Notes - 2/16

Saturday, March 12, 2011 5:43 PM

IMPORTANT

2/23 talk:
- Architecture of the demo
- Make sure that it doesn't leak
    - Contact them and add confidentiality
- Examples and demos
- Starts with Javascript ends with Javascript
- Idiomatic programming as the theme – people do it and do it wrong all the time
- We're at a juncture not wrt JS engines, how do we get to the next level
- |0 as an example of how yu run on every browser but run better on ours
    - Show on a slide
    - There are idioms like this for doubles and string
    - There are idioms like this for classes
- Culture of static typing

Longer term architecture:
- C++ codebase
- Ideally an open source JS implementation

Pattern for inheritance:
- Setting the prototype as an object literal doesn't work
- We're good for now

Constants:
- Comment for debugging
- Erasure for static class member
- Separate compilation issue with erasure
- "const"

Timeline:
- Mix 2012

Violation of constraints:
- Starts with JS ends with JS
- By default you don't
- Exceptions are okay – no guarantees you'll get these runtime checking on reach platforms
- Calling a typed function accepting int from outside with a string should work
-

```
class Constraint(int strength)
{
  protected int strength;
  abstract void addToGraph();
}


class UnaryConstraint(var v, int strength)
  : Constraint(strength)
{
   var myOutput  = v;
   bool satisfied = false;

   public override var addToGraph() {
     this.myOutput.addConstraint(this);
     this.satisfied = false;
```

```
    }
}
```

```
class Point(int x, int y) {
   protected int x = x;
   public int
}
```

```
class Constraint(int strength)
{
  protected int strength;
  abstract void addToGraph();
  abstract void addContraint();
}
```

```
class UnaryConstraint(var v, int strength)
  : Constraint(strength)
{
    private var myOutput  = v;
    bool satisfied = false;
    addConstraint();

    public override var addToGraph() {
      myOutput.addConstraint(this);
      satisfied = false;
    }

    public override void addConstraint() {
       //...
    }
}
```

```
var UnaryConstraint = function(v, strength) {
   Constraint.call(this, strength);
   this.myOutput = v;
   this.satisfied = false;
   this.addConstraint();
}

UnaryConstraint.inheritsFrom(Constraint)

UnaryContraint.prototype.addToGraph = function() {
   this.myOutput.addContraint(this);
   this.satisfied = false;
}
```

- Can leave off parens -> zero params

Questions:
- Public fields?
  - Want yes. Need to reconcile with rules for entering the box, as this is a very easy way to crss the boundary.
- Implied this.
- What is the syntax for members?
- Short syntax for single expression functions
  - Yes
- Scope modifiers on class params?
  - No
- Protected members?
  - Yes

```
class Foo(int x)
{
  int y = x;



  public int Add(int z) = z + y;


}
```

```
// NOT FOR REDISTRIBUTION

function echo(o) {
   try {
      document.write(o + "<br/>");
```

```
    } catch (ex) {
      try {
        WScript.Echo("" + o);
      } catch (ex2) {
        print("" + o);
      }
    }
}


// Original

var _v8StartDate = new Date();

/**
* A JavaScript implementation of the DeltaBlue constraint-solving
* algorithm, as described in:
*
* "The DeltaBlue Algorithm: An Incremental Constraint Hierarchy Solver"
*   Bjorn N. Freeman-Benson and John Maloney
*   January 1990 Communications of the ACM,
*   also available as University of Washington TR 89-08-06.
*
* Beware: this benchmark is written in a grotesque style where
* the constraint model is built by side-effects from constructors.
* I've kept it this way to avoid deviating too much from the original
* implementation.
