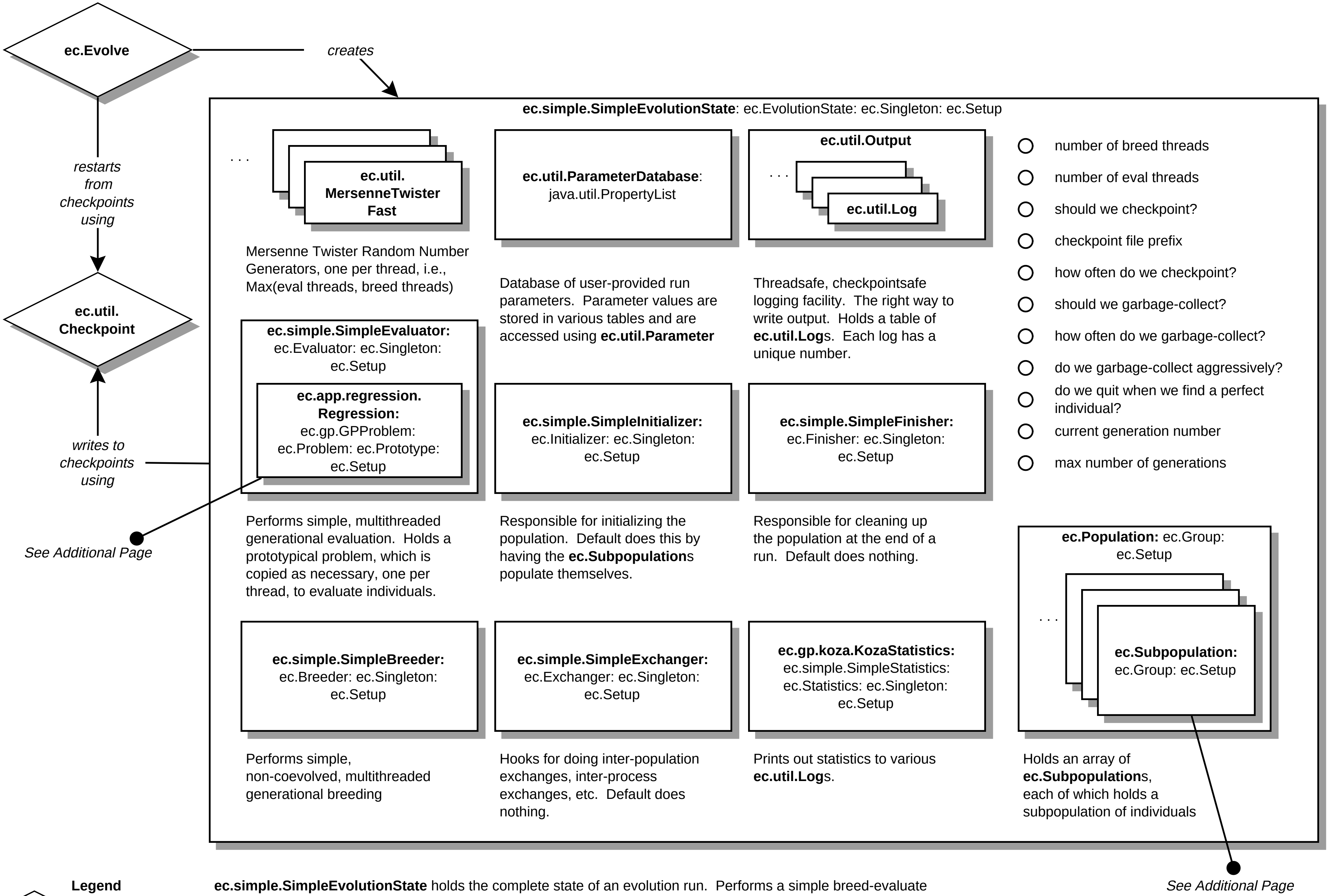


ec.simple.SimpleEvolutionState: ec.EvolutionState: ec.Singleton: ec.Setup

ec.util. MersenneTwister Fast

Mersenne Twister Random Number Generators, one per thread, i.e., Max(eval threads, breed threads)

ec.util.ParameterDatabase: java.util.PropertyList

Database of user-provided run parameters. Parameter values are stored in various tables and are accessed using **ec.util.Parameter**

ec.util.Output

ec.util.Log

Threadsafe, checkpointsafe logging facility. The right way to write output. Holds a table of **ec.util.Log**s. Each log has a unique number.

