

# *Basic Java Refresher*

---

A quick run-through of basic Java features and syntax in a single handout

## Student Java Example

- As a first example of a java class, we'll look at a simple "Student" class. Each Student object stores an integer number of units and responds to messages like `getUnits()` and `getStress()`. The stress of a student is defined to be `units * 10`.

## Implementation vs. Interface

- In OOP, every class has two sides...
  1. The **implementation** of the class -- the data structures and code that implement its features.
  2. The public **interface** that the class exposes for use by other classes.
- With a good OOP design, the interface is smaller and simpler than the implementation. The public interface is as simple and logical as possible -- exposing only aspects that the clients care about.
- We'll use the word "client" to refer to code that uses the public interface of a class and "implementation" when talking about the guts of a class.

## Student Client Side

- First we'll look at some client code of the Student class.
- Client code will typically allocate objects and send them messages.
- With good OOP design, being a client should be easy.
- Client code plan
  - Allocate objects with "new" -- calls constructor
  - Objects are always accessed through pointers -- shallow, pointer semantics
  - Send messages -- methods execute against the receiver (also known as "calling" a method on the receiver object)
  - Can access public elements, but not private/protected elements from client side

## Object Pointers

- The declaration "Student x;" declares a pointer "x" to a Student object, but does not allocate the object yet.
- Java has a very simple and uniform memory system. Objects and arrays are allocated in the heap and accessed through pointers.

- There is no "&" operator to make a pointer to something in the stack and there is no pointer arithmetic. The only pointers that exist in java point to objects and arrays in the heap -- simple.
  - Objects and arrays are allocated with the "new" operator (below).
  - Using = on an object pointer just copies the pointer, so there are multiple pointers to the one object (aka "shallow" or "sharing").
  - Likewise, using == on object pointers just compares the pointers (for some classes, the equals() method will do a "deep" comparison of two objects).

## new Student() / Constructor

- The "new" operator allocates a new object in the heap, runs a constructor to initialize it, and returns a pointer to it.
  - `x = new Student(12)`
- Classes define "constructors" that initialize objects at the time new is called. Constructors are similar to methods, but they cannot be called at will. Instead, they are run automatically by the JVM when a new object is created.
- The word "constructor" is generally written as "ctor"
- The constructor has the same name as the class. e.g. the constructor for the "Student" class is named "Student".
- There can be multiple constructors. They are distinguished at compile time by having different arguments -- this is called "overloading".
  - e.g. The Student class defines one ctor that takes an int argument, and one ctor that takes no arguments.
  - The ctor that take no arguments is called the "default" ctor. If it is defined, the system uses it by default when no other ctor is specified.

## Message send

- Send a message to an object.
  - `a.getUnits();`
  - `b.getStress();`
- Finds the matching method in the class of the receiver, executes that method against the receiver and returns.
- The Java compiler will only allow message sends that the receiver actually responds to.
- Instead of "sending a message to the object", we could say we "called that method on the object". The distinction between a message and a method is not important, until we get to inheritance later on.

## Object Lifecycle

- The client allocates objects and they are initialized by the class ctor code
- The client then sends messages which run class method code on the objects.

- The client essentially makes requests -- all the code that actually operates on the objects is defined by the class, not the client.
- As a result, if the class is written correctly, the client should not be able to introduce new bugs in the class and visa-versa.
- This is the benefit of using public/private to keep the client and the implementation separate.

