FALL 2003 CS211
JAVA BOOTCAMP

Organization:

- general help
- program level
- method level
  - tokens
  - statements
  - methods
- class level
  - classes
  - objects
  - fields and methods
  - references
- useful classes
  - strings
  - arrays

1. General Help

1.1 Books

- Appendix A of C&S
- Java Precisely
- Java in a Nutshell
- other books:

  - easy: a used copy of a CS100 book (L&L, …)

  - hard: The Java Programming Language

  - psycho: The Java Language Specification

1.2 Help on Java

- Java tutorial: http://java.sun.com/docs/books/tutorial/ Your First Cup of Java: Detailed instructions to help you run your first program: Win32, UNIX, Mac
- http://www.cs.cornell.edu/courses/cs100j/2001fa/ Notes and Exams will give you sample programs and problems
- CodeWarrior link on CS211 (under IDEs)
- more links! Java Resources link on CS211 website

2. Applications and Applets

2.1 Applet

- run as part of webpage or viewer
- see **`java.applet.Applet`** in API
- use **`<applet>`** tag in HTML
- not used in CS211

2.2 Application

- stand-alone program
- all code goes into ***classes***

```
modifiers class name morestuff {
    fields (data)
    constructors (methods to make objects)
    methods
    initializers
    inner classes
}
```

- compile classes to ***bytecode***
- put classes in own files
  - each class is public
  - or one public class per file (JDK rule)
  - when classes in 1 file, can be in any order

2.3 Main Method and Main Class

- one class must have a **`main`** method to start program

```
public class SomeThing {
    public static void main(String[] args) {
        // code for main
    }
    // rest of MainClass just like others
}
```

- other classes may have their own **`main`** methods
- to run a particular main method, must tell JDK which class's main to run

```
> java MainClass arguments
```

2.4 Related Things

- http://www.cs.cornell.edu/courses/cs100j/2003sp/ handouts/applications.html
- Section 1 notes CS211

```
public class Mains {
    public static void main(String[] args) {
        Test1.main(new String[] {"hello 1"});
        Test2.main(new String[] {"hello 2"});
    }
}

/* public */ class Test1 {
    public static void main(String[] args) {
        System.out.println(args[0]);
    }
}

/* public */ class Test2 {
    public static void main(String[] args) {
        System.out.println(args[0]);
    }
}

/* output:
    hello 1
    hello 2
 */
```


3. Language Elements

- write sentences (***statements***) out of words (***tokens***), which are formed from an alphabet (***character set***)
- write paragraph (***program***)
- sometimes write footnotes (***comments***)

4. Inert Elements

4.1 Comments

- type 1:

  single line: `// I am a comment`

- type 2:

  multi-line:

```
/*
     I am a comment
 */
```

  nesting allowed: `/* /* .... */ */`

- type 3:
  - Java Doc: see C&S D.7-D.13
  - see also Java Resources

4.2 White Space

- blanks, tabs ignored by Java compiler
- use as much WS as you want
- do not split tokens!

5. Characters

- Character Set: UNICODE
  - 16 bit encoding
  - store virtually every character out there
  - http://www.unicode.org/
  - enter character anywhere in program as **`\uxxxx`**
- Java will automatically understand ASCII
  - same codes in UNICODE

```
0 NUL   1 SOH    2 STX    3 ETX    4 EOT    5 ENQ    6 ACK    7 BEL
8 BS    9 HT     10 NL    11 VT    12 NP    13 CR    14 SO    15 SI
16 DLE  17 DC1   18 DC2   19 DC3   20 DC4   21 NAK   22 SYN   23 ETB
24 CAN  25 EM    26 SUB   27 ESC   28 FS    29 GS    30 RS    31 US
32 SP   33 !     34 "     35 #     36 $     37 %     38 &     39 '
40 (    41 )     42 *     43 +     44 ,     45 -     46 .     47 /
48 0    49 1     50 2     51 3     52 4     53 5     54 6     55 7
56 8    57 9     58 :     59 ;     60 <     61 =     62 >     63 ?
64 @    65 A     66 B     67 C     68 D     69 E     70 F     71 G
72 H    73 I     74 J     75 K     76 L     77 M     78 N     79 O
80 P    81 Q     82 R     83 S     84 T     85 U     86 V     87 W
88 X    89 Y     90 Z     91 [     92       93 ]     94 ^     95 _
96 ‘    97 a     98 b     99 c     100 d    101 e    102 f    103 g
104 h   105 i    106 j    107 k    108 l    109 m    110 n    111 o
112 p   113 q    114 r    115 s    116 t    117 u    118 v    119 w
120 x   121 y    122 z    123 {    124 |    125 }    126 ~    127 DEL
```