○ number of breed threads
○ number of eval threads
○ should we checkpoint?
○ checkpoint file prefix
○ how often do we checkpoint?
○ should we garbage-collect?
○ how often do we garbage-collect?
○ do we garbage-collect aggressively?
○ do we quit when we find a perfect individual?
○ current generation number
○ max number of generations

ec.simple.SimpleEvaluator: ec.Evaluator: ec.Singleton: ec.Setup

ec.app.regression. Regression: ec.gp.GPProblem: ec.Problem: ec.Prototype: ec.Setup

Performs simple, multithreaded generational evaluation. Holds a prototypical problem, which is copied as necessary, one per thread, to evaluate individuals.

ec.simple.SimpleInitializer: ec.Initializer: ec.Singleton: ec.Setup

Responsible for initializing the population. Default does this by having the **ec.Subpopulation**s populate themselves.

ec.simple.SimpleFinisher: ec.Finisher: ec.Singleton: ec.Setup

Responsible for cleaning up the population at the end of a run. Default does nothing.

ec.Population: ec.Group: ec.Setup

ec.Subpopulation: ec.Group: ec.Setup

ec.simple.SimpleBreeder: ec.Breeder: ec.Singleton: ec.Setup

Performs simple, non-coevolved, multithreaded generational breeding

ec.simple.SimpleExchanger: ec.Exchanger: ec.Singleton: ec.Setup

Hooks for doing inter-population exchanges, inter-process exchanges, etc. Default does nothing.

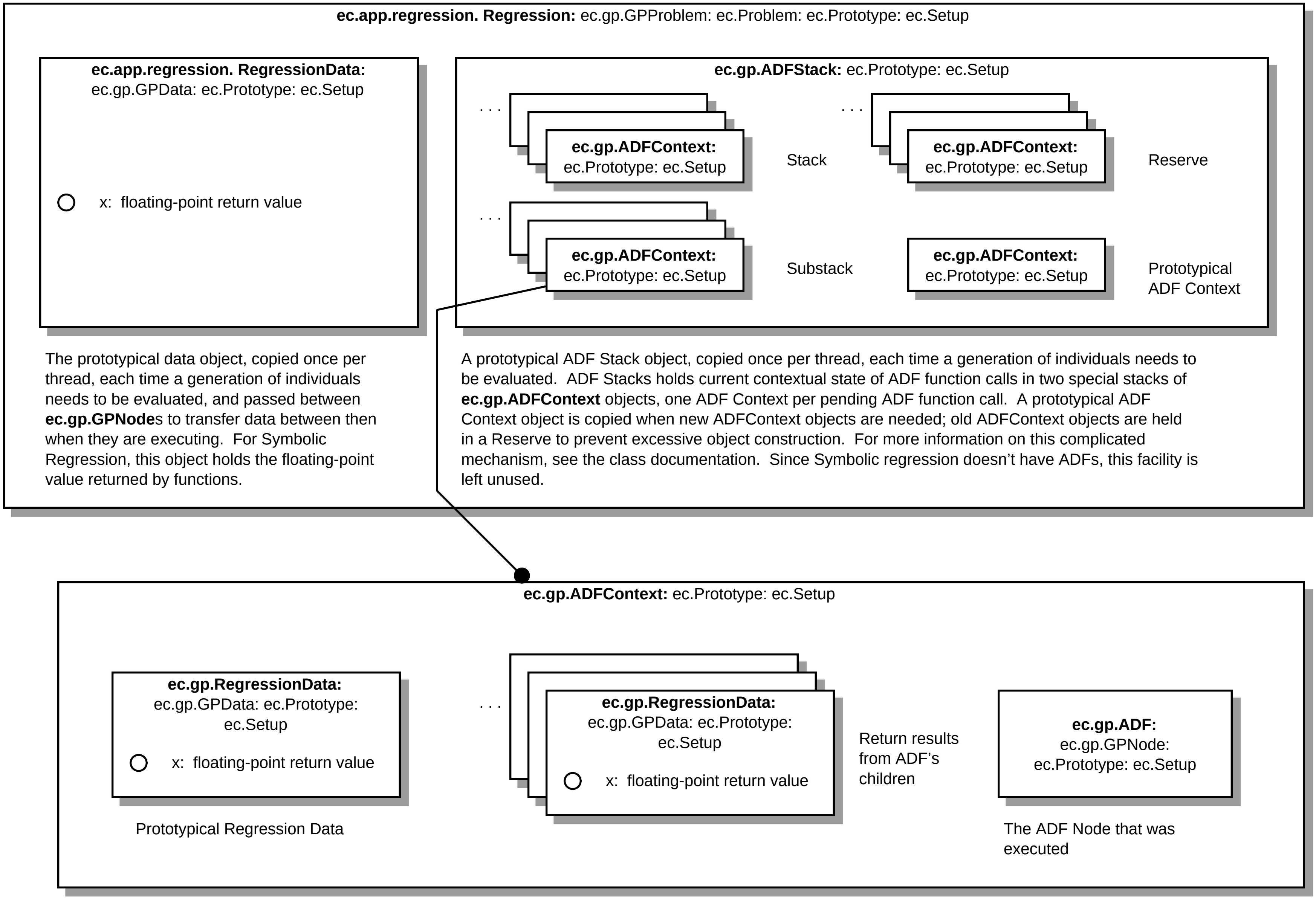
ec.gp.koza.KozaStatistics: ec.simple.SimpleStatistics: ec.Statistics: ec.Singleton: ec.Setup

