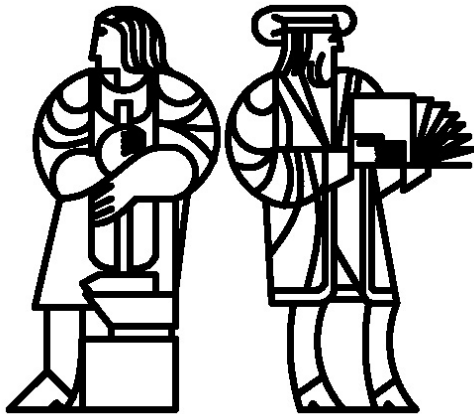




Introduction

MIT 6.170



MIT EECS



Course staff

Lecturers

Saman Amarasinghe
Michael Ernst

TAs

C. Scott Ananian
Natan Cliffer
Philip Guo
Tucker Sylvestro
Matthew Tschantz
Vincent Yeung

LAs

David Glasser
Katherine Hollenbach
Eric Lieberman
Scott Ostler
Clayton Sims
Robert Toscano
Stephanie Yeh

Continuous testing support

Arjun Dayal
David Saff

Ask us for help!



Main Topics Covered by 6.170: Managing Complexity

Abstraction and specification

Procedural, data, control flow

Why they are useful and how to use them

Writing, understanding, and reasoning about code

Examples in Java, but the issues are more general

Object-oriented programming

Program design and documentation

What makes a design good or bad (example: modularity)

The process of design and design tools

Pragmatic considerations

Testing

Debugging and defensive programming

Managing software projects

Teamwork



The goal of system building

To create a correctly functioning artifact!

All other matters are secondary

However, many of them are *essential* to producing a correct system

We insist that you learn to create correct systems



Why Is Building Good Software Hard?

Large software systems enormously complex

Millions of “moving parts”

People expect software to be malleable

After all, it’s “only software”

We are always trying to do new things with software

Relevant experience often missing

Software engineering is about:

Managing complexity

Managing change

Coping with potential defects

Customers, developers, environment, software



Programming is hard

It is surprisingly difficult to specify, design, implement, test, debug, and maintain even a simple program

6.170 will challenge you

If you are having trouble, *think* before you act

Then, look for help

We strive to create assignments that are reasonable if you apply the techniques taught in lecture

... but hard to do in a brute-force manner

Knowing Java is *not* a prerequisite for 6.170

We will use Java 1.5 (aka Java 5.0) this semester



Logistics

Website: <http://www.mit.edu/~6.170/>

See the website for all administrative details

Run `student-setup.pl` by 9pm tonight

Collaboration policy:

Discussion permitted

Everything you turn in must be your own work

You may not view others' work



Java Semantics: Objects

Declarations, assignments, method calls

Java primitives vs. objects

Variables vs. objects; null references

Mutable and immutable objects

Sharing

Equality testing

Intro to classes (more next time)



Java and Scheme

Java is not Scheme, but has similar semantics

Garbage-collected, call-by-value

Some important differences

Statically typed (discover errors earlier)

Built-in support for object-oriented programming

Wider acceptance

You will need to learn

Syntax, libraries, OO programming



Two Kinds of Values

Primitives: `int`, `float`, `boolean`, `char`, ...

Created by writing literals, e.g., `3` or `true`

Objects: `String`, `PrintStream`, `ArrayList`, ...

Composed of other values

Created by calling a constructor (sometimes implicit)

```
new ArrayList<Date>(); // empty sequence
```

