# A Performance Tale

## The evolution of binding in JavaFX

Robert Field
Brian Goetz

Sun Microsystems, Inc.

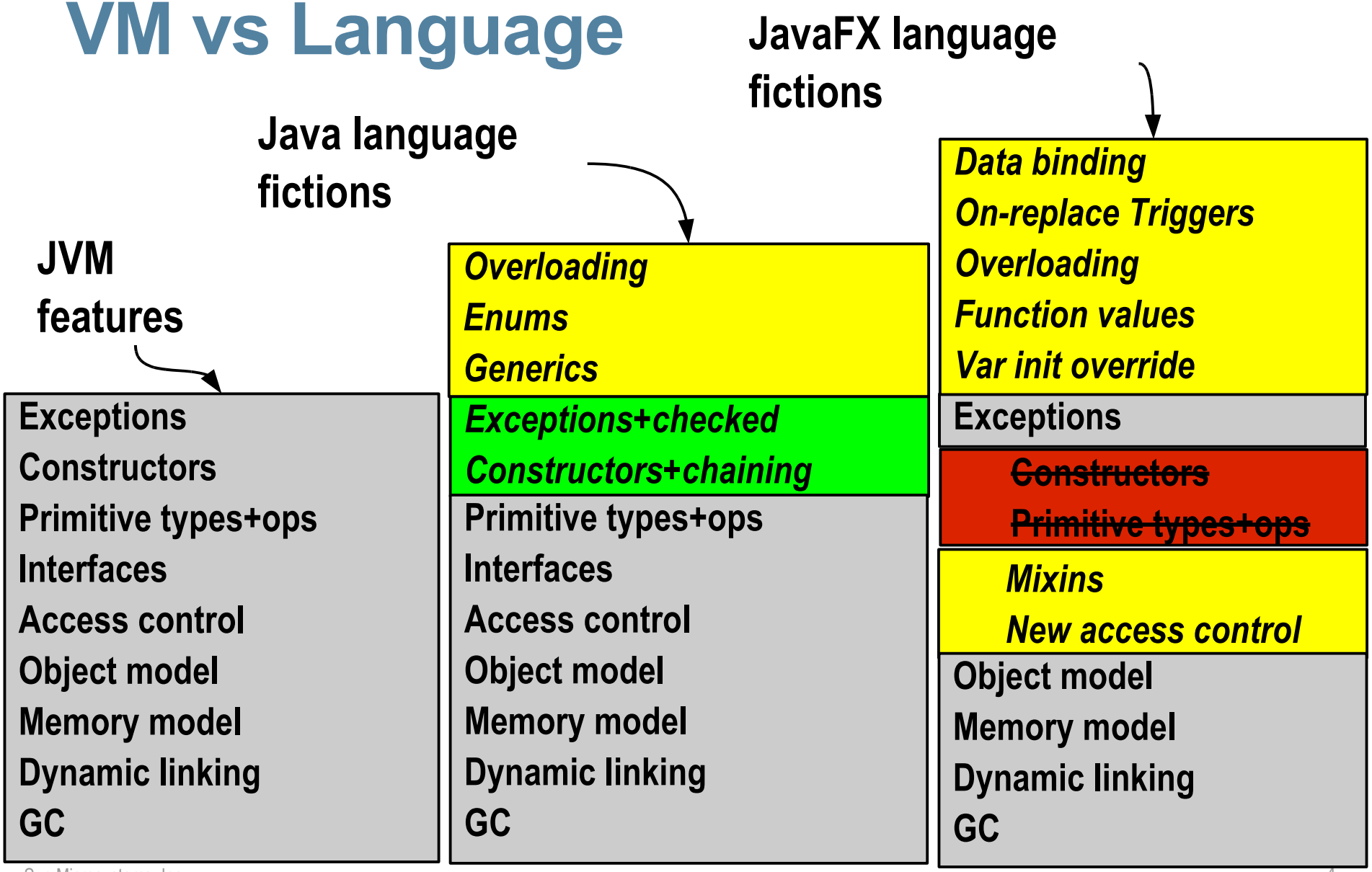
# Overview

- This talk will chronicle our ongoing work on making JavaFX Script not only powerful and fun but fast
  - We'll focus specifically on one feature:  binding
  - This is a work in progress...