## Student Client Side Code

```
// Make two students
Student a = new Student(12); // new 12 unit student
Student b = new Student();// new 15 unit student (default ctor)

// They respond to getUnits() and getStress()
System.out.println("a units:" + a.getUnits() +
    " stress:" + a.getStress());

System.out.println("b units:" + b.getUnits() +
    " stress:" + b.getStress());

a.dropClass(3); // a drops a class

System.out.println("a units:" + a.getUnits() +
    " stress:" + a.getStress());

// Now "b" points to the same object as "a" (pointer copy)
b = a;
b.setUnits(10);

// So the "a" units have been changed
System.out.println("a units:" + a.getUnits() +
    " stress:" + a.getStress());

// NOTE: public vs. private
// A statement like "b.units = 10;" will not compile in a client
// of the Student class when units is declared protected or private

/*
OUTPUT...
a units:12 stress:120
b units:15 stress:150
a units:9 stress:90
a units:10 stress:100
*/
```

# Student Implementation Side

- Now we'll look at the implementation of the Student class. The complete code listing for Student is given at the end of this section..

## Class Definition

- A class defines the instance variables and methods used by its objects.
- Each variable and method may be declared as "public" if it may be used by clients, or "private" or "protected" if it is part of the implementation and not for use by clients.
- The compiler and JVM enforce the public/private scheme.

## Public Interface

- The most common public/private scheme is...
- All ivars are declared private
- Methods to be used by clients are declared public -- these make up the interface that the class exposes to clients.
- Utility methods for the internal use of the class are declared private.

## Java Class

- The convention is that java classes have upper case names like "Student" and the code is in a file named "Student.java".
- By default, java classes have the special class "Object" as a superclass. We'll look at what that means later when we study superclasses.
- Inside the Student.java file, the class definition looks like...

```
public class Student extends Object {
    ... <definition of the Student ivars and methods> ....
}
```

- The "extends Object" part can be omitted, since java classes extend Object by default if there is no "extends" clause.
- There are not separate .h and .c files to keep in synch -- the class is defined in one place.
  - This is a nice example of the "never have two copies of anything" rule. Keeping duplicate info in the .h and .c files in synch was a bore -- better to just have one copy.

## Instance Variables

- Instance variables (ivars) are declared like ordinary variables -- a type followed by a name.

```
protected int units;
```

- An ivar defines a variable that each object of this class will have -- allocates a slot inside each object. In this case, every Student object has an int ivar called "units" within it.
- The object itself is allocated in the heap, and its ivars are stored inside it. The ivars may be primitive types, such as int, or they may be pointers to other objects or arrays.

## public/private/protected

- An ivar or other element declared private is not accessible to client code. The element is only accessible to the implementation inside the class.
- Suppose on the client side we have a pointer "s" to a Student object. The statement "s.units = 13;" in client code will not compile if "units" is private or protected.
- "protected" is similar to private, but allows access by subclasses or other classes in the same package (we will not worry about those cases)
- "public" makes something accessible everywhere
- There is also a "default" protection level that you get when no public, private, or protected keyword is specified. In that case, the element is accessible to all other classes in the same package as the compiled class. This is an odd case, and I recommend against using it.

## Constructor (ctor)

- A constructor has the same name as the class.
- It runs when new objects of the class are created to set up their ivars.

```
public Student(int initUnits) {
    units = initUnits;
}
```

- A constructor does not have a return type (unlike a method).
- New objects are set to all 0's first, then the ctor (if any) is run to further initialize the object.
- Classes can have multiple ctors, distinguished by different arguments (overloading)
- If a class has constructors, the compiler will insist that one of them is invoked when new is called.
- If a class has no ctors, new objects will just have the default "all 0's" state. As a matter of style, a class that is at all complex should have a ctor.

- Bug control
  - Ctors make it easier for the client to do the right thing since objects are automatically put into an initialized state when they are created.
- Every ivar goes in Ctor
  - Every time you add an instance variable to a class, go add the line to the ctor that inits that variable.
  - Or you can give an initial value to the ivar right where it is declared, like this...  
"private int units = 0;" -- there is not agreement about which ivar init style is better.

## Default Ctor

- A constructor with no arguments is known as the "default ctor".

```
public Student() {
    units = 15;
}
```

- If a class has a default ctor, and a client creates an instance of that class, but without specifying a ctor, the default ctor is automatically invoked.
- e.g. new Student() -- invokes the default ctor, if there is one.

