Java Bootcamp

David I. Schwartz

COMS/ENGRD 211

Read Step 1 first on Page 3!
It will explain what you need to do!

Table of Contents

Step 0: Understand the notation in this tutorial.

text	the stuff you need to read
`code`	something you would enter into a program; part of the Java language
`output`	something that a program would report as text; result of expression/method evaluation
`value`	the type of something you would enter as code e.g., name to represent any variable name
`file`	a filename
Menu	menu selection
link	link, which should bring you to somewhere in this document or a website the links tend to work best if open the document in a web browser
term	an important key term, which I try to define in nearby text

Step 1: Figure out how to do this tutorial.

1.1 Background

In the past, the ***Java Bootcamp*** was taught in a large lecture hall with an instructor or teaching assistant rattling off a blizzard of Java syntax while students' eyes rapidly glazed over. So, I have created this hopefully more engaging version. There may be a few rough edges, so I hope that you will help me by providing constructive criticism.

1.2 Time?

If you are unfamiliar with Java, you will likely need more that 3 hours. Otherwise, you might want to start with the OOP sections and refer back to previous sections.

1.3 On-line Help:

- Java 5 and JDK 1.5–explanation of these names mean effectively the same thing: http://java.sun.com/j2se/naming_versioning_5_0.html.
- Summary of Java 5 features/changes: http://java.sun.com/developer/technicalArticles/releases/j2se15/
- Java 5 complete release notes: http://java.sun.com/j2se/1.5.0/docs/relnotes/features.html
- Java tutorial: http://java.sun.com/docs/books/tutorial/
- Java API: http://java.sun.com/j2se/1.5.0/docs/api/index.html
- Data structures collection: http://www.nist.gov/dads/

1.4 Some suggestions

- Skim the sections to see what seems unfamiliar. Refer to the Table of Contents.
- If you are familiar with the basics of Java's language, we suggest starting with Step 14 (OOP in Java).
- If all you need is a quick introduction to ***Java 5***, see these sections: Step 13.13, Step 24.12, Step 27, Step 28.2, Step 29, and Step 31.
- If you get stuck, try the previous section. You might need to start from the beginning of this tutorial.
- Use companion notes, like the course books, their websites, Java's online tutorial at, and numerous examples posted on past CS100 Websites. Well…we can't cram *everything* about Java into this document.
- Try to do the suggested problems. If you only read the solutions, you have not practiced enough! Programming involves practice.

1.5 Problem 1

Where are the solutions posted for the tutorial?

1.6 Problem 2

What is *autoboxing*? Check out the Java 5 summary or release notes to find out.

Step 2: How to find Java at Cornell.

2.1 Background

Cornell has public labs (CIT), Engineering labs (ACCEL), and departmental labs, most of which you won't be able to access until you affiliate with a particular department. Review the links on the course website: http://www.cs.cornell.edu/courses/cs211 (you don't need the current year). See **Software** under the **Course Info** link.

Most likely you will work in a CIT lab if you don't use your own computer. To access course-specific software, click on Windows's **Start** and select **All Programs** → **Class Files**.

2.2 Java Environments

Use any type of Java development environment that you wish. If you think you are using something completely off-the-wall, check your code with something more standard.

2.3 Examples and Help

Review the CS211 website (see above). In CS211, we require that students understand how to compile and run their programs from the command line and, thus, use directly use the JDK (Java Development Kit). You can find help on these environments by following the links on the website.

2.4 Problem 1

Compile and run the following code (see class software in a public lab):

```
public class Step2 {
   public static void main(String[] args) {
      System.out.println("Hello, world!");
   }
}
```

2.5 Problem 2

In the above example, replace `"Hello, world!"` with `args[0]`. Now use the ***command line*** in a DOS window to compile and run your code. If you have never done this kind of programming, skim `applications.html`, which is posted along with this tutorial ("Companion Document"). Java's on-line tutorial also has a brief explanation of using command-line arguments.

To test your program, enter the following command at the DOS prompt:

```
> java Step2 Buh!
```

Step 3: What is a Java application?

3.1 Long Answer

Refer to **Applications.html**, which Step 2 mentions.

3.2 The Gist

- Except for **import** statements, all Java code must reside inside a class.
- Unless you want your program to run within a webpage (an ***applet***), you need to write an ***application***. A Java application has a **main** method that starts the program.
- Every class can have a **main** method.
- You need to "tell" Java which **main** method you wish to run when executing a program.
- When compiling from the command-line, **each file must include only one public class**. If you prefer to place multiple classes in one file, do not modify the classes with the word **public**–only the class containing method **main** should have **public** as part of the declaration.

3.3 Problem 1

Write a Java application that contains the following code in **one file** called **MyProgram.java**:

```
public class MyProgram {
   public static void main(String[] args) {
      System.out.println( new Person(args[0],args[1]) );
   }
}

class Person {
   private String first;
   private String last;
   public Person(String f, String l) {
      first = f;
      last = l;
   }
   public String toString() {
      return first + " " + last;
   }
}
```

From the command-line, compile and run the program with arguments **Dimmu** and **Borgir**.

Step 4: Java Language

4.1 Language Analogy

- alphabet ⇔ character set
- punctuation ⇔ punctuation
- whitespace ⇔ whitespace
- words ⇔ tokens
- sentences ⇔ statements
- paragraphs ⇔ modules

4.2 What's next?

The next few sections go over details and tricks about the Java language. I suggest that you try the problems to see how well you know/remember a lot of the nit-picky items. If you find yourself getting strange results, chances are you missed one of these rules.

4.3 Problem 1

Access http://java.sun.com/docs/books/tutorial/java/nutsandbolts, and review the Java file `BasicsDemo.java`. Rewrite the code in `BasicsDemo.java` to add backwards (start from 10 and end at 1).

Step 5: Java Character Set

5.1 Unicode and ASCII

- All characters use ***Unicode*** (www.unicode.org), not really practical for CS211. So, you could feasibly make the assignment statement `_☺_=27;`.
- We will use keyboard characters (***ASCII***) for language.
- Will show character values in token section.
- Java is case sensitive!

5.2 Problem 1

Look up ASCII codes at http://www.asciitable.com and answer these questions:

- What is the ASCII code for BEL? (Try MATLAB's **`beep`** function for more fun.)
- What is the ASCII code for the letter **`a`**?
- Is the ASCII value of letter **`A`** higher or lower than **`a`**?

5.3 Problem 2

What is the trick for changing the case of a letter using character arithmetic?

5.4 Submitting Your Homework in ASCII

Skim the submission format requirements in the CMS Info document on the course website. If you still do know what we mean by ASCII text, please ask!

Step 6: Java Comments and Whitespace

6.1 Java Comments

- Single line comments:
  ```
  // stuff
  /* stuff */
  ```
- Multiline comments:
  ```
  /*
      stuff
   */
  ```
- You can nest comments.

6.2 Java Whitespace

- Spacebar, return, new line, tab.
- You can separate tokens and statements with as much whitespace as you want.
- Do not split a token!

6.3 Problem 1

Correct the following program, which should print `hello`:

```
public Class Test {
   public static void ma in (String[] Args) {

      // hello
      /* hello
      System.out.println();

   }
}
```

Step 7: Java Tokens

7.1 Punctuation

- Statements end with semicolon! (`;`).
- Classes, methods, constructors, statement blocks, inner classes, initialization blocks, and initializer lists use braces (`{ }`). Inner classes and initialization blocks aren't covered in our CS100, so don't worry about them for now.
- Methods and expressions use parentheses (`( )`).
- Arrays use square brackets (`[ ]`).
- Parameter lists and variable declarations use commas (`,`).
- String literals (strings created w/o using a constructor) use double quotes (`" "`).
- Character values use single quotes (`' '`).
- Escape characters use a backslash (`\`).
- What about other symbols, like `+` and `.`? Those are operators–discussed soon!

7.2 Problem 1

The following program should print integers from 0 to 10. Unfortunately, we forgot to apply the punctuation! Please fix it.

```
public class Punctuation
   public static void main String args

      int count = 0
      final int STOP = 10
      while count <= STOP
         System.out.println count
         count++
```

7.3 Reserved Words

Reserved words are tokens that are reserved for the language and cannot be used to represent any value. See textbook for comprehensive list:

- values: `true false null this`
- types: `boolean int double char void`
- control: `if else for do while return switch`
- modifiers: `public private static final protected`
- classes: `new class extends implements super interface enum`
- amusing: `goto`

7.4 Values

Also thought of constants or literals: tokens whose representation means exactly the values expressed in their name. Java has ***primitive*** (non-object) types and ***reference*** (object) types.

Numbers:

- integers.
  We focus on `int`: `-2147483648` ↔ `2147483647`
  See also `long`.
- doubles
  decimal point: `0.1`, `.1`, `1.`, `1.0`
  scientific notation: `1e-6`, `1.23E2`

Boolean:

- `false` and `true`
- no 0 and 1 to represent truth!

Characters:

- Unicode: `'\uxxxx'`
- ASCII: use decimal `0`–`127`.
- Literal representation: single quote: `'a'`, `'1'`
- No empty character (`''`)!
- Escape characters: `\n` (new line), `\t` (tab), and others

References:

- Java has reference values to point to objects, but you cannot directly access those values.
- Special reference value you can use–"no object": `null`
- You'll get much more into references later in the tutorial.

7.5 Problem 1

If you mix characters and numerical values in an expression, Java promotes the expression to a value. Since `double` > `int` > `char`, you can actually mix all three together.

```
System.out.println(' ' * 2.0);   // what does this output? why?

System.out.println(true + 29);   // what does this output? why?

System.out.println("Hi\nthere!") // what does this output?

System.out.println((char)97);    // (type) is cast operator; output?
```

Why does the expression `(String) 9` cause Java to vomit? How do you create a string containing a primitive value? Hint: Use `+`. Actually, see next section.

7.6 Strings

Making strings:

- Java strings are not primitive values or arrays! Strings are *objects* in Java.
- Shortcut: use double quotes, which makes a string look like a value:
  ```
  String s = "I wuv u";
  ```
- Longer way:
  ```
  String s = new String("I wuv u");
  ```
- Also, see string promotion below.
- Once created, cannot change contents. See **`StringBuffer`** in API for mutable strings.

Null string (**`""`**) is not **`null`**!

- The null string is a string object with no character values.
- Very handy for ***string promotion***. See below.

