Java GUI Programming

A quick-start guide to building Swing applications.
With examples and pictures.

Introduction

Background

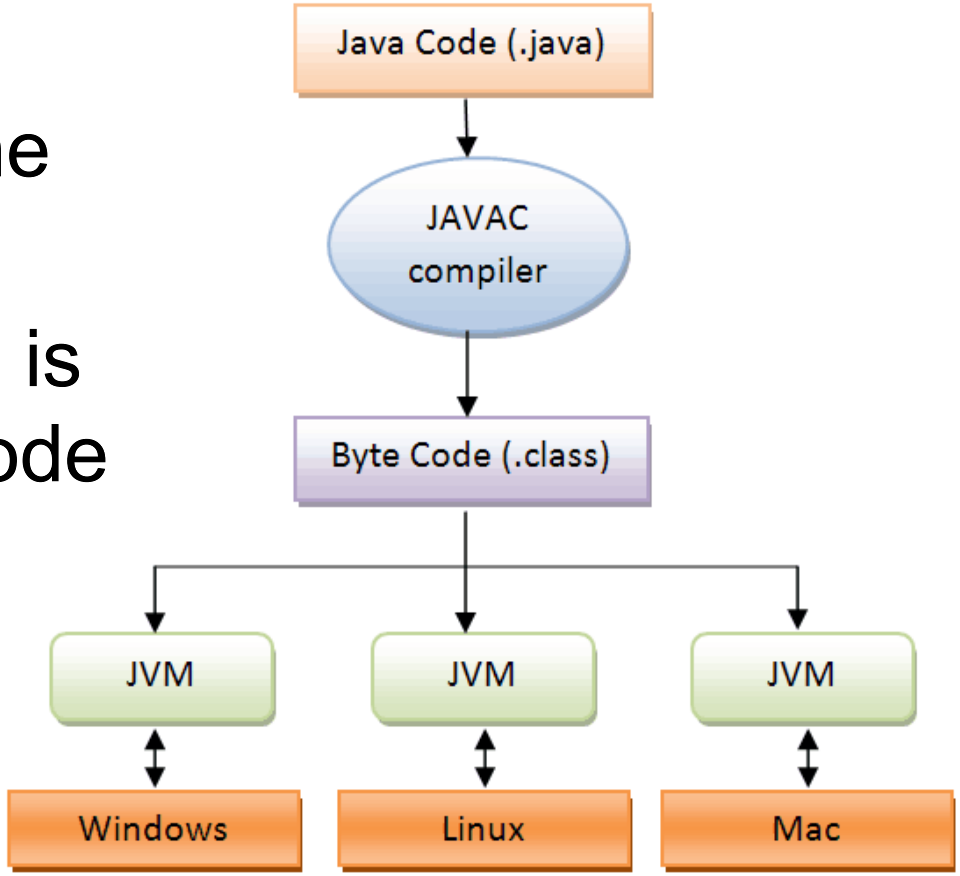
Design goals

Background

- Designed by James Gosling; released by Sun Microsystems in 1995.
  - Made open source under GNU GPL in 2007.
  - Sun and Java acquired by Oracle in 2010.
- Portable through virtualization.
  - Requires Java Virtual Machine (JVM) on each target platform.
  - Scale from small devices to large deployments.
- Class-based, object-oriented design.
  - C++ syntax
  - Strongly typed, interpreted language
  - Extensive class libraries included

Java Virtual Machine (JVM)

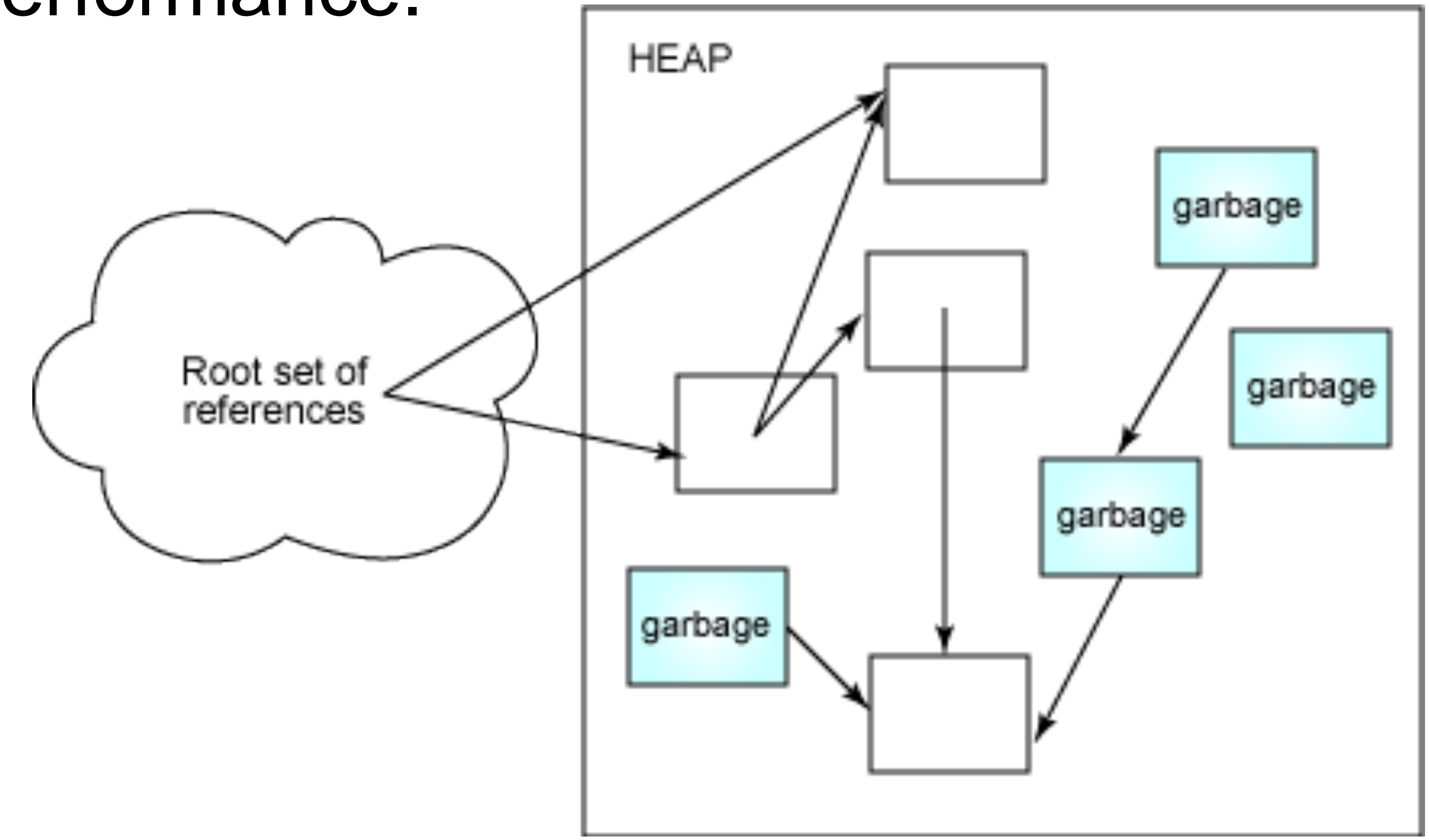
- Portability through virtualization
  - Java compiles to bytecode (IR).
  - Bytecode is executed by a Java virtual machine (JVM) on the target platform.
  - Interpreted bytecode is slower than native code BUT just-in-time compilation can give near-native performance.



http://viralpatel.net/blogs/java-virtual-machine-an-inside-story/

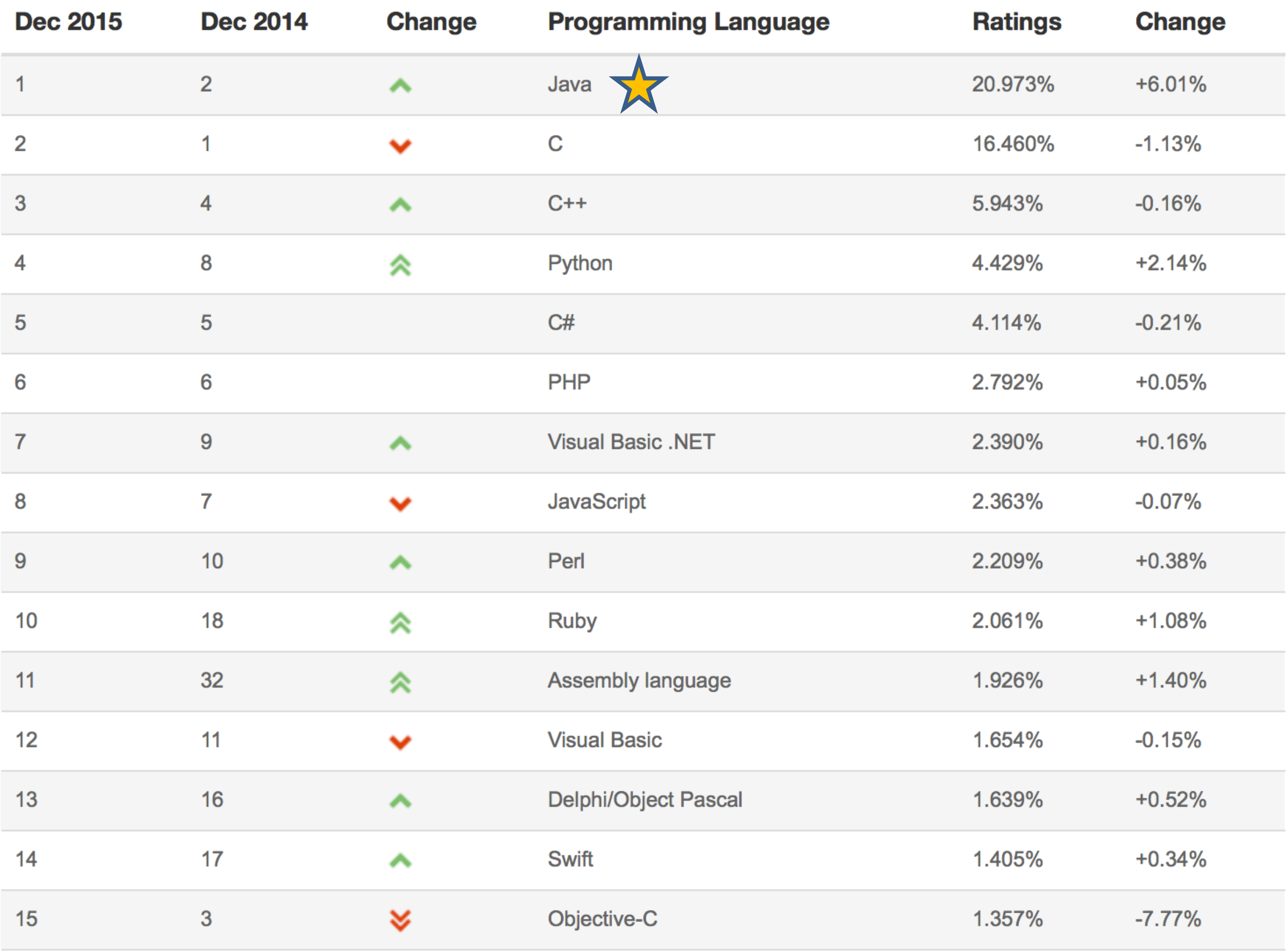
Garbage Collection (GC)

- In Java, there’s no need to free memory
  - Garbage collection runs periodically and frees up memory that’s not in use.
  - JVM attempts to do this without impacting performance.



http://www.ibm.com/developerworks/library/j-jtp10283/

Why Java?



