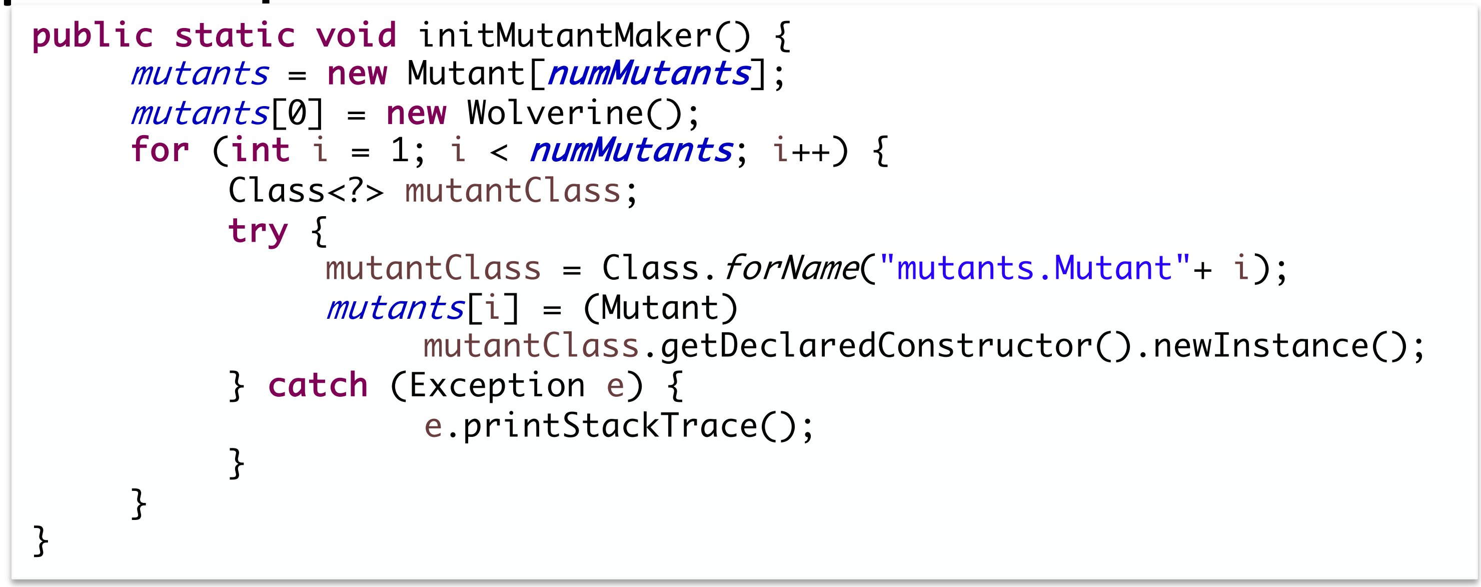
Dependency Inversion Principle

Depend upon Abstractions

“Inversion” from the way you think

Using Reflection in Java

- *Reflection* allows us to create objects of a class using the *name* of the class
- Example adapted from MutantMaker:



```java
public static void initMutantMaker() {
    mutants = new Mutant[numMutants];
    mutants[0] = new Wolverine();
    for (int i = 1; i < numMutants; i++) {
        Class<?> mutantClass;
        try {
            mutantClass = Class.forName("mutants.Mutant"+ i);
            mutants[i] = (Mutant)
                mutantClass.getDeclaredConstructor().newInstance();
        } catch (Exception e) {
                e.printStackTrace();
        }
    }
}
```

Using Reflection in Java

- Can create objects of a class through the *name* of the class
- Used in `SemanticAnalyzer`
  - Gets list of functions
    - Read from `conf/functions.conf`
  - Maps a token to the class responsible for parsing that type of token
  - When `SemanticAnalyzer` sees that token, calls the respective analyzer to parse
  - Example: `FloorToken` maps to the `FloorAnalyzer`
    - `FloorAnalyzer` pops the `Floor` token off the stack and then parses the (one) parameter for the *floor* function

TODO

- Project Analysis due Friday before class

- Note: the given code is not perfectly designed
  - What would “perfectly” designed even mean?
  - But, need to understand the given code.