String promotion:

- You can add *any* value or object to a string, which returns a string object!
  ```
  System.out.println(1+""+1); // what does this output?
  ```
- How does an object become a string? **`toString`** method, which all classes inherit, which you may implement. More later.
- Strings can't be demoted to other types. But you can use ***wrapper classes***. For example, **`Integer.parseInt("4")`** evaluates to the *integer* **`4`**.

Convert characters to strings and vice versa?

- You need arrays, which you haven't seen too much, yet.
- For now, here's a quick way (initializer list is shortcut for an array):
  ```
  char[] c = {'A','B','C'}; // array of 3 chars
  String s = new String(c);
  ```

Handy built-in methods!

- There are many useful string methods in the Java API. Refer to the **`String`** class.
- Number of characters in a string: **`length()`** method, e.g., **`s.length()`**.
- Others: **`charAt`**, **`indexOf`**, **`toUpperCase`**, **`toLowerCase`**, **`toCharArray`**, **`equals`**

String equality:

- Compare strings with **`equals`** method!
- Why? Strings are objects. So if you use **`==`**, you are comparing addresses, not contents.
- Actually **`==`** will work if you created strings with literals, but it's an obscure rule, so forget you heard it.
- As **`toString`**, **`equals`** is inherited by all classes. So, you can define it for other classes.

7.7 Problem 2

Are the following statements OK? Why or why not? What do they do?

```
String s1 = null;
String s2 = "";
System.out.println( s1.length() );
System.out.println( s2.length() );

System.out.println("abc" + "123");


String s2 = "abc";
s2[1] = 'd';


String a = "abcd";
String b = new String("abcd");
System.out.println( a == b );
System.out.println( a.equals(b) );


String start = "";
start = start + 1;
start = start + "\n";
start = start + 2;
System.out.println(start);
```

7.8 Identifiers

As long as you don't use reserved words, you can name elements of code:

- variables
- methods
- classes

Java names:

- Name contain alphanumeric characters, underscore, and currency symbols.
- Must not start with number!
- Case sensitive!

Variables:

- Store and refer to a value.
- Can be in method (***local variable***), method header (***formal parameter***), or field (***class variable*** or ***instance variable***–see **`static`** later).
- Must have declared type–Java is strongly typed! Discussed more in statements section. For now, ***`type name`*** to declare.

Methods:

- See above for naming rules.
- Convention: start with lowercase character.

Classes:

- See above for naming rules.
- Convention: start with uppercase letter.

Scope:

- How and when you can access variables, methods, and classes requires discussion of Java statements and OOP.
- This issue will appear all over the tutorial. The discussion of scope has to evolve because there is simply too much Java to learn all at once.

7.9 Problem 1

Is the following code legal? If so, what does it do? Should you use names such as these?

```
public class CLASS {
   public static void main(String[] Sgra) {
      int INT = 10;
      for (int Int=0;Int<INT;Int++)
         System.out.println(Int);
   }
}
```

7.10 Operators

Review: http://java.sun.com/docs/books/tutorial/java/nutsandbolts/opsummary.html

Basics:

- Arithmetic: `+`, `-`, `*`, `/`, `%` (mod), `-` (negation, subtraction)
- Logic: `&` (and), `&&` (short-circuit and), `|` (or) `||` (short-circuit or), `!` (not), `^` (xor)
- Comparison: `==`, `!=`, `<`, `<=`, `>`, `>=`
- Conditional: `?:`
- Assignment: `=`, `+=`, `*=`, `-=`, …
- Increment: `--`, `++`
- Cast: `(type)`
- Member access: `.` (no pointer, dereferencing)
- Array access: `[index]`
- String concatenation: `+`
- Object type comparison: `instanceof`

Power?

- Java does not use `**`; `^` means XOR (exclusive OR, not a power!).
- For x^y, use `Math.pow(x,y)`.

Precedence:

- See http://java.sun.com/docs/books/tutorial/java/nutsandbolts/expressions.html
- Handy to remember that precedence of `[]` and `.` are highest.
- Use parentheses (`( )`) to control the order of operation in an expression.

Associativity:

- Two main rules people run into:
  - Arithmetic works left to right.
  - Assignment works right to left.
- Example:

```
System.out.println("answer: "+1+2); // outputs answer: 12
x = y = 3; //
```

- Use parentheses (`( )`) to control the order of operation in an expression.

Promotion (revised):

- Number types: `double` > `int` > `char`.
- Cannot mix `boolean` with numbers.
- Everything promotes to string with `+`.

Arithmetic:

- Consequence of `int`s promoting to `double`?
- In mixed operations, result becomes `double`.
- Integer division results in integer result! Not rounded–result is floored!

Casting:

- `(type) expr`
- arrays and inheritance: cast has lower precedence! So, use `((type) expr)`

Conditional operator:

- `expr ? expr1 : expr2`
- Returns either `expr1` or `expr2`
- Example:
```
int x = 1; int y = 2;
System.out.println( x==y? 'a' : 'b');
```

Increment operators (some examples):

- prefix example:
```
int x = 1;
int y = ++x;
System.out.println(x); // outputs 2
System.out.println(y); // outputs 2
```
- postfix example:
```
int a = 1;
int b = a++;
System.out.println(a); // outputs 2
System.out.println(b); // outputs 1
```
- bonkers example (way beyond scope of CS211)
```
int x = 1;
x = x++;
System.out.println(x); // might not be what you think :-)
```

7.11 Problem 2

Write a program that checks if 13 is even or odd.

Why does `System.out.println(3/4)` output zero?

Write a program that generates a random integer between `1` and `100`, inclusive. Hint: `Math.random()` returns a `double` in the interval [0, 1).

Write a program that converts all of the lowercase letters to uppercase without using an array or strings. Hint: `for(int count=start;count<=stop;count++)`.

Step 8: Java Statements

8.1 Statement Types

- empty
- block
- expression
- declaration
- assignment
- selection
- repetition
- method call
- return
- object creation

8.2 Methods and Objects

The *return*, *method call*, and *object-creation* statements are discussed in later steps. The next few sections focus on the other statements.

8.3 Problem 1

Review http://java.sun.com/docs/books/tutorial/java/nutsandbolts/expressions.html.

What does the term ***control flow*** refer to?

What is a selection statement? Give an example in Java.

What is a repetition statement? Give an example in Java.

Step 9: Empty and Block Statements

9.1 Empty

The simplest form of a sentence:

- semicolon `;`
- empty block: `{}`

These statements simply succeed without further work.

9.2 Block

- syntax: `{ statements }`
- example

```
if (x == 10) {
   System.out.println("Hello!");
   System.out.println("x is " + x);
} else
System.out.println("Bye!");
```

9.3 Expression

- cannot make statements, like `1+1;`
- some expression statements are legal: method call, increment, object creation
- see operators for reminders of associativity and precedence

9.4 Problem 1

Write a program that pauses. Do not use a built-in method. Hint: Use a `for` loop.

Step 10: Declaration and Assignment Statements

10.1 The Rules

- all variables have types (strongly typed)!
- variables may be declared only once in a block!
- all variables must have values before used!
- all variables do not have initial values in methods!
- when variables declared but not assigned in class (fields), defaults are "zero" some tricky ones: references are **`null`**, booleans are **`false`**

10.2 Declaration Syntax

```
type name;
type name, …, name;
```

10.3 Assignment Syntax

- general:
  ```
  name = expr;
  ```
- combined syntax:
  ```
  type name = expr;
  final type name = expr;
  ```
- more about scope? coming up!

10.4 Problem 1

What is wrong with the following code, besides the fact that I did not comment it?

```
public void something( ) {
   int x = 1;
   boolean y;
   System.out.println(y);
   System.out.println(x==y);
}
```

Step 11: Introduction to Scope

11.1 Blocks

- Classes, methods, control structures, and blocks of statements.
- Blocks can enclose other blocks.

11.2 Variables

- Variables declared before an enclosed block are seen and used inside the enclosed block.
- Changes to the variable are retained while the outer block is still running.

```
{  int x = 2;
   { System.out.println(x); x = 3; } // output is 2
   System.out.println(x); } // output is 3
```

- Variable declared inside an inner block is not visible to the outer block.

```
{ int x = 3;
   { boolean y = true; }
  double y = 10; }
```

11.3 Methods

- Methods form their own blocks.
- So, variables declared inside method are local variables, meaning that their scope is strictly in the method.
- Method formal parameters (header) also act as local variables.

11.4 Classes

- Variables declared at class level (see braces) are visible to all methods and other variables declared at the class level. These variables are called fields.
- Actually, there's a lot more to discuss: **`this`**, **`private`**, **`public`**, **`protected`**, and **`static`**. Coming up!

11.5 Why nested blocks?

You can reuse the same variable name for loops:

```
void doStuff() {
   for (int i = 0 ; i < 10 ; i++ ) { /* stuff */ }
   for (char i = 'a'; i <= 'z'; i++ ) { /* stuff */ }
}
```

Note that reusing a variable with a different type is usually bad style.

11.6 Problem 1

What does the following code output?

```
public class Scope {
  public static void main(String[] args) {
     int x = 1;
     System.out.println("S1 for x: "+x);
      {
         System.out.println("S2 for x (before changing): "+x);
         x = x + 1;
         System.out.println("S2 for x (after changing):  "+x);
         int y = 3;
         { System.out.println("S3: "+(x + y)); }
      }
     String y = "yes, this is legal";
     System.out.println("S1: "+y);
  }
}
```

11.7 Problem 2

What does the following program output? Note that when creating an object (`new Data()`), Java first assigns the fields (`x`, `y`), then performs the constructor (`Data()`). Hint: The fields are in a block, which means the rules above still apply!

```
class Data {
   int x;
   int y = x;
   Data() {
      System.out.println(x);
      x = 1;
      System.out.println(y);
   }
}
public class Test {
   public static void main(String[] args) {
      new Data();
   }
}
```

Step 12: Control Flow

12.1 Execution

Execution in Java (and many other languages) is top-down, left-to-right. You can alter this flow with a ***control-flow*** statement, like **`while`** and **`if`**. Of course, the Java tutorial has a nice summary:
http://java.sun.com/docs/books/tutorial/java/nutsandbolts/flowsummary.html

12.2 Labeled Statements

In general, we're going to avoid these statements. Usually, you can avoid jumping to another statement (in the spirit of goto) with appropriate conditions in a selection or repetition statement.

12.3 Break Statement

Just as you can jump to another statement, you break execution to the statement that made the call to a particular point in code. Usually, people want to break out of a loop or if-statement. We're also going to avoid using break when possible. However, you will see a good break in the switch statement.