Dec 2015	Dec 2014	Change	Programming Language	Ratings	Change
1	2	▲	Java ★	20.973%	+6.01%
2	1	▼	C	16.460%	-1.13%
3	4	▲	C++	5.943%	-0.16%
4	8	▲▲	Python	4.429%	+2.14%
5	5		C#	4.114%	-0.21%
6	6		PHP	2.792%	+0.05%
7	9	▲	Visual Basic .NET	2.390%	+0.16%
8	7	▼	JavaScript	2.363%	-0.07%
9	10	▲	Perl	2.209%	+0.38%
10	18	▲▲	Ruby	2.061%	+1.08%
11	32	▲▲	Assembly language	1.926%	+1.40%
12	11	▼	Visual Basic	1.654%	-0.15%
13	16	▲	Delphi/Object Pascal	1.639%	+0.52%
14	17	▲	Swift	1.405%	+0.34%
15	3	▼▼	Objective-C	1.357%	-7.77%

Installing the Platform

- There are two main Java implementations
  - Oracle Java: https://docs.oracle.com/javase/8/
  - Open JDK: FOSS implementation for Linux.

- JRE: standalone JVM installation (runtime).
- JDK: JRE plus development tools and libraries.
  - This gives you command-line tools (`javac` compiler and `java` runtime).
- Third-party support is excellent
  - Editor support in VIM, Sublime, etc.
  - IDEs like IntelliJ, Eclipse.

Recommended Resources

- Required
  - Java SE 8 JDK : http://www.oracle.com/technetwork/java/javase/
- Reference
  - Java 8 SE Platform SDK Documentation: https://docs.oracle.com/javase/8/docs/api/overview-summary.html
  - Java 8 Tutorials: http://docs.oracle.com/javase/tutorial/java/index.html
- IDE
  - IntelliJ (Jetbrains Student Licenses): https://www.jetbrains.com/student/

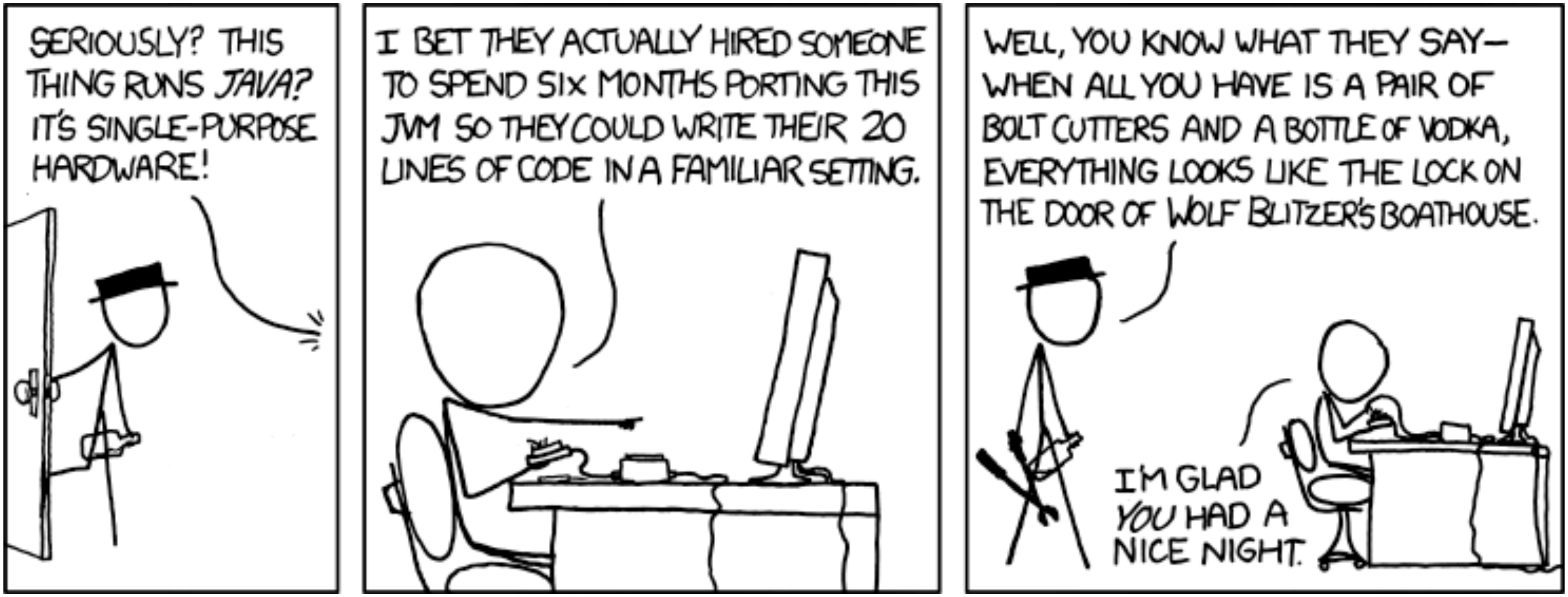
The Java Platform

Packages, libraries

Classes and objects

Program structure

xkcd


SERIOUSLY? THIS THING RUNS *JAVA?* IT'S SINGLE-PURPOSE HARDWARE!
I BET THEY ACTUALLY HIRED SOMEONE TO SPEND SIX MONTHS PORTING THIS JVM SO THEY COULD WRITE THEIR 20 LINES OF CODE IN A FAMILIAR SETTING.
WELL, YOU KNOW WHAT THEY SAY— WHEN ALL YOU HAVE IS A PAIR OF BOLT CUTTERS AND A BOTTLE OF VODKA, EVERYTHING LOOKS LIKE THE LOCK ON THE DOOR OF WOLF BLITZER'S BOATHOUSE.
I'M GLAD *YOU* HAD A NICE NIGHT.

http://xkcd.com/801/

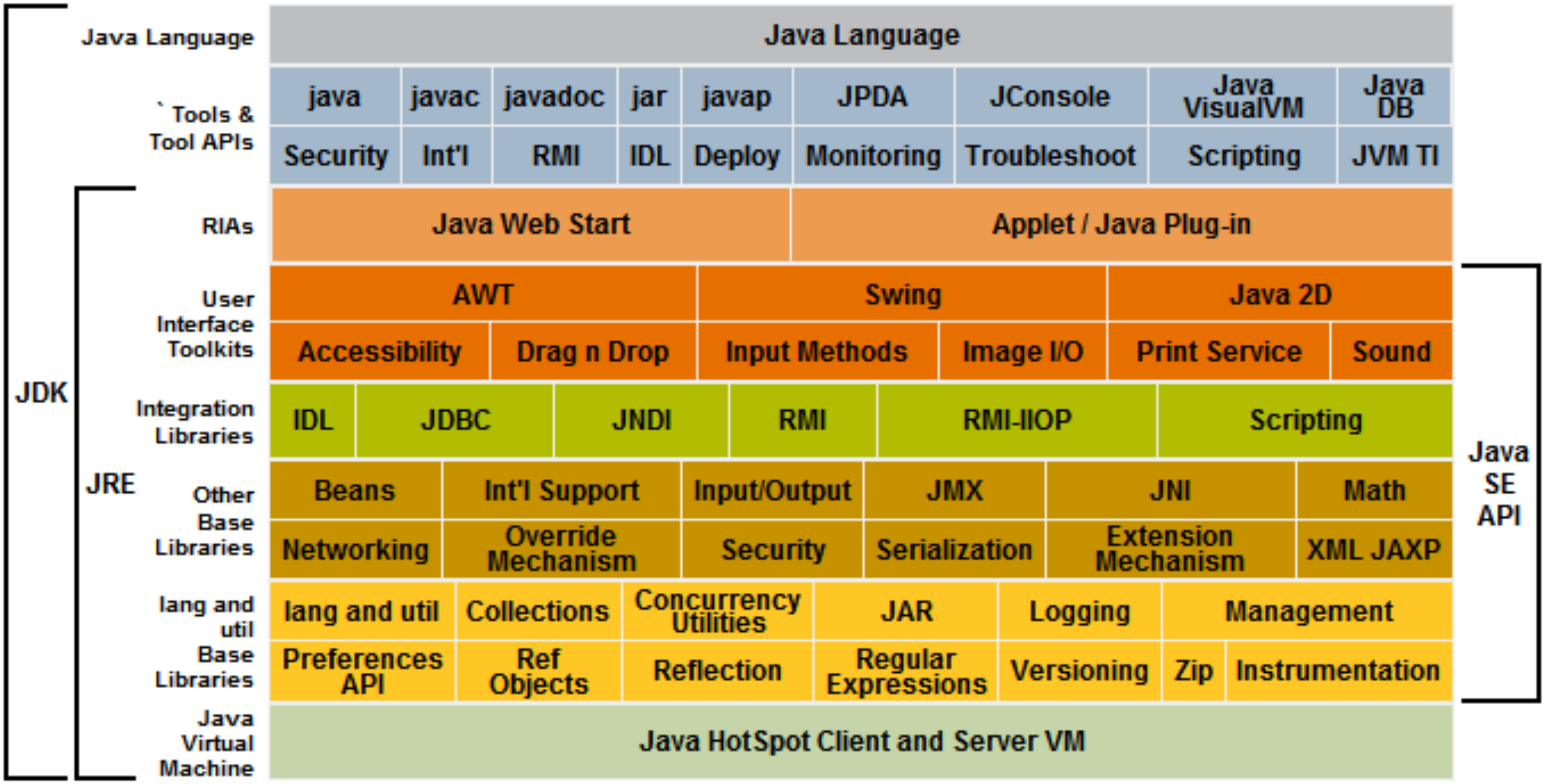
Everything is a class

- Classes, objects are core language constructs
- OO features: polymorphism, encapsulation, inheritance (more later)
- Static and member variables and methods
- Resembles C++ on the surface

- Language differences from C++
  - No pointers. Really. Not needed.
  - No type ambiguity; classes resolved at runtime
  - No destructor (due to garbage collector)
  - Single inheritance model

Java Platform (JDK)

Includes tools, and libraries - everything from threading, to database access, to UI toolkits – all cross platform and portable.



Java Language	Java Language								
` Tools & Tool APIs	java	javac	javadoc	jar	javap	JPDA	JConsole	Java VisualVM	Java DB
	Security	Int'l	RMI	IDL	Deploy	Monitoring	Troubleshoot	Scripting	JVM TI
RIAs	Java Web Start	Applet / Java Plug-in							
User Interface Toolkits	AWT	Swing	Java 2D						
	Accessibility	Drag n Drop	Input Methods	Image I/O	Print Service	Sound			
Integration Libraries	IDL	JDBC	JNDI	RMI	RMI-IIOP	Scripting			
Other Base Libraries	Beans	Int'l Support	Input/Output	JMX	JNI	Math			
	Networking	Override Mechanism	Security	Serialization	Extension Mechanism	XML JAXP			
lang and util Base Libraries	lang and util	Collections	Concurrency Utilities	JAR	Logging	Management			
	Preferences API	Ref Objects	Reflection	Regular Expressions	Versioning	Zip	Instrumentation		
Java Virtual Machine	Java HotSpot Client and Server VM								

Description of Java Conceptual Diagram

Java Class Library

- Classes are grouped into packages (i.e. namespaces) to avoid name collisions.
- To assign your source code to a package, use the ***package*** keyword at the top of source files.
- Typically, package = subdirectory
  - e.g. “graphics” package is in subdirectory of the same name
  - alt. it can be included in a JAR file.
- Use the ***import*** keyword to include a class from a different package in your code.
  - This is how you include bundled Java libraries.

