Java Bootcamp 2004

Expectations:
We assume that you have studied C, C++, or Java and know about the following:
• Variables and variable declarations
• Expressions (integer, boolean)
• Assignment statement
• If-statement and if-else statement
• While-loop and for-loop

DrJava
We use the IDE (Interactive Development Environment) DrJava in this course. We use it to demo during lecture. Its Interactions pane allows us to evaluate expressions and execute statements (including method calls) without having to have a complete Java application.

If you have your own computer, please get on the course website, download DrJava, and practice using it. Use it to learn about Java.

We’ll use it in this bootcamp.

We concentrate on the following Java constructs:
• Primitive types, variables, expressions
• Casting between types
• The class as a definition of the format of an object (instance, manilla folder)
• The new-expression
• Referencing instance variables and methods
• Methods (procedures, functions, constructors)
• Subclasses, inheritance, and overriding

Resources for learning Java
See website for reading material
• ProgramLive, by Gries & Gries. Has a CD, which has 250 2-4-minute lectures with synched animation. Used in CS100J this year. The glossary of the CD is a good source of information.
• Course textbook.
• Java Precisely.
• Java in a Nutshell.
• Java tutorial: http://java.sun.com/docs/books/tutorial/

Primitive types

<table>
<thead>
<tr>
<th>type</th>
<th>range of values</th>
<th>space used</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte</td>
<td>−128..127</td>
<td>1 byte</td>
</tr>
<tr>
<td>short</td>
<td>−32768..32767</td>
<td>2 bytes</td>
</tr>
<tr>
<td>int</td>
<td>−2³¹..2³¹</td>
<td>4 bytes ***</td>
</tr>
<tr>
<td>long</td>
<td>−2⁶³..2⁶³</td>
<td>8 bytes</td>
</tr>
<tr>
<td>float</td>
<td>6 significant digits, 10⁻⁴⁶..10³⁸</td>
<td>4 bytes</td>
</tr>
<tr>
<td>double</td>
<td>15 sig. digits, 10⁻³²⁴..10³⁰⁸</td>
<td>8 bytes ***</td>
</tr>
<tr>
<td>char</td>
<td>Unicode character</td>
<td>2 bytes ***</td>
</tr>
<tr>
<td>boolean</td>
<td>(false, true)</td>
<td>1 bit ***</td>
</tr>
</tbody>
</table>

Use mainly these types

<table>
<thead>
<tr>
<th>type</th>
<th>range of values</th>
<th>space used</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>−2³¹..2³¹</td>
<td>4 bytes</td>
</tr>
<tr>
<td>double</td>
<td>15 sig. digits, 10⁻³²⁴..10³⁰⁸</td>
<td>8 bytes</td>
</tr>
<tr>
<td>char</td>
<td>Unicode character</td>
<td>2 bytes</td>
</tr>
<tr>
<td>boolean</td>
<td>(false, true)</td>
<td>1 bit or 1 byte</td>
</tr>
</tbody>
</table>

Operations on type int

− h
h + k    h − k
h * k    h / k    h % k
h / k yields an int: 7 / 2 is 3!!!!
h % k is the remainder when h is divided by k.
min, max values for primitive types

- Short.MAX_VALUE: smallest short value
- Short.MIN_VALUE: largest short value
- Integer.MAX_VALUE: smallest int value
- Integer.MIN_VALUE: largest int value
- Double.MAX_VALUE: smallest POSITIVE double value
- Double.MIN_VALUE: largest POSITIVE double value

etc.

Type boolean

- Values: true and false
- Complement: ! b
- And (conjunction): b & & c
- Or (disjunction): b || c

Value: if b is false, then false; otherwise, whatever c is.

SHORT-CIRCUIT EVALUATION

```
x = 0 || 5 / x = 1 is true
5 / x = 1 || x = 0 GIVES AN EXCEPTION
```
### Casting

#### narrowest type \rightarrow widest type

- `byte` \rightarrow `short` \rightarrow `int` \rightarrow `long` \rightarrow `float` \rightarrow `double`

There are no operations in types `byte` and `short`. If they appear as operand of an operation, they are promoted (automatically cast) to `int` or `long` and the operation is performed in `int` or `long`.

If one operand of `x + y` is `long`, the other is cast to `long` and a `long` addition is done. Otherwise, the operation is an `int` addition.