*/


/* --- O b j e c t   M o d e l --- */

Object.prototype.inheritsFrom = function (shuper) {
  function Inheriter() { }
  Inheriter.prototype = shuper.prototype;
  this.prototype = new Inheriter();
  //this.superConstructor = shuper;
}

Foo.inheritsFrom(Bar)

function OrderedCollection() {
  this.elms = new Array();
}

OrderedCollection.prototype.add = function (elm) {
  this.elms.push(elm);
}

OrderedCollection.prototype.at = function (index) {
  return this.elms[index];
}

OrderedCollection.prototype.size = function () {
  return this.elms.length;
}

OrderedCollection.prototype.removeFirst = function () {
  return this.elms.pop();
}

OrderedCollection.prototype.remove = function (elm) {
  var index = 0, skipped = 0;
  for (var i = 0; i < this.elms.length; i++) {
    var value = this.elms[i];
    if (value != elm) {
      this.elms[index] = value;
```

```
      index++;
    } else {
      skipped++;
    }
  }
  for (var i = 0; i < skipped; i++)
    this.elms.pop();
}

/* --- *
* S t r e n g t h
* --- */

/**
* Strengths are used to measure the relative importance of constraints.
* New strengths may be inserted in the strength hierarchy without
* disrupting current constraints.  Strengths cannot be created outside
* this class, so pointer comparison can be used for value comparison.
*/
function Strength(strengthValue, name) {
  this.strengthValue = strengthValue;
  this.name = name;
}

Strength.stronger = function (s1, s2) {
  return s1.strengthValue < s2.strengthValue;
}

Strength.weaker = function (s1, s2) {
  return s1.strengthValue > s2.strengthValue;
}

Strength.weakestOf = function (s1, s2) {
  return this.weaker(s1, s2) ? s1 : s2;
}

Strength.strongest = function (s1, s2) {
  return this.stronger(s1, s2) ? s1 : s2;
}

Strength.prototype.nextWeaker = function () {
  switch (this.strengthValue) {
    case 0: return Strength.WEAKEST;
    case 1: return Strength.WEAK_DEFAULT;
    case 2: return Strength.NORMAL;
    case 3: return Strength.STRONG_DEFAULT;
    case 4: return Strength.PREFERRED;
    case 5: return Strength.REQUIRED;
  }
}

// Strength constants.
Strength.REQUIRED        = new Strength(0, "required");
Strength.STONG_PREFERRED = new Strength(1, "strongPreferred");
Strength.PREFERRED       = new Strength(2, "preferred");
Strength.STRONG_DEFAULT  = new Strength(3, "strongDefault");
Strength.NORMAL          = new Strength(4, "normal");
Strength.WEAK_DEFAULT    = new Strength(5, "weakDefault");
Strength.WEAKEST         = new Strength(6, "weakest");

/* --- *
* C o n s t r a i n t
* --- */

/**
* An abstract class representing a system-maintainable relationship
* (or "constraint") between a set of variables. A constraint supplies
```

```
* a strength instance variable; concrete subclasses provide a means
* of storing the constrained variables and other information required
* to represent a constraint.
*/
function Constraint(strength) {
  this.strength = strength;
}

/**
* Activate this constraint and attempt to satisfy it.
*/
Constraint.prototype.addConstraint = function () {
  this.addToGraph();
  planner.incrementalAdd(this);
}

/**
* Attempt to find a way to enforce this constraint. If successful,
* record the solution, perhaps modifying the current dataflow
* graph. Answer the constraint that this constraint overrides, if
* there is one, or nil, if there isn't.
* Assume: I am not already satisfied.
*/
Constraint.prototype.satisfy = function (mark) {
  this.chooseMethod(mark);
  if (!this.isSatisfied()) {
    if (this.strength == Strength.REQUIRED)
      echo("Could not satisfy a required constraint!");
    return null;
  }
  this.markInputs(mark);
  var out = this.output();
  var overridden = out.determinedBy;
  if (overridden != null) overridden.markUnsatisfied();
  out.determinedBy = this;
  if (!planner.addPropagate(this, mark))
    echo("Cycle encountered");
  out.mark = mark;
  return overridden;
}

Constraint.prototype.destroyConstraint = function () {
  if (this.isSatisfied()) planner.incrementalRemove(this);
  else this.removeFromGraph();
}

/**
* Normal constraints are not input constraints.  An input constraint
* is one that depends on external state, such as the mouse, the
* keybord, a clock, or some arbitraty piece of imperative code.