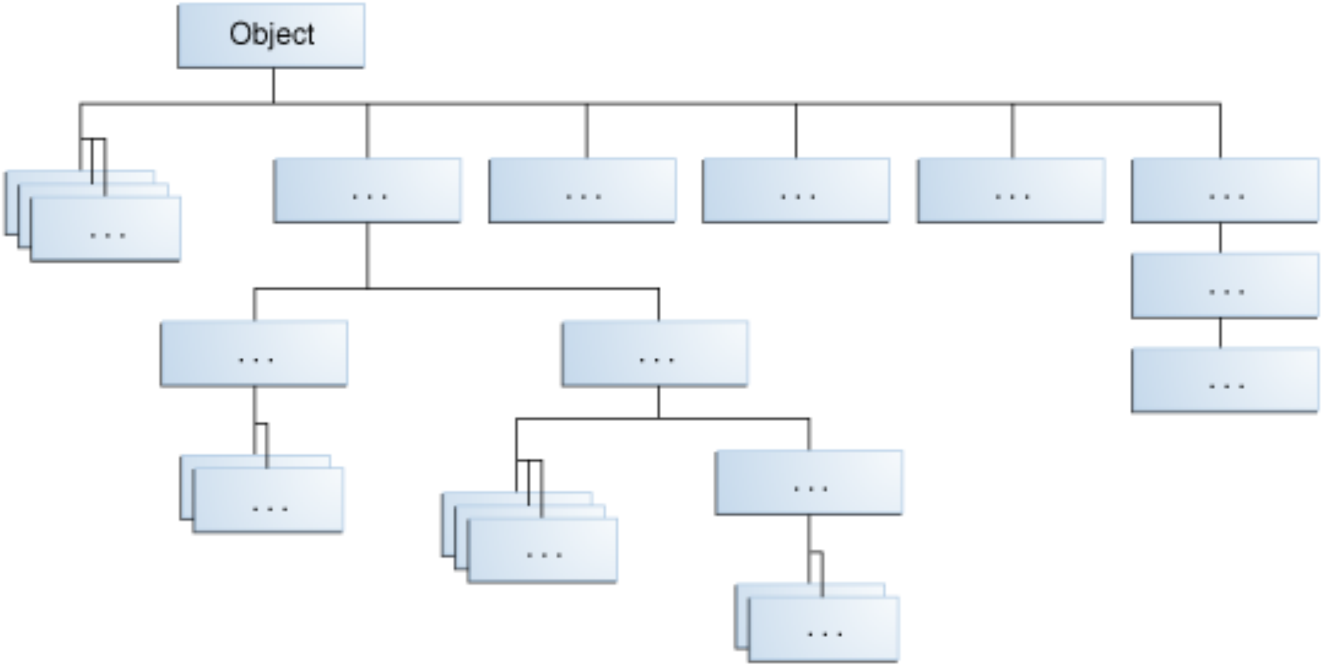
Common Classes/Packages

Package	Classes (Examples)	Description
java.awt	Color, Graphics, Graphics2D, event.	Contains all of the classes for creating user interfaces and for painting graphics and images.
javax.swing	JFrame, JButton, JList, JToolbar	Provides a set of "lightweight" (all-Java language) components that works the same on all platforms.
java.io	File, FileReader, FileWriter, InputStream	Provides for system input and output through data streams, serialization and the file system.
java.lang	Boolean, Integer, String, System, Thread, Math	Provides classes that are fundamental to the design of the Java programming language.
java.util	ArrayList, HashMap, Observable	Contains the collections framework, legacy collection classes, event model,…

Java 8 API Specification: https://docs.oracle.com/javase/8/docs/api/overview-summary.html

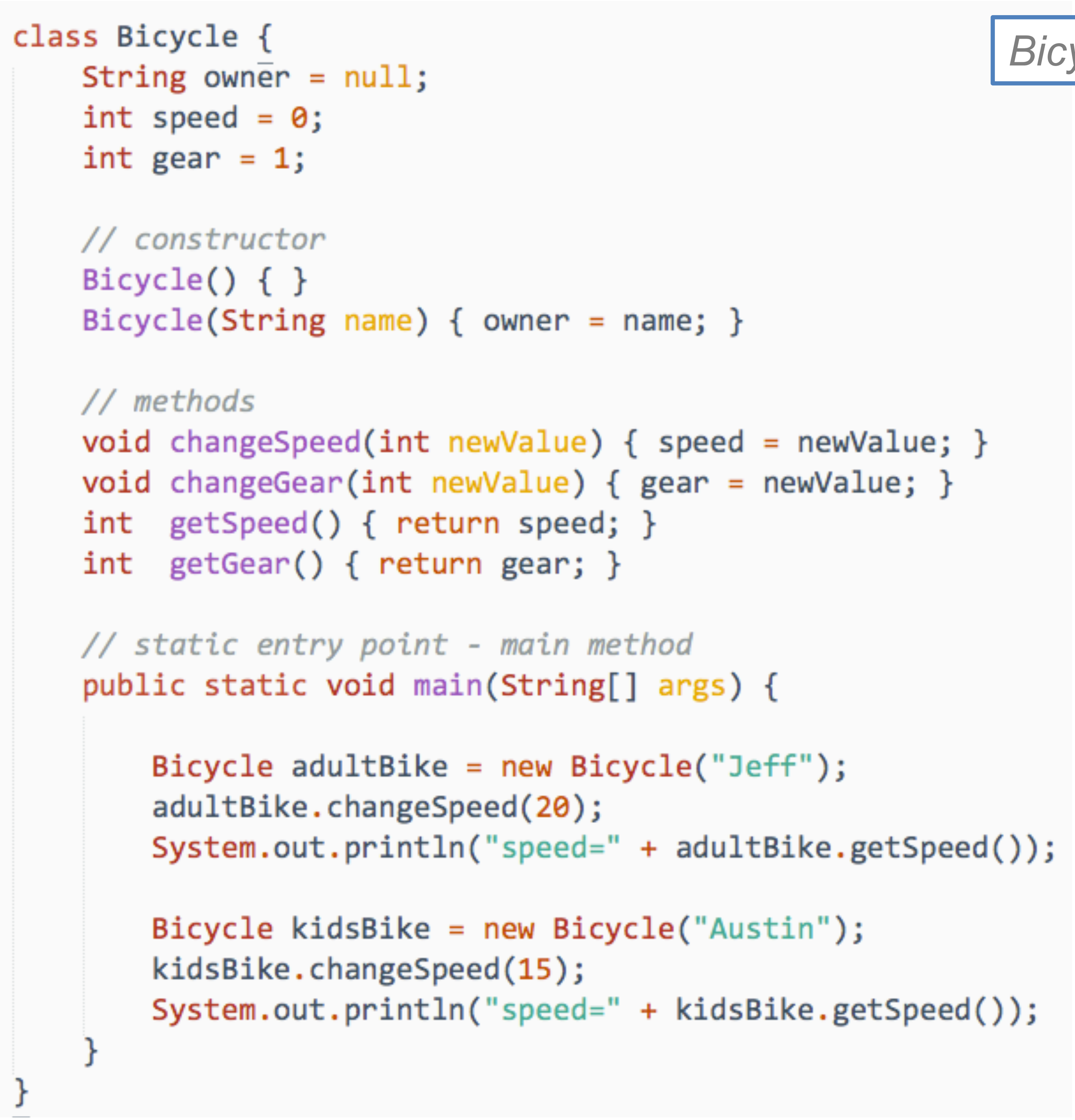
Java Class Hierarchy

- Implicit class hierarchy
  - All classes in Java are derived from the `Object` class in `java.lang` defines and implements common class behavior
    - e.g. `clone()`, `toString()`, `finalize()` methods
  - Classes you write inherit this basic behavior.



Structure of a Program

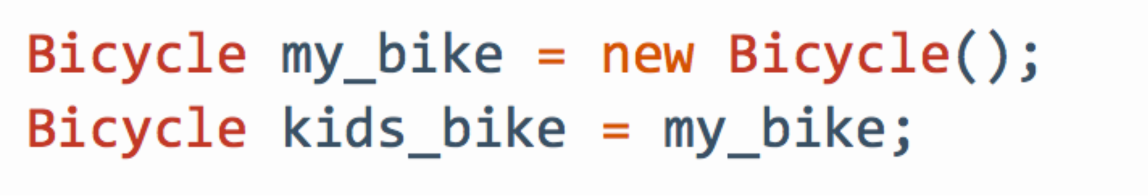
Bicycle.java



```java
class Bicycle {
    String owner = null;
    int speed = 0;
    int gear = 1;

    // constructor
    Bicycle() { }
    Bicycle(String name) { owner = name; }

    // methods
    void changeSpeed(int newValue) { speed = newValue; }
    void changeGear(int newValue) { gear = newValue; }
    int  getSpeed() { return speed; }
    int  getGear() { return gear; }

    // static entry point - main method
    public static void main(String[] args) {

        Bicycle adultBike = new Bicycle("Jeff");
        adultBike.changeSpeed(20);
        System.out.println("speed=" + adultBike.getSpeed());

        Bicycle kidsBike = new Bicycle("Austin");
        kidsBike.changeSpeed(15);
        System.out.println("speed=" + kidsBike.getSpeed());
    }
}
```

Instantiating Objects

- In Java,
  - Primitive types are allocated on the stack, passed by value.
  - Objects are allocated on the heap, passed by reference
    - Technically, value of address passed on the stack, but behaves like pass-by-reference.