Prints out statistics to various **ec.util.Log**s.

Holds an array of **ec.Subpopulation**s, each of which holds a subpopulation of individuals

Legend
Static objects
Instances
○ Data

ec.simple.SimpleEvolutionState holds the complete state of an evolution run. Performs a simple breed-evaluate loop on individuals in its population. Every object stored within this object is serializable, and so can be written out to a checkpoint file using **ec.util.Checkpoint.** The evolution state is also passed around to among objects a lot so they can access its random number generators, parameter database, and logs. This is done, of course, in a threadsafe manner.

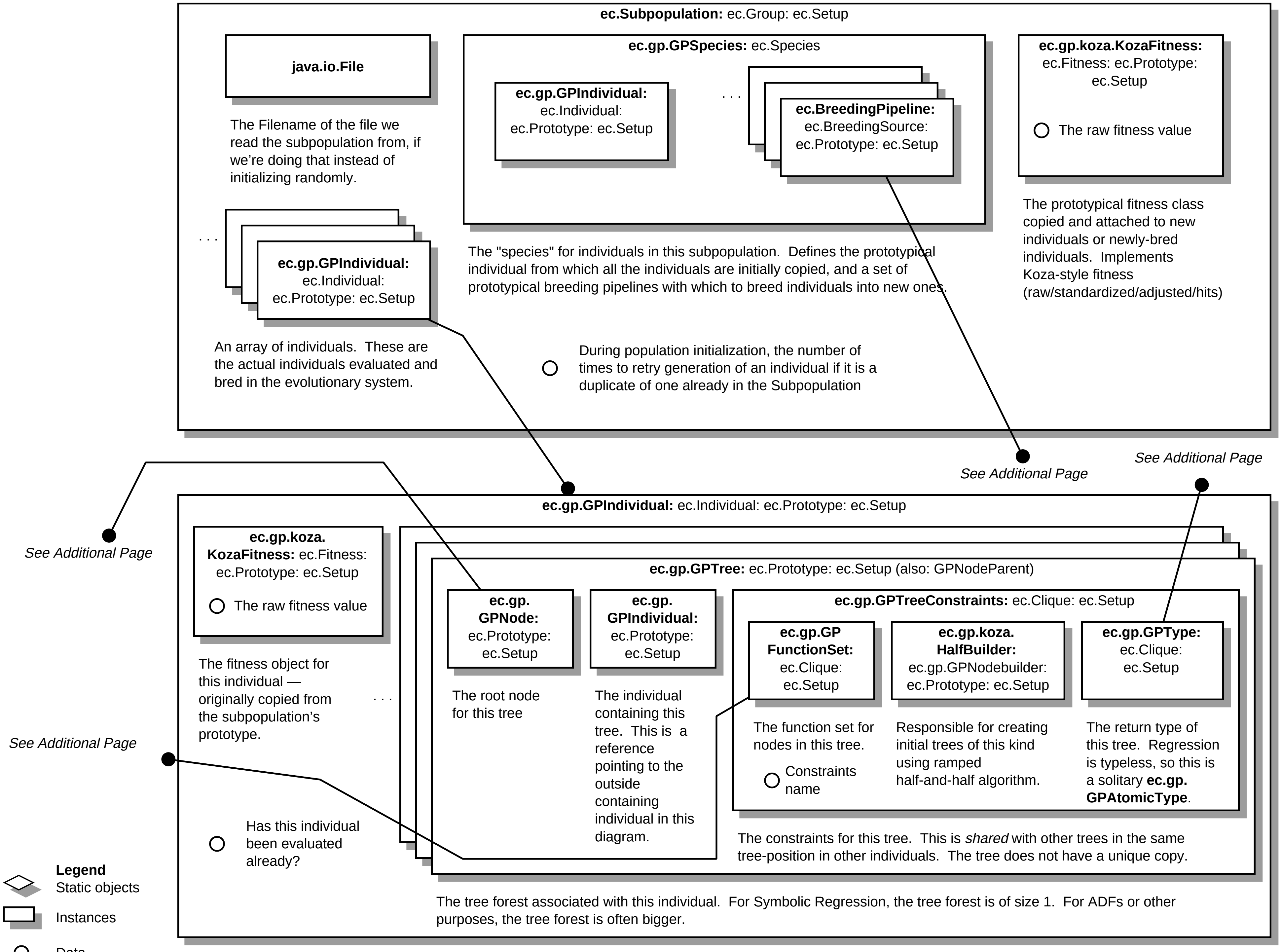

ec.app.regression. Regression: ec.gp.GPProblem: ec.Problem: ec.Prototype: ec.Setup
ec.app.regression. RegressionData:
ec.gp.GPData: ec.Prototype: ec.Setup
x: floating-point return value
ec.gp.ADFStack: ec.Prototype: ec.Setup
. . .
ec.gp.ADFContext:
ec.Prototype: ec.Setup
Stack
. . .
ec.gp.ADFContext:
ec.Prototype: ec.Setup
Reserve
. . .
ec.gp.ADFContext:
ec.Prototype: ec.Setup
Substack
ec.gp.ADFContext:
ec.Prototype: ec.Setup
Prototypical
ADF Context
ec.gp.ADFContext: ec.Prototype: ec.Setup
ec.gp.RegressionData:
ec.gp.GPData: ec.Prototype:
ec.Setup
x: floating-point return value
. . .
ec.gp.RegressionData:
ec.gp.GPData: ec.Prototype:
ec.Setup
x: floating-point return value
Return results
from ADF's
children
ec.gp.ADF:
ec.gp.GPNode:
ec.Prototype: ec.Setup
Prototypical Regression Data
The ADF Node that was
executed

The prototypical data object, copied once per thread, each time a generation of individuals needs to be evaluated, and passed between **ec.gp.GPNode**s to transfer data between then when they are executing. For Symbolic Regression, this object holds the floating-point value returned by functions.

