

# Java

---

This is the handout for the special "Basic Java" section -- basic OOP style, classes, methods, this, receiver, static, strings, arrays, collections

## Java -- Buzzword Enabled

From the Sun Java whitepaper: "Java is a simple, object-oriented, distributed, interpreted, robust, secure, architecture-neutral, portable, high-performance, multi-threaded, and dynamic language." By law, any introductory Java lecture must mention the original buzzwords.

### Simple

Simpler than C++ -- no operator overloading

Mimics C/C++ syntax, operators, etc. where possible

To the programmer, Java's garbage collector (GC) memory model is much simpler than C/C++

### Object-Oriented

Java is fundamentally based on the OOP notions of classes and objects

Java uses a formal OOP type system that must be obeyed at compile-time and run-time. This is helpful for larger projects, where the structure helps keep the various parts consistent. Contrast to Perl, which has more of a quick-n-dirty feel.

### Distributed / Network Oriented

Java is network friendly -- both in its portable, threaded nature, and because common networking operations are built-in to the Java libraries.

### Robust / Secure

Java is very robust -- both vs. unintentional memory errors and vs. malicious code such as viruses. Java makes a tradeoff of robustness vs. performance.

1. The JVM uses a verifier on each class at runtime to verify that it has the correct structure
2. The JVM checks certain runtime operations, such as pointer and array access, to make sure they are touching only the memory they should. Memory is managed automatically by the garbage collector (GC).
3. The Security Manager can check which operations a particular piece of code is allowed to do at runtime

## Architecture Neutral / Portable

Java is designed to "Write Once Run Anywhere", and for the most part this works. Not even a recompile is required -- a Java executable can work, without change, on any Java enabled platform.

## High-performance

Java performance has gotten a lot better with aggressive just-in-time-compiler (JIT) techniques. Java performance is often similar to the speed of C, and is faster than C in some cases. However memory use and startup time are both significantly worse than C.

## Multi-Threaded

Java has a notion of concurrency wired right in to the language itself. This works out more cleanly than languages where concurrency is bolted on after the fact.

## Dynamic

Class and type information is kept around at runtime. This enables runtime loading and inspection of code in a very flexible way.

## Java Compiler Structure

Compile classes in .java files -- produce bytecode in .class files

On unix, the java compiler is called "javac". To compile all the .java files in a directory use "javac \*.java".

## Bytecode

A compiled class stored in a .class files or .jar file

Represent a computation in a portable way -- as PDF is to an image

## Java Virtual Machine

Loads and runs the bytecode for a program + the library classes

The JVM runs the code with the various robustness/safety checks in place -- robustness vs. performance tradeoff

On unix, the JVM is called "java" -- run the Java program main() in the class MyClass with "java MyClass".

## JITs and Hotspot

Just In Time compiler -- the JVM may compile the bytecode to native code at runtime (with the robustness checks still in). (This is one reason why java programs have slow startup times.)

The "hotspot" project tries to do a sophisticated job of which parts of the program to compile. In some cases, hotspot can do a better job of optimization than a

C++ compiler, since hotspot is playing with the code at runtime and so has more information.

Java performance is now similar to C performance -- faster in some cases, slower in others. Memory use and startup time are worse than C.

## Java Lang + Its Libraries

The core java language is not that big

However, it is packaged with an enormous number of "library" or "off the shelf" classes that solve common problems for you

e.g. String, ArrayList, HashMap, StringTokenizer, HTTPConnection, Date, ...

Java programmers are more productive in part because they have access to a large set of standard, well documented library classes.

## Java: Programmer Efficiency

Faster Development

Building an application in Java takes about 30% less time than in C or C++

Faster time to market

Java is said to be "programmer efficient"

OOP

Java is thoroughly OOP

Robust memory system

Memory errors largely disappear because of the safe pointers and garbage collector. I suspect the lack of memory errors accounts for much of the increased programmer productivity.