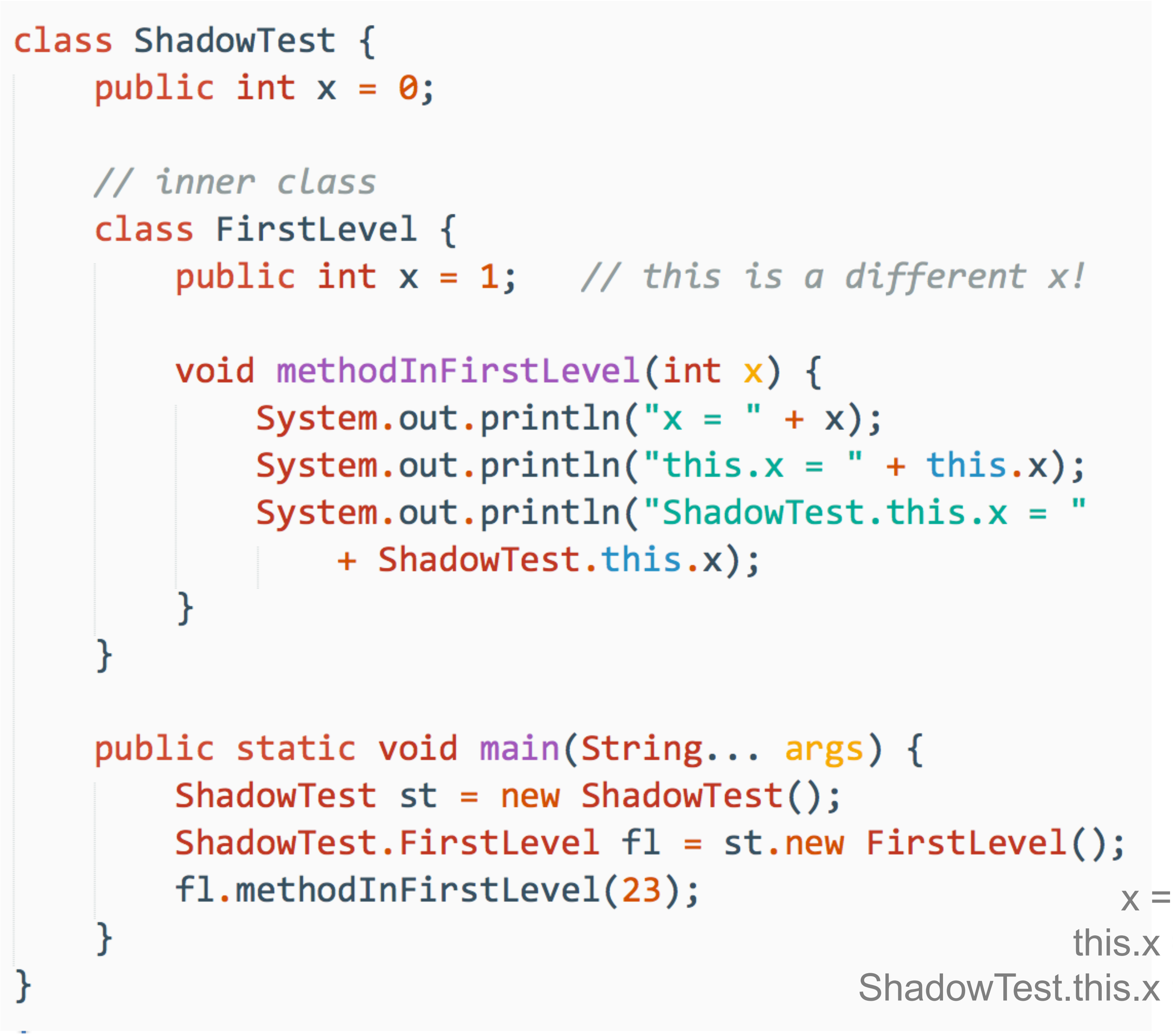
```
Bicycle my_bike = new Bicycle();
Bicycle kids_bike = my_bike;
```

Both refer to the same memory on the heap

- Practically, this means that you don’t need to worry about pointer semantics in parameter passing.

Composition: Inner Classes

Classes can be nested as inner classes. Watch scope!

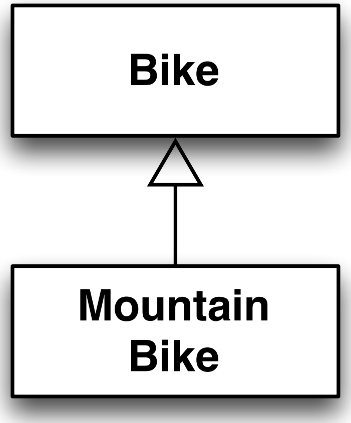


```
class ShadowTest {
    public int x = 0;

    // inner class
    class FirstLevel {
        public int x = 1;   // this is a different x!

        void methodInFirstLevel(int x) {
            System.out.println("x = " + x);
            System.out.println("this.x = " + this.x);
            System.out.println("ShadowTest.this.x = "
                + ShadowTest.this.x);
        }
    }

    public static void main(String... args) {
        ShadowTest st = new ShadowTest();
        ShadowTest.FirstLevel fl = st.new FirstLevel();
        fl.methodInFirstLevel(23);
    }
}
```

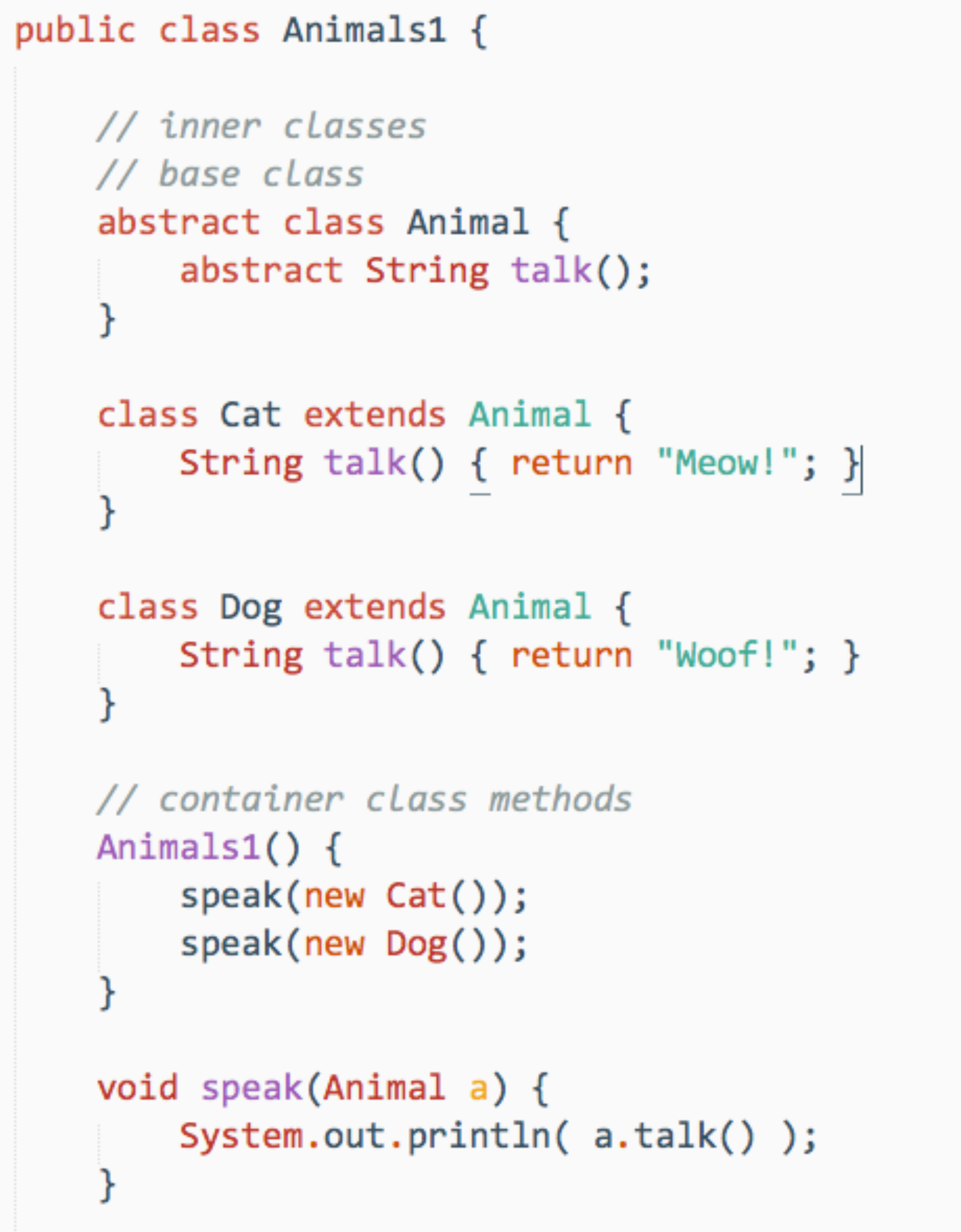
```
x = 23
this.x = 1
ShadowTest.this.x = 0
```

Inheritance

- Inheritance: increasing code reusability by allowing a class to inherit some of it's behavior from a base class ("is a" relationship).
  - Classes inherit common attributes and behavior from base classes.
  - e.g. "Mountain Bike" is-a "Bike".
    - Common: speed, gear
    - Unique: engine
  - *In Java, use* ***extends*** keyword.



Subtype Polymorphism



```java
public class Animals1 {

    // inner classes
    // base class
    abstract class Animal {
        abstract String talk();
    }

    class Cat extends Animal {
        String talk() { return "Meow!"; }
    }

    class Dog extends Animal {
        String talk() { return "Woof!"; }
    }

    // container class methods
    Animals1() {
        speak(new Cat());
        speak(new Dog());
    }

    void speak(Animal a) {
        System.out.println( a.talk() );
    }
```

Animals1.java

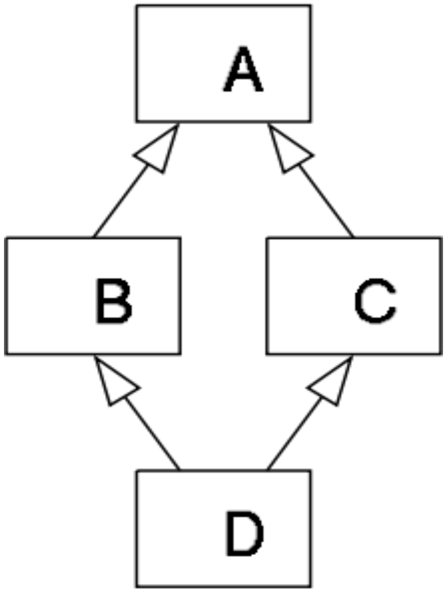
The *Animal* class is abstract, and cannot be instantiated.

It’s *talk()* method is abstract, so derived classes must override it.

```
” Meow! “
“ Woof! “
```

Single Inheritance

- Java only supports single inheritance!
  - In practice, this simplifies the language.
  - See the “Diamond Problem” for one example of multiple inheritance being a hard problem.
  - Solutions to this problem vary by language; Java prevents you from doing it.



- It’s *very* common in Java to derive an existing class and override behavior (even provided classes).
- All classes have Object as their ultimate base class (implicit).

Interfaces

- An interface represents a set of methods that must be implemented by a class (“contract”).