## Method

- A method corresponds to a message that the object responds to

```
public int getStress() {
    return(units * 10);
}
```

- When a message is sent to an object, the corresponding method runs against that receiver.
- Methods may have a return type, int in the above example, or may return void.
- Message-Method Lookup sequence
  - Message sent to a receiver
  - Receiver knows its class and looks for a matching method
  - The matching method executes against the receiver

## Receiver Relative Style (Method, Ctor)

- Method code runs "on" or "against" the receiving object
- Ivar read/write operations in the method code use the ivars of the receiver
- Method code is written in a "receiver relative" style where the state of the receiver is implicitly present. This makes data access very convenient in method code compared to writing in straight C.

- e.g. the "units" ivar in the Student methods is automatically that of the receiver
- Likewise, sending a message to the same receiver from inside a method requires no extra syntax.
- e.g.. inside the Student dropClass() method, the code sends the setUnits() message to change the number of units with the simple syntax:  
setUnits(units - drop);

## "this" -- receiver

- "this" in a method
  - "this" is a pointer to the receiver
  - Don't write "this.units", write: "units"
  - Don't write "this.setUnits(5)", write "setUnits(5);"
- Some programmers, like sprinkling "this" around to remind themselves of the OOP structure involved, but I find it distracting. The nice thing about OOP is the effortlessness of the receiver-relative style.
- A common use of "this" is when one object is trying to register itself with another object.

## ivar vs. local var

- Usually, you simply refer to each ivar by its name -- e.g. "units". Sometimes you have a local var or parameter with the same name as the ivar. In that case, "units" is the local var, and the expression "this.units" refers to the ivar (see the Student.setUnits() method). Having a local var with the same name as an ivar is a stylistically questionable, but it can be handy sometimes. Some people prefer to give ivars a distinctive name, such as always starting with an "my" -- e.g. myUnits.

## Student.java Code Example

```
// Student.java
/*
Demonstrates the most basic features of a class.

A student is defined by their current number of units.
There are standard get/set accessors for units.

The student responds to getStress() to report
their current stress level which is a function
of their units.

NOTE A well documented class should include an introductory
comment like this. Don't get into all the details -- just
introduce the landscape.
*/
public class Student extends Object {
    // NOTE this is an "instance variable" named "units"
    // Every Student object will have its own units variable.
```

```

// "protected" and "private" mean that clients do not get access
protected int units;

/* NOTE
"public static final" declares a public readable constant that
is associated with the class -- it's full name is Student.MAX_UNITS.
It's a convention to put constants like that in upper case.
*/
public static final int MAX_UNITS = 20;
public static final int DEFAULT_UNITS = 15;

// Constructor for a new student
public Student(int initUnits) {
    units = initUnits;
    // NOTE this is example of "Receiver Relative" coding --
    // "units" refers to the ivar of the receiver.
    // OOP code is written relative to an implicitly present receiver.
}

// Constructor that that uses a default value of 15 units
// instead of taking an argument.
public Student() {
    units = DEFAULT_UNITS;
}

// Standard accessors for units
public int getUnits() {
    return(units);
}

public void setUnits(int units) {
    if ((units < 0) || (units > MAX_UNITS)) {
        return;
        // Could use a number of strategies here: throw an
        // exception, print to stderr, return false
    }
    this.units = units;
    // NOTE: "this" trick to allow param and ivar to use same name
}

/*
Stress is units *10.

NOTE another example of "Receiver Relative" coding
*/
public int getStress() {
    return(units*10);
}

/*
Tries to drop the given number of units.
Does not drop if would go below 9 units.