# VM vs Language

- The VM provides certain base services
  - Compiler is free to decide which to expose and which not
  - Compiler can build additional features atop these – we call these *language fictions*
    - Some language fictions have no runtime cost
      - Checked exceptions
    - Others are hard to implement *efficiently*
      - Dynamic dispatch in JRuby
      - Data binding in JavaFX

# VM vs Language



# Typical implementation evolution

- Most languages follow a fairly standard evolution
  - For simple things like 32-bit arithmetic, let the VM do it
  - For most language fictions, implement in a runtime library

# Language Fictions in Runtime

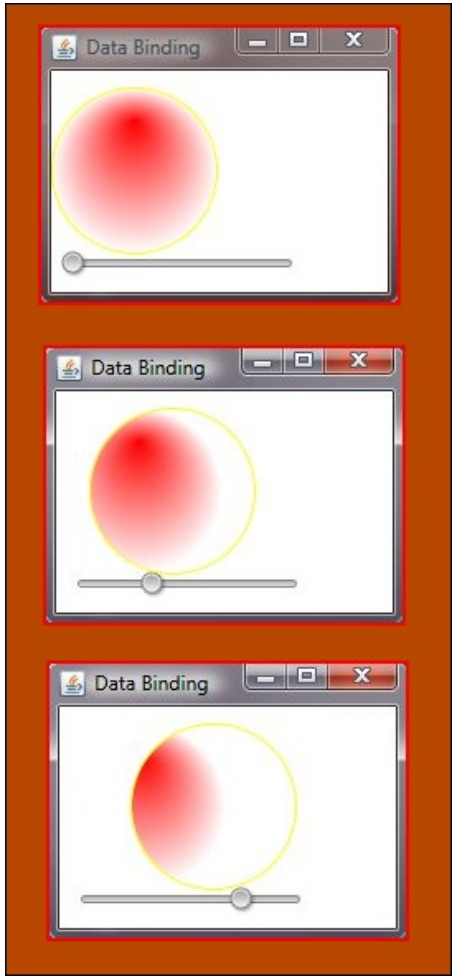
- Implement fictions in a runtime library (usually written in Java)
  - Have compiler emit calls to the runtime
  - Fast time-to-implementation
  - Flexible
  - Testable
- Problem with runtime-centric approach can be performance
  - People can try out the cool features
    - And then notice how slow they are
  - Go from fan mail to hate mail in almost no time...

# JavaFX Script

- JavaFX Script is statically-typed, object-oriented
  - Compiles to Java class files
  - Built for Rich Internet Applications
    - Programming model is based on a *scene graph*
    - Language is optimized for "care and feeding" of scene graph
      - Object literals
      - Animations
      - Declarative data binding

# JavaFX Script: Example

```
def slider = Slider{
                min: 0
                max: 60
                value: 0
                translateX: 10  translateY: 110
             }
Stage {
   title: "Data Binding"
   width: 220  height: 170
   scene: Scene {
      content: [
         slider,
         Circle {
            centerX: bind slider.value+50
            centerY: 60
            radius: 50
            stroke: Color.YELLOW
            fill: RadialGradient {
               centerX: 50  centerY: 60  radius: 50
               focusX: 50  focusY: 30
               proportional: false
               stops: [
                  Stop {offset: 0  color: Color.RED},
                  Stop {offset: 1  color: Color.WHITE},
               ]
            }//RadialGradient
         }//Circle
      ]}//Scene
}//Stage
```



# Background: Binding in JavaFX

- Binding keeps a variable updated to the value of an expression:

```
def x = bind a + b
```

- The value of `x` will always be the sum of `a` and `b` even as `a` and `b` change
- Relationship is fixed and one-directional
  - `x` cannot be assigned to or re-bound
  - values of `a` and `b` are not impacted

# Binding in JavaFX

- Bindings may be arbitrarily complex:

```
Polyline {
  transform: bind
    Transform.translate(0, la.currentHeight-1)
  points: bind
    for (i in [0.0 .. la.currentWidth step 2.0]) {
      def x = i + la.currentWidth mod 2;
      [x, if(indexof i mod 2 == 0) 1.5 else -1.5]
    }
}
```