Libraries

Code re-use at last -- String, ArrayList, Date, ... available and documented in a standard way

## Microsoft vs. Java

Microsoft hates Java, since a Java program is not tied to any particular operating system. If Java is popular, then programs written in Java might promote non-Microsoft operating systems. For basically the same reason, all the non-Microsoft vendors think Java is a great idea.

Microsoft's C# is very similar to Java, but with some improvements, and some questionable features added in, and it is not portable in the way Java is.

Microsoft has used its power to try to derail Java with some success, but Java remains very popular on its merits.

## Java Is For Real

Java has a lot of hype, but much of it is deserved.

Java is very well matched for many modern problem

Using more memory and CPU time but less programmer time is an increasingly appealing tradeoff.

Robustness and portability can be very useful features

I suspect we will be using some version of the Java language for the next 10 or 20 years.

# Procedural vs. OOP

Compare the traditional procedural paradigm to OOP

## Nouns and Verbs

Nouns -- data

Verbs -- operations

## Procedural Structure

C/Pascal/etc. ...

Verb oriented

decomposition around the verbs -- dividing the big operation into a series of smaller and smaller operations.

Nouns/Verb structure is not formal

The programmer can group the verbs and nouns together (ADTs), but it's just a convention and the compiler does not especially help out.

# OOP Structure

## Objects

Storage

Objects store state at runtime, just like a regular variable

Behavior

Objects will also in some sense take an active role. Each object has a set of operations that it can perform, usually on itself.

Class

Every object belongs to a class that defines its storage and behavior.

An object always remembers its class (in Java).

"Instance" is another word for object -- an "instance" of a class.

Anthropomorphic -- self-contained

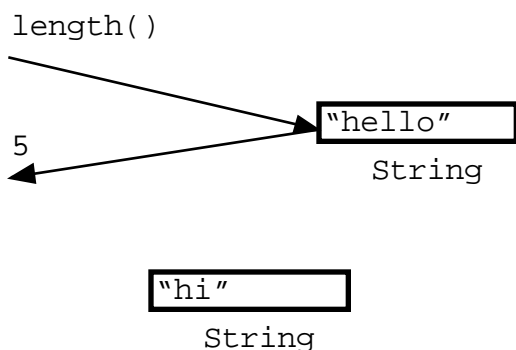
Procedural variables are passive -- they just sit there. A procedure is called and it changes the variable.

Objects are anthropomorphic-- the object has both storage and behavior to operate on that state.

String example

Could have a couple String objects, each of which stores a sequence of characters. The objects belong to the String class. The String class defines the storage and operations for the String objects...

Sending the length()  
message to a String  
object...



String class

```
length() {
  ---;
  ---;
}

reverse() {
  ---;
  ---;
}
```

## Class

Exists once -- there is one copy of the class in memory.

Contains definitions for its objects

Storage

Define the storage that objects of this class will have.

"instance variables" -- the variables that each object will use for its own storage. Instance variables are usually just called "ivars"

Behavior

Define the behaviors that objects of this class can execute (methods).

String example

The String class defines the storage structure used by all String objects -- probably an array of chars of some sort

The String class also defines the operations that String objects can perform on themselves -- length(), reverse(), ...

## Message / Receiver

Suppose we have Student objects, each of which has a current number of units.

The message getUnits() requests the units from a student, and setUnits(int u) sets the units of a student.

Java syntax: a.getUnits() -- send the "getUnits()" message to the receiver "a"

Receiver

The "receiver" is the object receiving the message. Typically, the operation itself (reading and writing memory) is on the receiver's memory.

```
obj.units = 15;           // NO, do not manipulate the obj state directly (see
                          "encapsulation" below)
```

The receiver should operate on its state, not the client

```
obj.setUnits(15);        // YES, send a message to the obj, it operates on itself
```

Sending a message to the receiver object, maps to a method (below). The method is code that actually changes the receiver state.

String example

The string object responds to messages like "length()" and "reverse()" which operate on the receiver they are sent to.