```

```

    Returns true if the drop succeeds.
    */
public boolean dropClass(int drop) {
    if (units-drop >= 9) {
        setUnits(units - drop);    // NOTE send self a message
        return(true);
    }
    return(false);
}

/*
Here's a static test function with some simple
client-of-Student code.
NOTE Invoking "java Student" from the command line runs this.
It's handy to put test/demo/sample client code in the main() of a class.
*/
public static void main(String[] args) {
    // Make two students
    Student a = new Student(12);    // new 12 unit student
    Student b = new Student();    // new 15 unit student

    // They respond to getUnits() and getStress()
    System.out.println("a units:" + a.getUnits() +
        " stress:" + a.getStress());

    System.out.println("b units:" + b.getUnits() +
        " stress:" + b.getStress());

    a.dropClass(3);    // a drops a class

    System.out.println("a units:" + a.getUnits() +
        " stress:" + a.getStress());

    // Now "b" points to the same object as "a"
    b = a;
    b.setUnits(10);

    // So the "a" units have been changed
    System.out.println("a units:" + a.getUnits() +
        " stress:" + a.getStress());

    // NOTE: public vs. private
    // A statement like "b.units = 10;" will not compile in a client
    // of the Student class when units is declared protected or private

    /*
    OUTPUT...
    a units:12 stress:120
    b units:15 stress:150
    a units:9 stress:90
    a units:10 stress:100
    */
}

```

}

# Java Features

## Inheritance -- ignore for now

- OOP languages have an important feature called "inheritance" where a class can be declared as a "subclass" of another class, known as the superclass.
- In that case, the subclass inherits the features of the superclass. This is a tidy way to give the subclass features from its superclass -- a form of code sharing.
- This is an important feature in some cases, but we will cover it a little later.
- By default in Java, classes have the superclass "Object" -- this means that all classes inherit the methods defined in the Object class.

## Java Primitives

- Java has "primitive" types, much like C. Unlike C, the sizes of the primitives are fixed and do not vary from one platform to another, and there are no unsigned variants.
  - boolean -- true or false
  - byte -- 1 byte
  - char -- 2 bytes (unicode)
  - int -- 4 bytes
  - long -- 8 bytes
  - float -- 4 bytes
  - double - 8 bytes
- Primitives can be used for local variables, parameters, and ivars.
- Local variables are allocated on the runtime stack when the code runs, just as in C.. At runtime, primitives are simple and work fast.
- Primitives may be allocated **inside** objects or arrays, however, it is not possible to get a pointer to a primitive itself (there is no & operator). Pointers only work for objects and arrays in the heap -- this makes pointers much simpler in Java than in C or C++.
- Java is divided into two worlds: primitives work in simple ways and there are no pointers, while objects and arrays only work through pointers. The two worlds are separate, and the boundary between the two can be a little awkward.
- There are "wrapper" classes Integer, Boolean, Float, Double... that can hold a single primitive value. These classes are "immutable", they cannot be changed once constructed. They can finesse, to some extent, the situation where you have a primitive value, but need a pointer to it. Use intValue() to get the int value out of an Integer object.
- Use the static method Integer.parseInt(String) -> int to parse a String to an int
- Use the static method Integer.toString(int) -> String to make a String out of an int

## Arrays

- Java has a nice array functionality built in to the language.
- An array is declared according to the type of element -- an `int[]` array holds ints, and a `Student[]` array holds Student objects.
- Arrays are always allocated in the heap with the "new" operator and accessed through pointers (like objects)
- An array may be allocated to be any size, although it may not change size after it is allocated (i.e. there is no equivalent to the C `realloc()` call).
- Array Declaration
  - `int[] a;` -- a can point to an array of ints (the array itself is not yet allocated)
  - `int a[];` -- alternate syntax for C refugees -- do not use!
  - `Student[] b;` -- b can point to an array of Student objects. Actually, the array will hold pointers to Student objects.
- `a = new int[100];`
  - Allocate the array in the heap with the given size
  - Like allocating a new object
  - The array elements are all zeroed out when allocated.
  - The requested array length does not need to be a constant -- it could be an expression like `new int[2*i + 100];`
- Array element access
  - Elements are accessed `0..len-1`, just like C and C++
  - Java detects array-out-of-bounds access at runtime
  - `a[0] = 1;` -- first element
  - `a[99] = 2;` -- last element
  - `a[-1] = 3;` -- runtime array bounds exception
- `a.length` -- returns the length of the array (read-only)
  - Arrays know their length -- cool!
  - NOT `a.length()`
- Arrays have compile-time types
  - `a[0] = "a string";` // NO -- int and String don't match
  - At compile time, arrays know their element type and detect type mismatches such as above
  - The other Java collections, such as `ArrayList`, do not have this compile time type system error catching, although compile time types are being added for Java 1.5
- `Student[] b = new Student[100];`
  - Allocates an array of 100 Student pointers (initially all null)
  - Does not allocate any Student objects -- that's a separate pass