- Limited bidirectional binding allowed:

```
def g = bind v.y with inverse
```

# Background: On-Replace Triggers

- An on-replace runs a block of code on a change:

```
var w = 4 on replace { println(“w: {w}”) };
w = 1234;
```

- This prints:

```
w: 4
w: 1234
```

- Sequence triggers give slice info:

```
def seq = bind m.seqB`
  on replace
    oldValue[firstIdx..lastIdx]=newElements{
    … }
```

# Binding: Dependencies

- What is different about JavaFX variables?
  - Can have on-replace triggers
  - Can be bound to expressions
  - Can be used in other bindings
- What this means is
  - changes to the value of a variable must be tracked
  - dependency relationships must be maintained between variables

# Implementation #1 – Locations

- We invented structures called Locations
  - Hold and track changes in values
    - value of simple variables (to allow being bound to)
    - value of bound expression for bound variables
  - Maintain dependency relationships
    - to fire update of bound expressions
    - to trigger on-replace

# Implementation #1 – Locations

- Locations: many specialized classes
  - Location interface:
    `get(), set(T), addChangeListener(…), …`
  - Location implementation for variables add:
    `bind(…, Location…), …`

# Implementation #1 – Locations

- Oops!  Footprint explosion
- Problem is two-fold:
  - Simple non-bound variables represented as Location much bigger than just value
  - Dependencies between variables and components of a bind expression form complex data structures
- Well let's optimize these ...

# Early optimizations: Locations

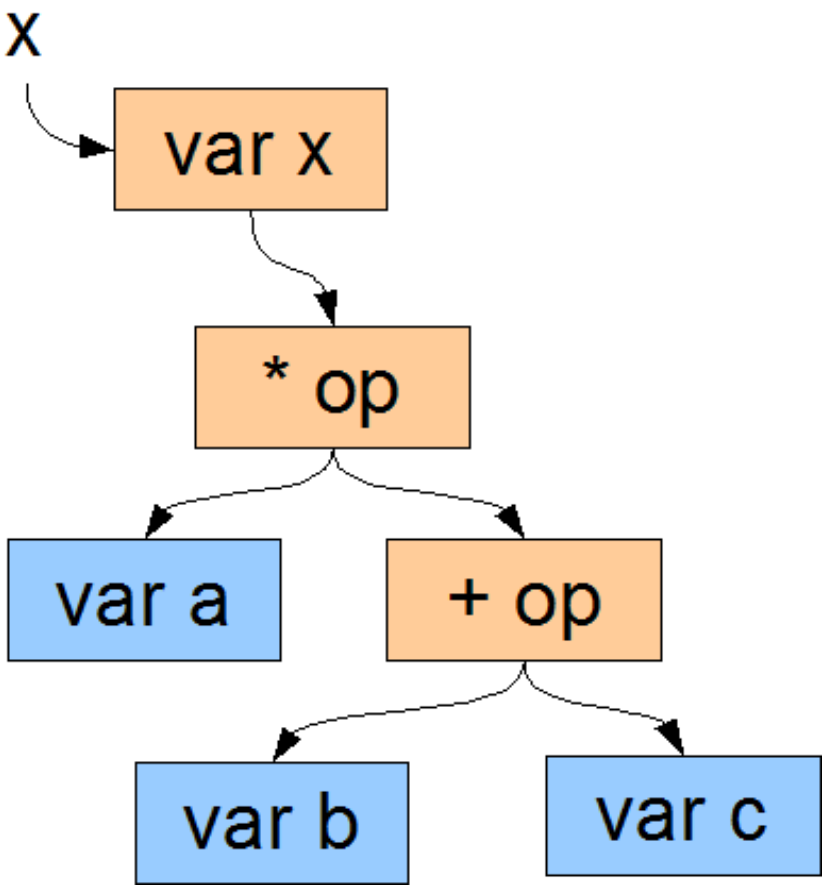
- Reduce size of Location
  - Not every var will be triggered on or bound
    - Lazily inflate those portions of the data structure
  - Initial Location implementation: ~48 bytes
  - "Optimized" Location implementation: 24
- Static elision of locations
  - Compiler can sometimes prove a var can *never* be referenced in a bind or will *never* change
    - Can demote those down to regular fields
    - Analysis aided by the fact that default visibility is script-private. Lesson learned! Pick good defaults.