## Method (code)

A "method" is executable code that defined in a class.

The objects of a class can execute all the methods the class defines.

The String class defines the code for length() and reverse() methods. The methods are run by sending the "length()" or "reverse()" message to a String object.

## Message -> Method resolution

Suppose a message is sent to an object --- string.reverse();

1. The receiver is of some class -- suppose the object is of the String class
2. Look in that class of the receiver for a matching reverse() method (code)
3. Execute that code "against" the receiver -- using its memory (instance variables)

In Java this is "dynamic" -- the message/method resolution uses the true, run-time class of the receiver.

## OOP Design - Anthropomorphic, Modular

1. Objects responsible for their own state
2. Objects can send messages to each other -- requests
3. The object/message paradigm makes the program more modular internally. Each class deals with its own implementation details, but can be largely independent of the details of the other classes. They just exchange messages.

## OOP Design Rule #1 -- Encapsulation

Objects "protect" their own state from direct access by other objects -- "encapsulation". Other objects can send requests, but only the receiver actually changes its own state. This allows more reliable software -- once a class is correct and debugged, putting it in a new context should not create new bugs.

Abstraction vs. Implementation

This is the old Abstract Data Type (ADT) style of separating the abstraction from the implementation, but re-cast as messages (abstraction) vs. methods (implementation)

## OOP Design Process

Think about the objects that make up an application

Think about the behaviors or capabilities those objects should have

Endow the objects with those abilities as methods

If a capability does not occur to you in the initial design, that's ok. Add it to the appropriate class when needed -- the just needs to go in the right class

### Co-operation

Objects send each other messages to co-operate  
But each method operates on its own receiver

### Tidy style

Experience shows that having each object operate on its own state is a pretty intuitive and modular way to organize things.

# Student Java Example

## Student Example

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

## Java Client Side

First we'll look at Java code on the "client side" -- code that uses the Student class, but not the code that defines the Student class.

Client will typically allocate objects and send them messages.

With good OOP design, being a client should be easy -- the hard stuff should be hidden inside the class.

Allocate objects with "new" -- calls constructor

Objects are always accessed through pointers -- shallow, pointer semantics

Send messages -- methods execute against the receiver

Can access public, but not private/protected from client side

## Object Pointers

```
Student x;
```

Declares a pointer "x" to a Student object, but does not allocate the object yet.

All objects and arrays in Java are accessed through pointers and so have "shallow" semantics...

Must allocate object with `new` before using

Using `=` on an object, just copies the pointer, so there are multiple pointers to the one object.

Using `==` with objects just compares pointers (see the `equals()` message below for a "deep" comparison of two objects)

## new Student() / Constructor

```
new Student(12)
```

The "new" operator allocates a new object in the heap, runs a constructor to initialize it, and returns a pointer to it.

Classes define "constructors" that initialize objects at the time `new` is called.

Constructors are similar to methods, but they cannot be called. They are only invoked when new objects are created.

The word "constructor" is generally written as "ctor"

The constructor uses the same name as the class. e.g. the constructor for the "Student" class uses the name "Student"

There can be multiple constructors. They are distinguished 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. It is invoked automatically when an object is created if it exists and no other ctor is specified.

## Message send

Send messages 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.

## 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 definition of the Student class -- in the file Student.java

## Instance Variables

```
protected int units;
```

"ivars"

Defines the variables that each object of this class will have -- allocates space inside the object. In this case, every Student object has a int ivar called "units" in it.

## public/private/protected

An ivar or other element declared private is not accessible to client code. The element is only accessible to the class itself.

Suppose on the client side with have a pointer "s" to a Student object. The statement "s.units = 13;" will not compile since "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

## Constructor (ctor)

```

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

```

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 always put into an initialized state when 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 currently agreement about which ivar init style is better.