12.4 Selection Statements

Java has three kinds of selection statements: **`if`**-**`else`**, **`switch`**, and **`try`**-**`catch`**-**`finally`**. Since exception handling is covered later in CS211, we will cover only **`if`**-**`else`** and **`switch`** statements here.

12.5 `if` Statement

```
if (booleanexpression)
   block


if (booleanexpression)
   block
else
   block


if (booleanexpression)
   block
else if (booleanexpression)
   block
.
.
.
else
   block
```

12.6 `switch` Statement

If your test condition is **`int`**, **`short`**, **`char`**, **`byte`**, or an **`enum`** type (Step 27) there is a cool way to express a selection statement if you can identify the possible values of the condition. For example, suppose that a variable **`x`** might have three values: **`10`**, **`20`**, and **`30`**. Instead of saying

```
if (x == 10)
   do_something;
else if (x==20)
   do_something;
else if (x==30)
   do_something;
else do_something;
```

Each of these three constants can be used as cases in a **`switch`** on **`x`**:

```
switch(x) {
   case 10:
      do_something;
      break;
   case 20:
      do_something;
      break;
   case 30:
      do_something;
      break;
   default:
      do_something;
}
```

The general syntax of **`switch`** is as follows:

```
switch(expr) {
   case constant1:
      statements
      break;
   case constant2:
      statements
      break;
   …
   default:
      statements
  }
```

Java has a nitpicking requirement though: the constants must be known at compile-time, which means using literal values or constants declared with **`final`**. So, no variables, because they’re assigned at run-time.

12.7 Problems

Write a program that generates a random integer between `0` and `10`, inclusive. Report whether the number is even or odd.

Write a program that performs a logical and operation without actually using the `and` operators. So, assign two boolean variables and then test then using `if` statements.

Suppose that a variable `test` might be assigned to one of these characters: `'a'`, `'b'`, `'c'`, or `'d'`. Use a `switch` statement to do the following:

- If `test` is `'a'`, the program tells the user `"You did great!"`.
- If `test` is `'b'`, the program tells the user `"You did well!"`.
- If `test` is `'c'`, the program tells the user `"You passed!"`.
- If `test` is `'d'`, the program tells the user `"Well, you didn't fail!"`.
- Otherwise, the program tells the user `"Mission Control, we've got problems"`.

12.8 Repetition Statements

There are three general structures: `while`, `do-while`, `for`.

`while` syntax:

```
while (conditionalexpression)
   block
```

`do-while` syntax:

```
do
   block
while(conditionalexpression);
```

`for`-statement:

```
for ( initializations ; checks ; increment_decrements )
   block
```

`for`-each statement–will show later in arrays (Step 24) and `Vector`s (Step 30):

```
for ( object : collection )
   block

for ( int : array )
   block
```

12.9 Repetition Examples

```
int i = 0;
while (i < 5) {
   System.out.println(i);
   i++;
}
for ( int i = 0; i < 5 ; i++ )
   System.out.println(i);
for ( int i = 10; i < 100 ; i=i+10 )
   System.out.println(i);
```

12.10 Problem 1

Write a program that prints the following grid of characters without using an array:

```
a n
b o
. .
. .
. .
m z
```

Write the program with **`for`** and then with **`while`**.

12.11 Blocks in Control Flow and Scope Reminders

Note how I use ***`block`*** for the body of each statement. The block can be empty, one statement, or multiple statements:

- The empty case can be an empty statement or empty block.
- The single statement does not require braces.
- The multiple statements do require braces.

Because the bodies of the control statements are indeed blocks, you can indeed declare variables inside those blocks.

- Since those blocks are *inner* blocks, those declared variables must be visible only in those inner blocks.
- For the **`for`** statement, the initializations can also be newly declared variables.
- For repetition statements, each time the body repeats, the variables declared inside the body are simply re-declared without any problem.

If you intend for a variable to be seen from outside the control structure, you should declare that variable before the control statement.

12.12 Problem 2

```
int j = 0;
for ( int i = 0 ; i + j < 10 ; i++, j++ )
   ;
System.out.println( ___ ) ; // can you print the value of i? j?
```

Step 13: Methods

13.1 Where do they go?

- Inside a class. There are no "stand-alone" methods.
- You can write methods in any order in a class without having to use "function prototypes."
- Each class can have one **`main`** method. When running a program, you indicate which class's **`main`** method you wish to use. Refer to the beginning of the tutorial for more discussion.

13.2 Method Syntax

Syntax:

```
modifiers returntype name(params) stuff
   block
```

`modifiers`:

- privacy: **`private`**, **`public`**, **`protected`**, package (see OOP)
- **`static`** or non-static

`returntype`:

- must be valid type: class or primitive
- may be **`void`** (no return value)

`name`:

- must be valid Java identifier
- usually start with lowercase letter

`params`:

- arguments to the method
- must be declared
- for methods with no parameters, use just **`()`**. For example, **`int test() {code}`**.
- you may supply a variable amount of parameters with syntax ***`type … var`***. The ellipsis (**`…`**) is indeed part of the syntax! See Step 13.13.

`stuff`:

- methods can throw ***exceptions***, which is something you'll learn in CS211.
- there are other "things" you can put here, but we don't usually discuss them in CS211.

13.3 Problem 1: Random Integers

Method to generate a random integer:

```
public static int randInt(int low, int high) {
   if (low > high) {
      System.out.println("myRand Failure!");
      System.exit(0);
   }
   return (int) (Math.random()*(high-low+1)) + (int) low;
}
```

Now, use that method:

```
public class NumberGuess {

   public static void main(String[] args) {

      final int LOW = Integer.parseInt(args[0]);
      final int HIGH = Integer.parseInt(args[1]);

      // insert statement to find random integer between LOW and HIGH,
      // inclusive

   }

   // put randInt method here
}
```

13.4 Scope

If you are jumping around the tutorial, check out the other Scope sections. There is some new material about methods, below. You will see even more when you reach objects and classes:

- **`static`** methods cannot access non-**`static`** fields or methods (explained better when we hit objects)
- each method treated as an outer block (including params): so, all params and declared local variables inside the method are visible *only* inside the method
- methods must be in a class, which forms a block that encloses the method blocks: so, the methods inside a class all share the same fields

13.5 Static vs. Non-static and OOP (more later!)

The downside to teaching methods before objects is that some students prefer to make all methods **`static`** thereafter. When working with classes and objects, your methods will usually be *instance* methods, meaning that they are *not* modified as **`static`**. **`static`** methods are good style when you have methods that should influence *all* objects created from a class, or the methods would require nonsensical objects. Assuming you're actually reading this portion before OOP, I'll repeat this discussion in more detail when you learn how to access a method from an object or class.

13.6 Problem 2

Trace the following code. Does it run? Why or why not?

```
public class Methods1 {

   public static void test3() {
      System.out.println("test3");
   }

   public static void main (String[] args) {
      test1();
      test2();
      test3();
      test4();
   }

   public static void test2() {
      System.out.println("test2");
   }

   public static void test1() {
      System.out.println("test1");
   }

   public void test4() {
      System.out.println("Can you make this print?");
   }

}
```

13.7 Parameters

Java passes variables by value. So, Java passes the values of the actual arguments to the formal arguments. The variables cannot be passed.

```
public class Params {
   public static void main(String[] args) {
      int x = 1;
      change(x);
      System.out.println(x); // output is 1
   }
   public static change(int x) {
      x = 2;
      System.out.println(x); // output is 2
   }
}
```

You can also determine that the value of **x** in method **main** doesn't change based on the scope of **x**. Method **change** has its own local variable **x**. There is no conflict with **main**'s **x**, because both **main** and **change** are independent blocks inside the outer block of **Params**.

13.8 Return Type and Values

Return types:

- Java methods must have a return type.
- The types may be any valid type (primitive, object).
- If you prefer not to return anything, you must give a return type of `void`.

`return` statements:

- Can written anywhere in a method body.
- `void` methods are allowed to have a return statement (with no return expression) anywhere in the method. Use `return;`.
- Java passes the value of the return expression back to the code that made method call. So, the code that makes the method call (e.g., `Math.sqrt(4)`), gets replaced by the returned value (`2`).

13.9 Problem 3

What does the following code output?

```
public class Methods2 {
   public static void main (String[] args) {
      System.out.println( and(true,false) );
      boolean t1 = nand(true,false);
      boolean t2 = xor(true,false);
      System.out.println( and(t1,t2) );
   }

   // Return short-circuiting AND of t1,t2:
   public static boolean and(boolean t1, boolean t2) {
      return t1 && t2;
   }

   // Return exclusive OR of t1,t2:
   public static boolean xor(boolean t1, boolean t2) {
      return t1 != t2;
   }

   // Return NOT-AND of t1,t2:
   public static boolean nand(boolean t1, boolean t2) {
      return !(t1 & t2);
   }
}
```

13.10 Problem 4

Trace the following code. What does the program output? Which method (`search1` or `search2`) is more reusable (and thus, arguably better style)?

```
public class LinearSearch {
   public static void main (String[] args) {
      int[] x = {1, 4, 5, -1}; // shortcut to create 1D array of values
      search1(x,-1);
      System.out.println(search2(x,-1));
      System.out.println(search2(x,0));
   }

   // linear search for target in data x, return early if found:
   public static void search1(int[] x, int target) {
      for (int i = 0; i < x.length; i++)
         if (x[i] == target) {
            System.out.println("Success!");
            return;
         }
      System.out.println("Fail!");
   }

   // linear search for target in data x,
   // return true if found, else false:
   public static boolean search2(int[] x, int target) {
      for (int i = 0; i < x.length; i++)
         if (x[i] == target)
            return true;
      return false;
   }

}
```

13.11 Method Overloading

Method overloading:

- Write more than one method with the same name in same class.
- Helps with creating too many silly names, like `search1`, `search2`, ….
- Common example: `System.out.println`. See `PrintStream` in API.

Rules:

- You may change order of arguments, types, number of params and combinations of these changes.
- These two changes do not constitute overloading:
  - changing just the return type
  - changing just the parameter names

13.12 Problem 5

Write two more **myRand** methods:

- One to generate a random bit.
- Another to generate a random character.