*/
Constraint.prototype.isInput = function () {
  return false;
}

/* --- *
* U n a r y   C o n s t r a i n t
* --- */

/**
* Abstract superclass for constraints having a single possible output
* variable.
*/
function UnaryConstraint(v, strength) {
  UnaryConstraint.superConstructor.call(this, strength);
  this.myOutput = v;
  this.satisfied = false;
```

```
  this.addConstraint();
}

UnaryConstraint.inheritsFrom(Constraint);

/**
* Adds this constraint to the constraint graph
*/
UnaryConstraint.prototype.addToGraph = function () {
  this.myOutput.addConstraint(this);
  this.satisfied = false;
}

/**
* Decides if this constraint can be satisfied and records that
* decision.
*/
UnaryConstraint.prototype.chooseMethod = function (mark) {
  this.satisfied = (this.myOutput.mark != mark)
    && Strength.stronger(this.strength, this.myOutput.walkStrength);
}

/**
* Returns true if this constraint is satisfied in the current solution.
*/
UnaryConstraint.prototype.isSatisfied = function () {
  return this.satisfied;
}

UnaryConstraint.prototype.markInputs = function (mark) {
  // has no inputs
}

/**
* Returns the current output variable.
*/
UnaryConstraint.prototype.output = function () {
  return this.myOutput;
}

/**
* Calculate the walkabout strength, the stay flag, and, if it is
* 'stay', the value for the current output of this constraint. Assume
* this constraint is satisfied.
*/
UnaryConstraint.prototype.recalculate = function () {
  this.myOutput.walkStrength = this.strength;
  this.myOutput.stay = !this.isInput();
  if (this.myOutput.stay) this.execute(); // Stay optimization
}

/**
* Records that this constraint is unsatisfied
*/
UnaryConstraint.prototype.markUnsatisfied = function () {
  this.satisfied = false;
}

UnaryConstraint.prototype.inputsKnown = function () {
  return true;
}

UnaryConstraint.prototype.removeFromGraph = function () {
  if (this.myOutput != null) this.myOutput.removeConstraint(this);
  this.satisfied = false;
}
```

```
/* --- *
* S t a y   C o n s t r a i n t
* --- */

/**
* Variables that should, with some level of preference, stay the same.
* Planners may exploit the fact that instances, if satisfied, will not
* change their output during plan execution.  This is called "stay
* optimization".
*/
function StayConstraint(v, str) {
  StayConstraint.superConstructor.call(this, v, str);
}

StayConstraint.inheritsFrom(UnaryConstraint);

StayConstraint.prototype.execute = function () {
  // Stay constraints do nothing
}

/* --- *
* E d i t   C o n s t r a i n t
* --- */

/**
* A unary input constraint used to mark a variable that the client
* wishes to change.
*/
function EditConstraint(v, str) {
  EditConstraint.superConstructor.call(this, v, str);
}

EditConstraint.inheritsFrom(UnaryConstraint);

/**
* Edits indicate that a variable is to be changed by imperative code.
*/
EditConstraint.prototype.isInput = function () {
  return true;
}

EditConstraint.prototype.execute = function () {
  // Edit constraints do nothing
}

/* --- *
* B i n a r y   C o n s t r a i n t
* --- */

var Direction = new Object();
Direction.NONE     = 0;
Direction.FORWARD  = 1;
Direction.BACKWARD = -1;

/**
* Abstract superclass for constraints having two possible output
* variables.
*/
function BinaryConstraint(var1, var2, strength) {
  BinaryConstraint.superConstructor.call(this, strength);
  this.v1 = var1;
  this.v2 = var2;
  this.direction = Direction.NONE;
  this.addConstraint();
}

BinaryConstraint.inheritsFrom(Constraint);
```

```
/**
* Decides if this constraint can be satisfied and which way it
* should flow based on the relative strength of the variables related,
* and record that decision.