- use single quotes: **`'c'`**, **`'C'`**
- characters are integers! **`'a'+1`** → **`98`**
- escape characters: **`\"`**, **`\'`**, **`\\`**, **`\n`**, **`\r`**, **`\t`**

6. Tokens

6.1 Reserved words

- part of language
- cannot use as variables
- see front cover of C&S

6.2 Identifiers

- name/variable that refers to something: variables, methods, class names, constants
- must begin with letter, underscore (_), or currency symbol ($)
- may contain any number of digits, letters (even from UNICODE), underscores, currency symbols after the first character
- Java is case sensitive!
- must not use reserved words

6.3 Variables

- identifier holds value:

```
x = 3 ;
```

- must have a declared type:

```
int x ;
```

- must have value before used:

```
int x ;
x = 3 ;
System.out.println(x) ;
```

- variables as fields of a class have defaults (“zero”)

```
public class Thing {
    int i ;
    char c ;
    double d ;
    Thing t ;
    String s ;
    boolean b ;
}
```

- variables as method parameters or local variables (vars declared in methods) have unknown defaults!

6.4 Types

- Java is strongly typed: Variables always have a type!
- Primitive types:
  - 8 types: **`boolean`**, **`char`**, **`byte`**, **`short`**, **`int`**, **`long`**, **`float`**, **`double`**
  - why *primitive*? one solid value; not composite
  - typical for CS211: **`boolean`**, **`int`**, **`double`**, **`char`**
- Reference types:
  - using a class or interface or to declare a type
  - class type and array type
  - discussed more later

6.5 Constants

- constants: variables can get values, but the values themselves cannot change (e.g., 10 can't become 9)
- primitives:
  - **`boolean`**: **`true`**, **`false`** (no 0s and 1s allowed!)
  - **`char`**: see Section 5.
  - **`int`**: **`-2147483648`** to **`2147483647`**
  - **`double`**:
    use decimal point (**`.`**) (e.g., **`0.1`**, **`.1`**, **`1.`**, **`1.0`**)
    use scientific notation (e.g., **`1e-6`**, **`1.23E02`**)
- references:
  - reference values (see **`toString`** and **`hashCode`** in class **`Object`** in API)
  - **`null`**: no object (e.g., **`Thing x = null;`**)
- string literals:
  - actually objects, but easier to create on-demand
  - **`"`*`characters`*`"`** (can include escape chars)
  - **`""`**: empty string; not same as **`null`**
- modifier **`final`**: **`final type `*`VAR = value`*
  makes ***`VAR`*** constant during scope of ***`VAR`***

6.6 Operators

- arithmetic: mostly borrowed from C/C++
- precedence, associativity:
  - follows rules (see books)
  - parentheses to force order
  - all operands evaluated before operation except with **`&&`**, **`||`**, **`?:`** (these short-circuit)
    e.g., **`x = false; y = true; x && y;`**
- increment/decrement:
  - **`x=x+1`** could be written as **`x++`** or **`++x`**
  - **`a=1; b=a++;`** means that **`b`** gets the current value of **`a`**, then a increments
- modulus (**`%`**) (remainder operator)
  - **`4 % 3`** gives **`1`**
  - **`2 % 3`** gives ?
- beware of **`=`** (assign) vs **`==`** (equals)!