Judicious use of myRand for integers can help reduce the amount of code you write.

```
public class Overloading {
   public static void main(String[] args) {
      System.out.println("int:  "+myRand(1,10));
      System.out.println("bit:  "+myRand());
      System.out.println("char: "+myRand('a','z'));
   }

   // Return random int, low <= high:
   public static int myRand(int low, int high) {
      if (low > high) {
         System.out.println("myRand Failure!");
         System.exit(0);
      }
      return (int) (Math.random()*(high-low+1)) + (int) low;
   }

   // Generate random bit:









   // Generate random character:









}
```

13.13 Problem 6: Variable Arguments

Java 5 introduced ***variable arguments*** using the syntax ***`type ... var`***. So, you can call a method with 0 to many arguments without having to specify new methods using overloading. Refer to the following example:

```
public class Varargs {
   public static void main(String[] args) {
      test(1, 2, 3);
   }

   public static void test(int ... x) {
      for (int i = 0; i < x.length; i++)
         System.out.println(x[i]);
   }
}
```

What does the above example output? Write another method test2 that accepts multiple `String`s. Method `test2` will return how many of the inputs are equivalent to `"Metroid"`.

Using `Object` and other reference types can help you to create very generic code. You will see a review of `Object` throughout later sections.

13.14 The Stack

- Each time you call a method, a portion of computer memory is set aside to hold the parameters, local variables, and "administrative information" for that method.
- The most recent method's memory goes on top of the previously called method.
- Eventually you reach the last method, which executes.
- The ***stack*** is literally a stack of each method's portion of memory.

13.15 Recursion

A Java method can call itself. CS211 will go over this notion in great detail in the first couple of weeks of classes.

The notion of the stack is very handy in keeping track of which method call is happening, especially when a method calls itself. At some point the method needs to know when to stop calling itself.

13.16 Advanced Problem

You can skip this problem if you've never tried recursion.

Write a method called `sum` that finds the sum of each number less than or equal to the a supplied number. For example, sum(5) = 5 + 4 + 3 + 2 + 1 = 15.

Step 14: Building A Class

14.1 Should you do this section?

This entire section provides a summary of the concepts of creating classes for OOP. You can get a quick reminder of what you should already know. If you are very skilled at OOP, skip ahead to Step 14.5 and then to Step 15, which focus more on Java syntax.

14.2 ADTs

Abstract Data Type:

- primitive and built-in types don't anticipate all the modules we might encounter
- ADT: type of thing with associated data and actions
- Use ADT to form customized types

Classes and objects?

- Classes implement ADTs
- Classes provide code to create and use objects. Think of classes as user-defined types. For example, `class String`.
- Objects are unique instances of a class. Think of objects as specific values that have the type of the class from which they were created. For example,

```
String s = new String("abc");
```

14.3 Designing a Class

Find nouns and verbs in problem:

- verbs become either
  - operators
  - methods
- nouns become either
  - classes (code used to make objects)
  - fields (variables in classes)
  - parameters (variable in methods)
  - local variables (in methods)
  - constants (known values like numbers, chars, booleans, enums)

Elements of ***class*** (specification–a blueprint–of behaviors and actions of an entity):

- ***fields***: data, properties, states, attributes of a thing
- ***methods***: actions done by/on/for a thing
- ***constructors***: how is the thing made

Will see these terms again in subsequent sections.

14.4 Design By Abstraction

Abstraction:

- Try to create general structures that hide nitty-gritty details.
- Helps a programmer deal with more English-like code, making it easier to program.
- Use named constants:
  - Rather than using specific values inside the body of a block, define constants as fields or local variables that store those values.
- Use methods calling other methods:
  - The “early” methods (first in the stack) can have really high-level sounding names.
  - The “later” methods (uppermost in the stack) can have very detailed, low-level code.
- Use encapsulation (has-a relationships):
  - If a thing has a something, then that something belongs in the class (a member)
  - The class encapsulates states and behaviors.
  - Provides a mechanism to “bury” things away in another class.
- Use information hiding:
  - Determine how an object should communicate with other code.
  - Make high-level, **`public`** methods to communicate with the code.
  - Make all other class members **`private`**.

Java rules for privacy modifiers:

- **`private`** member cannot be accessed by a class different from the class in which the private member is defined.
- **`public`** members can be accessed by any class.
- ***package*** (blank) visibility. Acts as **`public`** when not using packages.
- **`protected`** visibility relates to inheritance, which means we’ll deal with it later.

Style for privacy:

- Make fields **`private`** when possible. For inheritance, **`private`** fields are irritating, so we generally make them **`protected`**.
- Utility methods (methods used only by other methods inside a classes) should be **`private`**. No need for another class to see how you hacked together a weird solution.
- Setters, getters, and other accessors should be **`public`**, so that you can use your objects.

Static vs. Non-Static Members:

- Make a member **`static`** if you wish to access a member without creating an object (because that member should influence or be shared by all objects created from a class).
- Will discuss more later when you’ve reviewed how to access an object’s members.

14.5 Overall Structure

In English:

- classes contain members and constructors
- constructor is essentially a method that makes an object and returns the object's location in memory (reference)
- members are fields and methods (and other things you'll learn about in CS211)

In code:

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

14.6 Problems

What does the following program output? How would you create another `Person` with the name `"Harlan Ellison"`? How would you modify class `Person` to include a middle name (don't bother inheritance for now–CS211 will cover that later)?

```
public class Glance {

   public static void main(String[] args) {
      Person p = new Person("Dimmu");
      p.setLastName("Borgir");
      System.out.println(p);
   }

}

class Person {

   private String firstName;
   private String lastName;

   public Person(String fn) {
      firstName=fn;
   }

   public void setLastName(String ln) {
      lastName=ln;
   }

   public String toString() {
      return firstName+" "+lastName;
   }

}
```

Step 15: Creating Objects

15.1 Where are we?

To write program:

- Form the ADTs: You need to determine the composite “things” you need to model, including their properties and actions/behavior.
- Implement the ADTs by writing the code for the classes.
- Write code that uses the classes: Make ***objects*** (specific instances of classes):

So, we need a way to make objects.

15.2 Constructor Syntax

```
modifiers Classname(params)
   block
```

- Constructors resemble methods, but:
  - Constructors have same as class. (Actually, a method could have the same name as a class, but you end up with strange looking code.)
  - Constructors have no specified return type: they do in fact return the address of the object that they create.
- Constructors are usually **`public`** so that another class can make objects.

Examples:

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

Ways to call a constructor:

```
new Person("Shagrath");

Person p;
p = new Person("Horgh");

Person p = new Person("Dani");
```

15.3 Problem 1

Write a class **`Complex`** that represents a complex number ADT. Include an **`add`** method.

15.4 Constructor Rules: No Inheritance

Assuming you aren't worried about inheritance:

- Classes can have more than one constructor
  - same idea as method overloading
  - number and types of arguments must change
  - provides multiple ways to make the same kind of object
- Constructors usually set field values.
- Constructors can call another constructor with **`this(…)`** as the first statement in the constructor. Handy for reducing the number of variable assignments if another constructor already does that work.
- Default, or empty, constructor:
  - If you do not provide a constructor, Java automatically provide the empty.
  - Empty constructor syntax: **`public Classname() {}`**.
  - If you provide any constructor, Java does *not* give you the empty constructor!

15.5 Constructor Rules: Inheritance

This stuff will show up later in CS211. You can skip for now. But, if you are curious…

- The first statement in a constructor must be **`super(…)`** or **`this(…)`**.
  - If it's **`this(…)`**, then the last constructor in the chain must have as its first statement **`super(…)`**.
  - If there is no **`super(…)`**, Java will automatically call **`super()`** (no arguments).
- The **`super(…)`** calls the immediate superclass constructor.
  - You cannot say **`super.super.…`**.
  - If Java is forced to call the default **`super()`**, you had better make sure that the superclass has an empty constructor.

Order of construction when making an object:

- Set all fields to default values of "zero" (all classes!).
- Invoke each constructor without executing the bodies:
  - Start with the first constructor you call.
  - Follow the chain of calls to **`this(…)`** and **`super(…)`** all the way to class **`Object`**.
- Working top-down, then left-to-right, assign the fields if any assignment is specified.
- Execute the body of the code in the uppermost constructor.
- Go to the next highest class in the hierarchy and repeat the process.

15.6 Problems

Write a Java program that creates two **`Complex`** numbers. In the next section, you will learn the code to add them.

Write a Java program that creates a **`Rectangle`** object. In class **`Rectangle`**, there should be two constructors:

- Use four fields for the sides.
- **`Rectangle(int s1, int s2, int s3, int s4)`**. If you want to apply rudimentary error checking, compare the sides and call **`System.exit(0)`** for illegal inputs.
- **`Rectangle(int s1, int s2)`**. Call **`this(s1, s2, s1, s2)`** instead of writing all four assignment statements.

15.7 Problem 2: Really Bonkers (Not Recommended)

I don't recommend that you try to trace this code. In fact, stop looking at it. Just skip ahead if you don't want to risk losing your mind. OK, you asked for it:

```
public class Shadowing {
   public static void main(String[] args) {
   A a = new B();
   a.test4();
   }
}
class A {
   public int x;
   public A() { test1(); test2(); test3(); }
   private void test1() { System.out.println(x); }
   public void test2() { System.out.println(x); }
   public void test3() { System.out.println(x); }
   public static void test4() { System.out.println("Hi"); }
}
class B extends A {
   public boolean x = true;
   private void test1() { System.out.println(x); }
   public void test3() { System.out.println(x); }
   public static void test4() { System.out.println("Bye!"); }
}
```

Step 16: Storing and Accessing Objects (References)

16.1 Where are we?

- You can write a class by modeling a thing's data and actions.
- The class provides the code from which you build and use unique objects.
- You need to figure ways to build objects with constructors.
- A constructor returns a reference to the newly created object.

Remaining questions:

- What's a ***reference***?
- How to access an object with the reference?

16.2 What's a reference?

Short answer:

- ***Reference*** (or reference value): the address of an object, which is indeed a value.
- References are returned from constructor calls.

Example:

- Run the following code:
```
public class RefTest {
   public static void main(String[] args) {
      System.out.println( new Person() );
   }
}
class Person { }
```
- Java will output a strange looking value, like `Person@092abc`.

What happened?

- `new Person()` creates an object of type `Person`.
  - class Person has the empty constructor because you didn't provide one.
- The return value of the constructor call is the address of the `Person` object that you created.
- There is a "secret" method in Java that has a default behavior, which returns the reference value as a string.
  - The method is `toString`, which is inherited from class `Object`. All classes extend `Object`.
  - Refer to the API on `toString`'s default behavior. It's actually not guaranteed to be the address, but for our JVM, it is.
  - You will usually override `toString` because of this weirdness.