## Int Array Code

- Here is some typical looking int array code -- allocate an array and fill it with square numbers: 1, 4, 9, ...

- (also, notice that the "int i" can be declared right in the for loop -- cute.)

```
{
    int[] squares;
    squares = new int[100];    // allocate the array in the heap

    for (int i=0; i<squares.length; i++) { // iterate over the array
        squares[i] = (i+1) * (i+1);
    }
}
```

## Student Array Code

- Here's some typical looking code that allocates an array of 100 Student objects

```
{
    Student[] students;

    students = new Student[100]; // 1. allocate the array

    // 2. allocate 100 students, and store their pointers in the array
    for (int i=0; i<students.length; i++) {
        students[i] = new Student();
    }
}
```

## Array Literal

- There's a syntax to specify an array and its contents as part of an array variable declaration.
- This is called an "array constant" or an "array literal".
  - String[] words = { "hello", "foo", "bar" };
  - int[] squares = { 1, 4, 9, 16 };
  - 
  - // in this case, we call new to create objects in the array
  - Student[] students = { new Student(12), new Student(15) };

## Anonymous array

- Alternately, you can create outside of a variable declaration like this...
  - ... new String[] { "foo", "bar", "baz" } ...

## Array Utilities

- Java has a few utility functions to help with arrays...
- There is a method in the System class, System.arraycopy(), that will copy a section of elements from one array to another. This is likely faster than writing the equivalent for-loop yourself.
  - System.arraycopy(source array, source index, dest array, dest index, length);
- Arrays Class

- The Arrays class contains many convenience methods that work on arrays -- filling, searching, sorting, etc.

## Multidimensional Arrays

- An array with more dimensions is allocated like this...
  - `int[][] big = new int[100][100];` // allocate a 100x100 array
  - `big[0][1] = 10;` // refer to (0,1) element
- Unlike C, a 2-d java array is not allocated as a single block of memory. Instead, it is implemented as a 1-d array of pointers to 1-d arrays.

## String

- Java has a great built-in String class. See the String class docs to see the many operations it supports.
- Strings (and char) use 2-byte unicode characters -- work with Kanji, Russian, etc.
- String objects use the "immutable" design style
  - Never change once created
  - i.e. there is no `append()` or `reverse()` method that changes the string state
  - To represent a different string state, create a new string with the different state
  - The immutable style, has an appealing simplicity to it -- easy for clients to understand.
  - The immutable style happens to avoid many complexities when dealing with (a) multiple pointers sharing one object, and (b) multiple threads sharing one object.
  - On the other hand, the immutable style can cause the program to work through a lot of memory over time which can be expensive.
- String constants
  - Double quotes (") build String objects
  - `"Hello World!\n"` -- builds a String object with the given chars and returns a pointer to it
  - The expression `new String("hello")` is a little silly, can just say `"hello"`.
  - Use single quotes for a char `'a'`, `'B'`, `'\n'`
- `System.out.print("print out a string");` // or use `println()` to include the endline
- String + String
  - `+` concatenates strings together -- creates a new String based on the other two

```
String a = "foo";
String b = a + "bar";    // b is now "foobar"
```

- String + int
  - `+` between a String and an int, converts the int to a String and concatenates it all together:

```
int i = 6;
String s = "hello " + i + "there " + (i*10);
// s is now "hello 6there 60"
```

- toString()
  - Many objects support a toString() method that creates some sort of String version of the object -- handy for debugging. print(), println(), and + will use the toString() of any object passed in. The toString() method is defined up in the Object class, so that's why all classes respond to it. (More on this when we talk about inheritance.)