- **`instanceof`** (not covered in CS100)
- object access (**`.`**)
- array element access (**`[]`**)
- object creation (**`new`**)
- casting (**`(`*`type`*`)`**)

6.7 Punctuation

- **`( )`** expressions, methods
- **`{ }`** blocks of statements, members
- **`;`** ending statements

7. Statements

7.1 Empty

- **`;`**
- useful for placeholder in control structures:

```
for(int ii=1;ii<1e6;ii++);
```

7.2 Block

- collection of other statements inside **`{}`**
- treated as if it were a "subprogram" with its own variables if they're declared inside
- many issues concerning scope of variables (will see more later)

7.3 Expression

- legal expression statements are assignments, increments/decrements, method calls , and object creation
- combine constants with operators
- see operators (Section 6.6)
- beware of mixing types!
  - **`System.out.println(1/2)`** yields **`0`**, not **`0.5`**
  - mixing **`double`**s and **`int`**s makes **`double`**s: **`System.out.println(1.0/2)`** yields **`0.5`**
  - called **`promotion`**
- promotion: ***`char`*** < ***`int`*** < ***`double`***

```
char + int → int
int + double → double
char + double → double
```



```
public class Promotion {
    public static void main(String args[]) {

        System.out.println('a'+1);    // 98
        System.out.println(1+1.0);    // 2.0
        System.out.println('a'+1.0);  // 98
        char x='a';int y=1;test(x,y); // works!
    }
    public static void test(int x, double y) { }
}
```

- Java allows assigning thinner type to wider type
- is Java really strongly typed?
  - polymorphism and subtyping
  - same for primitives???
- use ***cast*** to force different value
  - syntax: ***`(type)expression`***
  - **`System.out.print((int)9.8)`** → **`9`**
  - usually for wider type to get "part" of thinner type
  - shows up a lot with subtyping in OOP

7.4 Assignment

- special kind of expression statement
- store value in a variable that's been declared
- cannot use variable until it's declared!
- syntax: ***`declaredvar = value`***
- ***`value`***'s type *must* match the ***`declaredvar`***'s type

```
boolean x ;
x = true ;
```

- shortcut to ***initialize*** (declare and assign) variable:
  - syntax: **`type`** ***`var = value`***
  - e.g., **`int y = 7;`**
- use **`final`** to make a constant: **`final int x = 1;`** (cannot change now!)

7.5 Labeled

- name a labeled statement
  - syntax: ***`label:statement`***
  - avoid this! (Java's way of doing a GOTO)
- method call
  - special kind of expression statement
  - syntax: ***`methodname(expressions)`***
- arguments can be empty
- arguments evaluated left to right before method body
- method does not have to return anything:

```
System.out.println("Hello");
```

7.6 Selection

- use **`if`**, **`if-else`**, **`if-else-if`**, **`switch`**
- syntax:

```
if (c)           // if c is true
        s;       // do s
// otherwise, skip this statement
```

- for multiple statements use a statement block **`{s1, s2, … }`** instead of ***`s`***

```
if (c)           // if c is true
        x;       // do x
else   // otherwise
        y; // do y

switch(expression) {
case constantexpr:
    statements;
    break; // optional
// more cases
default:
    statements;
}
```


- can have as many **`else-if`**s as you wish
- indent statements, but entire body counts as single statement! (indenting does not make a new statement, just improves clarity)
- can do multiple statements under each condition:
- language elements:
    - if, else are keywords
    - conditions? must evaluate to true or false (Boolean)
    - relations for conditions:
      `<` (less than)
      `>` (greater than)
      `<=` (less than or equal)
      `>` (greater than or equal)
      `==` (equal, but do NOT use =)
      `!=` (not equal)
      logic: **`&&`** (and) **`||`** (or) **`!`** (not)
      values: **`true false`**

7.7 Repetition

- use **`while`**, **`do-while`**, **`for`**

```
while(condition)
        s;

while(condition) {
        statements;
}

do {
    statements;
} while(condition)

for(inits; conds; increments) {
    statements;
}
```

```
// example:
int i = 1;
while ( i < 4) {
    System.out.println(i);
}

// becomes:
for (int i = 1 ; i < 4 ; i++ )
        System.out.println(i);
}
```


7.8 More Statements (coming up)

- more about methods
- object creation
- others: return, break, continue, return, synchronized, throw, try

8. Methods

8.1 Where?

- must be written inside a class, but in any order
- no function prototypes!

```
public class Thing {
        method1
        method2
}
```