# Early optimizations: Expressions

- Initial version built a full Location tree for each bound expression

```
def x =
  bind a*(b+c)
```

  - > Lots of runtime classes for operators
  - > Lots of objects



# Early optimizations: Before

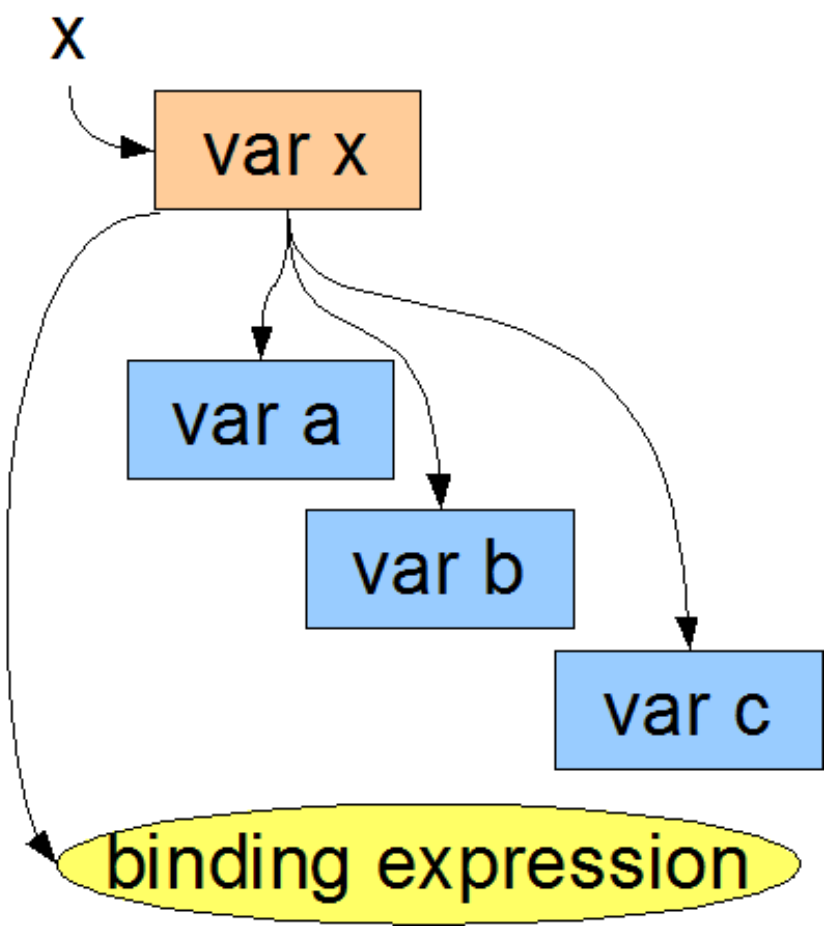
```
public final IntVariable loc$x = IntVariable.make();

public IntVariable loc$x() {
    return loc$x;
}

public static void applyDefaults$x(final b0$Intf receiver$) {
    receiver$.loc$x().bind(
        BoundOperators.plus_ii(
            receiver$.loc$a(),
            receiver$.loc$b()));
}
```

# Early optimizations: Expressions

- Instead represent many bound expressions with compiled code and a list of dependencies



# Early optimizations: After

```
public final IntVariable loc$x = IntVariable.make();

public IntVariable loc$x() {
  return loc$x;
}

public static void applyDefaults$x(final b0$Intf receiver$) {
 receiver$.loc$x().bind(false, new _SBECL(0,   loc$a(),  loc$b(),
                        null, 3));
}

private static class _SBECL<T> extends SBECL<T> {
  public void compute() {
    switch (id) {
      case 0:
        pushValue(((IntLocation)arg$0).getAsInt() + ((IntLocation)arg$1).getAsInt());
        break;
    }
  }

  private _SBECL(final int id, final Object arg$0, final Object arg$1, final Object[] moreArgs,
                 final int dependents) {
    super(id, arg$0, arg$1, moreArgs, dependents);
  }
}
```

