

## A Very Brief Introduction to Java

16.410-13  
Paul Robertson

## Things you need

- Sun Tutorial
  - <http://java.sun.com/docs/books/tutorial/information/download.html>
- Java: java (j2sdk-1\_4\_2\_09-windows-i586-p.exe)
  - <http://java.sun.com/j2se/1.4.2/download.html>
- Eclipse: (eclipse-SDK-3.1-win32.zip)
  - <http://eclipse.org/downloads/index.php>
- Junit: (junit3.8.1.zip)
  - <http://www.junit.org/index.htm>
- Online API reference
  - <http://java.sun.com/j2se/1.4.2/docs/api/index.html>
- Java in a Nutshell – a reference book

## Java vs. 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, char, boolean, ...

Created by writing literals, e.g., 3 or "a"

### Objects: String, PrintStream, ArrayList, ...

Composed of other values  
Created by calling a constructor (sometimes implicit)  
`new ArrayList<Date>();` // empty sequence

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

+ with `int`  
`addElement(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

### Objects exist at runtime

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

## Declarations, Assignments and Method Calls

### Declaration creates a new variable

```
String a;
```

### Assignment binds a variable to a value

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

### Method call invokes a procedure

```
System.out.println(a);  
String b = a.toUpperCase();  
System.out.println(b);  
int i = a.indexOf('i');  
int j = b.indexOf('i');
```

### Methods are invoked on objects (this)

## A Few Interesting Types

**int** Primitive values, not objects

**Integer** - containers for ints

**ArrayList** - sequences of containers

**What happens when one compiles ?**

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

## Mutable and Immutable Objects

**Immutable objects: can never be changed after creation**

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

**Mutable objects: can be changed**

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

**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.addElement(a);
System.out.println(v.lastElement());
v.addElement(b);
System.out.println(v.lastElement());
```

**Why do languages have (im)mutable types?**

## Equality

**Object (reference) equality: ==**

**Value equality: equals**

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

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

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

## Aliasing

**What about this?**

```
ArrayList<String> al = new ArrayList<String>();
ArrayList<String> al2 = al;
String a = "mit";
al.addElement(a);
System.out.println(al2.lastElement());
```

**al and al2 are *aliased***

**What if we now do this?**

```
if (al == al2) System.out.println("same object");
if (al.equals(al2)) System.out.println("same value");
```

## New types

### 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 = "Kerry Healey";
p1.inOffice = false;
Pol p2 = new Pol("Mitt Romney", true);
```

## 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));
```

### And if we didn't supply toString?

## Bank account system

Will represent bank accounts and transactions

### A class representing (modeling) a transaction:

```
class Transaction {
    int amount;
    Date date;
    Transaction(int amount, Date date) {
        this.amount = amount;
        this.date = date;
    }
    Transaction(int amount) {
        this(amount, new Date());
    }
}
```

### Use of the class:

```
Transaction t = new Transaction(100);
```

## A bank account

```
class Account {
    String name;
    List<Transaction> transactions = new ArrayList<Transaction>();
    int balance = 0;
    Account(String name) {
        this.name = name;
    }
    boolean check(Transaction t) {
        return (balance + t.amount) >= 0;
    }
    void post(Transaction t) {
        if (check(t)) {
            transactions.add(t);
            balance += t.amount;
        }
    }
}
```

## Use of the class

```
Account copAcct = new Account("Copperfield");
Transaction t1 = new Transaction(100);
copAcct.post(t1);
```

## Subclassing

```
class OverdraftAccount extends Account {
    int creditLimit;
    OverdraftAccount(String name, int limit) {
        super(name);
        creditLimit = limit;
    }
    boolean check(Transaction t) {
        return (balance + t.amount) >= -creditLimit;
    }
    void improveCredit(int amount) {
        creditLimit += amount;
    }
}
```