- must access a method from a class or from within a class
- same for **`main`** method!


```
// method example:

class Thing {
    public void t2() { }
    public void t1( ) {
        t2();
        Blah.b2();
    }
}

class Blah {
    public static void b2() { }
}

public class Whatever {
    public static void main(String[] args) {
        Thing t = new Thing();
        t.t1();
    }
}
```


8.2 Syntax

```
    modifiers returntype name(arguments) throws exceptions {
        statements;
    }
```

8.3 Arguments (formal parameters):

- Java is strongly typed
- arguments is composed of series of inputs with form: ***`type1 var1, type2 var2, ...`***
- allowed to have no arguments: use **`()`**, not **`(void)`**

```
System.out.println()
```

8.4 Call by value:

- value of actual parameter copied into formal parameter
- method cannot change the value of an actual parameter

8.5 Name:

- use a legal identifier
- ***`name`*** should be an action

8.6 Return Type

- methods return either a value or nothing
- nothing to return?
    - use **`void`** keyword as method's return type
    - may use **`return`** statement to break from a void method anywhere in the method
- value to return:
    - use **`return`** expression
    - statement somewhere in method
    - Java is strongly typed, so type of returned value must be ***`returntype`***

8.7 Local Variables

- variables declared inside a method
- same rules as formal parameters, but declared after formal params
- variables cannot be "seen" (are invisible) to other methods
    - think of method body as a statement block:

```
header { stuff; }
```

    - ***stuff*** may have declarations which are local (visible) only in the block!

8.8 Overloading:

- write several methods with the same name
- change order of arguments, types of arguments, number of arguments, or any combination of these.
- the following do not constitute method overloading:
    - Changing *just* the return type
    - Changing *just* the names of the formal parameters

8.9 Modifiers

- privacy: **`public`**, **`protected`**, blank, or **`private`**
- class method (access w/o object): **`static`**
- no overriding (inheritance): **`final`**
- others: **`native`**, **`synchronized`**

8.10 Exceptions

- discussed later (likely) in CS211

9. Classes

9.1 Blueprint for creating objects

- except for import & package statements, your code goes into a class
- CS100 syntax:

```
modifiers class name {
    fields
    constructors
    methods
}
```

- other things inside a class: inner classes, static/instance initialization blocks (not CS100)
- fields and methods are members of a class

9.2 Fields:

- syntax:

```
modifiers type name = expression ;
```

    - modifiers: **`public`**, **`private`**, **`protected`**, or blank; **`static`**; **`final`**
    - the expression assignment is optional
- fields are assigned from top-down, left-right as object is created
- fields get default values of "zeros":
    - **`int`**s: **`0`**, **`double`**s: **`0.0`**, **`char`**s: ASCII code **`0`** or **`\u0000`**, **`boolean`**: **`false`**;
    - all reference variables: **`null`**
    - **`String`**s (which are objects): **`null`**

9.3 Method Syntax Reminders

- must write methods in a class! any order is OK
- every method in the *same class* can access any other method in the same class

9.4 Constructors

- resemble methods, but no return type written by programmer
    - the returned value is the address of the object created by the constructor
    - object will have the type of the class
- syntax: ***`modifiers name(arguments){body}`***
- every class has at least one constructor
    - if you do not give a constructor, Java automatically provides the empty constructor ***`name`*`() {}`**
- the first statement in the constructor must either
    - call a constructor of the same class with **`this(`*`arguments`*`)`**
    - call the super class constructor with **`super(`*`arguments`*`)`** (inheritance)
- if there is no **`super(…)`** or **`this(…)`** in a constructor, Java will automatically call **`super()`**
- for a class that does not extend from another, Java calls class **`Object`** when calling **`super()`**


```
public class Person {

    private String name;

    public Person(String n) {
        name=n;
    }

    public toString() {
        return name;
    }

    public addLastName(String ln) {
        name+=ln;
    }

}
```


10. Objects and References

10.1 Objects

- object is an instance of a class
- object creation: as part of an expression or an individual statement
- syntax: **`new `*`Classname`*`(`*`arguments`*`)`**
    - statement: **`new Thing(1,2,3);`**
    - expression: **`Thing t = new Thing(1,2,3);`**