# Early optimizations: Alas!

- Significant improvements
  - Smaller and faster
  - In some cases many times over
- But still **way** too big
  - Simple variables x6 Java size
  - Some bound expressions hundreds of bytes
- And all that complexity is ***slow***
- Time to roll up our sleeves

# Implementation #2: Slacker Location

- Same two cases to optimize: simple & bound vars
- Variable representation:
  - Location – for cases we can't optimize
  - Non-Location – as before provable
  - Slacker – lazily inflated to Location
- Slacker:
  - For variables that are externally accessible
  - So, could be bound
  - But often never are

# Implementation #2: Slacker Location

- Optimization: represent each var as value+Location
  - Location initially null

```
int $x;
IntVariable loc$x = null;
```

- Only if Location is requested is it created
  - Getting and setting does not inflate
- Good wins as most vars are never bound
- But what about ones that are ...

# Implementation #2: Slacker Location

- For binds without side-effects, why not lazy eval:

```
def x = bind a + b
```

- The bound expression can be inlined into getter

```
int get$x() { return a + b; }
```

- Similarly, on-replace can be inlined into setter
- Now slacker Locations can be applied to bound vars
- When inflated, slacker binds create: binding expression, Location, change listeners

# Implementation #2: After

```
public static int VOFF$x = 2;
public int $x = 0;
public IntVariable loc$x;

public int get$x() {
    if (loc$x != null)
        return loc$x.getAsInt();
    else
        return (VFLGS$0 & 4) == 0 ?
              get$a() + get$b()
            : $x;
}
```

```
public int set$x(int varNewValue$) {
    if (((VFLGS$0 & 32) != 0 ? loc$x() : loc$x) != null) {
        varNewValue$ = loc$x.setAsInt(varNewValue$);
        VFLGS$0 |= 4;
        return varNewValue$;
    } else {
        $x = varNewValue$;
        VFLGS$0 |= 4;
        return $x;
    }
}
```

# Implementation #2: After – part 2

```
public IntVariable loc$x() {
    if (loc$x != null)
        return loc$x;
    if ((VFLGS$0 & 4) != 0)
        loc$x = IntVariable.make($x);
    else {
        loc$x = IntVariable.make();
        if ((VFLGS$0 & 32) != 0)
            loc$x().bind(false, new _SBECL(1, loc$a(), loc$b(),
                        null, 3));
    }
    return loc$x;
}

public void applyDefaults$(final int varNum$) { … }

private static class _SBECL<T> extends SBECL<T> { … }
```

# Implementation #2: Slacker Location

- Well... how did that work?
- On micro-benchmarks of bind: x20 improvement
- But many binds have potential side-effects, are too complex, have on-replace – so implemented as Locations
- And bound expressions, implemented as Locations, want Location as dependents, so it unravels
- But, lesson learned, if we can inline it all ...

# Take #3: Compiled Bind

- All bound expressions inlined into getters
- All on-replace triggers inlined into setters
- Intra-class dependencies: hard-coded
- Inter-class dependencies: dependent lists
- Variables referenced by (instance, var#) pair
- Default lazy eval using invalidation
- No Locations.  No ChangeListeners.
- One month in ...

# Take #3: where we are today

```
public int $x;

public int get$x() {
  if (!isValidValue$(VOFF$x)) {
    be$x(get$a() + get$b());
  }
  return $x;
}

public void be$x(int varNewValue$) {
  final int varOldValue$ = $x;
  if (varOldValue$ != varNewValue$) {
    $x = varNewValue$;
    invalidate$x();
    setValidValue$(VOFF$x);
    onReplace$x(varOldValue$);
  }
}
```