Exception: **Strings** are created via a literal: `"6.170"`

Operations are associated with the type, e.g.,

`+` with `int`

`add(Date)` with `ArrayList<Date>`



Variables and Assignment

Variables exist in program text, e.g., `int x`

Environment binds variables either to

Primitive value (e.g., `1` or `"abc"`) or to object reference

Objects exist at run time

Containers for primitive values or objects

Assignment binds identifier to object

Changes environment

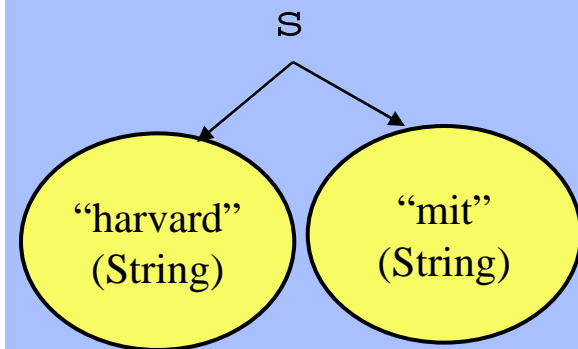
Does not change value of object

Objects have a type

Governs what can be done with object

Including valid assignments

```
String s;  
s = "harvard";  
s = "mit";
```





Declarations, Assignments and Method Calls

Declaration creates a new variable

```
String a;
```

Assignment binds a variable to a value

```
a = "mit";
```

Evaluate an assignment by evaluating each side in turn; "mit" is a literal that evaluates to a String object

Method call invokes a procedure

```
System.out.println(a);
```

System.out is of type PrintStream

```
String b = a.toUpperCase();
```

toUpperCase takes a String, returns a String

```
System.out.println(b);
```

```
int i = a.indexOf('i');
```

1 (indexing starts at 0)

```
int j = b.indexOf('i');
```

-1 (by the Java spec)

Methods are invoked on objects (**this**, a.k.a. the receiver)



A Few Important Types

int - the integers you learned about in grade school (almost)

Primitive values, not objects

Integer - container for ints

ArrayList - sequence of containers

More precisely, a container whose element is a sequence of containers

What happens when one compiles this code?

```
int i = 3;
Integer iobj = new Integer(i);
ArrayList<Integer> al = new ArrayList<Integer>();
al.add(iobj);
al.add( new Date() );
al.add(i);
```

Compile-time error

Autoboxing:

```
al.add(new Integer(i));
```



Variables, References, and Assignment

A variable is declared to hold:

a primitive value, or
a reference to an object

Uninitialized variables

```
String a;  
System.out.println(a);  
String b = a.toUpperCase();
```

a holds no reference

prints "null" (because println checks)

throws NullPointerException:
receiver must be an object



Mutable and Immutable Objects

Immutable objects: can never be changed after creation

```
String tute = "mit";  
tute.toUpperCase();  
tute = tute.toUpperCase();
```

has no effect; result is discarded

Mutable objects: can be changed

```
Date now = new Date();  
...  
System.out.println(now);  
now.setMonth(1);  
System.out.println(now);
```

Thu Jul 04 1776

Tue Jun 04 1776

Assignment changes the variable binding

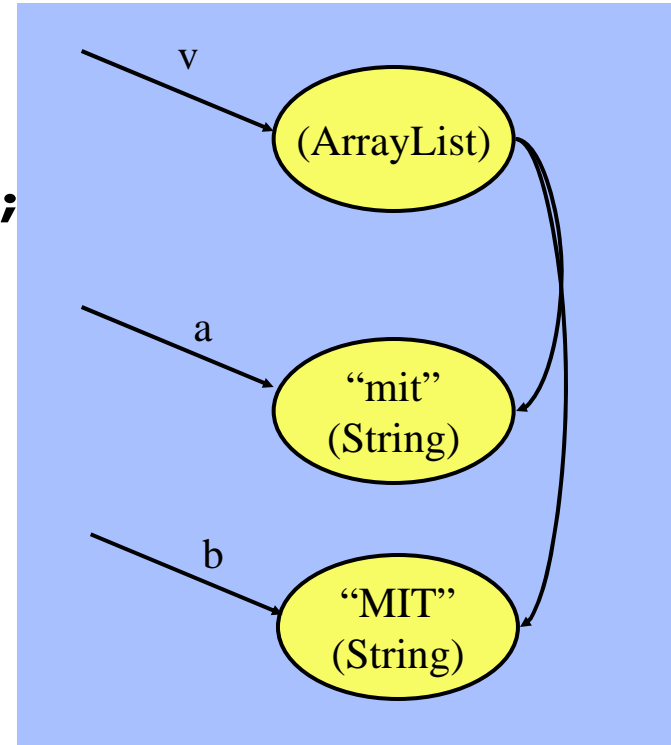
Mutation changes the object



Mutable Objects

What happens when you run this?

```
ArrayList<String> v
    = new ArrayList<String>();
String a = "mit";
String b = "MIT";
v.add(a);
System.out.println(
    v.lastElement());
v.add(b);
System.out.println(
    v.lastElement());
```



Why do languages have (im)mutable types?