10.2 References

- variables do not store objects!
- to reuse an object, you store an object's address (which is a value) in a variable
- the variable must have a compatible type of the object
- ***reference variable***: a variable that stores the address of an object
    - **`Thing x = new Thing();`**
    - **`x`** is a variable of type **`Thing`**
    - **`x`** stores the address of a newly created **`Thing`**


```
// example

class One {
    public One() { }  // is this necessary?
}

class Two {
    public void something() {
        One x;          // does x have a value?
        new One();      // what happens here?
        x = new One(); // what happens now?
    }
}

public class TestOneTwo {
    public static void main(String[] args) {
        new Two();
    }
}
```

10.3 toString

- to print the contents of an object, include a public method in your class: **`toString()`**
- **`toString`** returns a **`String`** that describes the object's contents
- you have the job putting something reasonable in the body of your **`toString`**

```
class Person {
    private String name;
    public Person(String n) { name = n; }
    public String toString() {
        return "Name: "+name;
    }
}

public class ToStringTest {
    public static void main(String[] args) {
        Person p = new Person("Dani");
        System.out.println(p);
    }
}

// Name: Dani
```



```
// another example

class Thing1 { }

class Thing2 {
    public String toString() {
        return "I am a Free Thing2!";
    }
}

public class ThingTest{
    public static void main(String[] args) {

        new Thing1();
        new Thing2();

        System.out.println(new Thing1());

        Thing1 t1 = new Thing1();
        Thing2 t2 = new Thing2();

        System.out.println(t1);
        System.out.println(t2);
    }
}

/* output:
    Thing1@17182c1
    Thing1@13f5d07
    I am a Free Thing2!
*/
```


10.4 Aliases

- reference variables store an address of an object
- so, changing the content of a ref var means storing the address of a different object

```
class Person {
    private String name;
    Person(String n) { name=n; }
    public String toString() { return name; }
    public void setName(String n) { name = n; }
}

public class AliasTest {
    public static void main(String[] args) {
        Person boss;
        Person p1 = new Person("A");
        Person p2 = new Person("B");
        boss = p1;
        p2 = boss;
        p2.setName("C");
        System.out.println(p1); // C
        System.out.println(p2); // C
    }
}
```


10.5 Passing References

- methods have have formal parameters, which are essentially local variables
- variables store values, so…
  - "passing an object" to a method means passing the value of a reference!
  - "returning an object" from a method means returning a value of a reference!

```
// references and methods

class Person {
    private String name;
    public Person(String n) { name = n; }
    public String toString() { return name; }
}

public class ReferenceTest {
    public static void main(String[] args) {
        Person p = new Person("Borknagar");
        test1(p);
        System.out.println(p); // Borknagar
        p = test2();
        System.out.println(p); // Dani
    }

    private static void test1(Person p) {
        p = new Person("Shagrath");
    }

    private static Person test2() {
        return new Person("Dani");
    }
}
```


10.6 This (keyword)

- represents a reference to the current object
- can be treated as a value because it is a reference

```
class Person {
    private String name;
    private Person friend;
    public Person(String name) {
        this.name = name;
    }
    // Set current Person's friend:
    public void makeFriends(Person friend) {
        this.friend = friend;
        friend.friend = this;
    }
    public String friend() {
        return friend.name;
    }
}

public class ThisTest {
    public static void main(String[] args) {
        Person p1 = new Person("A");
        Person p2 = new Person("B");
        p1.makeFriends(p2);
        System.out.println(p2.friend()); // A
    }
}
```



```
// messing with you

class Friend {
    private String Friend;
    private Friend friend;
    public Friend(String Friend) {
        this.Friend = Friend;
    }
    // Set current Friend's friend:
    public void Friend(Friend friend) {
        this.friend = friend;
        friend.friend = this;
    }
    public String friend() {
        return friend.Friend;
    }
}

public class Friends {
    public static void main(String[] friends) {
        Friend Friend = new Friend("Friend");
        Friend friend = new Friend("friend");
        Friend.Friend(friend);
        System.out.println(friend.friend());
    }
}

// yes, this really does work and is legal
```