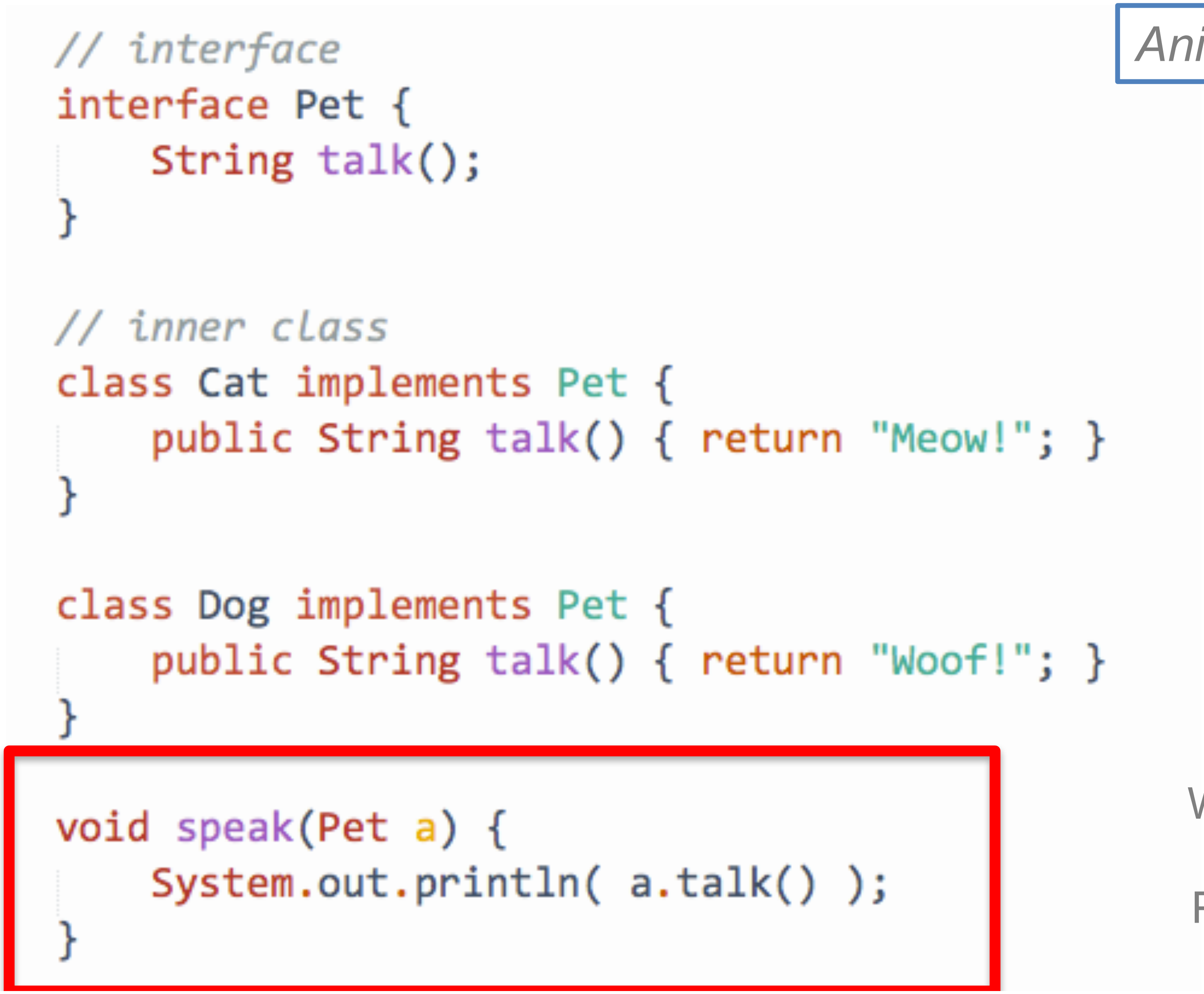
- Similar to pure abstract classes/methods.
  - Can’t be instantiated
  - Class implementing the interface must implement all methods in the interface.
  - *Use **implements*** keyword.

- In Java,
  - *extend* a class to derive functionality from it
  - *implement* an interface when you want to enforce a specific API.

Interfaces Example

- We can replace the abstract class/method with an interface.
  - Polymorphism still applies to interfaces.

Animals2.java



```
// interface
interface Pet {
    String talk();
}

// inner class
class Cat implements Pet {
    public String talk() { return "Meow!"; }
}

class Dog implements Pet {
    public String talk() { return "Woof!"; }
}

void speak(Pet a) {
    System.out.println( a.talk() );
}
```

We're treating the interface, Pet, as a type

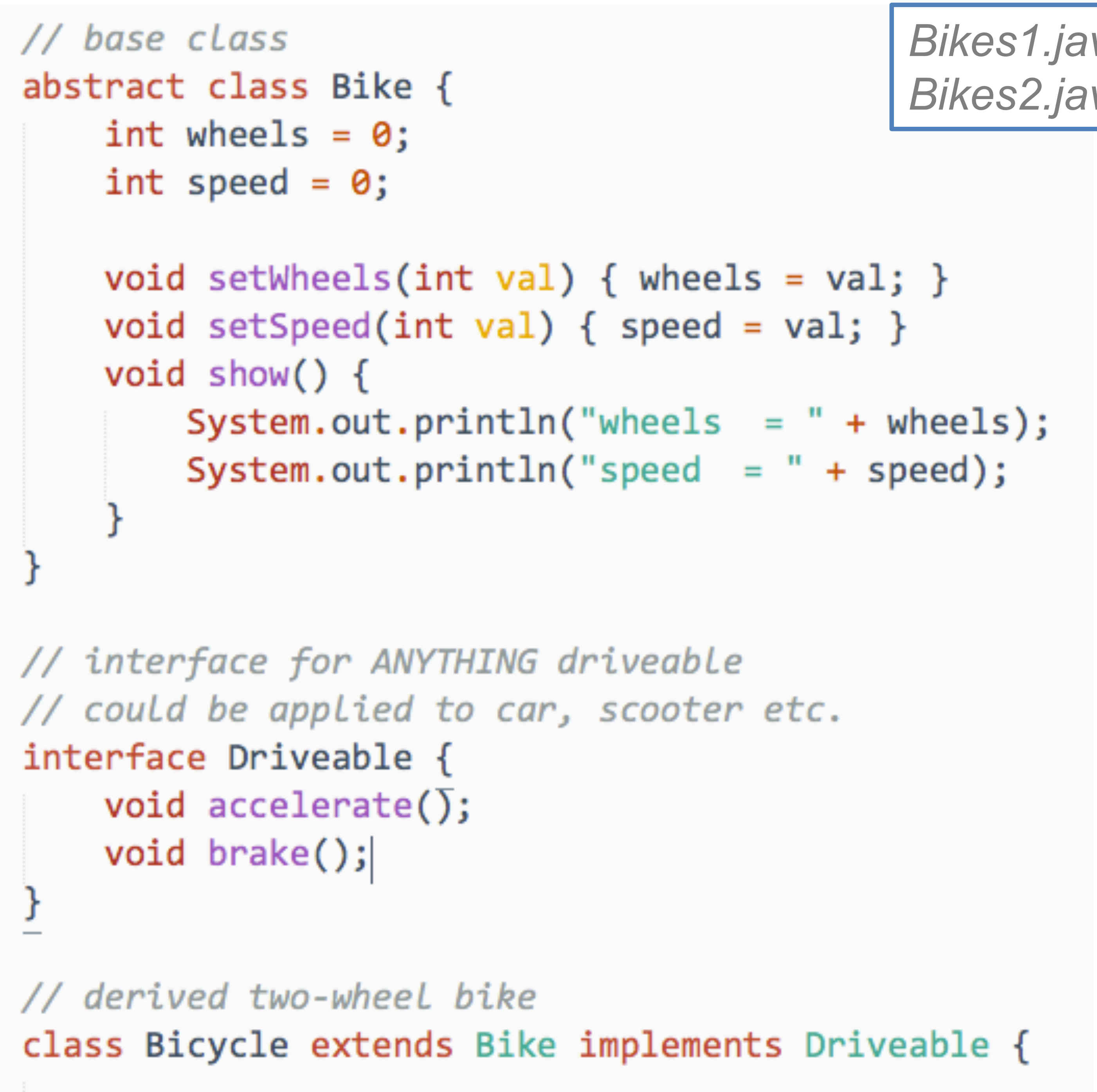
Interfaces Example

- You can (and will often) mix approaches.
  - e.g. Drivable interface could be applied to any type of drivable vehicle, including our Bike class hierarchy.

```
interface Driveable {
  public void accelerate();
  public void brake();
  public int getSpeed();
}

// implement interface only
class Car implements Driveable {}
class Train implements Driveable {}

// derive from base class, and implement interface
class Motorcycle extends Bike implements Driveable {}
```

Extended Example

Bikes1.java – classes
Bikes2.java - interfaces



```
// base class
abstract class Bike {
    int wheels = 0;
    int speed = 0;

    void setWheels(int val) { wheels = val; }
    void setSpeed(int val) { speed = val; }
    void show() {
        System.out.println("wheels  = " + wheels);
        System.out.println("speed  = " + speed);
    }
}

// interface for ANYTHING driveable
// could be applied to car, scooter etc.
interface Driveable {
    void accelerate();
    void brake();
}

// derived two-wheel bike
class Bicycle extends Bike implements Driveable {
```

Building User Interfaces

Swing components

Creating a window

Adding Swing components

Listening for events

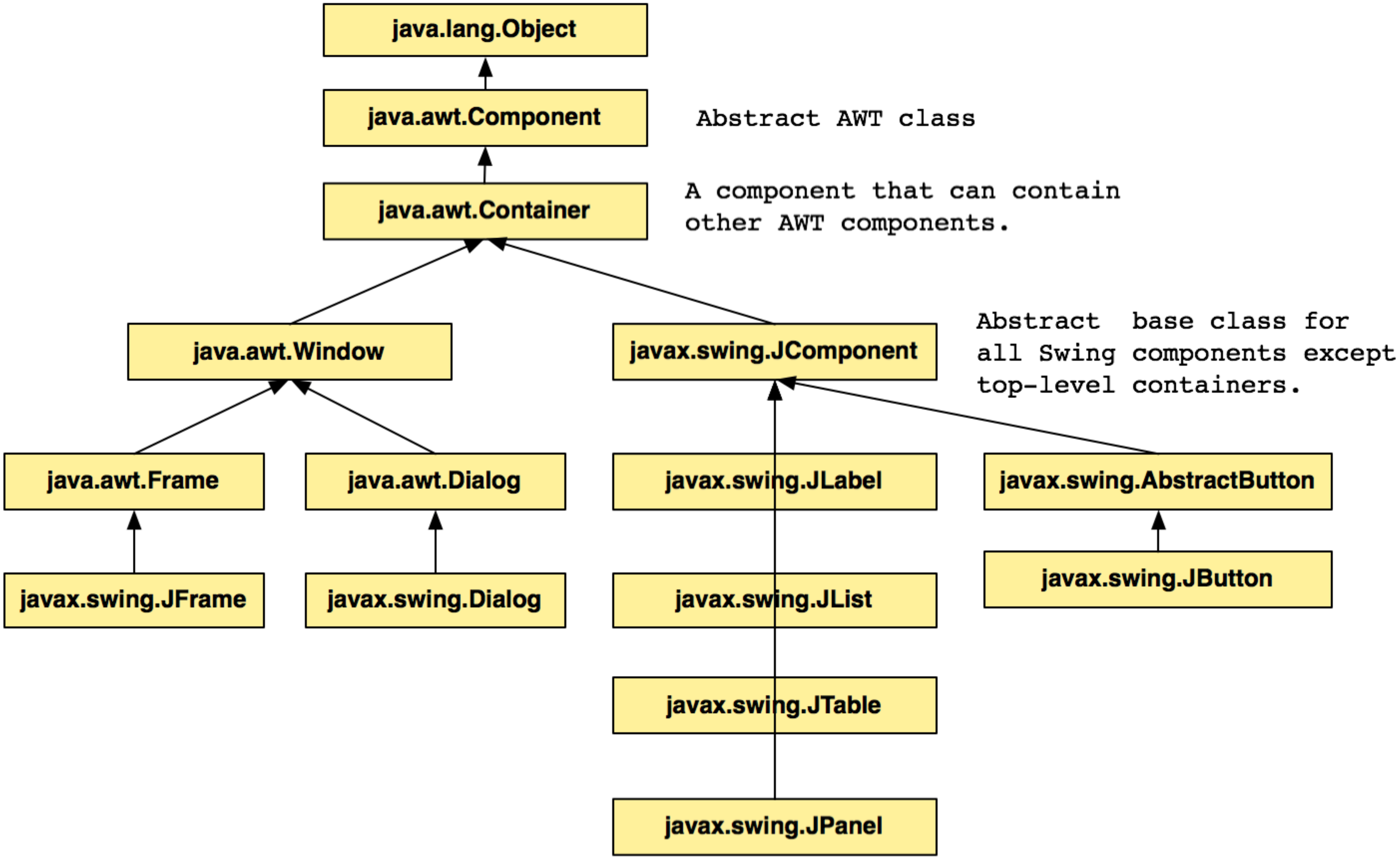
PaintDemo

Java UI Toolkits

Java has four user-interface libraries, each with different types of widgets (and strengths/tradeoffs)..