16.3 Comparing Primitive and Reference Values

Primitive types:

- When you say `int x; x = 1;`,
  - Java allocates memory to an integer.
  - Java adds the association of `x` and its address in a table somewhere else in memory.
  - For the assignment, Java looks up the address of `x` and then puts the value of `1` there.
- The idea is that all variables have a specific address, allocated memory, and possibly a value.
- Refer to the stack from methods. Each method call allocates memory because of the variables inside the methods.

Object types:

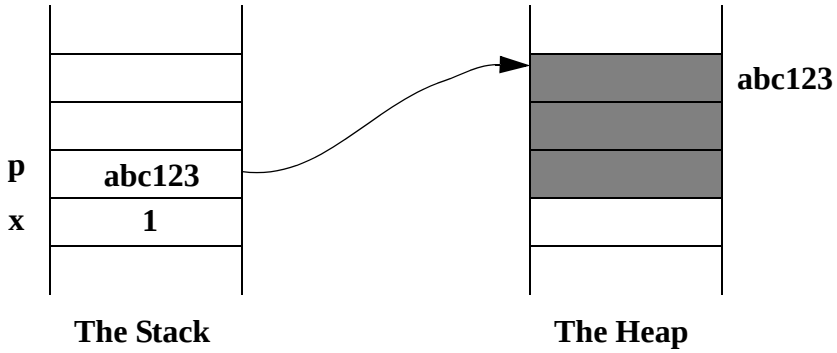
- When you say `Person p;`,
  - Java allocates a small amount of space to hold an address value, but not the entire object. In fact, Java has not created an object yet!
  - The variable `p` has a location in memory. When you create an object and copy its address to store in `p`'s location, `p` then refers to the object.
- When you say `p = new Person();`,
  - Java creates an object.
  - Java allocates enough memory to hold the object in another location called the heap. This memory holds instance variables and administrative information.
  - The constructor returns the address of the object, which is stored in `p`'s portion of memory. The object is not stored in `p`!
- So, there are two portions of memory:
  - `p`, which has it's own location (local variable, parameter, field), which holds an address of an object.
  - the object, which resides somewhere else in memory (the heap).
- Java does not allow you to change a reference value.
  - You can still make and change objects, but while an object's reference is "alive," you cannot change the value itself.
  - C/C++ people might be disappointed–what Java is effectively disallowing is pointer arithmetic.

16.4 Picture

Suppose you have a method with the following code:

```
int x = 1;
Person p = new Person();
```

Refer to the figure below. Both **x** and **p** are local variables that are allocated on the ***stack***. The object created from class **Person** has space allocated on the ***heap***.



16.5 Problem 1

Review this code and answer the questions that follow:

```
public void m1() {
   Person p = new Person();
   Person q = new Person();
   // before
   q = p;
   // after
}
```

At the point where the code says **// before**, do **p** and **q** refer to the same object? Why or why not?

At the point where the code says **// after**, do **p** and **q** refer to the same object? Why or why not?

You'll see me repeat similar code in the upcoming section on aliases.

Step 17: Special reference–`null`

17.1 What happens if you have no object?

Two important Java rules:

- You cannot use variable if it doesn't have a value.
  - Local variables do not receive default variables!
  - Sometimes you have to provide dummy or initial values.
- Fields always have defaults of "zero."

So, what is the "zero" value for a reference variable? It can't be a primitive value, like 0! Instead, Java has **`null`**, which means "no object." You could also think of it as "address 0, which has no object."

17.2 Example

```
public class TestNull {
   public static void main(String[] args) {
      Person p = null;
      System.out.println(p);
      Person q = new Person();
      q = p; // q now refers to no object
   }
}
```

17.3 Problem 1

What does the following code output? Why?

```
class Person {
   private String name;
   public Person() {
      System.out.println(name);
   }
}
public class TestNull2 {
   public static void main(String[] args) {
      new Person();
   }
}
```

Step 18: Special method–`toString`

18.1 Why bother?

Reference values are irritating! It is very handy to be able to "print an object." So, Java provides a built-in method called **`toString`**, which all classes inherit from class **`Object`**. **`toString`** will "stringify" an object. From before, the default behavior is usually returning the object's address as a string.

To make **`toString`** useful, your classes should override the default behavior.

- Write a **`toString`** method with the exact same header that **`toString`** has in **`Object`**:
  `public String toString()`
- Give a body that returns a string, which you can define.

`Object` has other methods, like **`equals`**, that you will often override. In fact, when extending any class, you will often override methods, not just those in **`Object`**.

18.2 Example

This program outputs **`Slim Shady`** instead of something weird, like **`Person@abc123`**:

```
class Person {
   private String name;
   public Person(String n) {
      name = n;
   }
   public String toString() {
      return "My name is "+name;
   }
}
public class PersonTest {
   public static void main(String[] args) {
      Person p = new Person("Slim Shady");
      System.out.println(p); // isn't this cool?
   }
}
```

Remember that **`toString`** only returns a string and does not "print the object!" Above, the **`println`** method forces the stringification and the prints the result. In general, you can actually force **`toString`** to stringify an object with string promotion:

```
   Person p = new Person();
   String s = "" + p;
```

18.3 Problem 1

Write a useful **`toString`** method for class **`Complex`**. Hint: Consider real and imaginary components. Sometimes the imaginary component with be negative.

Step 19: Accessing an object's members

19.1 Where are we?

So far,

- You can write a class.
  - A class has fields, constructors, and methods.
  - There's other stuff you will eventually see in CS211.
- You can create objects using constructors in a classes.
  - Constructor returns reference to newly created object.
  - You can store addresses of newly created objects in reference variables.
  - You cannot store entire object in a variable!

Well, how you access the members of an object?

19.2 Member Access

Encapsulation reminder:

- has-a relationship.
- State (fields): An object might have a name, an age, ….
- Behavior (methods): An object might have a way to report its age, might tell you who its parents are, …,

So, we need a way to say "object's something" or "object has-a something."

- Java uses the dot (`.`) operator to get to the "something."
- Syntax:
  `reference_variable.member`
  `constructor_call.member`
- You can shorten these two to just ***`ref.member`***.
- There's a 3rd syntax (***`Classname.member`***), which requires ***`static`***. And yet again, I want to delay formal discussion. But it's coming up really soon!

The member must be **`public`** (or accessible within the scope) for the dot to work!

19.3 Examples

Practice accessing an object's field. Below, I'm breaking choosing clarity over style:

```
class Person { public int age; }
public class TestDot {
   public static void main(String[] args) {
      Person p1 = new Person();
      System.out.println(p1.age); // outputs 0
      System.out.println( new Person().age ); // outputs 0
   }
}
```

Accessing a method is very similar because of the ***ref.member*** syntax:

```
class Person {
   private int age;
   public Person(int a) { age = a; }
   public int getAge() { return age; }
}
public class TestDot {
   public static void main(String[] args) {
      Person p = new Person(50);
      System.out.println( p.getAge() ); // outputs 50
   }
}
```

19.4 Objects are unique (reminder)

Refer to the example, above. If you say ,

```
Person p = new Person(10);
Person q = new Person(20);
```

the objects the **p** and **q** refer are different. There are two entirely different addresses. However, the code used is identical. Why?

- Objects are created from a class.
- The class provides the blueprint (the code) used to “stamp out” an object.
- When calling an object’s members,
  - Java uses the code from the class to see *how* to process the member.
  - But, the specific values Java uses come from the specific *object*.

So, both **p** and **q** have **getAge** methods, but each call to **getAge** will access different values of the field age:

```
System.out.println(p.getAge());
System.out.println(p.getAge());
```

If you want to make things more interesting,

```
Person p = new Person(20);
Person q = new Person(20);
```

Both objects are still different. You just happen to have **p**’s **age** be 20 and **q**’s **age** be 20. After all, there certainly many people in the world who happen to have the same **age**. And they are not all the same person!

19.5 Changing an object's fields

Generally, fields should be private to prevent access, which is a consequence of maintain information hiding. However, you should at least see a small example:

```
class Data {
   public int x;
   public int y=20;
   private int z;
}

public class TestPublic {
   public static void main(String[] args) {
      Data d = new Data();
      System.out.println(d.x);
      d.x = 10;
      System.out.println(d.x);
   }
}
```

You should get `0` and `10` as output. Why `0` on the first output? Recall the fields always have a default of "zero" unless you initialize their values directly in the class.

19.6 Privacy: A Nitpicking Detail

Now and then you'll see code that looks as follows…I've extracted a bit of `Complex`:

```
private double r, i;
public Complex add(Complex c) {
   return new Complex ( r+c.r , i+c.i );
}
```

How could I say `c.r` and `c.i`, though `r` and `i` are `private`? Well, `c` is type `Complex`, so as far as Java is concerned, `c` and the current object are allowed to see each other's members. Yes, it's kind of kinky.

19.7 Problem 1

In the above example, try to access and change `d`'s `y` and `z` fields. Follow the same order of output statements:

- Why does `d.y` output as `20` before you change it?
- Why does your program not allow you to access `d.z`?

Rewrite `Data` such that it has no `public` fields, but you can still access and change its fields.

Expand class `Person` to include a last name, first name, and methods to access those names.

Write a program that adds two complex numbers together. Report the output.

Step 20: What is `static`?

20.1 Mystery revealed!

Not everything is unique!

- Sometimes objects share the same data, e.g., students with the graduation year, population all living in the same place, and many other examples.
- Sometimes a member doesn't make sense belonging to an object: `Math.PI`. You could make a `new Math()` object, but why? In this case, you could say that `PI` is shared by "everything," which improves the case to make it `static`.

When fields are shared (modified as `static`), then you need a way to access the fields without worrying about objects. So, we'll need `static` methods. Note that people who use language with global values tend to like `static`, because it allows for member access without objects.

20.2 Syntax and Rules

Syntax:

```
classname.member
```

Rules:

- The member must still satisfy scope rules. So, if a field is private `static`, an outside class still cannot access/see the field!
- You can still access a `static` member via references. What may seem weird (but shouldn't) is that once you change an object's `static` field via a reference, all the objects will be that same field value. If you are confused, try Step 20.5.

Terminology:

- class variable/ method: a `static` field/method
- instance variable/method: a non-`static` field/method

20.3 Example

```
class Person {
   public static final String fate1 = "death";
   public static final String fate2 = "taxes";
   private String name;
   ...
}
```

In a method somewhere, you can say

```
System.out.println( Person.fate1 );
```

20.4 `static` and other modifiers

Why do I make the fields **`public`** and **`final`**? (see next page)

Making a field **`static`** because it is shared makes sense. Since it's shared, you might as well make it easy to access. However, easy access can be disastrous. So, the field is also **`final`**. The triple modification of **`public static final`** provides the closest representation of a global constant in Java. Sometimes people prefer **`private static`**, which might remove the need for **`final`**.