*/
BinaryConstraint.prototype.chooseMethod = function (mark) {
 if (this.v1.mark == mark) {
   this.direction = (this.v2.mark != mark && Strength.stronger(this.strength, this.v2.walkStrength))
     ? Direction.FORWARD
     : Direction.NONE;
 }
 if (this.v2.mark == mark) {
   this.direction = (this.v1.mark != mark && Strength.stronger(this.strength, this.v1.walkStrength))
     ? Direction.BACKWARD
     : Direction.NONE;
 }
 if (Strength.weaker(this.v1.walkStrength, this.v2.walkStrength)) {
   this.direction = Strength.stronger(this.strength, this.v1.walkStrength)
     ? Direction.BACKWARD
     : Direction.NONE;
 } else {
   this.direction = Strength.stronger(this.strength, this.v2.walkStrength)
     ? Direction.FORWARD
     : Direction.BACKWARD
 }
}

/**
* Add this constraint to the constraint graph
*/
BinaryConstraint.prototype.addToGraph = function () {
 this.v1.addConstraint(this);
 this.v2.addConstraint(this);
 this.direction = Direction.NONE;
}

/**
* Answer true if this constraint is satisfied in the current solution.
*/
BinaryConstraint.prototype.isSatisfied = function () {
 return this.direction != Direction.NONE;
}

/**
* Mark the input variable with the given mark.
*/
BinaryConstraint.prototype.markInputs = function (mark) {
 this.input().mark = mark;
}

/**
* Returns the current input variable
*/
BinaryConstraint.prototype.input = function () {
 return (this.direction == Direction.FORWARD) ? this.v1 : this.v2;
}

/**
* Returns the current output variable
*/
BinaryConstraint.prototype.output = function () {
 return (this.direction == Direction.FORWARD) ? this.v2 : this.v1;
}

/**
* Calculate the walkabout strength, the stay flag, and, if it is
```

```
* 'stay', the value for the current output of this
* constraint. Assume this constraint is satisfied.
*/
BinaryConstraint.prototype.recalculate = function () {
 var ihn = this.input(), out = this.output();
 out.walkStrength = Strength.weakestOf(this.strength, ihn.walkStrength);
 out.stay = ihn.stay;
 if (out.stay) this.execute();
}

/**
* Record the fact that this constraint is unsatisfied.
*/
BinaryConstraint.prototype.markUnsatisfied = function () {
 this.direction = Direction.NONE;
}

BinaryConstraint.prototype.inputsKnown = function (mark) {
 var i = this.input();
 return i.mark == mark || i.stay || i.determinedBy == null;
}

BinaryConstraint.prototype.removeFromGraph = function () {
 if (this.v1 != null) this.v1.removeConstraint(this);
 if (this.v2 != null) this.v2.removeConstraint(this);
 this.direction = Direction.NONE;
}

/* --- *
* S c a l e   C o n s t r a i n t
* --- */

/**
* Relates two variables by the linear scaling relationship: "v2 =
* (v1 * scale) + offset". Either v1 or v2 may be changed to maintain
* this relationship but the scale factor and offset are considered
* read-only.
*/
function ScaleConstraint(src, scale, offset, dest, strength) {
 this.direction = Direction.NONE;
 this.scale = scale;
 this.offset = offset;
 ScaleConstraint.superConstructor.call(this, src, dest, strength);
}

ScaleConstraint.inheritsFrom(BinaryConstraint);

/**
* Adds this constraint to the constraint graph.
*/
ScaleConstraint.prototype.addToGraph = function () {
 ScaleConstraint.superConstructor.prototype.addToGraph.call(this);
 this.scale.addConstraint(this);
 this.offset.addConstraint(this);
}

ScaleConstraint.prototype.removeFromGraph = function () {
 ScaleConstraint.superConstructor.prototype.removeFromGraph.call(this);
 if (this.scale != null) this.scale.removeConstraint(this);
 if (this.offset != null) this.offset.removeConstraint(this);
}

ScaleConstraint.prototype.markInputs = function (mark) {
 ScaleConstraint.superConstructor.prototype.markInputs.call(this, mark);
 this.scale.mark = this.offset.mark = mark;
}
```

```
/**
* Enforce this constraint. Assume that it is satisfied.