10.7 Object Literals

- **`null`**: value that represents the absence of an object

  `Thing t = null;`

  - use **`null`** when you need to use a reference variable before creating an object (variables must have values before you use them)
  - also shows up as default value for fields that are references
- String literals: **`"characters"`**
  - creates a **`String`** object automatically
  - must be on one line in your code!
  - use **`+`** to add **`String`**s together
  - can add other types to **`String`** to make another **`String`**:

```
System.out.println("A" +
                                "B");

System.out.println("A"+1+2);
```

- other literals:
  - **`Class Class`** (yes, there is such a thing)
  - anonymous inner class (later on in CS211)

11. Scope and Visibility

11.1 Scope

- ***scope*** essentially means visibility or accessibility
- overall structure you've seen in terms of nesting of blocks:

```
class name {
        fields;
        method(params) {
            vars;
            { blockvars; }
        }
}
```

- rough rules:
  - generally, names see each other in the same "outer" block
  - generally, names from "outside" a block can be seen "inside" a block

11.2 Scoping rules for code in a class

- fields can access "previous" fields but not see "ahead" (fields are initialized top-down, left-to-right)
- all methods see all the fields
- all methods can see each other regardless of order
- all members in a class can see each other regardless of modifier

11.3 Scoping rules for code in a method

- parameters and variables are declared within the method
- they cannot be seen *outside* of the method
- thus, params and vars declared in method are often called ***local variables*** because of the outside inaccessibility
- locals and params can use the same names as fields, because local variables are not seen by the class! (thus, one use of **`this`**, because methods can see the **`this`**)
- If Java cannot find a local var in a method, then Java looks for a field
- locals declared before a block inside the method are seen throughout the method (just like methods seeing fields of a class!)

11.4 Scoping for a given block

- variables can be declared inside a block if not declared outside (and thus, before) the block
- variables declared inside a block do NOT exist outside that block (which summarizes the entire scope issue in the first place!)
- blocks can be re-initialized in loops (loop essentially repeat a block)


```
// Scope example)

class blah {
    int x1=10;
    int x2=17;
    String method1(double x1) {
        if (x1 > 0) { int x2 = 1; }
            { boolean x2 = true; }
        return method2(x1+x2);
    }
    String method2(double x2) {
        return "the value is: "+(x2+x1);
    }
}

public class TestScope {
    public static void main(String[] args) {
        System.out.println(new blah().method1(1));
    }
}

// output: 28.0
```

11.5 Information Hiding

- general rule: make fields and members used only in the class private
- everything else is public (but not variables inside a method!)
- public class members can be accessed from outside the class
- private class members cannot be accessed from outside the class
- objects created from the *same* class can indeed access members of that class regardless of privacy

11.6 Static

- **`static`** means you can access a member w/o creating an object
- does not mean "unchanging" for Java!
- use ***`Classname.member`*** for access
- all objects will share a **`static`** member, so you can also use ref.member to access


```
// example:

class Student {
     private String name;
     private static int count;
     public static int currentYear;
     public static final int GRADYEAR = 2005;
     public Student(String name) {
          this.name=name;
          count++;
     }
     public static int getCount() { return count; }
}

public class StaticTest {
     public static void main(String[] args) {
          System.out.println(Student.GRADYEAR);
          Student s1 = new Student("Dani");
          Student s2 = new Student("Shagrath");
          Student.currentYear = 2001;
          System.out.println(s2.currentYear);
          System.out.println(Student.getCount());
     }
}

/* output:
     2005
     2001
     2
*/
```


12. Strings and Characters:

12.1 Characters

- see Tokens and Character set
- remember that chars are primitives

12.2 Strings

- **`String`**: a collection of characters
- objects in Java
- string literal: **`"stuff"`** (saves hassle of calling a constructor)

12.3 String Operations

- **`"stuff1"+"stuff2"`** → **`"stuff1stuff2"`**
- **`"stuff"`**+primitive type promotes to **`String`**
- must put **`String`** on one line (no continuation character!)

12.4 Constructors

```
String s1 = new String();
String s2 = new String("stuff");
char[] tmp = {'a','b','c'};
String s3 = new String(tmp);
```

- What is a string literal? an instance of class **`String`**
- namely, a shortcut from calling a constructor!