Toolkit (Release)	Description
AWT (1995)	“Heavyweight” with platform-specific widgets. AWT applications were limited to common-functionality that existed on all platforms.
Swing (1997)	“Lightweight”, full widget implementation. Commonly used and deployed cross-platform.
Standard Window Toolkit / SWT (~2000)	”Heavyweight” hybrid model: native, and tied to specific platforms. Used in Eclipse.
Java FX (~2010)	Intended for rich desktop + mobile apps. Still in development.

Swing Component Hierarchy



Abstract AWT class

A component that can contain other AWT components.

Abstract base class for all Swing components except top-level containers.

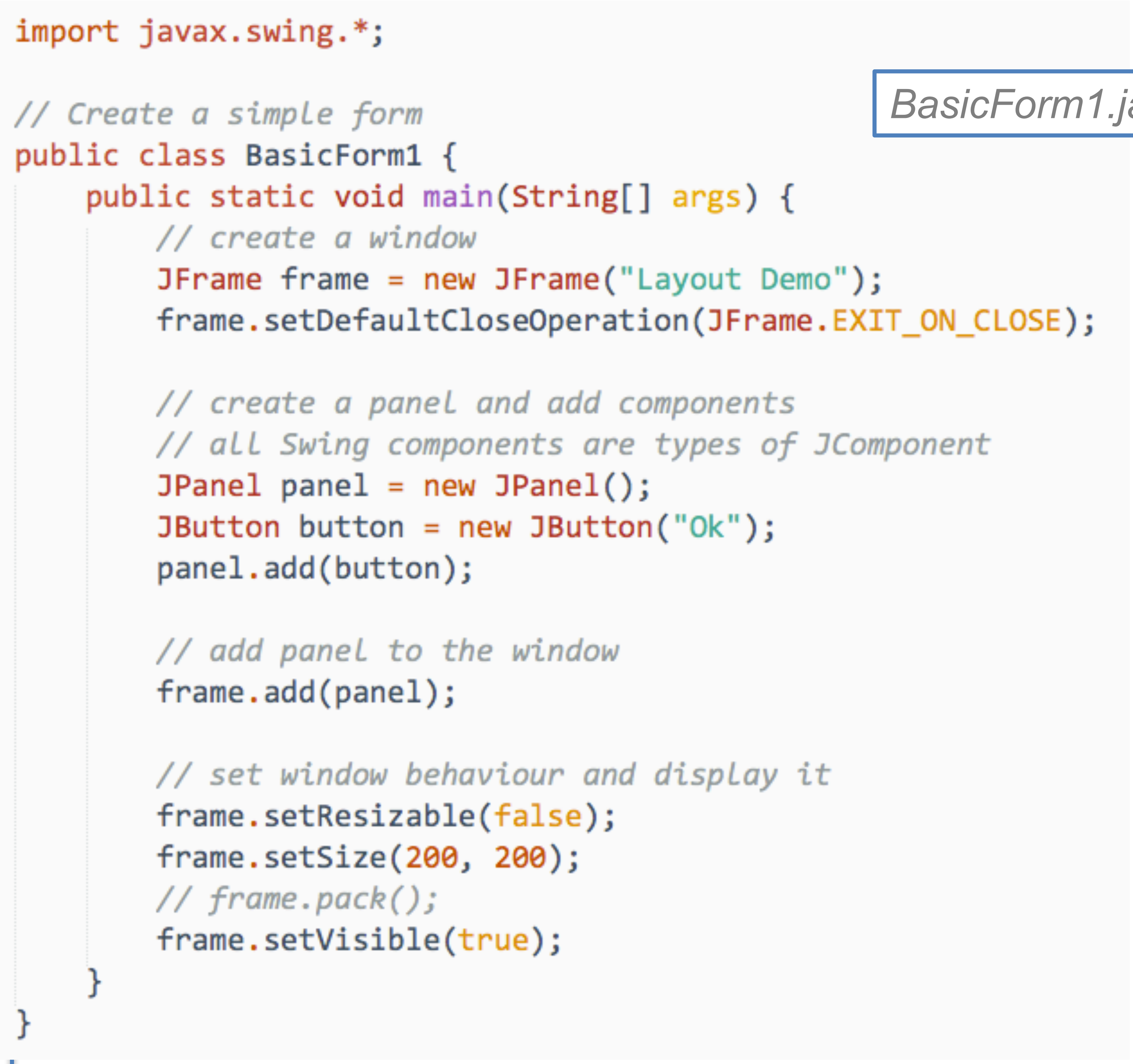
- `java.awt.Window` is the base for all containers.
- `javax.swing.Jcomponent` is the root for all widgets.

How to build a Swing UI

1. Create a top-level application window, using a Swing container (`JFrame` or `JDialog`).
2. Add Swing components to this window.
   - Typically, you create a smaller container (like a `JPanel`) and add components to the panel.
   - This makes dynamic layouts easier (more on that later in the course!)
3. Register for events: add listeners, like keyboard (press), mouse (down, up, move)
4. Write code to respond to these events.
5. Make components update and paint themselves based on events.

Creating a Window

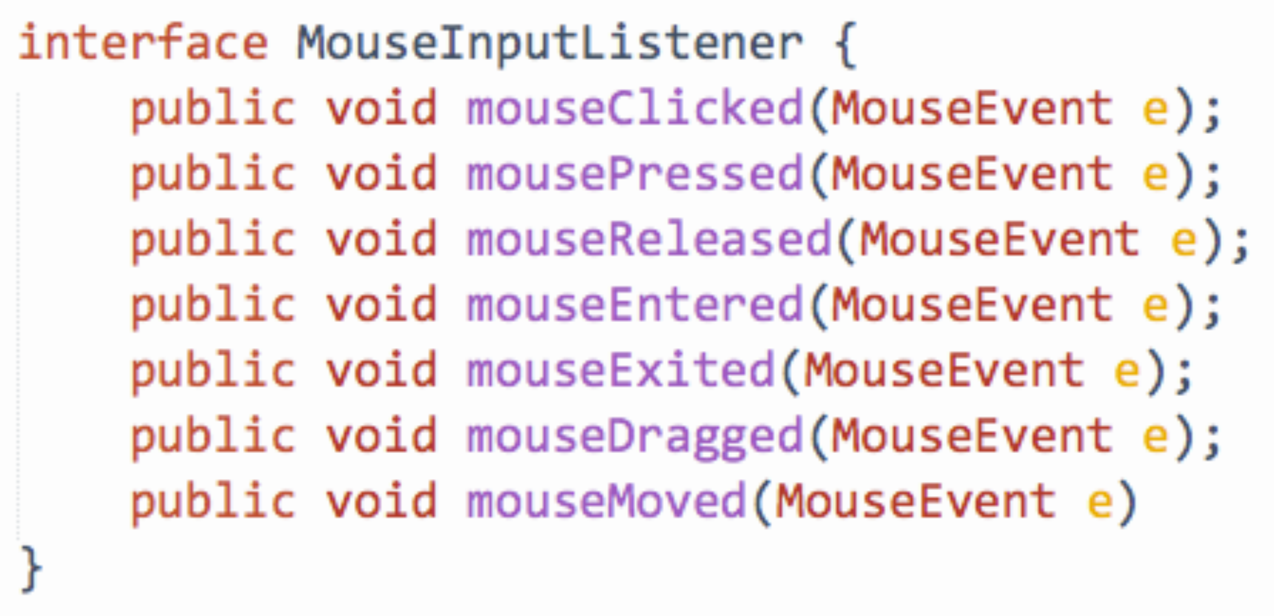
BasicForm1.java



```java
import javax.swing.*;

// Create a simple form
public class BasicForm1 {
    public static void main(String[] args) {
        // create a window
        JFrame frame = new JFrame("Layout Demo");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        // create a panel and add components
        // all Swing components are types of JComponent
        JPanel panel = new JPanel();
        JButton button = new JButton("Ok");
        panel.add(button);

        // add panel to the window
        frame.add(panel);

        // set window behaviour and display it
        frame.setResizable(false);
        frame.setSize(200, 200);
        // frame.pack();
        frame.setVisible(true);
    }
}
```

Java Listener Model

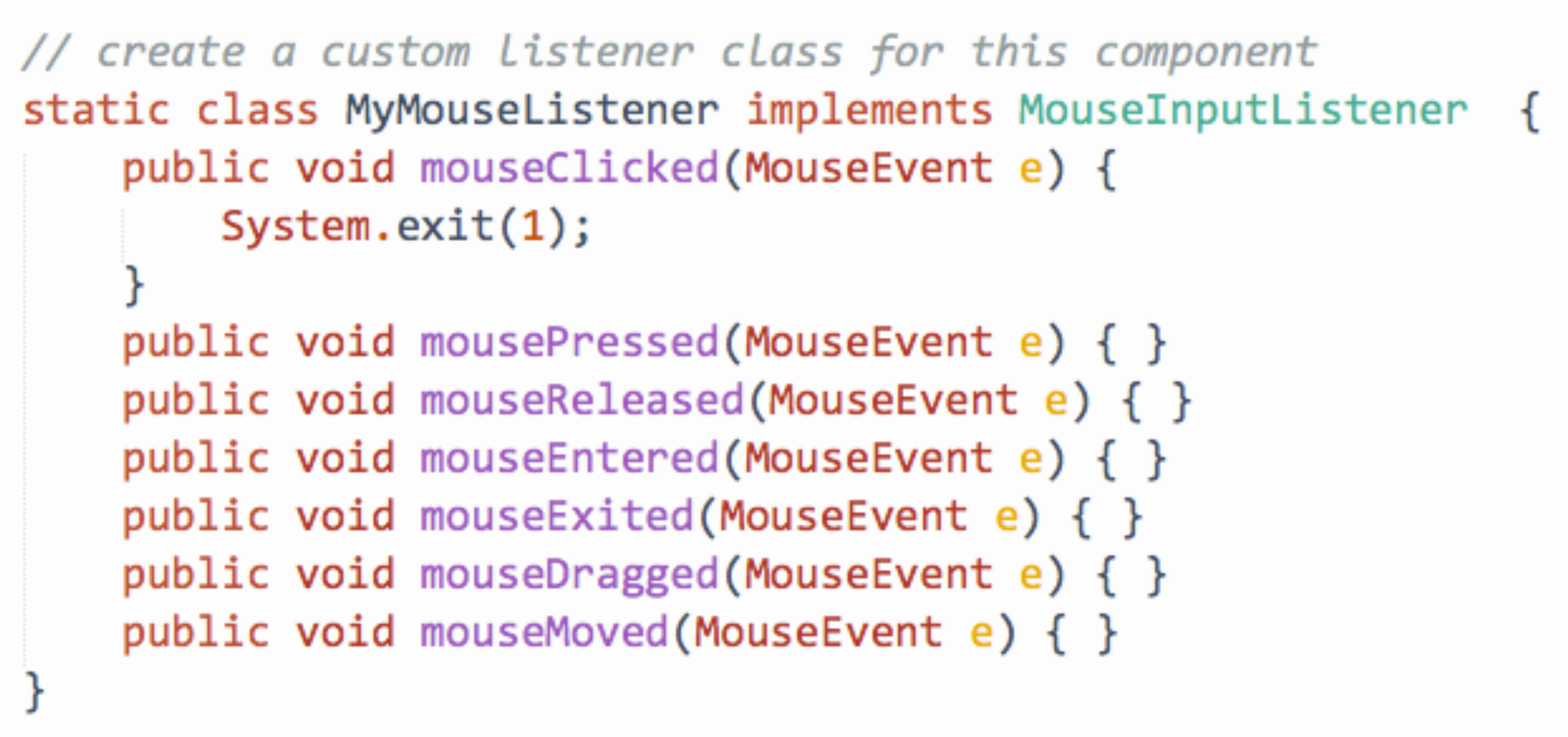
- Java has interfaces specialized by event type.
  - Each interface lists the methods that are needed to support that device's events
- To use them, write a class that implements this interface, and override the methods for events you care about.



```
interface MouseInputListener {
    public void mouseClicked(MouseEvent e);
    public void mousePressed(MouseEvent e);
    public void mouseReleased(MouseEvent e);
    public void mouseEntered(MouseEvent e);
    public void mouseExited(MouseEvent e);
    public void mouseDragged(MouseEvent e);
    public void mouseMoved(MouseEvent e)
}
```

