```
 _____ _            _     _           _ _          ___   __
|_   _| |__   ___  | |   (_)_ __ ___ (_) |_ ___   / _ \ / _|
  | | | '_ \ / _ \ | |   | | '_ ` _ \| | __/ __| | | | | |_
  | | | | | |  __/ | |___| | | | | | | | |_\__ \ | |_| |  _|
  |_| |_| |_|\___| |_____|_|_| |_| |_|_|\__|___/  \___/|_|

                 _  __
                | |/ /___ _ __  _   _  __ _
                | ' // _ \ '_ \| | | |/ _` |
                | . \  __/ | | | |_| | (_| |
                |_|\_\___|_| |_|\__, |\__,_|
                                |___/


/* Stacks
   ------
   ------ */

/* Ideal interface
   ---------------
*/
<T> Stack<T> newStack() {
}

<T> boolean isEmpty(Stack<T> s) {
}

<T> T top(Stack<T> s) {
    assert !isEmpty(s) : "Cannot top an empty stack";
}

<T> T pop(Stack<T> s) {
    assert !isEmpty(s);
}

<T> void push(Stack<T> s, T item) {
}


/* Stack implementation 1: As a linked-list
   ----------------------------------------
note there are some /bugs/ in Kenya */

void main() {
                               //v-shouldn't need this (kenya bug)
    Stack<String> ns = newStack("");
    push(ns, "hello");
    push(ns, "world");
    println(pop(ns));
    println(top(ns));
    println(pop(ns));
    println(isEmpty(ns));
}

                     // v-shouldn't need this argument (kenya bug)
<T> Stack<T> newStack(T ignored) {
    Stack<T> s;
    return s;
}

<T> boolean isEmpty(Stack<T> s) {
```



```
    return s.node == null;
}

<T> T top(Stack<T> s) {
    assert !isEmpty(s) : "Cannot top an empty stack";
    return s.node.data;
}

<T> T pop(Stack<T> s) {
    assert !isEmpty(s) : "Cannot pop an empty stack";
    T data = s.node.data;
    s.node = s.node.next;
    return data;
}

<T> void push(Stack<T> s, T item) {
    StackNode<T> sn;
    sn.data = item;
    sn.next = s.node;
    s.node = sn;
}


class Stack<T> {
    StackNode<T> node;
}

class StackNode<T> {
    T data;
    StackNode<T> next;
}


/* Why is/Is the StackNode<T> class necessary?
    - what happens without it?              */



/* How about a different implementation, using Arrays?
    note we are now limited by Java in one respect: */

void main() {
    String[] ss = makeMeAnArray("hello", "world");
    println(ss[0]);
    println(ss[1]);
}

<T> T[] makeMeAnArray(T a, T b) {
                //v - Java doesn't know what the T is!
    T[] out = new T[2];
    return out;
}

/* Stack implementation 2: With arrays:
   ------------------------------------
*/

void main() {
    Stack<String> ns = newStack(new String[1]);
    push(ns, "hello");
    push(ns, "world");
    println(pop(ns));
```

```
    println(top(ns));
    println(pop(ns));
    println(isEmpty(ns));
}

<T> Stack<T> newStack(T[] buffer) {
    Stack<T> s;
    s.dataItems = buffer;
    s.size = 0;
    return s;
}

<T> boolean isEmpty(Stack<T> s) {
    return s.size == 0;
}

<T> T top(Stack<T> s) {
    assert !isEmpty(s) : "Cannot top an empty stack";
    return s.dataItems[s.size - 1];
}

<T> T pop(Stack<T> s) {
    assert !isEmpty(s) : "Cannot pop an empty stack";
    T data = top(s);
    s.dataItems[s.size - 1] = null;
    s.size = s.size - 1;
    return data;
}

<T> void push(Stack<T> s, T item) {
    assert s.size < s.dataItems.length : "Can't push onto a full stack";
            /*  In an ideal world we would create a new array that is longer
                than the original one, copy data over to that and use that.