A prototypical ADF Stack object, copied once per thread, each time a generation of individuals needs to be evaluated. ADF Stacks holds current contextual state of ADF function calls in two special stacks of **ec.gp.ADFContext** objects, one ADF Context per pending ADF function call. A prototypical ADF Context object is copied when new ADFContext objects are needed; old ADFContext objects are held in a Reserve to prevent excessive object construction. For more information on this complicated mechanism, see the class documentation. Since Symbolic regression doesn't have ADFs, this facility is left unused.

Legend
◇ Static objects
▭ Instances
○ Data

An ADF Context holds the current execution context of an ADF function call. The ADF ("Automatically Defined Function", see Koza-I and Koza-II) called is stored in the context, as is the returned data from its children nodes. A prototypical data object is also held from which new children results are created if necessary. This data object is typically the same kind of prototype as the one used in the problem, in this case, **ec.app.regression.Regression.** ADF Contexts are also used for ADMs ("Automatically Defined Macros"). Since Symbolic Regression doesn't have ADFs, this facility is left unused.



ec.Subpopulation: ec.Group: ec.Setup

java.io.File

The Filename of the file we read the subpopulation from, if we're doing that instead of initializing randomly.

ec.gp.GPIndividual: ec.Individual: ec.Prototype: ec.Setup

. . .

An array of individuals. These are the actual individuals evaluated and bred in the evolutionary system.

ec.gp.GPSpecies: ec.Species

ec.gp.GPIndividual: ec.Individual: ec.Prototype: ec.Setup

. . .

ec.BreedingPipeline: ec.BreedingSource: ec.Prototype: ec.Setup

The "species" for individuals in this subpopulation. Defines the prototypical individual from which all the individuals are initially copied, and a set of prototypical breeding pipelines with which to breed individuals into new ones.