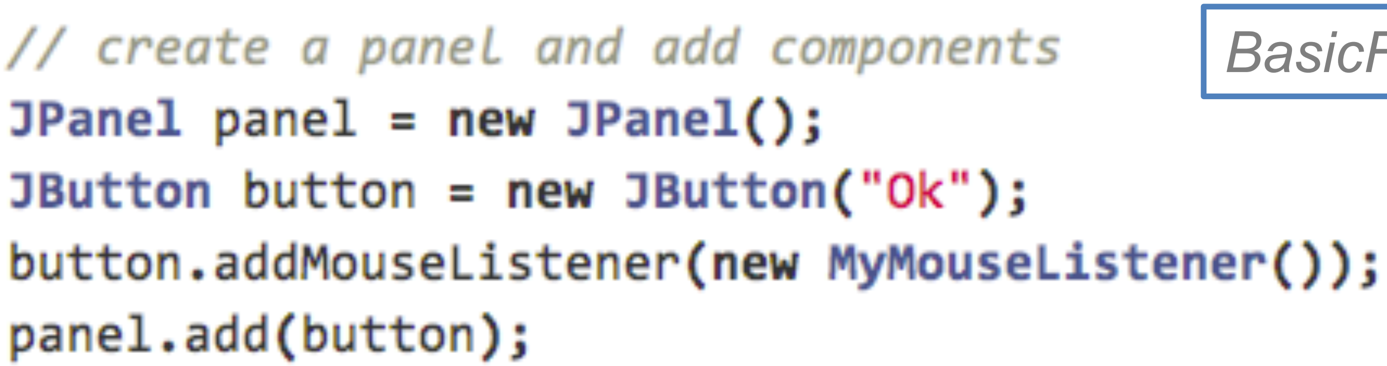
- Because it's an interface, you have to override all of these methods – even for events you don't care about!

Using Listeners



```java
// create a custom listener class for this component
static class MyMouseListener implements MouseInputListener  {
    public void mouseClicked(MouseEvent e) {
        System.exit(1);
    }
    public void mousePressed(MouseEvent e) { }
    public void mouseReleased(MouseEvent e) { }
    public void mouseEntered(MouseEvent e) { }
    public void mouseExited(MouseEvent e) { }
    public void mouseDragged(MouseEvent e) { }
    public void mouseMoved(MouseEvent e) { }
}
```

BasicForm2.java

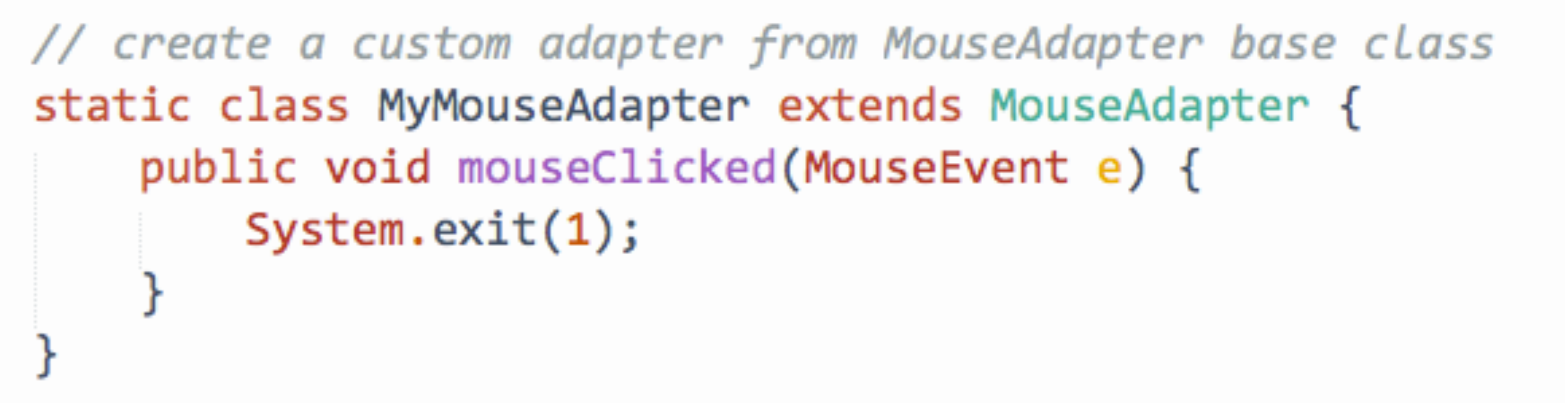


```java
// create a panel and add components
JPanel panel = new JPanel();
JButton button = new JButton("Ok");
button.addMouseListener(new MyMouseListener());
panel.add(button);
```

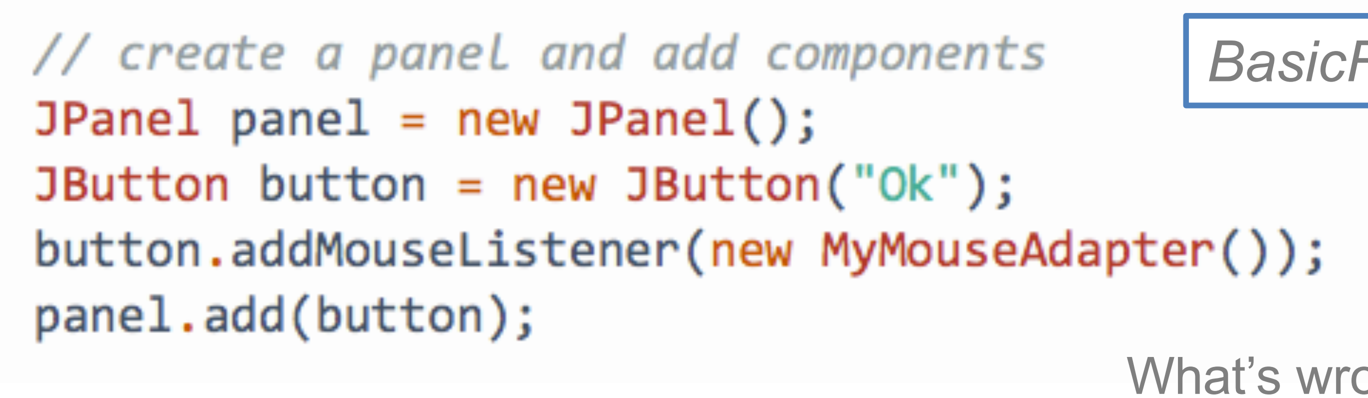
What's wrong with this approach?

Adapters vs. Listeners

- Java also has adapters, which are base classes with empty listeners.
  - Extend the adapter and override the event handlers that you care about; avoids bloat.



```java
// create a custom adapter from MouseAdapter base class
static class MyMouseAdapter extends MouseAdapter {
    public void mouseClicked(MouseEvent e) {
        System.exit(1);
    }
}
```



```java
// create a panel and add components
JPanel panel = new JPanel();
JButton button = new JButton("Ok");
button.addMouseListener(new MyMouseAdapter());
panel.add(button);
```

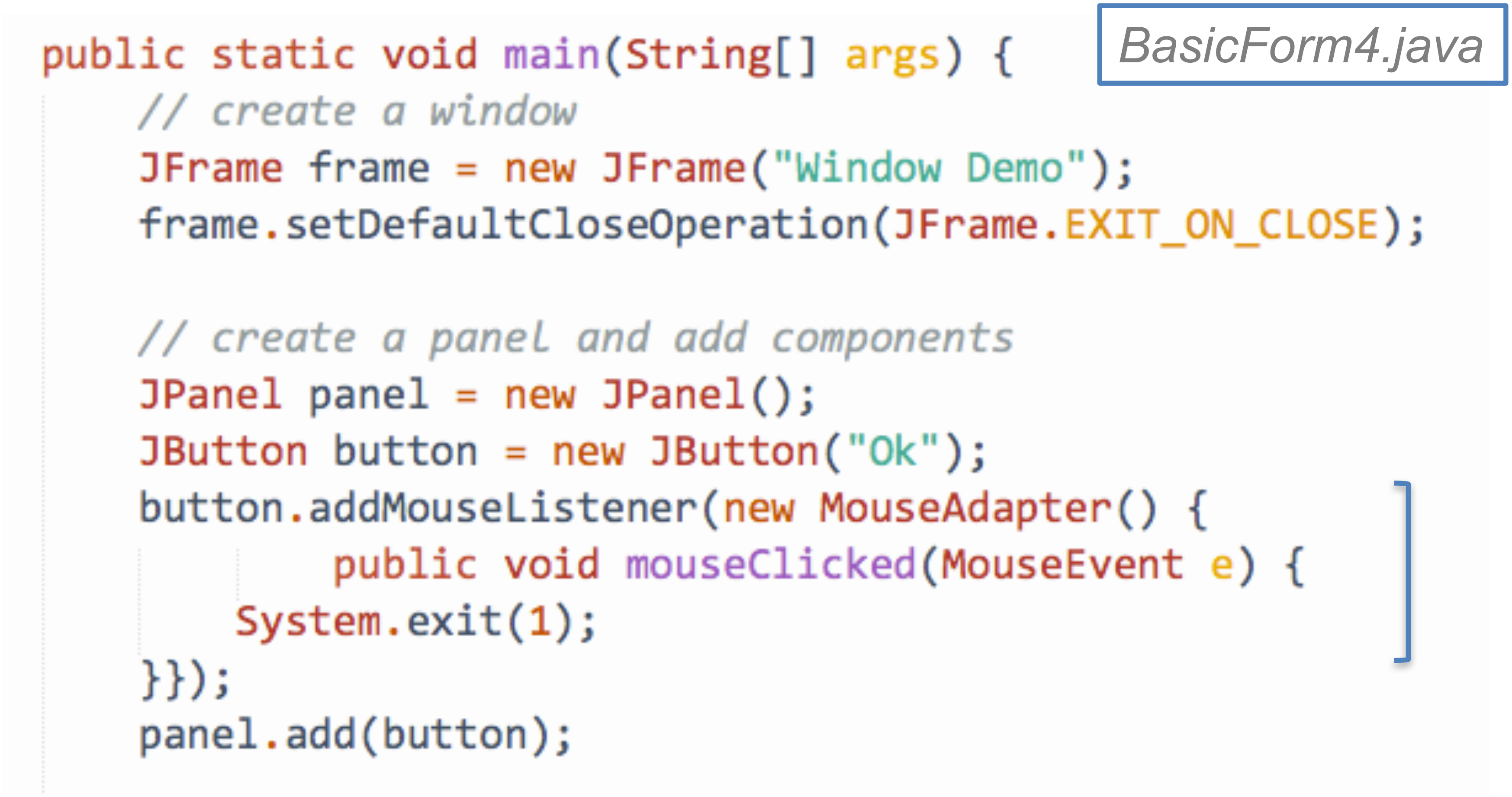
BasicForm3.java

What's wrong with this approach?