## String Methods

- Here are some of the representative methods implemented in the String class
- Look in the String class docs for the many messages it responds to
  - int length() -- number of chars
  - char charAt(int index) -- char at given 0-based index
  - int indexOf(char c) -- first occurrence of char in the string, or -1
  - int indexOf(String s)
  - boolean equals(Object) -- test if two strings have the same characters
  - boolean equalsIgnoreCase(Object) -- as above, but ignoring case
  - String toLowerCase() -- return a new String, lowercase
  - String substring(int begin, int end) -- return a new String made of the begin..end-1 substring from the original

## Typical String Code

```
{
    String a = "hello"; // allocate 2 String objects
    String b = "there";
    String c = a;      // point to same String as a -- fine

    int len = a.length(); // 5
    String d = a + " " + b; // "hello there"

    int find = d.indexOf("there"); // find: 6

    String sub = d.substring(6, 11); // extract: "there"

    sub == b; // false (== compares pointers)
    sub.equals(b); // true (a "deep" comparison)
}
```

## StringBuffer

- StringBuffer is similar to String, but can change the chars over time. More efficient to change one StringBuffer over time, than to create 20 slightly different String objects over time.

```
{
    StringBuffer buff = new StringBuffer();
    for (int i=0; i<100; i++) {
        buff.append(<some thing>); // efficient append
    }
    String result = buff.toString(); // make a String once done with appending
}
```

}

## System.out

- System.out is a static object in the System class that represents standard output. It responds to the messages...
  - println(String) -- print the given string on a line (using the end-line character of the local operating system),
  - print(String) -- as above, but without and end-line
- Example
  - System.out.println("hello"); -- prints to standard out

## == vs equals()

- == -- compare primitives or pointers
  - Use == to compare primitives, such as int or car or boolean
  - Use == to compare a pointer to null
  - With two pointers, == will test if the two pointers point to the same object, which is not the same as testing if they are byte-for-byte the same.
- boolean equals(Object other)
  - There is a default definition in the Object superclass that just does an == compare of (this == other), so it's just like using == directly.
  - However, many of the library classes such as String, Integer, Color, etc. override equals() to provide "deep" byte-by-byte compare version. See the docs for a particular class to see if it overrides equals().
- String Example
  - String a = new String("hello"); // in reality, just write this as "hello"
  - String a2 = new String("hello");
  - a == a2 // false
  - a.equals(a2) // true
  -
- Foo Example
  - Foo a = new Foo("a");
  - Foo a2 = new Foo("a");
  - a == a2 // false
  - a.equals(a2) // ??? -- depends on Foo overriding equals()

## Garbage Collector GC

- String a = new String("a");
- String b = new String("b");
- a = a + b; // a now points to "ab"
- Where did the original a go?

- It's still sitting in the heap, but it is "unreferenced" or "garbage" since there are no pointers to it. The GC thread comes through at some time and reclaims garbage memory.
- GC slows Java code down a little, but eliminates all those `&/malloc()/free()` bugs. The GC algorithm is very sophisticated.
- Stack vs. Heap
  - Remember, stack memory (where locals are allocated for a method call), is much faster than heap memory for allocation and deallocation.
- Destructor
  - In C++, the "destructor" is an explicit notification that the object is about to be destroyed.
  - In Java, the "finalizer" is like a destructor -- it runs when an object is about to be GC'd. However, when or even if the finalizer runs is very random because of the odd scheduling of the GC. Because the timing of the finalizer is imprecise, depending on them can make the whole program behavior a little unpredictable. Therefore, I recommend not using finalizers if at all possible.