○ During population initialization, the number of times to retry generation of an individual if it is a duplicate of one already in the Subpopulation

ec.gp.koza.KozaFitness: ec.Fitness: ec.Prototype: ec.Setup

○ The raw fitness value

The prototypical fitness class copied and attached to new individuals or newly-bred individuals. Implements Koza-style fitness (raw/standardized/adjusted/hits)

See Additional Page

See Additional Page

See Additional Page

ec.gp.GPIndividual: ec.Individual: ec.Prototype: ec.Setup

ec.gp.koza. KozaFitness: ec.Fitness: ec.Prototype: ec.Setup

○ The raw fitness value

The fitness object for this individual — originally copied from the subpopulation's prototype.

See Additional Page

○ Has this individual been evaluated already?

. . .

ec.gp.GPTree: ec.Prototype: ec.Setup (also: GPNodeParent)

ec.gp. GPNode: ec.Prototype: ec.Setup

The root node for this tree

ec.gp. GPIndividual: ec.Prototype: ec.Setup

The individual containing this tree. This is a reference pointing to the outside containing individual in this diagram.

ec.gp.GPTreeConstraints: ec.Clique: ec.Setup

ec.gp.GP FunctionSet: ec.Clique: ec.Setup

The function set for nodes in this tree.

○ Constraints name

ec.gp.koza. HalfBuilder: ec.gp.GPNodebuilder: ec.Prototype: ec.Setup

Responsible for creating initial trees of this kind using ramped half-and-half algorithm.

ec.gp.GPType: ec.Clique: ec.Setup

The return type of this tree. Regression is typeless, so this is a solitary **ec.gp. GPAtomicType**.

The constraints for this tree. This is *shared* with other trees in the same tree-position in other individuals. The tree does not have a unique copy.

The tree forest associated with this individual. For Symbolic Regression, the tree forest is of size 1. For ADFs or other purposes, the tree forest is often bigger.