20.5 Problem 1

Demonstration of **`static`**'s syntax and its danger…what does the following code do to all **`CUStudent`**s?

```
class CUStudent {
   public static String location;
}

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

      // All CUStudents will now live in Ithaca:
      CUStudent.location = "Ithaca";

      // Let's check a CUStudent's location:
      CUStudent c1 = new CUStudent();
      System.out.println(c1.location);

      // Let's demonstrate the syntax and danger of static...
      // An object can access a static member if the member is
      // in the scope:
      CUStudent c2 = new CUStudent();
      c2.location = "Timbuktu";

      // Check what happened to all students:
      System.out.println(c1.location);
      System.out.println(CUStudent.location);

   }
}
```

20.6 Problem 2

Special geeky problem: why did Java pick the name “static” for **`static`**?

Why is the **`main`** method modified as **`static`**?

20.7 Problem 3

What’s wrong with the following code?

```
public class TestMain {
   public int x;
   public static void main(String[] args) {
      System.out.println(x);
   }
}
```

20.8 Problem 4

Trace the following code. What is special about the **`Student`** constructor? (There is an interesting “trick” that I am using.) What is the output?

```
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 StaticTest2 {
   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());
   }
}
```

Step 21: Aliases

21.1 Demonstration

What happens when you do run this code?

```
class Person {
   public int x;
   public Person(int n) { x=n; }
   public String toString() { return "#"+x; }
}

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

      Person p = new Person(1);
      Person q = new Person(2);

      System.out.println(p); // output: #1
      System.out.println(q); // output: #2

      p = q;

      System.out.println(p); // output: #2
      System.out.println(q); // output: #2
   }
}
```

Why does Java use **q**'s **toString** and not **p**'s **toString** after you say **p=q?** The statement **p = q** forges an alias between **p** and **q**. An ***alias*** is another name for the same thing. So, **p** becomes another name for **q**. See the next section for an explanation of how the alias is formed.

21.2 Relevant rules

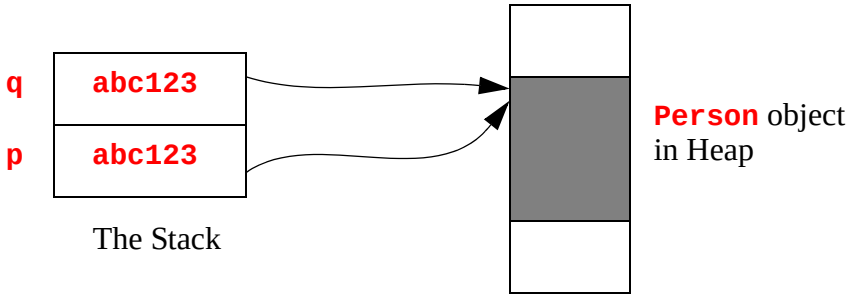
There are some interesting consequences of the following rules:

- Java variables must obey scope.
- Java is strongly typed.
- Java allows to copy variable values to other variables.
- You store address of object in a variable, not the entire object.

So, to say **p=q**,

- Both variables must be visible to each other, which they are, because they are local variables in the same method block.
- Both references must refer to the same type of object. Actually, **p** must be supertype or same type as **q**. We'll be reviewing inheritance later in CS211.
- **p** is the left side of an assignment, which means it will get the value of the expression on the right side of the assignment.
- **q** stores an address value, which is copied into **p**.

Since **p** gets the value of **q**, **p** must now point to the same object that **q** does! See below:



Consequences:

- If you say **p.x = 10**, **q.x** is now **10**, because both variables refer to the same object.
- The object that **p** originally referred to is now toast. Why?
  - That object was created in the current method and associated only with **p**. Since **p** has been reassigned, there is no connection (via a reference) to that original object.
  - To retain a connection to **p**'s original object, you would need to insert a statement before **p=q**, like **Person tmp = p**.

21.3 Problem 1

What does the following code output?

```
class Person {
   private String name;
   public 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);
      System.out.println(p2);
   }
}
```

21.4 Problem 2

Suppose that you have three **Person**s, each with a different age. Write a program that figures out which person is the oldest.

Step 22: Methods and objects (and aliases)

22.1 Problem 1: Scope revisited (yet again)

Review the following code and try to figure out the output:

```
public class Pass1 {
   public static void main(String[] args) {
      Person p = new Person();
      p.name = "Dimmu";
      change(p);
      System.out.println(p);
   }
   public static void change(Person p) {
      p.name = "Borgir";
      p = null;
   }
}
class Person {
   public String name;
   public String toString() { return name; }
}
```

If you got **Borgir** instead of **null**, congratulations! If not, read below and then try again.

22.2 The rules

Again, I try to summarize the rules that should help you understand how objects are handled:

- Java methods pass all parameters by value.
- The value of a reference variable is the address of an object.
- Another variable becomes an alias for an object when that variable is assigned to another reference.
- A local variable and parameter are only visible inside the method in which they are defined. They are not accessible outside that method's scope.

So, when I pass **p** to the change method,

- The **p** in **main** and **p** in **change** are completely different variables!
- The value of **p** passed to **change** is the address of the **Dimmu** object.
- The value of **p** inside the **change** method becomes an alias of the **Dimmu** object.
- The object's field **name** is changed when assigning **p.name**, because **change**'s **p** still refers to the **Dimmu** object.
- Setting **p** to **null** removes the address stored in **change**'s **p**. But, **main**'s **p** is still alive and stores the address of the **Dimmu** object.
- Exiting the **change** method and returning to the point in which it was called deallocates all the memory from **change**. But, **main**'s variables are still alive! That method isn't finished yet.

22.3 "Return an object" (*actually, return a reference to an object*)

Because a constructor creates an object and then returns a reference to it, you can think of the constructor call as an expression. For instance, just as `Math.sqrt(4)` returns `2`, the expression `new Person("Shagrath")` returns a value, which is a reference. So, instead of saying something like this:

```
public Something method() {
   Something s = new Something();
   return s;
}
```

try this:

```
public Something method() {
   return new Something();
}
```

22.4 Example

In your `Complex` class, you might have an add method that looks something like this:

```
public Complex add(Complex other) {
   return new Complex( r + other.r , i + other.i );
}
```

By relying on references and aliases, you can add two `Complex` numbers as follows:

```
Complex c = new Complex(1,2).add( new Complex(3,4) );
```

22.5 Problem 2

If you can demonstrate that the following code outputs `the value: 28.0`, you've pretty much nailed the rules about scope:

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

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

Step 23: Using Java's `this`

23.1 "The Current Object"

Recall that the code in a class provides a blueprint to create an object, which provides the following design for Java:

- When you call a constructor, you create one object at a time.
- When you access that object's members via the dot operator, you access just t*hat object*'s members, because *each object is unique*.

These reminders might seem obvious, but there is a subtle concept–the notion of ***the current object***.

Refer to this example that I've been using:

```
class Complex {
   private double re, im; // components of current object
   public Complex(double r, double i) {
      re = r; im = i;
   }
   public Complex add(Complex other) {
      return new Complex( re+other.re, im+other.im );
   }
}
```

Note how the **`add`** method knows that **`re`** and **`im`** refer to the current object's fields? In case you're not sure *who* that current object is, I'll show how the rules about the current object apply. The following code uses **`Complex`** from above:

```
public class TestCurrent {
   public static void main(String[] args) {
      Complex c1 = new Complex(1,2);
      Complex c2 = new Complex(3,4);
      c1.add(c2); // c1 is current, c2 is supplied
      c2.add(c1); // c2 is current, c1 is supplied
   }
}
```

When I call **`c1.add(c2)`**, Java works with **`c1`**'s members (**`re`**, **`im`**, **`add`**–of course, only **`add`** can be accessed from outside the class). The trick from a few sections ago is that the dot resembles as "has-a" or "pointer." Inside the **`add`** method, **`other`** becomes an alias for the object that **`c2`** refers to. So, when Java calls **`other.re`**, Java thinks of **`other`** as the current object, which has its own **`re`**.

If you're wondering how I circumvented the **`private`**s, look back a few sections on a nitpicking rule (Step 19.6) that explains how objects from the same class can access private members. Normally you really should use a getter method.

23.2 Representing the current object

Now and then you actually need to express the current object inside a class. There are three general situations:

- You want to call another constructor of the same class as part of another constructor.
- You want to distinguish between a field and local variable that have identical names.
- You need to assign the current object to a supplied object.

Java's keyword for the current object is **`this`**, which I think it the funniest keyword in the history of programming. The *Whitespace* programming language is really funny, too.

23.3 Constructor Chaining

Suppose you don't want an "empty" **`Person`** to be created. You could chain one constructor to another by calling a constructor of the current class, which means calling **`this(args)`**. For example, below, **`Person()`** ends up executing **`Person(String f, String l)`**, because the arguments of **`this("John","Doe")`** match it:

```
class Person {
   private String first, last;
   public Person() {
      this("John","Doe");
   }
   public Person(String f, String l) {
      first = f; last = l;
   }
}
```

Remember that the *first* statement of a constructor can be **`this(args)`** or **`super(args)`**. Eventually, the last constructor in the chain of calls in the current class must have just **`super(args)`**.

23.4 Field and Variable Name Conflicts

All methods know about the **this**, which helps you to resolve name conflicts. If a method has a variable (local or parameter) with the same name as a field, the method chooses the variable before the field! To access the field, you must tell Java, "the current's object version of name." So, in code, you would say **this.*name***.

```
class Person {
   private String name;
   public Person(String name) {
      this.name = name;
   }
}
```

In the constructor, the **String name** is a declaration of the parameter, which has nothing to do with the field! To access the field (**private String name**), you need to tell Java, "give me the current object's field **name**" with **this.name**. Note: If you do *not* have a name conflict, do *not* use **this.*name***! I don't care if you "just want to be safe!"

23.5 Storing Current Object Reference

Sometimes you need to say "the supplied object refers the current object" in some fashion. Unfortunately, until you learn more about data structures, the examples are really contrived.

Here is one of those classic problems: Suppose you want to make two **Person**s be each other's friends. To do so,

- Create a **Person** who has an unknown friend.
- Create another **Person** who has an unknown friend.
- Set the first **Person**'s friend to the second **Person**.
- Set the second **Person**'s friend to the first **Person**.