                You can't do this in Kenya, there is a <hack> way of doing
                this in Java... */
    s.dataItems[s.size] = item;
    s.size = s.size + 1;
}


class Stack<T> {
    T[] dataItems;
    int size;
}


/* As an aside, you also can't do (in Kenya or Java at the moment)
    - modulo the Kenya bug */

void main() {
  String s = makeMeAT("");
}

              //  v - the argument bug back again
<T> T makeMeAT(T ignored) {
    T x;
    return x;
}


/* Other data structures...how about Haskell Trees? */
```



```
data Tree a = Leaf | Branch (Tree a) a (Tree a)

/* Could represent this (the data structure itself) as: */

class Tree<T> {
    TreeNode<T> node;
}

class TreeNode<T> {
    TreeNode<T> lhs;
    T data;
    TreeNode<T> rhs;
}

/* But what about? */

data Tree a = Leaf a | Branch (Tree a) a (Tree a)
                // ^ //

/* Could represent this as...

  ... erm ...

  Well a <horrible> solution:
*/

class Tree<T> {
    TreeNode<T> node;
}

enum TreeNodeType {
    Leaf, Node;
}
class TreeNode<T> {
    TreeNodeType type;
    TreeNode<T> lhs;
    T data;
    TreeNode<T> rhs;
}

/*  BUT DO *NOT* EVER EVER EVER EVER EVER EVER WRITE
    CODE LIKE THIS IN JAVA. EVER.
                          ----
*/


/*  We are reaching the limits of the expressivness of Kenya...

    In the haskell Both Leaves and Branches are Trees.
    In Kenya there can only be TreeNodes that are TreeNodes.


    ...

    So how did we make it work before we added data to the Leaf?

    By using null as an implicit Leaf.

*/

/* Some haskell tree method signatures */
```

```
mapTree :: (Tree a) -> (a -> b) -> (Tree b)

buildSortedTree :: (Ord a) => [a] -> Tree a


/*
  Can't do either of these (literally) in Kenya:

  mapTree requires the ability to pass a function around.
  Just not possible in Kenya.  In Java, its possible using REFLECTION
  (and hopefully in Java 7.0 it'll be nicer using CLOSURES).

  Note, if you had a Map data structure written in Kenya, you can write a
  similar in spirit method:
*/

mapTree :: (Tree a) -> (Data.Map a b) -> (Tree b)

/* Kenya interface */

<A,B> Tree<B> mapTree(Map<A,B> map, Tree<A> inTree) {}

<K,V> Map<K,V> newMap(K ignored, V ignored2) {}

<K,V> void addMapping(Map<K,V> map, K key, V value) {}

<K,V> V getValue(Map<K,V> map, K key) {}

class Map<K,V> {}

/*
    But how would you implement Map?

    Well you could create a lookup list
*/
class Map<K,V> {
    List<Pair<K,V>> mappings;
}

/*
  The mechanism is then very similar to haskell builtin
*/

Prelude> :t lookup
lookup :: (Eq a) => a -> [(a, b)] -> Maybe b

/*
  buildSortedTree has a constraint on the 'a' that gives
  knowlege about the capabilities of the 'a'.  In Kenya
  there is no way to place those constraints.
  (There is also no way to generate a 'type' with capabilities
  separate from their type.

  In Java there are several ways (SUBCLASSING, using INTERFACES)
*/


{-- Haskell reminder on constraints --}

-- From the prelude:
class Eq a where
  (==) :: a -> a -> Bool
```



```
  (/=) :: a -> a -> Bool

  a /= b = not (a == b)
  a == b = not (a /= b)

-- My data
data Tree a = Leaf | Branch (Tree a) a (Tree a)

instance Eq a => Eq (Tree a) where
  Leaf == Leaf                     = True
  Branch l v r == Branch l' v' r' = (l == l') &&
                                     (v == v') &&
                                     (r == r')
  _ == _                           = False

{-- Or just:
data Tree a = Leaf | Branch (Tree a) a (Tree a)
  (deriving Eq)



{--
Random haskell syntax reminder that's needed for the next bit:

\a ->  notation introduces an anonymous function in Haskell.
