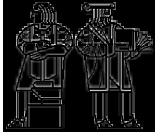


6.170 Lecture 2 Modularity, Types & Objects



John Guttag
MIT EECS

Achieving Modularity

Reduce and order complexity

Using decomposition and abstraction

Decompose programs into units s.t.

Each has independent requirements
Interactions limited in complexity
Clear how to implement and test each
Can be combined to solve original problem

Why is this not enough

Dreaded system integration failure

Abstraction Makes Decomposition Work

The design loop

Abstract to get a simpler problem
Decompose the simpler problem
Define the interface between modules
Apply process recursively to new problem

Over-simplification

Never strictly top-down

Next week's topic

Decomposition and Abstraction in Java

Variables and objects

Today

Types and classes

Packages and interfaces

Variables and Objects

What happens when you run this?

```
String a = "mit";  
System.out.println (a);
```

It prints mit

What is "mit?"

String literal that evaluates to a String object

What is a?

A variable whose value is an object reference

What is String a = "mit?"

A declaration and an assignment in one

Assignment

Variables in program text

Environment binds variables to

Primitive values, e.g., 1 or "abc"
Object (variable contains a reference to an object)

Objects exist at runtime

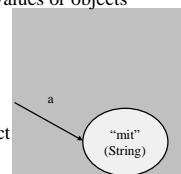
Think of as containers for primitive values or objects

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



A Few Interesting Types

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

Integer - containers for ints

Vector - sequences of containers

What happens when one compiles

```

\\Lect3Ex1
int i = 3;
Integer i1 = new Integer (i);
String s = new String ("ABC");
Vector v = new Vector ();
v.addElement (i1);
v.addElement (s);
v.addElement (i);
    
```

© John Chapin/John Guttag Spring 2000 Slide 7

Typing in Java

Java is *strongly typed*
Language enforces rules
Mostly, but not entirely, at compile-time

Type of a variable known at compile time
Governs what types of objects can be assigned to it
As we just saw

Type of an object may not be known until runtime
Governs methods available for manipulating it


© John Chapin/John Guttag Spring 2000 Slide 8

Compile Time and Runtime Typing

Consider the code

```

//Lect2Ex2
Integer i1 = new Integer (3);
String s = new String ("ABC");
Vector v = new Vector ();
v.addElement (i1);
v.addElement (s);
i1 = (Integer) v.elementAt(0);
i1 = (Integer) v.elementAt(1);
    
```



What happens when one compiles it?

What happens when one runs it?

Why is this disturbing?

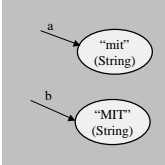
© John Chapin/John Guttag Spring 2000 Slide 9

Method Calls

What happens when you run this?

```

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



It prints MIT

toUpperCase a method of class **String**
type is String ♦ String
declared as public String toUpperCase ()

a.toUpperCase () is a **method call on the object a**
First argument (*a* in this case) has a special status

Does it change a?
No, it creates a new string
A function with no side-effect

© John Chapin/John Guttag Spring 2000 Slide 10

Null References

What happens when you run this?

```

String a = null;
System.out.println (a);
    
```

It prints null
Signifying that *a* is not bound to any object

What happens when you run this?

```

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

It throws a NullPointerException
Because a method call must have an object

© John Chapin/John Guttag Spring 2000 Slide 11

Sharing & Equality

What happens when you run this?

```

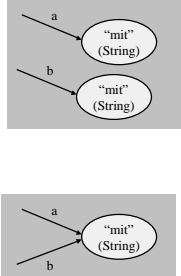
String a = "mit";
String b = "mit";
System.out.println (b);
    
```

It prints mit

Consider the following

```

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



© John Chapin/John Guttag Spring 2000 Slide 12

6.170 **Mutable and Immutable Objects**

Depends upon type

Immutable objects
 Given a value at time of creation
 Can never be changed

Mutable objects
 Can be changed at during execution
 Not to be confused with assignment

Assignment changes binding of variable to object
Mutation changes binding of object to value

© John Chapin/John Guttag Spring 2000 Slide 13

6.170 **Mutable Objects**

What happens when you run this?

```
Vector v = new Vector ();
String a = "mit";
String b = "MIT";
v.addElement (a);
System.out.println (v.lastElement ());
v.addElement (b);
System.out.println (v.lastElement ());
```

It prints
 mit
 MIT

The diagram shows three variables: v, a, and b. Variable v points to a Vector object containing the elements 'mit' and 'MIT'. Variable a points to the String object 'mit', and variable b points to the String object 'MIT'.

© John Chapin/John Guttag Spring 2000 Slide 14

6.170 **Aliasing**

What about this?

```
Vector v = new Vector ();
Vector q = v;
String a = "mit";
v.addElement (a);
System.out.println (q.lastElement ());
```

It prints mit
Because v and q are aliased

What if we now do this?
 if (v == q) System.out.println ("same object");
 if (v.equals (q)) System.out.println ("same value");

It prints
 same object
 same value

The diagram shows three variables: v, q, and a. Variable v points to a Vector object containing the element 'mit'. Variable q also points to the same Vector object, illustrating aliasing. Variable a points to the String object 'mit'.

© John Chapin/John Guttag Spring 2000 Slide 15

6.170 **Aliasing, cont.**

What about this?

```
Vector v = new Vector ();
Vector v1 = new Vector ();
Vector q = v;
if (v == q) System.out.println ("same object");
if (v.equals (q)) System.out.println ("same value");
if (v == v1) System.out.println ("same object");
if (v.equals (v1)) System.out.println ("same value");
```

It prints
 same object
 same value
 same value

© John Chapin/John Guttag Spring 2000 Slide 16

6.170 **An Informal Test**

Should (x == y) imply x.equals(y)?

Is the following a reasonable thing to do?
 v.addElement (5);

Is aliasing a meaningful notion for immutable types?

Why do language have immutable types?

What does the following fragment do?

```
//Lect2Ex3
Vector v = new Vector ();
v.addElement (v);
System.out.println (v.lastElement ());
```

© John Chapin/John Guttag Spring 2000 Slide 17

6.170 **Key Concepts**

Variables hold
 References to objects
 Primitive values, e.g., 5

Sharing, equality & mutability
 An object can be mutable (state may change) or immutable
 Two variables can point to the same object
 Sharing useful only for mutable objects

Compile-time & Runtime types
 An object has a type at runtime
 A variable has a declared, compile-time type
 Runtime type is a specialization of compile-time type

© John Chapin/John Guttag Spring 2000 Slide 18