## Declare Vars As You Go Style

- In Java, it's possible to declare new local variables on any line.
- This is a handy way to name and store values as you go through a computation...

```
public int method(Foo foo) {
    int a = foo.getA();
    int b = foo.getB();
    int sum = a + b;
    int diff = Math.abs(a - b);
    if (diff > sum) {
        int prod = a * b;
        for (int i = 0; i < a; i++) {
            ...
        }
    }
}
```

## Static

- Ivars or methods in a class may be declared "static".
- Regular ivars and methods are associated with objects of the class.
- Static variables and methods are not associated with an object of the class. Instead, they are associated with the class itself.

## Static variable

- A static variable is like a global variable, except it exists inside of a class.

- There is a single copy of the static variable inside the class. In contrast, regular ivars such as "units" exist many times -- once for each object of the class.
- Static variables are rare compared to ordinary ivars.
- The name of a static variable starts with the name of its class -- so a static variable named "count" in the Student class would be referred to as "Student.count".
- Output Example
  - "System.out" is a static variable in the System class that represents standard output.
- Monster Example
  - Suppose you are implementing the game Doom. You have a Monster class that represents the monsters that run around in the game. Each monster object needs access to a Sound object that holds the sound "roar.mp3". so the monster can play that sound at the right moment. With a regular ivar, each monster would have their own copy of the variable. Instead, the Monster class contains a static Sound variable, and all the monster objects share that one variable.

## Static method

- A static method is like a function that is defined inside a class.
- A static method does not execute against a receiver. Instead, it is like a plain C function -- takes arguments, but there is no receiver.
- The full name of a static method begins with the name of its class, so a static foo() method in the Student class is called Student.foo().
- The Math class contains the common math functions, such as sin(), cos(), etc.. These are defined as static methods in the Math class. Their full names are Math.sin(), Math.cos(), ...
- The System.arraycopy() method is another example of a static method. The static method does not have a receiver that it executes against. Instead, we call it like a regular function, and pass it the arguments to work on.
- A "static int getCount()" method in the Student class is invoked as Student.getCount();
- In contrast, a regular method would be invoked with a message send to a receiver like s.getStress(); where s points to a Student object.
- The method "static void main(String[] args)" is special. To run a java program, you specify the name of a class. The system then starts the program by running the static main() function in that class, and the String[] array represents the command-line arguments.
- Call a static method like this: Student.foo(), NOT s.foo(); where s points to a Student.
  - s.foo() actually compiles, but it discards s as a receiver and translates to the same thing as Student.foo() using the compile-time type of the receiver variable. The s.foo() syntax is misleading, since it makes it look like a regular message send.

## static method/var example

- Suppose we modify the Student example so it has a static variable and a static method.
- -Add a "static int count;" variable that counts the number of Student objects constructed -- increment it in the Student ctor. Both static and regular methods can see the static count variable. There is one copy of the count variable, shared by all the Student objects.
- -Add a static method getCount() that returns the current value of count.

```
public class Student {
    private int units;

    // Define a static int counter
    private static int count = 0;

    public Student(int init_units) {
        units = init_units;

        // Increment the counter
        count++;
    }

    public static int getCount() {
        // Clients invoke this method as Student.getCount();
        // Does not execute against a receiver, so
        // there is no "units" to refer to here
        return(count);
    }

    // rest of the Student class
    ...
}
```

## Typical static method error

- Suppose in the static getCount() method, we tried to refer to a "units" variable...
  - public static int getCount() {
  - units = units + 1; // error
  - }
- This gives an error message -- it cannot compile the "units" expression because there is no receiver with ivars in it. The "static" and the "units" are contradictory -- something is wrong with the design of this method.
- Static vars, such as count, are available in both static and regular methods. However, ivars like "units" only work in regular methods that have a receiver.
-