(a function where we don't care about its name.

So
--}

Prelude> (\a -> a + 1) 3
4

Prelude> (\x y -> x ++ y) [1] [2,3,4]
[1,2,3,4]


/* Types and kinds
(dipping into type theory - an aside)

Another way to think about generics in Kenya / Type parameters in haskell involv
es
thinking about the 'type of types'.

Using haskell:


True is a value.  It has type Bool.
  Prelude> :t True
  True :: Bool

3 is a value.  It has type Integer (ish)
  Prelude> :t 3
  3 :: (Num t) => t

(\a -> a || False ) is a value (function).  It has type (Bool -> Bool)
  Prelude> :t (\a -> a || False)
  (\a -> a || False) :: Bool -> Bool

Just False is a value.  It has type "Maybe Bool"
  Prelude> :t Just False
  Just False :: Maybe Bool
```

```
Types can also have types.  These are called 'kinds'.

Bool is a type.  It has kind *
  Prelude> :k Bool
  Bool :: *

Integer is a type.  It has kind *
  Prelude> :k Integer
  Integer :: *

Bool -> Bool is a type.  It has kind *
  Prelude> :k (Bool -> Bool)
  (Bool -> Bool) :: *

(Maybe Bool) is a type.  It has kind *
  Prelude> :k Maybe Bool
  Maybe Bool :: *

Maybe is...something else.  It has kind * -> *
  Prelude> :k Maybe
  Maybe :: * -> *

alternatively think of "Maybe" as (\a -> Maybe a)
so Maybe Bool is ((\a -> Maybe a) Bool)


Similarly in Kenya this then becomes:

true is a value.  It has type boolean.

Pair<Integer,Integer> x
x.a = 3
a.y = 4
x is a value.  It has type Pair<Integer, Integer>

(\A B -> Pair<A,B>) is a type.  It has kind * -> * -> *


What is the type of kinds?

*/


/*
In Kenya/Java there is a difference between 'primitive' types and 'reference' ty
pes.

null is a value that is a subtype of all reference types

many of the primitive types have reference equivalents.

int -- Integer
boolean -- Boolean
double -- Double

In Java generics can't use primitive types.  The kenya translator automatically
does
this translation for you.

There is also a notion of "autoboxing" in Java/Kenya, which will turn a primitiv
```



```
e value into its associated reference value:
*/

void main() {
    println(usesAnInt(3));
    println(usesAnInteger(3));
    println(usesAnInteger(null));
}

Integer usesAnInt(int x) {
    return x;
}

int usesAnInteger(Integer x) {
    return x;
}

/* Just be careful of null... causing NullPointerExceptions.  aka NPE's.  You wi
ll see
    a lot of these in the future. */


/* Finally: Kenya / Java has automatic promotion between ints and doubles.
    However generic parameters do not support promotion / conversion.
    This will be covered in a lot more depth next term.
*/

void main() {

    int ii = 3;
    double dd = ii;

    Holder<Integer> intHolder = newHolder(3);
    Holder<Double> doubleHolder = newHolder(3.3);

    /*
      doubleHolder = intHolder;

      This line is not allowed.  Otherwise you could do:

    put(doubleHolder, 3.3);
    int y = get(intHolder);   // BOOM!
    */
}

<T> Holder<T> newHolder(T value) {
    Holder<T> a;
    a.a = value;
    return a;
}

<T> void put(Holder<T> holder, T value) {
    holder.a = value;
}

<T> T get(Holder<T> holder) {
    return holder.a;
}

class Holder<T> {
    T a;
}
```