*/
ScaleConstraint.prototype.execute = function () {
  if (this.direction == Direction.FORWARD) {
    this.v2.value = this.v1.value * this.scale.value + this.offset.value;
  } else {
    this.v1.value = (this.v2.value - this.offset.value) / this.scale.value;
  }
}

/**
* Calculate the walkabout strength, the stay flag, and, if it is
* 'stay', the value for the current output of this constraint. Assume
* this constraint is satisfied.
*/
ScaleConstraint.prototype.recalculate = function () {
  var ihn = this.input(), out = this.output();
  out.walkStrength = Strength.weakestOf(this.strength, ihn.walkStrength);
  out.stay = ihn.stay && this.scale.stay && this.offset.stay;
  if (out.stay) this.execute();
}

/* --- *
* E q u a l i t y   C o n s t r a i n t
* --- */

/**
* Constrains two variables to have the same value.
*/
function EqualityConstraint(var1, var2, strength) {
  EqualityConstraint.superConstructor.call(this, var1, var2, strength);
}

EqualityConstraint.inheritsFrom(BinaryConstraint);

/**
* Enforce this constraint. Assume that it is satisfied.
*/
EqualityConstraint.prototype.execute = function () {
  this.output().value = this.input().value;
}

/* --- *
* V a r i a b l e
* --- */

/**
* A constrained variable. In addition to its value, it maintain the
* structure of the constraint graph, the current dataflow graph, and
* various parameters of interest to the DeltaBlue incremental
* constraint solver.
**/
function Variable(name, initialValue) {
  this.value = initialValue || 0;
  this.constraints = new OrderedCollection();
  this.determinedBy = null;
  this.mark = 0;
  this.walkStrength = Strength.WEAKEST;
  this.stay = true;
  this.name = name;
}

/**
* Add the given constraint to the set of all constraints that refer
* this variable.
*/
```

```
Variable.prototype.addConstraint = function (c) {
  this.constraints.add(c);
}

/**
* Removes all traces of c from this variable.
*/
Variable.prototype.removeConstraint = function (c) {
  this.constraints.remove(c);
  if (this.determinedBy == c) this.determinedBy = null;
}

/* --- *
* P l a n n e r
* --- */

/**
* The DeltaBlue planner
*/
function Planner() {
  this.currentMark = 0;
}

/**
* Attempt to satisfy the given constraint and, if successful,
* incrementally update the dataflow graph.  Details: If satifying
* the constraint is successful, it may override a weaker constraint
* on its output. The algorithm attempts to resatisfy that
* constraint using some other method. This process is repeated
* until either a) it reaches a variable that was not previously
* determined by any constraint or b) it reaches a constraint that
* is too weak to be satisfied using any of its methods. The
* variables of constraints that have been processed are marked with
* a unique mark value so that we know where we've been. This allows
* the algorithm to avoid getting into an infinite loop even if the
* constraint graph has an inadvertent cycle.
*/
Planner.prototype.incrementalAdd = function (c) {
  var mark = this.newMark();
  var overridden = c.satisfy(mark);
  while (overridden != null)
    overridden = overridden.satisfy(mark);
}

/**
* Entry point for retracting a constraint. Remove the given
* constraint and incrementally update the dataflow graph.
* Details: Retracting the given constraint may allow some currently
* unsatisfiable downstream constraint to be satisfied. We therefore collect
* a list of unsatisfied downstream constraints and attempt to
* satisfy each one in turn. This list is traversed by constraint
* strength, strongest first, as a heuristic for avoiding
* unnecessarily adding and then overriding weak constraints.
* Assume: c is satisfied.
*/
Planner.prototype.incrementalRemove = function (c) {
  var out = c.output();
  c.markUnsatisfied();
  c.removeFromGraph();
  var unsatisfied = this.removePropagateFrom(out);
  var strength = Strength.REQUIRED;
  do {
    for (var i = 0; i < unsatisfied.size(); i++) {
      var u = unsatisfied.at(i);
      if (u.strength == strength)
        this.incrementalAdd(u);
    }
```

```
    strength = strength.nextWeaker();
  } while (strength != Strength.WEAKEST);
}

/**
* Select a previously unused mark value.