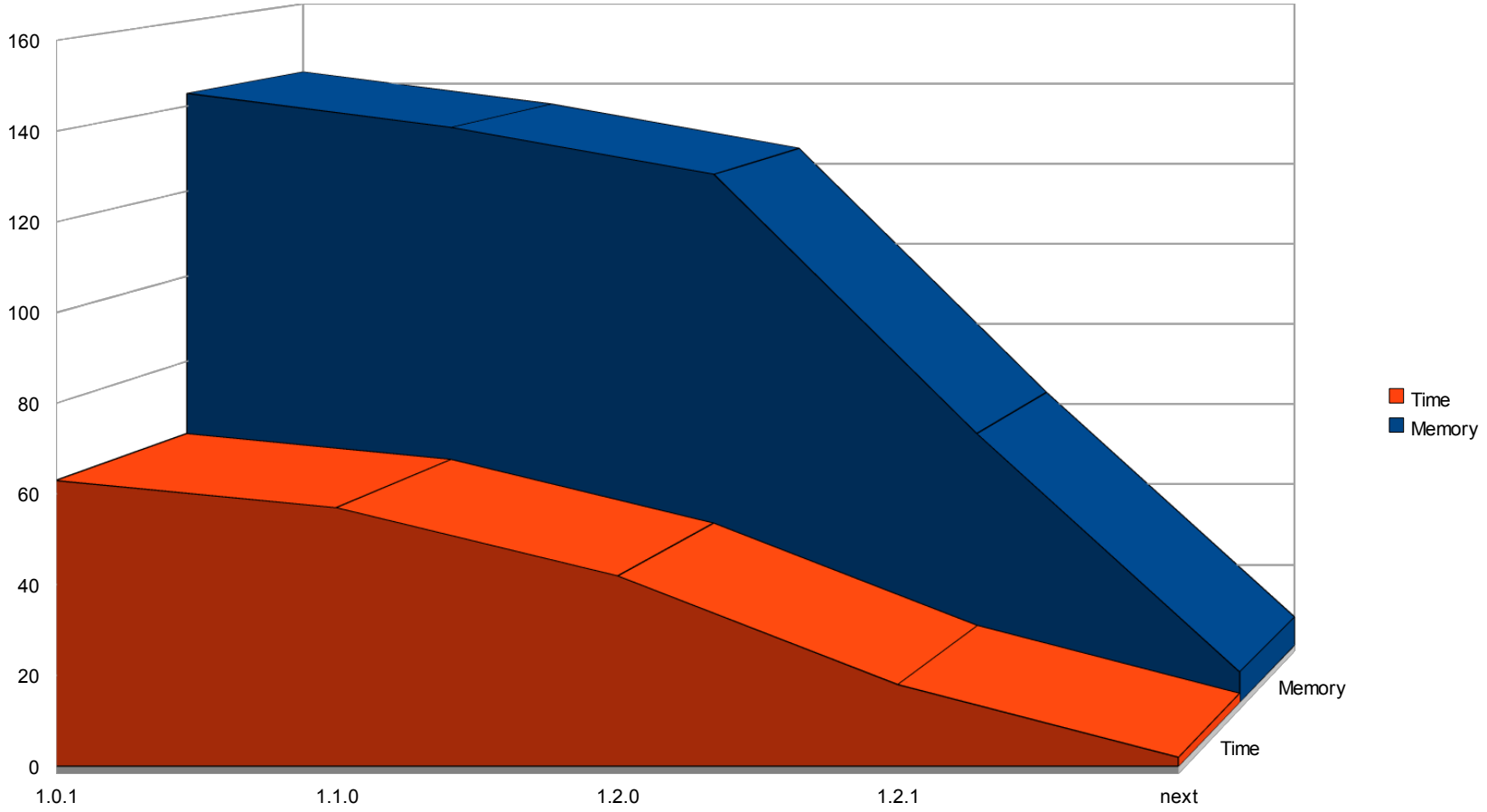
# Take #3: where we are today

```
public void invalidate$x() {
  if (isValidValue$(VOFF$x)) {
    clearValidValue$(VOFF$x);
    notifyDependents$(VOFF$x);
  }
}

public int set$x(int varNewValue$) {
  throw new AssignToBoundException();
}

public void onReplace$b(int varOldValue$) {
}
```

# JavaFX Performance – By Release



# Q & A