## Default Ctor

Ctor with no args 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

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

Code stored in class

This code will execute against instances of the class

Message-Method Lookup

Message sent to a receiver

Receiver looks in its class for matching method

That method executes against the receiver

## Receiver Relative (Method, Ctor)

The code runs "on" or "against" the receiving object

Any ivar read/write operations happen to the receiver

Method code is written with a "receiver relative" style where the state of the receiver is implicitly present. This style will really grow on you.

e.g. "units" instance variables automatically that of the receiver

Self message send -- just name the message -- the receiver is the same as the current receiver.

"setUnits(units - drop);" -- easy to send a message keeping the same receiver

## "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 think this is a bad idea. The nice thing about OOP is the effortlessness of the receiver-relative style.

## ivar vs. local var

Suppose we have a

Suppose you declare a

Usually, just refer to the ivar by name directly. Sometimes you have a local var with the same name as the ivar, in which case the expression `this.ivar` refers to the ivar. 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 "m" -- `mUnits`, etc.

Receiver/Noun Style

You think a little differently about your code -- code is grouped around the noun it operates on

## "private"

Implementation visibility

Essentially, only code that is implementing the class can access the method or ivar.

Applies to all elements defined in the class: ivars, methods, ctors...

"Sibling Access"

Private does not prevent one object in the class access the state of **another object in the same class**. Such "sibling" access is slightly less desirable OOP style than an object accessing its own state, but the private keyword allows it.

## "public"

Visible to all

"Official" class interface

aka the interface the class "exposes" for other classes to use.

Public = supported

public features will not be removed in a future rev

Other classes can depend on these features

Sun deprecates some public features, so new code won't be written with them, but it very rarely removes a formerly public feature

private things can be removed from an implementation at will

## "protected"

Similar to "private" but allows access to subclasses and other classes in the same package.

## 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
*/
}
}

/*
Things to notice...

-Demonstrates the Object-lifecycle -- clients create the object (must go
through constructor), then send it messages. Hard for the client to mess
up the state of the object. Note how setUnits() can maintain the internal
correctness of the object.

-The implementation code can refer to instance variables like "units"
by name. It automatically binds to the ivar of the receiver.

-"units" is declared protected. Therefore, a client cannot write something like
"a.units++". The client must go through public messages like setUnits().
This promotes a less fragile design. The client may access things declared
"public".

-State vs. Computation -- notice that the client can't really tell if stress is
stored or computed. It just appears to be a property that Students have. Whether
it is stored or computed is just a detail. This is a nice separation between
the abstraction exposed by client and how it is actually implemented.
*/

```

# Other Java Features

## Inheritance

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 it is not needed for ordinary programming and we will try to avoid it.

By default in Java, classes have the superclass "Object" -- this means that all classes inherit the methods defined in the Object class.

# Static

Ivars or methods may be declared to be static

Static = exists once for the whole class

Not associated with a particular instance. Instead, the static exists once for the whole class.

## Static variable

A static variable is like a global variable for that class. The variable exists just once in the class, instead of once for each instance. The variable is in the namespace of the class, such as `Student.someStaticVariable`.

Static variables are somewhat rarely used.

## Static method

A static method is like a function that is defined inside the class.

A static method does not execute against a particular instance. There is no receiver.

A "static void hello()" method in the Student class is invoked as `Student.hello()`;

In contrast, a regular method would be invoked with a message send 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()`. 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 in two ways...

-Add a static int "ctor\_count" variable that counts the number of Student objects constructed -- increment it in the Student ctor

-Add a static method `hello()` that prints a greeting to standard output

```
public class Student {
    private int units;

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

    public Student(int init_units) {
        units = init_units;

        // Increment the counter
        ctor_count++;
        // ( "Student.ctor_count" )
    }
}
```

```

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

    System.println("Hello there");
    System.out.println("We've made " + ctor_count + " students");
}

// rest of the Student class
...
}

```

## Typical static method error

Suppose in the static `hello()` method, we tried to refer to a "units" variable...