---

### Type char

#### narrowest type \rightarrow widest type

- `char` \rightarrow `int` \rightarrow `long` \rightarrow `float` \rightarrow `double`

Values of type `char`: the characters,
- `'b'`, `'5'`, `'&'`
- `'\n'`: new-line character
- `'\'`: backslash char

*(int) 'A'* is the integer that represents char ‘A’: 65
*(char) 65* is the character that is represented by 65: ‘A’

You don’t need to remember much about type `char`. Just that it exists, and you can look it up whenever you want.

Best reference: ProgramLIVE.

---

### Class String

Mentioned now because you may hear about it from time to time. An object of class `String` is a sequence of `chars`:

```
"abcd123\n"
```

**Note:** double quotes for `String`, single quotes for `char`

- `1 + "abc" + 2.1` is `"1abc2.1"`
- `1 + 2 + "abc"` is `3 + "abc"` is `"3abc"

If at least one operand of `+` is a `String`, then `+` denotes “catenation”. The other operand is converted to a `String`, if necessary.

- `s.length()` number of characters in `String` `s`
- `s.charAt(i)` character at position `i` of `s` (0 is first)
- `s.substring(h,k)` substring `s[h..k-1]`

---

### The Class

- All variables, methods (procs, funcs) are declared in a **class**.
- **Class def.** defines format of objects (instances) of the class.

```java
public class C {
    // Declaration of instance variable (field) x;
    // Declaration of instance variable (field) y;
    // Declaration of instance method m1(int);
    // Declaration of instance method m2();
}
```

#### class name

**Tab contains name of object (address in memory)**

**Object drawn like a manilla folder**

---

### The Class

**public components can be referenced anywhere; private ones only in the class itself.**

Generally, but not always, fields are private and methods are public.
Three kinds of method

- **public void proc(par. decs) {**
  - body
- **public int func(par. decs) {**
  - body
- **public C(par. decs) {**
  - body

**body:** sequence of statements and declarations.

The new-expression

- **new C1(5)**

1. Create an object of class C1 (that's what “new C1” says), initializing fields acc. to their declarations
2. Execute constructor call C1(5).
3. Yield as the value of the new-expression the name of the object.

```
public class C1 {
    private int x = 4;
    private double y = 2.0;
    public void m1(int) { ... }
}
```

3. **Yield as the value of the new-expression the name of the object.**

The new-expression –used in an assignment

```
public class C1 {
    private int x = 4;
    private double y = 2.0;
    public void m1(int) { ... }
    public C1(int p) { x = p; }
}
```

```
C1 c;
... c = new C1(5);
```

```
public class C1 {
    private int x = 4;
    private double y = 2.0;
    public void m1(int) { ... }
    // Constructor: an instance with x field equal to p
    public C1(int p) { x = p; }
}
```

```
c = new C1(5);
```

```
c = a0
```
Our analogy for explaining classes and objects

Class is a file drawer: contains the objects (manilla folders) of the class.

public class C1 {
    private int x = 4;
    private double y = 2.0;
    /** field x */
    ... a good software engineering technique.
The example to the right shows conventions for naming and specifying getter methods.

Static components

public class CS {
    private static int c = 4;
    private int s = 2.0;
    public static double m1(int) {
        ... a file drawer:
        Only ONE copy of each static component.
    }
}

Our analogy for explaining classes and objects

public class C1 {
    private int x;
    private double y;
    public void m1(int) {
        ... a file drawer C1
        Class is a file drawer: contains the objects (manilla folders) of the class.
    }
}

Default constructor

public class C1 {
    private int x = 4;
    private double y = 2.0;
    /** field x */
    ... a good software engineering technique.
The example to the right shows conventions for naming and specifying getter methods.

static components

public class CS {
    private static int c = 4;
    private int s = 2.0;
    public static double m1(int) {
        ... a file drawer C1
        Class is a file drawer: contains the objects (manilla folders) of the class.
    }
}

Our analogy for explaining classes and objects

public class C1 {
    private int x;
    private double y;
    public void m1(int) {
        ... a file drawer C1
        Class is a file drawer: contains the objects (manilla folders) of the class.
    }
}

Default constructor

public class C1 {
    private int x = 4;
    private double y = 2.0;
    /** field x */
    ... a good software engineering technique.
The example to the right shows conventions for naming and specifying getter methods.

Static components

public class CS {
    private static int c = 4;
    private int s = 2.0;
    public static double m1(int) {
        ... a file drawer C1
        Class is a file drawer: contains the objects (manilla folders) of the class.
    }
}

Our analogy for explaining classes and objects

public class C1 {
    private int x;
    private double y;
    public void m1(int) {
        ... a file drawer C1
        Class is a file drawer: contains the objects (manilla folders) of the class.
    }
}

default constructor

public class C1 {
    private int x = 4;
    private double y = 2.0;
    /** field x */
    ... a good software engineering technique.
The example to the right shows conventions for naming and specifying getter methods.