*/
Planner.prototype.newMark = function () {
  return ++this.currentMark;
}

/**
* Extract a plan for resatisfaction starting from the given source
* constraints, usually a set of input constraints. This method
* assumes that stay optimization is desired; the plan will contain
* only constraints whose output variables are not stay. Constraints
* that do no computation, such as stay and edit constraints, are
* not included in the plan.
* Details: The outputs of a constraint are marked when it is added
* to the plan under construction. A constraint may be appended to
* the plan when all its input variables are known. A variable is
* known if either a) the variable is marked (indicating that has
* been computed by a constraint appearing earlier in the plan), b)
* the variable is 'stay' (i.e. it is a constant at plan execution
* time), or c) the variable is not determined by any
* constraint. The last provision is for past states of history
* variables, which are not stay but which are also not computed by
* any constraint.
* Assume: sources are all satisfied.
*/
Planner.prototype.makePlan = function (sources) {
  var mark = this.newMark();
  var plan = new Plan();
  var todo = sources;
  while (todo.size() > 0) {
    var c = todo.removeFirst();
    if (c.output().mark != mark && c.inputsKnown(mark)) {
      plan.addConstraint(c);
      c.output().mark = mark;
      this.addConstraintsConsumingTo(c.output(), todo);
    }
  }
  return plan;
}

/**
* Extract a plan for resatisfying starting from the output of the
* given constraints, usually a set of input constraints.
*/
Planner.prototype.extractPlanFromConstraints = function (constraints) {
  var sources = new OrderedCollection();
  for (var i = 0; i < constraints.size(); i++) {
    var c = constraints.at(i);
    if (c.isInput() && c.isSatisfied())
      // not in plan already and eligible for inclusion
      sources.add(c);
  }
  return this.makePlan(sources);
}

/**
* Recompute the walkabout strengths and stay flags of all variables
* downstream of the given constraint and recompute the actual
* values of all variables whose stay flag is true. If a cycle is
* detected, remove the given constraint and answer
* false. Otherwise, answer true.
* Details: Cycles are detected when a marked variable is
```

```
* encountered downstream of the given constraint. The sender is
* assumed to have marked the inputs of the given constraint with
* the given mark. Thus, encountering a marked node downstream of
* the output constraint means that there is a path from the
* constraint's output to one of its inputs.
*/
Planner.prototype.addPropagate = function (c, mark) {
 var todo = new OrderedCollection();
 todo.add(c);
 while (todo.size() > 0) {
  var d = todo.removeFirst();
  if (d.output().mark == mark) {
   this.incrementalRemove(c);
   return false;
  }
  d.recalculate();
  this.addConstraintsConsumingTo(d.output(), todo);
 }
 return true;
}


/**
* Update the walkabout strengths and stay flags of all variables
* downstream of the given constraint. Answer a collection of
* unsatisfied constraints sorted in order of decreasing strength.
*/
Planner.prototype.removePropagateFrom = function (out) {
 out.determinedBy = null;
 out.walkStrength = Strength.WEAKEST;
 out.stay = true;
 var unsatisfied = new OrderedCollection();
 var todo = new OrderedCollection();
 todo.add(out);
 while (todo.size() > 0) {
  var v = todo.removeFirst();
  for (var i = 0; i < v.constraints.size(); i++) {
   var c = v.constraints.at(i);
   if (!c.isSatisfied())
    unsatisfied.add(c);
  }
  var determining = v.determinedBy;
  for (var i = 0; i < v.constraints.size(); i++) {
   var next = v.constraints.at(i);
   if (next != determining && next.isSatisfied()) {
    next.recalculate();
    todo.add(next.output());
   }
  }
 }
 return unsatisfied;
}

Planner.prototype.addConstraintsConsumingTo = function (v, coll) {
 var determining = v.determinedBy;
 var cc = v.constraints;
 for (var i = 0; i < cc.size(); i++) {
  var c = cc.at(i);
  if (c != determining && c.isSatisfied())
   coll.add(c);
 }
}

/* --- *
* P l a n
* --- */
```

```
/**
* A Plan is an ordered list of constraints to be executed in sequence
* to resatisfy all currently satisfiable constraints in the face of
* one or more changing inputs.