So, here is the main class, which presents the above algorithm:

```
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(p1);
      System.out.println(p2);
   }
}
```

Note that friends must be of type **Person**! Refer to the following example:

```
class Person {

   private String name;    // a Person has a name
   private Person friend;  // a Person has a friend (who is a Person)

   // Create a Person object with name:
   public Person(String name) {
      this.name = name;
   }

   // Set current Person's friend to supplied friend:
   public void makeFriends(Person friend) {
      this.friend = friend;
      friend.friend = this;
   }

   // Stringify current Person:
   public String toString() {
      return "I am "+name+"\nand my friend is "+friend.name;
   }
}
```

Explanation:

```
// Set current Person's friend to supplied friend:
   public void makeFriends(Person friend) {
      this.friend = friend;
      friend.friend = this;
   }
```

- The *parameter* **friend** and *field* **friend** have a name conflict! So, **this.friend = friend** means that the *current object's field* **friend** gets the *value of the supplied parameter* **friend** (the supplied **Person** object).
- The supplied **Person**'s **friend** needs to become the current **Person**. So, the **friend**'s **friend** (supplied object's field **friend**) gets the current **Person** (the reference to the current object).

The trick above is to remember the scope rules. When I declare **Person friend** as a parameter, Java will use that parameter when I say **friend** and **friend.*something***. C++ programmers should note that there is nothing special about the name **friend** in Java. If you prefer, change the name friend to **blurgo**, and you'll get the same result.

23.6 Problem

The above example is pretty intense already if you have never seen Java. If you feel comfortable with it, try programming this old adage: "the enemy of my enemy is my friend," but add a bit more: "…and I will be that friend's friend."

I suppose you could try to make this problem more complicated by setting more enemies and friends–throw in a few allies and neutral parties for good measure. If you want to be even more bonkers, use only underscores for names.

Step 24: Arrays

24.1 Java Arrays

General principles:

- Java arrays are objects.
- Java arrays are 1-D. To make a multidimensional array, you need to make an array of arrays.
- Java arrays have a special kind of constructor.
- Java arrays are strongly typed–all elements must have the same type.
- You cannot change the size of an array after you create it. Arrays are not dynamic!

24.2 Creating 1-D Arrays

Declaration:

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

- The type can be any valid type: primitive and reference.
- You can declare multiple names in the same statement, but we advice against it.

Assignment:

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

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

- Think of `new type[size]` as a constructor.
- `size` must be an integer (`0` or greater) or expression that evaluates to an integer.
- All values inside the array are “zero.” Think of each element as a “field” of the object. Refer to the rules for default values of fields in classes.

Example:

```
int[] x = new int[10];        // default values? 0
Person[] p = new Person[2];   // default values? null!
```

24.3 Indexing/Accessing/Storing

Expression syntax:

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

Statement syntax:

```
name[index] = expression;
name[index].method(…);
```

More rules:

- Labeling of indices starts at zero!
- ***index*** must be an integer or integer expression that corresponds to the position of an element in the array.
- If you attempt to access an index that does not exist, Java complains with an with out-of-bounds exception.
- To access an array's length, use the "secret field" **`length`**, e.g., **`x.length`**. Note that Java strings use the **`length()`** method.
- The **`[]`**s have higher precedence than dot (**`.`**).
  - So, if I want to access a **`Person`** object's name, for example, I can say **`p[2].getName();`**.
  - Of course, **`p[2]`** must first store a **`Person`** object!

24.4 Array of Objects

People often forget to fill an array with objects! Suppose that you want to print an array of **`Complex`** numbers:

```
final int SIZE = 4;
Complex[] c = new Complex[SIZE];
for (int i = 0 ; i < c.length; i++)
   System.out.println(c[i]);
```

When you run the above code, all you get is four **`null`**s. Why? The default value of all objects is zero for fields, as discussed in the previous section. So, you need first to fill the array!

```
final int SIZE = 2;
Complex[] c = new Complex[SIZE];
c[0] = new Complex(0,1);
c[1] = new Complex(1,2);
for (int i = 0 ; i < c.length; i++)
   System.out.println(c[i]);
```

24.5 Problem 1

Write a program that creates an array of **`10`** random **`Complex`** numbers.

24.6 Problem 2

Write a method **`rotate`** that returns a character array that contains the rotation of the lowercase English alphabet for an input of **`shift`**. A *rotation* is a simple form of encryption in which a character is shifted a given number of characters to the left. Assume that **`shift`** is a legal integer between 0 and 26. For instance, **`rotate(3)`** produces a character array in the order **`xyzabcdefghijklmnopqrstuvw`**. Test your program with a few strings. If you need to refresher on characters, refer to the earlier sections on the Java language.

Note that there is nothing special about a method that returns an array other than the fact that it really returns a reference to an array. Your method header for **`rotate`** might be as follows:

```
public static char[] rotate(int shift)
```

The last statement will return an expression of type **`char[]`**.

24.7 Initializer Lists

If you have a small array and known values, you can use an ***initializer list*** to declare, assign, and store in one statement.

```
type[] name = { e1 , … , en } ;
```

There are some interesting rules that seem to depart from standard Java convention:

- The entire statement must be in on one line.
- You need to place a semicolon after the last brace.

Examples:

```
int[] x = { 1 , 2 , 3 } ;
Person[] p = { new Person("A"), new Person("B") } ;
```

24.8 Anonymous Arrays

Sometimes someone decides to return to return an initializer list, as in

```
return {1,2,3} ;
```

Unfortunately, Java does not permit this syntax. Instead you have to use the more general form of the initializer list, which is called an ***anonymous array***:

```
new type[] { e1 , … , en }
```

CS211 will cover anonymous classes later on in the semester, so don’t worry too much about this particular topic for now.

24.9 Arrays of Arrays (Multidimensional Arrays)

To make a multidimensional array, you need to use an array of arrays. Why?

- An array can hold references to objects.
- An array is an object.

Syntax:

- completely specified: hyper-dimensional rectangular box:
  ***type*`[][][]…` *name* `=new` *type*`[`*size1*`][`*size2*`][`*size3*`]…`**
- partially specified: hyper-dimensional ragged:
  ***type*`[][]…` *name* `=new` *type*`[`*size1*`][]…`**

The **`length`** field:

- size of 1st dimension: ***name*`.length`**
- size of 2nd dimension: ***name*`[i].length`**
  (also size of **`i`**th array, like row or column)
- size of 3rd dimension: ***name*`[i][j].length`**

2-D examples:

```
public class Test2DArray {
   public static void main(String[] args) {

      int[][] x = new int[2][3]; // just defaults
      printArray(x);

      int[] a = { 1 , 2 , 3 } ;
      int[] b = { 4 , 5 } ;
      int[][] y = new int[2][];
      y[0] = a;
      y[1] = b;
      printArray(y);
   }

   public static void printArray(int[][] x) {
      for (int row = 0 ; row < x.length ; row++ ) {
         for (int col = 0 ; col < x[row].length ; col++ )
            System.out.print(x[row][col] + "");
         System.out.println();
      }
   }
}
```

3-D Example:

```
public class Test3DArray {
   public static void main(String[] Sgra) {

      int[][][] x = new int[3][][];
      x[0] = new int[2][];
      x[0][1] = new int[4];

   }
}
```

24.10 Problem 3

Trace the following code. Write the **printArray** method. To print 3-D, print each "layer" of the 3-D array by thinking of the array as a 1-D array of 2-D arrays.

```
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();
   } // 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);
            }
         }
      }
   }

}
```

24.11 Simulating Dynamic Arrays

Java arrays must have a size, which cannot be changed once the array is created. But there are a couple of useful tricks.

Objects:

```
class IntArray {
   private int[] x;
   public IntArray(int n) {
      x = new int[n];
   }
}
```

- You can call `new IntArray(size)` with an any valid size.
- Of course, once you create that object, its internal array x is now fixed.

Methods:

- Actually, we've already been using the trick of copying reference values.
- When you call a method, like `something(int[] x)`, the parameter `x` gets the address of the array that you supply.

There is a formal "dynamic array" in Java, which is called a `Vector`. See next section.

24.12 Iterating through an array with `for`-each

Java 5 introduced the `for`-each control structure, which I show briefly in Step 12.8. When using arrays, the index for iteration can be a nuisance. So, to condense the following common pattern:

```
Type[] a = Expression;
for (int i = 0; i < a.length; i++) {
   Type e = a[i];
   statements using e
}
```

you could use the following pattern:

```
Type[] a = Expression;
for (Type e : Expression) {
   statements using e
}
```

The pattern above means "for each `e` in `expression`, do something."

(continued on next page)

The following example demonstrates the **for**-each by printing a collection of strings:

```
String[] s = {"A", "B", "C"};
for (String e : s)
   System.out.print(e);
System.out.println();
```

The above example will output **ABC**.

You can also iterate over general collections of data, which we will explore when reaching **Vector**s in Step 30. For more information, refer to Java's general explanation of syntax and semantics that I used to write this section at this website: http://jcp.org/aboutJava/communityprocess/jsr/tiger/enhanced-for.html.

24.13 Problem 4

Write a program that does the following two tasks:

- Adds and outputs the sum of all elements in an array of integers. Use this array in your **main** method:**int[] a = {1 , 2 , 3 }**.
- Searches the same array **a** for the number 2 in a method called **find2**. Upon finding 2, method **find2** returns **true** without inspecting any further elements.

Step 25: Class `Object`

25.1 Background

I have tried to avoid too much conversation about inheritance, because it is a subject we go over again in CS211. But, class **`Object`** is important–all classes in Java extend **`Object`**. So, **`Object`** is a supertype of all reference types. What makes this notion extremely useful for data structures is that a supertype reference can store a subtype reference with no hassle. You can then write general data structures that have contents of type **`Object`**, which allows you to store every kind of object. Pretty handy!

25.2 Problem

How many methods do all objects in Java automatically inherit?

25.3 Example: `equals` revisited

You can define equality for any class using **`Object`**'s **`equals`** method. In the example, below, I define equality for class **`A`** as equivalence of **`A`**'s instance variable **`k`**:

```
public class Equals {
   public static void main(String[] args) {
      A a1 = new A(1);
      A a2 = new A(1);
      System.out.println(a1.equals(a2));
   }
}

class A {
   public int k;
   public A(int k) { this.k = k; }
   public boolean equals(Object other) {
      return k == ((A) other).k;
   }
}
```

25.4 Problems

What is the output of **`Equals`**? Would you get the same output if you print out **`a1==a2`**? Why or why not?