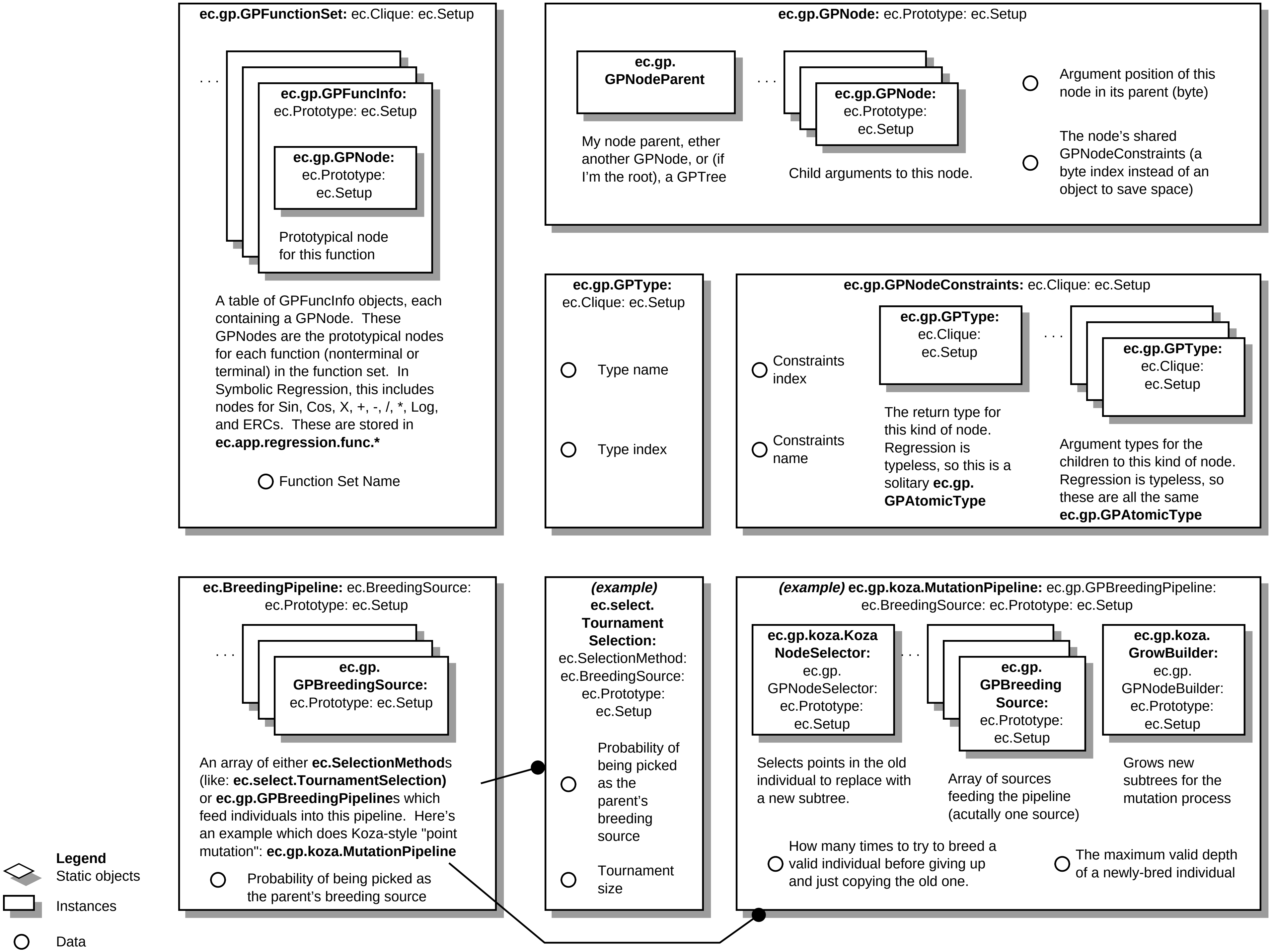
Legend

◇ Static objects

▭ Instances

○ Data

An individual is the fundamental unit of evolution — it is the thing that is bred, evaluated, and assigned a fitness.



ec.gp.GPFunctionSet: ec.Clique: ec.Setup

. . .

ec.gp.GPFuncInfo: ec.Prototype: ec.Setup

ec.gp.GPNode: ec.Prototype: ec.Setup

Prototypical node for this function

A table of GPFuncInfo objects, each containing a GPNode. These GPNodes are the prototypical nodes for each function (nonterminal or terminal) in the function set. In Symbolic Regression, this includes nodes for Sin, Cos, X, +, -, /, *, Log, and ERCs. These are stored in **ec.app.regression.func.***

○ Function Set Name

ec.gp.GPNode: ec.Prototype: ec.Setup

ec.gp. GPNodeParent

My node parent, ether another GPNode, or (if I'm the root), a GPTree

. . .

ec.gp.GPNode: ec.Prototype: ec.Setup

Child arguments to this node.

○ Argument position of this node in its parent (byte)

○ The node's shared GPNodeConstraints (a byte index instead of an object to save space)

ec.gp.GPType: ec.Clique: ec.Setup

○ Type name

○ Type index

ec.gp.GPNodeConstraints: ec.Clique: ec.Setup

○ Constraints index

○ Constraints name

ec.gp.GPType: ec.Clique: ec.Setup

The return type for this kind of node. Regression is typeless, so this is a solitary **ec.gp. GPAtomicType**

. . .

ec.gp.GPType: ec.Clique: ec.Setup

Argument types for the children to this kind of node. Regression is typeless, so these are all the same **ec.gp.GPAtomicType**

ec.BreedingPipeline: ec.BreedingSource: ec.Prototype: ec.Setup

. . .

ec.gp. GPBreedingSource: ec.Prototype: ec.Setup

An array of either **ec.SelectionMethod**s (like: **ec.select.TournamentSelection)** or **ec.gp.GPBreedingPipeline**s which feed individuals into this pipeline. Here's an example which does Koza-style "point mutation": **ec.gp.koza.MutationPipeline**

○ Probability of being picked as the parent's breeding source

(example) **ec.select. Tournament Selection:** ec.SelectionMethod: ec.BreedingSource: ec.Prototype: ec.Setup

○ Probability of being picked as the parent's breeding source

○ Tournament size

(example) **ec.gp.koza.MutationPipeline:** ec.gp.GPBreedingPipeline: ec.BreedingSource: ec.Prototype: ec.Setup

ec.gp.koza.Koza NodeSelector: ec.gp. GPNodeSelector: ec.Prototype: ec.Setup

Selects points in the old individual to replace with a new subtree.

. . .

ec.gp. GPBreeding Source: ec.Prototype: ec.Setup

Array of sources feeding the pipeline (acutally one source)

ec.gp.koza. GrowBuilder: ec.gp. GPNodeBuilder: ec.Prototype: ec.Setup

Grows new subtrees for the mutation process

○ How many times to try to breed a valid individual before giving up and just copying the old one.

○ The maximum valid depth of a newly-bred individual

Legend

Static objects

Instances

Data