Static components

public class CS {
    private static int c = 4;
    private int s = 2.0;
    public static double m1(int) {
        ... a file drawer C1
        Class is a file drawer: contains the objects (manilla folders) of the class.
    }
}

Our analogy for explaining classes and objects

public class C1 {
    private int x;
    private double y;
    public void m1(int) {
        ... a file drawer C1
        Class is a file drawer: contains the objects (manilla folders) of the class.
    }
}

default constructor

public class C1 {
    private int x = 4;
    private double y = 2.0;
    /** field x */
    ... a good software engineering technique.
The example to the right shows conventions for naming and specifying getter methods.

Static components

public class CS {
    private static int c = 4;
    private int s = 2.0;
    public static double m1(int) {
        ... a file drawer C1
        Class is a file drawer: contains the objects (manilla folders) of the class.
    }
}
Method toString

```java
public class Point /* The point is (x, y) */ {    private int x = 4;    private int y = 2;    /** Constructor: instance with name n */    public Point(String n) { /* The point is (x, y) */        name = n;    }    /** = a description of this Animal */    public String toString() {        return ("(x = " + x + ", y = " + y + ")");    }    System.out.println(new Point());    // prints (4, 2)
}
```

Summary of classes in Java

- Class defines content of file drawer and format of objects:
- File drawer contains static components and created objects, drawn as manilla folders. The name of an object — its location in memory — is drawn on the tab of the folder.
- new-expression, used to create objects. Know the 3 steps in evaluating it.
- Constructor: called in new-expression to initialize fields.
- Use of private and public.
- Getter and setter methods.
- static vs. non-static variables (instance variables or fields).
- static vs. non-static methods (instance methods).
- Method toString.
- Two uses of keyword this.

Keyword this. When it appears in a method, this refers to the object in which the method occurs.

```java
public class Point /* The point is (x, y) */ {    private int x = 4;    private int y = 2;    /** = point p equals point q */    public static boolean equals(Point p, Point q) {        return p.x == q.x && p.y == q.y;    }
}
```

Subclasses

The subclass definition has this form:

```java
public class subclass-name extends superclass-name {
    declarations of
    • instance variables
    • instance methods
    • class variables
    • class methods
}
```

only difference between a subclass definition and a superclass definition

Class Animal

```java
public class Animal {    private String name = "";
    /** Constructor: instance with name n */
    public Animal(String n) { /* the animal makes */        name = n;
    }    /** = description of this Animal */
    public String toString() {        return null;
    }
}
```

Subclass Dog

```java
public class Dog extends Animal {    private int friendliness;
    /** Constructor: instance with name n, friendly rank r */
    public Dog(String n, int r) { /* the animal makes */        Animal(name, n, friendliness);
    }    /** = desc. of this Dog */
    public String toString() { /* the animal makes */        return null;
    }    /** = noise this animal makes */
    public String noise() { /* the animal makes */        return null;
    }    /** = friendliness of this Dog */
    public int getFriendliness() { /* the animal makes */        return friendliness;
    }
}
```

subclass inherits all components of superclass

```
```
Subclass Dog: constructor

```java
public class Dog extends Animal {
    private int friendliness;
    // Constructor: instance with name n, friendly rank r */
    public Dog(String n, int r) {
        super(n);
        friendliness = r;
    }
    ... // constructor can't reference field name — it's private.

    FIRST statement of constructor can be a call on a superclass constructor.
    Example shows how to do it.
}
```

Subclass Dog: Overriding

```java
public class Dog extends Animal {
    public Dog(String n, int r) {
        super(n);
        friendliness = r;
    }
    ... // constructor can't reference field name — it's private.

    // declaration of toString in Dog overrides declaration of toString in Animal.
    When determining which method to call, use the bottom-up rule: start at bottom of folder and work upward

    // implicit
    a0
    Animal
    name fido
    Animal
    Animal(String)
    toString() noise()
    Dog(String,int) Dog
toString() noise()
getFriendliness()
friendliness 5

    // explicit
    d = new Dog("fido", 10);

    // object d is cast up (widened) to Animal
    a = (Animal) d;

    // object a is cast down (narrowed) to Dog
    d = (Dog) a;

    // illegal
    d = a;
```

Calling the inherited method: another use of super

```java
public class Dog extends Animal {
    public Dog(String n, int r) {
        super(n);
        friendliness = r;
    }
    ... // constructor can't reference field name — it's private.

    // Within toString, a call toString() refers to the same method toString! To refer to the toString method of the superclass, prefix the call with "super."
```

Casting up and down (narrowing and widening)

```java
Animal a;
Dog d;
Cat c;
a = d;
a = (Animal) d;
```

```java
Animal[ ] x = new Animal[100];
x[1] = new Dog("Fido", 10);
x[2] = new Dog("Pitty", 0);
x[3] = new Cat("Tabby", ...);
```

Why cast?