If you define a class **`B`** that extends **`A`**,

```
class B extends A {
   public B(int k) { super(k); }
}
```

will an object of **`B`** equal an object **`A`** and if they have the same value of **`k`**? Demonstrate your answer with **`new A(1).equals(new B(1))`** and
`new B(2).equals(new A(2))`.

Step 26: Java API and `import`

26.1 The Gist

Java provides many, many kinds of classes that can help you rapidly build programs without having to worry about nitty-gritty details of system calls, painting pixels, and "recreating the wheel" for all kinds of general classes. Check out the ***Java API*** (application programming interface) link on the CS211 website: http://java.sun.com/j2se/1.5.0/docs/api/index.html.

CS211 tends to use the following packages (collections of classes), which you can scroll through on the top, left corner of the API website:

- **`java.lang`**: all classes automatically loaded for your programs
- **`java.util`**: data structures and algorithms
- **`java.awt.event`**: GUIs
- **`javax.swing`**: GUIs

26.2 Loading Classes

Except for **`java.lang`**'s classes, you need to import an API's entire package or particular class if you have identified something that you wish to use.

To access a particular class, use **`import package.class`** at the top of your program:

```
import java.util.Arrays;

public class API {
   public static void main(String[] args) {
      String[] s1 = {"A", "B", "C"};
      String[] s2 = {"A", "B", "D"};
      System.out.println(Arrays.deepEquals(s1,s2));
   }
}
```

Occasionally, you may see programmer write the entire class path directly in the code when the class is used. To import *all* classes from a package, use the wildcard **`*`**, as follows:

```
import java.util.*;

public class API {
   public static void main(String[] args) {
      String[] s1 = {"A", "B", "C"};
      String[] s2 = {"A", "B", "D"};
      System.out.println(Arrays.deepEquals(s1,s2));
   }
}
```

You may have multiple **`import`** statements in your file.

Step 27: Constants: Static Import and Enumerations (enum)

27.1 Motivation

As you go through the API, you will discover that the API's classes not only contain useful methods, but several defined constants as well. You may eventually define your constants. Up until Java 5, programmers would define constants in interfaces, and then implement those interfaces, as follows:

```
interface Blah {
   public static final int ANSWER=42;
}

public class Something implements Blah {
   public static void main(String[] args) {
      System.out.println(ANSWER);
   }
}
```

Note that type **Blah** is meaningless with respect to any class that might implement it.

27.2 Static Import

You can actually import constants from a class or interface, as discussed in http://java.sun.com/j2se/1.5.0/docs/guide/language/static-import.html.

27.3 Problem 1

Write a program that imports all of the constants from class **Math** in the API. Output some of those values.

27.4 Enumerations (also, Enumerated Types or just enums)

An **enum** is a special kind of class in Java that allows you define types and unique values. One way to think of an **enum** is a user-defined "primitive value." In other languages, an enum is effectively an integer, though you see names, not numbers. However, in Java, an **enum** is object oriented.

Check out http://java.sun.com/j2se/1.5.0/docs/guide/language/enums.html to learn a number of **enum** tricks. The following example demonstrates a few:

```
enum Color {BLUE, RED}

enum Coin {
   penny(1), dime(10);
   private final int value;
   Coin (int value) { this.value=value;}
   public int value() {return value;}
   public String toString() {
      return this==penny? "p" : "d";
   }
}

public class Enums {

   public static void main(String[] args) {

   Color marble = Color.BLUE;
   Coin c = Coin.penny;
   System.out.println(Coin.dime);
   System.out.println(c.value()+Coin.dime.value());

   }
}
```

27.5 Problem 2

Review the above link concerning **enum**s. Write a program that uses **enum**s to model playing cards. Your program randomly shuffle a random deck of cards and then print out each card in order, from top to bottom.

Step 28: Wrapper Classes

28.1 Java Types

For better or for worse, Java has two kinds of types:

- *primitive*: **`int`**, **`double`**, **`boolean`**, **`char`**, …
- *reference*: all classes, interfaces, enums, **`null`**

To make generic data structures, you will generally use class **`Object`**, which means that you might need a way to convert primitives to objects. Java provides ***wrapper classes*** for all primitive types, which you can find in the **`java.lang`** package. You never need to import any classes from this package, since Java automatically uses them.

For example, to create an **`Integer`** object from the primitive value 4, you would do this:

```
Integer i = new Integer(4);
```

You can now store **`i`** in an **`Object`** variable:

```
Object o = i;
```

To get a handle on how **`Integer`** works, it has this approximate structure:

```
class Integer {
   private int value;
   public Integer(int value) {
      this.value = value;
   }
   // more methods--see java.lang.Integer in API
}
```

28.2 Do we still need wrapper classes?

Java 5 has autoboxing of primitive types, which automatically converts them to their equivalent wrapper types. See Step 29 for more detail. However, you may still need to wrap object types within others, depending on the application. For example, consider the "dynamic array" of Step 24.11–we wrap an array, which is a class, with another kind of class.

28.3 Problem 1

Without relying on autoboxing, create an array of integers

```
int[] a = { 1, 2, 3, 4 };
```

that can be passed into a method

```
int addInts(Object[] x)
```

that returns the integer sum of all elements in **`x`**.

Step 29: Autoboxing

29.1 Common Irritation

In older versions of Java, you could not mix reference and primitive types. For example, given a data structure of **`Object`** types, if a user tried to enter an integer, you had to convert the value to its wrapper class Integer. For example, you might see this sort of code in older Java:

```
Object[] o = { new Integer(1), new Integer(2), new Integer(3) };
```

29.2 Autobox/Unbox

To ease your frustration, Java 5 will automatically "upgrade" (***autoboxing***) or "downgrade" (***unboxing***) a primitive type to its wrapper equivalent.

For example, the following code demonstrates autoboxing of the integer 1:

```
Object o = 1 + new Integer(2);
```

When unboxing, an reference type can be converted to its primitive type, as follows:

```
int x = 1 + new Integer(2);
```

29.3 Some issues

Do not rely on autoboxing/unboxing if you have several numerical computations, as these features involve a performance hit. For more information, review http://java.sun.com/j2se/1.5.0/docs/guide/language/autoboxing.html.

29.4 Problem 1

If you output **`x`** and **`o`** in the above example, you get the same answer. Why?

29.5 Problem 2

An **`Integer`** object can be **`null`**. So, can you unbox **`null`**? What happens if you do?

Step 30: `Vector`s

30.1 Dynamic data structure

In CS211, you will learn about special ADTs that hold information and provide methods to store, access, and analyze that data. Of course, arrays provide an extremely quick and easy way to store and access information. In fact, most languages provide them as part of the language. However, arrays are ***static data structures***, which means they do not change in size. Once created, the only way to insert more information is to create another array, which can be computationally expensive if you have huge large sets. A ***dynamic data structure*** can indeed change in size.

30.2 Vector

A nice bridge between arrays and dynamic data structures is a ***vector***, which resembles an array. However, a vector can grow and shrink! So, when you need to store information, a vector allows you to make an initial guess on much space you need. If need more space, you can simply add more elements. The vector will automatically grow. You can also remove elements, which will cause the vector to shrink.

30.3 Java API and `import` statements reminder

Class `Vector` is part of `java.util`. According to Step 26, you will need to import `Vector`. You can use either of the following `import` statements at the top of your file:

- Import all classes in the `java.util` package, which contains `Vector`:
  ```
  import java.util.*;
  ```
- Import just `Vector`:
  ```
  import java.util.Vector;
  ```

In general, I tend to import all of `java.util`, figuring that I'm bound to need something else–it's a very nice collection of data structures!

30.4 Generics

You might get a compiler warning if you do not use something called ***generics***. For example, if you have a `Vector` of `String`s and/or `Integer`s, you should use the following syntax when declaring your `Vector`:

```
Vector<String,Integer> v = new Vector<String,Integer> ();
```

For more information about generics, check out http://java.sun.com/j2se/1.5.0/docs/guide/language/generics.html.

30.5 Problem 1

Without relying on Java's autoboxing feature, write a program that starts with a `Vector` of no elements. Add a sequence of random integers. For more fun, try a bunch of different types. Print the entire contents of `Vector` in order. Then, remove all the elements until the `Vector` size is zero.

30.6 Problem 2

Repeat Problem 1, except that now you can rely on autoboxing.

30.7 Problem 3

Write a program that creates a randomly-sized (0 to 20, inclusive) vector of random positive integers. Your program will print out the vector, delete all the odd integers from the `Vector`, and then print the remaining elements. Note that you must use only the original `Vector` that you create. So, there should be only one `Vector` for adding and deleting elements!

30.8 Problem 4

Write a program that does the following:

- Builds a `Vector` of 10 random integers that range from 1 to 10, inclusive.
- Searches the `Vector` for all of its even values.
- Outputs the sum of the even values.

For the iteration, use a for-each structure. This structure will work because Vector conveniently implements the `Collection` interface, which is part of the java.util API. For more information, refer to http://jcp.org/aboutJava/communityprocess/jsr/tiger/enhanced-for.html.

Step 31: User I/O

31.1 Class `Scanner`

If you have learned Java with version 1.4.2 or earlier, you were likely using a non-standard class to assist with user and/or file input. Java 5 introduced a great new class called **`Scanner`** that simplifies the process of obtaining input.

31.2 Example

```
import java.util.*;

public class UserIO {

   public static void main(String[] args) {

      // "plain" user I/O (no type checking):
      Scanner in = new Scanner(System.in);
      System.out.print("Please enter a legal integer: ");
      in.nextInt();

      // fancier user I/O:
      boolean stop = false;
      System.out.print("Please enter an integer: ");
      while(!stop) {
         Scanner s = new Scanner(System.in);
         try {
            s.nextInt();
            stop = true;
         }
         catch (Exception e) {
            System.out.print("Not legal! Re-enter: ");
         }
      }
   }

}
```

31.3 Problem 1

Run the above program. What happens if you enter non-integer input at the first prompt?

31.4 Problem 2

Although you might not have seen a **`try-catch`** structure beforehand, explain what the second input structure does when you run the program.

Step 32: Other things for the future…?

32.1 Output and Formatting

32.2 File I/0

32.3 Move Applications.html and Java links to this tutorial

32.4 Javadoc

32.5 Define and use packages

32.6 Exceptions

32.7 Generics