*/
function Plan() {
  this.v = new OrderedCollection();
}

Plan.prototype.addConstraint = function (c) {
  this.v.add(c);
}

Plan.prototype.size = function () {
  return this.v.size();
}

Plan.prototype.constraintAt = function (index) {
  return this.v.at(index);
}

Plan.prototype.execute = function () {
  for (var i = 0; i < this.size(); i++) {
    var c = this.constraintAt(i);
    c.execute();
  }
}

/* --- *
* M a i n
* --- */

/**
* This is the standard DeltaBlue benchmark. A long chain of equality
* constraints is constructed with a stay constraint on one end. An
* edit constraint is then added to the opposite end and the time is
* measured for adding and removing this constraint, and extracting
* and executing a constraint satisfaction plan. There are two cases.
* In case 1, the added constraint is stronger than the stay
* constraint and values must propagate down the entire length of the
* chain. In case 2, the added constraint is weaker than the stay
* constraint so it cannot be accomodated. The cost in this case is,
* of course, very low. Typical situations lie somewhere between these
* two extremes.
*/
function chainTest(n) {
  planner = new Planner();
  var prev = null, first = null, last = null;

  // Build chain of n equality constraints
  for (var i = 0; i <= n; i++) {
    var name = "v" + i;
    var v = new Variable(name);
    if (prev != null)
      new EqualityConstraint(prev, v, Strength.REQUIRED);
    if (i == 0) first = v;
    if (i == n) last = v;
    prev = v;
  }

  new StayConstraint(last, Strength.STRONG_DEFAULT);
  var edit = new EditConstraint(first, Strength.PREFERRED);
  var edits = new OrderedCollection();
  edits.add(edit);
  var plan = planner.extractPlanFromConstraints(edits);
  for (var i = 0; i < 100; i++) {
    first.value = i;
```

```
    plan.execute();
    if (last.value != i)
      echo("Chain test failed.");
  }
}

/**
* This test constructs a two sets of variables related to each
* other by a simple linear transformation (scale and offset). The
* time is measured to change a variable on either side of the
* mapping and to change the scale and offset factors.
*/
function projectionTest(n) {
  planner = new Planner();
  var scale = new Variable("scale", 10);
  var offset = new Variable("offset", 1000);
  var src = null, dst = null;

  var dests = new OrderedCollection();
  for (var i = 0; i < n; i++) {
    src = new Variable("src" + i, i);
    dst = new Variable("dst" + i, i);
    dests.add(dst);
    new StayConstraint(src, Strength.NORMAL);
    new ScaleConstraint(src, scale, offset, dst, Strength.REQUIRED);
  }

  change(src, 17);
  if (dst.value != 1170) echo("Projection 1 failed");
  change(dst, 1050);
  if (src.value != 5) echo("Projection 2 failed");
  change(scale, 5);
  for (var i = 0; i < n - 1; i++) {
    if (dests.at(i).value != i * 5 + 1000)
      echo("Projection 3 failed");
  }
  change(offset, 2000);
  for (var i = 0; i < n - 1; i++) {
    if (dests.at(i).value != i * 5 + 2000)
      echo("Projection 4 failed");
  }
}

function change(v, newValue) {
  var edit = new EditConstraint(v, Strength.PREFERRED);
  var edits = new OrderedCollection();
  edits.add(edit);
  var plan = planner.extractPlanFromConstraints(edits);
  for (var i = 0; i < 10; i++) {
    v.value = newValue;
    plan.execute();
  }
  edit.destroyConstraint();
}

// Global variable holding the current planner.
var planner = null;

function deltaBlue() {
  chainTest(100);
  projectionTest(100);
}

//////////////////////////////////////////////////////////////////////
for (var i = 0; i < 100; i++)
    deltaBlue();
```

```
var _v8Interval = new Date() - _v8StartDate;

echo("### DeltaBlue Original: " + _v8Interval + " ms");
```