Assume Dog and Cat are subclasses of Animal

```java
Animal a;
Dog d;
Cat c;
a = d;
... if (d instanceof Dog) {
    ...
} else {
    ...
```

Casting up and down (narrowing and widening)

Assume Dog and Cat are subclasses of Animal

```java
Animal a;
Dog d;
Cat c;
a = d;
... if (d instanceof Dog) {
    ...
} else {
    ...
```

Why cast?

Assume Dog and Cat are subclasses of Animal

```java
Animal a;
Dog d;
Cat c;
a = d;
... if (d instanceof Dog) {
    ...
} else {
    ...
```
Apparent and real class-type of a variable?
Assume Dog and Cat are subclasses of Animal

```
Animal[ ] x= new Animal[100];
x[1]= new Dog("Fido", 10);
```

**Apparent type** of `x[1]` is Animal.

**Syntactic property**: Apparently, looking at the declaration of `x`, `x[1]` contains an Animal.

**Real type** of `x[1]` is Dog.

**Semantic property**: Really, `x[1]` is a Dog. Real type can change at runtime when `x[1]= e;` is executed.

Apparent class-type determines what components of object can be referenced

```
Animal[ ] x= new Animal[100];
x[1]= new Dog("Fido", 10);
```

**Apparent type** of `x[1]` is Animal.

**Syntactically legal**: `x[1].name` and `x[1].toString()`

Other components are there but cannot be referenced.

```
Animal Fido name
```

Class Object
the superest class of them all

```
public class Object {
    /** = description of this Object */
    public String toString() { ... }
    /** Object ob and this object are the same */
    public boolean equals(Object ob) {
        return this = ob; }
    ... 
}
```

Object, in package java.lang, automatically is the super-class of all classes that do not extend another class.

```
Object, in package java.lang, is the superclass of all classes that do not extend another class.
```

Overriding function equals

```
/** An instance is a point in the plane */
public class Object {
    private int x;
    private int y;
    /** = description of this Point */
    public String toString() {
        return "(" + x + "," + y + ");";
    }
    /** Object ob and this object describe the same point */
    public boolean equals(Object ob) {
        return ob != null && ob instanceof Point &&
               x = ((Point)ob).x &&
               y = ((Point)ob).y;
    }
}
```
Overriding function equals

```java
public class Object {
    private int x;
    private int y;

    /**
     * Object ob and this object describe the same point */
    public boolean equals(Object ob) {
        return ob != null && ob instanceof Point &&
        x == ((Point) ob).x && y == ((Point) ob).y;
    }
}
```

Java says: equals should be an equivalence relation, i.e.

- Reflexive: `b.equals(b)`
- Symmetric: `b.equals(c) = c.equals(b)`
- Transitive: if `b.equals(c)` and `c.equals(d)`, then `b.equals(d)`

Four kinds of variable

```java
public class C {
    private int ins;
    public static int cla;

    public void p(int par) {
        if (...) {
            par = par + 1;
            int loc;
            ...
        }
    }
}
```

- `ins`: instance variable or field. Belongs in each folder (object, instance) of class `C`. Created when folder is created.
- `cla`: class variable. Belongs in file drawer class `C`. Created at start of program.
- `par`: parameter. Created when frame for a call on `p` is created; destroyed when frame is erased. Scope: method body.
- `loc`: local variable. Created when frame for a call on `p` is created; destroyed when frame is erased. Scope: all the statements after its declaration in the block in which it is declared.

Array: an object that contains a bunch of variables of the same type.

```java
int[] b; // Declaration of variable b of type int[] (int array).
b = new int[4]; // Create an assign to b an object that is an array of 4 int vars.
    // Vars are named b[0], b[1], b[2], b[3].
    // Number of elements in the array is b.length.
b[k] = b[j] + 2; // Evaluate b[j] + 2, store the value in b[k].
```

```text
0 1 2 3 4
8 3 0 4 2
```

```java
C[] b; // Declaration of variable b of class-type C[] (C array).
b = new C[4]; // Create and assign to b an object that is an array of 4 C vars.
```

```text
0 1 2 3 4
null null null null
```

```java
```

```text
0 1 2 3 4
null null null a1 null
```