Anonymous Inner Classes

- We really, really don’t want to create custom adapters for every component.
  - Solution? Anonymous inner class.

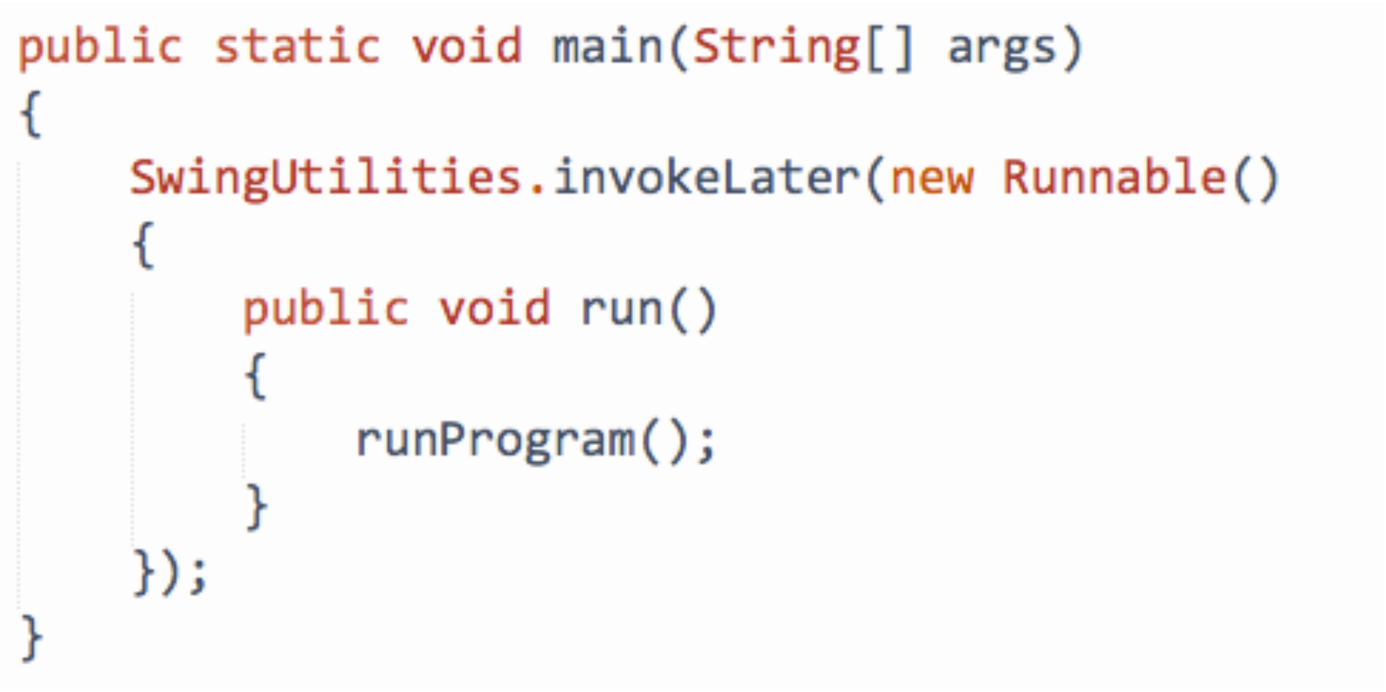
BasicForm4.java



```java
public static void main(String[] args) {
    // create a window
    JFrame frame = new JFrame("Window Demo");
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    // create a panel and add components
    JPanel panel = new JPanel();
    JButton button = new JButton("Ok");
    button.addMouseListener(new MouseAdapter() {
            public void mouseClicked(MouseEvent e) {
        System.exit(1);
    }});
    panel.add(button);
```

Swing UI Thread

- Swing needs to make sure that all events are handled on the Event Dispatch thread.
- If you just “run” your application from main, as we’ve been doing in the examples, you risk the main program accepting input before the UI is instantiated!
  - Use `invokeLater()` to safely create the UI.
  - We’ll discuss in greater detail later in the course.



```
public static void main(String[] args)
{
    SwingUtilities.invokeLater(new Runnable()
    {
        public void run()
        {
            runProgram();
        }
    });
}
```

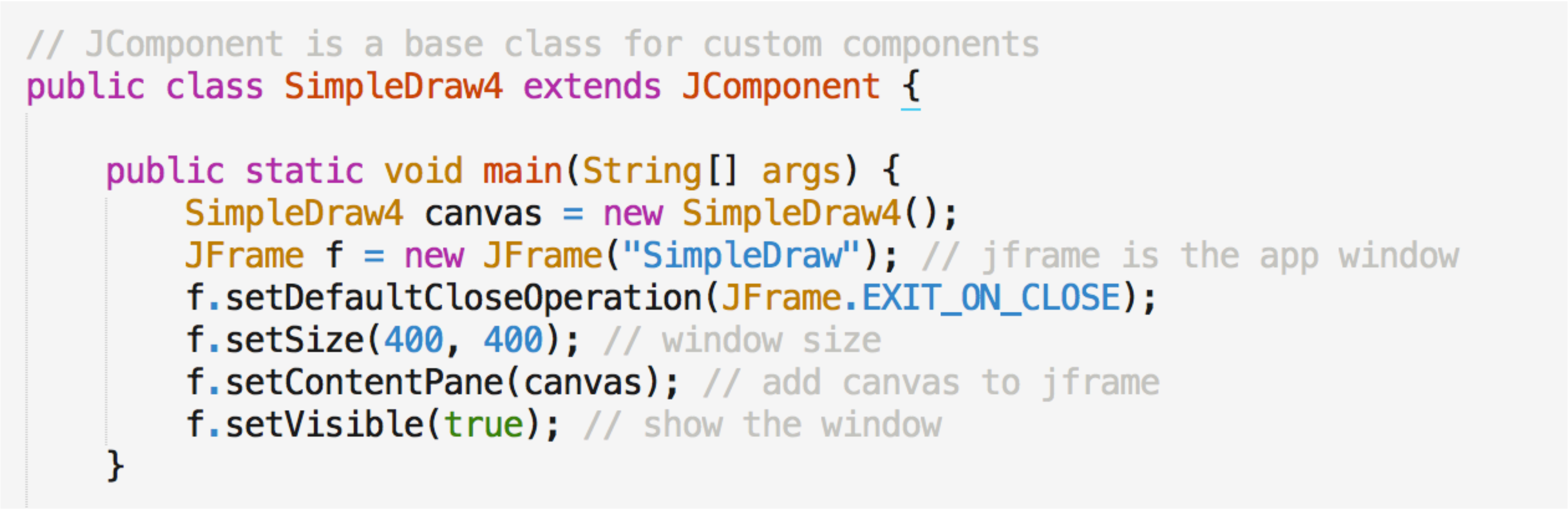
Drawing in Java

Overriding paintComponent()

Graphics object

Graphics and Painting

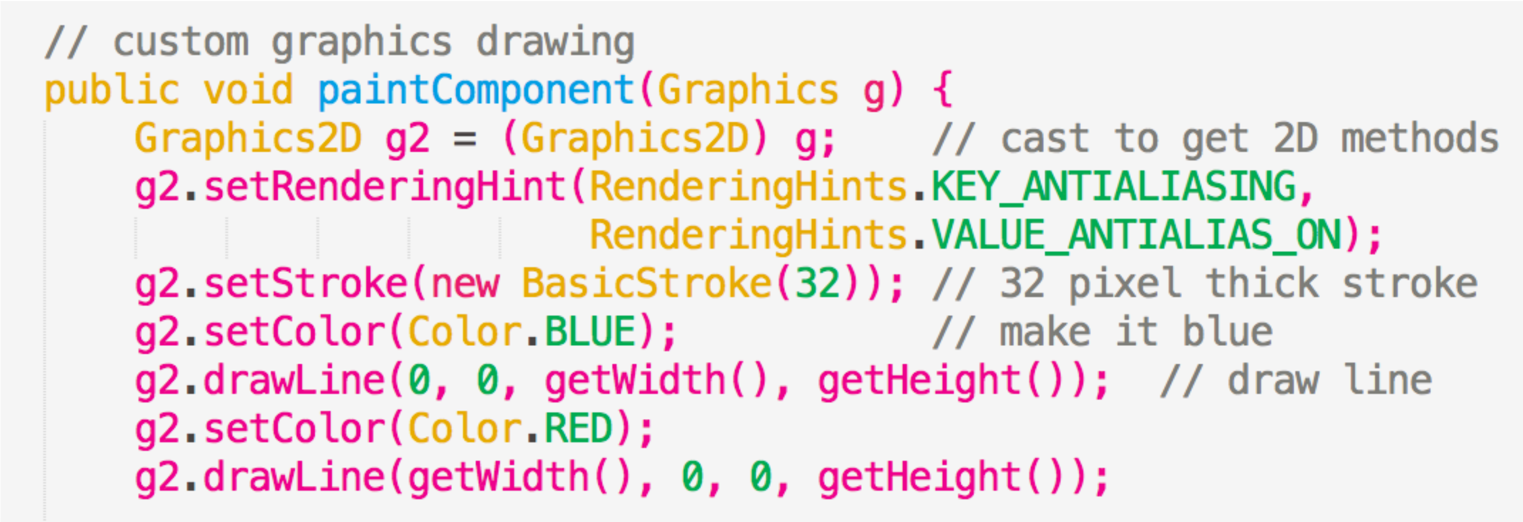
- Applications consist of a JFrame (window) containing one or more Swing components.
- We often define a top-level canvas (container)
  - This can hold other components (like text fields, buttons, scroll bars etc).
  - We can also draw directly on this canvas.



```
// JComponent is a base class for custom components
public class SimpleDraw4 extends JComponent {

    public static void main(String[] args) {
        SimpleDraw4 canvas = new SimpleDraw4();
        JFrame f = new JFrame("SimpleDraw"); // jframe is the app window
        f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        f.setSize(400, 400); // window size
        f.setContentPane(canvas); // add canvas to jframe
        f.setVisible(true); // show the window
    }
```

Graphics and Painting

- Each component has a `paintComponent()` method, which describes how it paints itself.
- You can override this `paintComponent()` method and draw primitive objects using the `java.awt.Graphics` object (basically, the Graphics Context).
- This is a common technique for defining *drawables* in Java.



```
// custom graphics drawing
public void paintComponent(Graphics g) {
    Graphics2D g2 = (Graphics2D) g;     // cast to get 2D methods
    g2.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
                        RenderingHints.VALUE_ANTIALIAS_ON);
    g2.setStroke(new BasicStroke(32)); // 32 pixel thick stroke
    g2.setColor(Color.BLUE);           // make it blue
    g2.drawLine(0, 0, getWidth(), getHeight());  // draw line
    g2.setColor(Color.RED);
    g2.drawLine(getWidth(), 0, 0, getHeight());
```

What’s left?

Topics that we’ll cover in later lectures

- Animation
- Advanced graphics
- Design patterns
- Features (undo-redo, copy-paste)