- **`String`**s are immutable
  - once created, cannot change!
  - see **`StringBuffer`** class for mutable strings

12.5 Resemblance of Strings to arrays:

- index of characters starts count at zero
- find number of characters with **`length()`** (not **`length`**)

12.6 String Methods

- **`equals`**: compare contents of two **`String`**s **`s1`** and **`s2`** with **`s1.equals(s2)`**!!!
  - why not **`==`**? **`==`** tests equality of *references*, not contents!
  - so, **`s1==s2`** tests if **`s1`** and **`s2`** refer to the same object
  - sometimes **`==`** works…why?
  - **`==`** will check contents when comparing two **`String`** literals only
  - see **`string_equals`**
- more methods? see **`string_methods`**

13. Arrays

13.1 Arrays Structure

- all elements must have the same type (or class)
- indices must be integers or expressions that evaluate to integers
- most basic array is a 1-D collection of items
- when we say ***array***, we mean 1D array in Java
- multidimensional arrays are created from collections of 1D arrays

13.2 Creating Arrays

- declare:

```
type[] name;
type name[];
```

- assign:

```
name = new type[size];
```

  - shortcut version:

```
type[] name = new type[size]
```

  - more versions show up later
- example:

```
int[] x = new int[2];
x[0]=10; x[1]=20;
```


13.3 Rules

- **`[]`** is an operator and in same category of precedence with **`.`** and a few others
- can declare arrays in same statement but be careful:
  - **`int a[], b;`** → **`b`** is not an array
  - why use new? arrays are objects
- ***`size`*** is number of elements (must be integer or integer expression 0 or greater)
- labeling of indices starts at zero!
- if you attempt to access index that does not exist, Java complains with an with out-of-bounds exception (not all languages do this!)
- can find length automatically with ***`name`*****`.length`**
- all values in array are "zeros" by default (just like instance variables)

13.4 Initializer Lists

- handy way to creating an array and storing small amount of values:

```
type[] var = {val0,val1,…};
```

  - note: the semicolon is mandatory!
- essentially a shortcut, but not useful as a trick to "pass arrays":
  - **`return {1, 2, 3};`** won't work
  - need to use ***anonymous array***

13.5 Anonymous Arrays

```
type[] var = new type[] { val0,val1,… };
```

- may use anonymous arrays to pass a ref to an array that contains values
- handy way to create and "pass" an array simultaneously
- has a connection to inner classes, which you'll learn later

13.6 Objects With Arrays

- one way of "faking" a collection of different types
- some classes may have fields as arrays
- typically initialize size in the constructor:

```
class ArrayTest {
        int[] x;
        ArrayTest(int n) {
            x = new int[n];
        }
}
```


13.7 Arrays of Objects

- think of the syntax ***type*`[]`**
- ***type*** can be any valid type, including objects!
  - examples: **`Worker[]`**, **`Tray[]`**, **`Blah[]`**, …
- array of objects means *array of references*
  - elements hold reference values, not objects!
  - need to create objects for each element of array!
  - default values are **`null`**
  - example:

```
int size = 3;
Thing[] things = new Thing[size];
for(int i=0; i<size; i++)
    Thing[i]=new Thing();
```

- to access members of objects in an array (the refs are in the array),

name*`[`*index*`]`.*member

  - the **`[]`** "operates" before the **`.`** because Java works left to right when has operators in same category of precedence

13.8 Multidimensional Arrays

- rectangular array concept:

```
type[][][] var =
        new type[size1][size2][size3]
```

- array of array concept:

```
type[][][] var =
        new type[size1][][]
```

- assign arrays to "2nd" and "3rd" dimensions of **`var`**


```
// last example

public class aoa3d {

    private static int[][][] x;

    public static int myRandom(int low, int high) {
        return (int) (Math.random()*(high-low+1)) + low;
    }

    public static void main(String[] args) {
        createArray();
        // printArray(); // left as an exercise
    } // main

    private static void createArray() {
        x = new int[2][][];

        for (int d1 = 0; d1 < x.length ; d1++) {
            x[d1] = new int[2][];

            for (int d2 = 0; d2 < x[d1].length ; d2++) {
                x[d1][d2] = new int[myRandom(1,2)];

                for (int d3 = 0; d3 < x[d1][d2].length ;
d3++) {
                    x[d1][d2][d3] = myRandom(0,1);

                }
            }
        }

    } // method createArray

}
```

-