immutable types simplify reasoning;
mutable types can increase efficiency



Equality

Object (reference) equality: `==` (like Scheme `eq`)

Value equality: `equals` (like Scheme `equal`)

```
if (v1 == v2) System.out.println("same object");  
if (v1.equals(v2)) System.out.println("same value");
```

Should `(x == y)` imply `x.equals(y)` Yes: same object => same value

Should `x.equals(y)` imply `(x == y)` No: same value $\not\Rightarrow$ same object



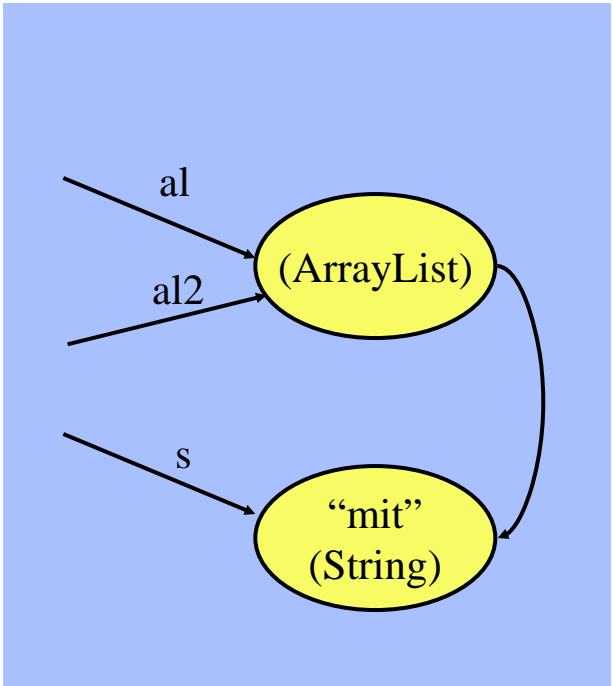
Aliasing

What about this?

```
ArrayList<String> a1  
    = new ArrayList<String>();  
ArrayList<String> a12 = a1;  
String a = "mit";  
a1.add(a);  
System.out.println(  
    a12.lastElement());
```

a1 and a12 are aliased

mit



What if we now do this?

```
if (a1 == a12)  
    System.out.println("same object")  
if (a1.equals(a12))  
    System.out.println("same value");
```

same object
same value



Declaring a new type of Object:

```
class Pol {  
    String name;  
    boolean inOffice;  
}
```

Creating a new object (instance) of the new type

```
Pol p1 = new Pol();  
p1.name = "Mitt Romney";  
p1.inOffice = true;  
  
// A better way to instantiate:  
Pol p2 = new Pol("Jane Swift", false);
```



Adding methods: toString and outranks

```
class Pol {  
    String name;  
    boolean inOffice;  
    String toString() {  
        if (inOffice) {  
            return "The Honorable " + name;  
        } else {  
            return name;  
        }  
    }  
    boolean outranks(Pol p) {  
        return this.inOffice && (! p.inOffice);  
    }  
}  
System.out.println(p2);  
System.out.println(p2.outranks(p1));
```

toString is used by println; previously, println used a default for Pol; toString really needs a "public" modifier

"inOffice" and "name" refer to fields in the receiver object

"this.inOffice" is equivalent to "inOffice"

The Honorable Mitt Romney

true

And if we didn't supply toString?

Some junk



Compile-time and Run-time Typing

Consider the code

```
Integer i = new Integer(3);
Number n = i;
List<Integer> intlist = new ArrayList<Integer>();
List<Number> numlist = new ArrayList<Number>();
intlist.add(i);
intlist.add(n);
numlist.add(i);
numlist.add(n);
int x = i.intValue();
int y = n.intValue();
```

OK
compile error
OK
OK

Both lines invoke the same version of intValue

What happens when one compiles it?

What happens when one runs it?

What is the subtyping relationship between ArrayList<Integer> and List<Integer>?

What is the subtyping relationship between ArrayList<Integer> and ArrayList<Number>?



Key Java Concepts in Lecture

Variables hold

References to objects

Primitive values, e.g., 6

Sharing, equality & mutability

An object can be mutable (state may change) or immutable

Two variables can point to the same object

Sharing can be useful for mutable objects

Compile-time & run-time types

An object has a type at run time

A variable has a declared type at compile time

Methods are associated with types

Run-time type is a specialization of compile-time type