```

public static void hello() {
    units = units + 1;    // error
}

```

This gives an error message -- it cannot compile the "units" expression because there is no receiver. The "static" and the "units" are contradictory -- something is wrong with the design of this method.

## Array

Arrays are built-in to Java

An array is declared according to the type of element

Arrays are always allocated in the heap and accessed through pointers

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

`a = new int[100];`

Allocate the array in the heap with the given size

Like allocating a new object

The array is zeroed out when allocated.

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

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 type (int in this case) and trap errors such as above

The other Java collections WILL NOT have this compile time type system error catching (d'oh!), although it is rumored that 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 -- fill the array with squares: 1, 4, 9, ...

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

    int i;
    for (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 100 Student objects

```
{
    Student[] students;

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

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

## String

Strings are built-in to the Java language

There is a built-in String class that implements many handy methods -- see the docs for the String class for a listing of its many methods

Strings (and char) use 2-byte unicode characters -- work with Kanji, Russian, etc.

String objects are "immutable"

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 is one way of building a class

The immutable style is one way to finesse memory sharing and multi-threading issues

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"

`System.out.print("print out a string");` // or use `println()` to include the newline  
**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"
```

**toString()**

Many objects support a `toString()` method that creates some sort of String representation of the object -- handy for debugging. `print()`, `println()`, + will use the `toString()` of any object passed in.

## 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 index (0 based)

`int indexOf(char c)` -- first occurrence of char, or -1

`int indexOf(String s)`

`boolean equals(Object)` -- test if two strings are the same

`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 -- 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"

d == b; // false (generally, don't use == on objects)
d.equals(b); // true (a "deep" comparison)
}
```

## StringBuffer

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  
 print(String) -- as above, but without and end-line

Example

System.out.println("hello"); -- prints to standard out

## Java Primitives

Java has "primitive" types, much like C. Unlike C, the sizes of the primitives are fixed, 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 automatically when the code runs, just as in C...

```

public void foo() {
    int a, b;
    char c;
    Student s;
    ...
}

```

Here's a method that allocates two ints and a char as locals.

The Student **pointer** "s" is also allocated as a local, but the Student object is not -- that requires a call to new.

However, it is not possible to get a pointer to a primitive. Pointers only work for objects and arrays.

Java is divided into two worlds: primitives work in simple ways and there are no pointers, while objects and arrays always use pointers. The two worlds are separate, and that makes some things awkward.

There are "wrapper" classes Integer, Boolean, ... 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.

The boundary between the primitive parts of Java and the true OOP parts are a little awkward. The primitives were included in the language, because without them it ran too slow.

## == vs equals()

== -- compare primitives or pointers  
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, such as String, 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, 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. Java does not really have that feature -- the constructor marks creation, but destruction is indefinite. The "finalizer" feature is a lame attempt, but I do not recommend using it.

## Declare Vars As You Go

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;
    ...
}

```

## Collections

Built-in classes for storage (like the C++ STL)

Collection type (sequence/set) -- ArrayList is the most useful

Map type (hash table/dictionary)-- HashMap is the most useful

See the Sun docs: <http://java.sun.com/docs/books/tutorial/collections/>

## Collection Design

As much as possible, all the various classes implement the same interface so they use the same method names (e.g. add()), so you can substitute one type of collection for another. This also makes it easier to learn, since the method names are all consistent.

Only store pointers to elements

Can store pointers to objects such as Strings or arrays, but cannot store a primitive like an int. If you need to store an int, use the Integer class (or Float, or Boolean, or Double). This is one irritating area of java, but it's understandable from a language implementation point of view.

CT collection source forgets class -- cast back

Internally, the collection classes treat all elements as Object pointers.

The client needs to cast the pointer to the correct class when getting elements out.

At runtime, the objects remember their class.

## Collection Details

There are a few key, basic methods...

constructor() -- collection with no elements

Actually, a Java interface cannot specify a ctor or static method, but all the collection classes implement the default ctor at a minimum.

int size() -- number of elements in the collection

This could have been called getSize() or getLength(), but they kept the name size() to remain compatible with the old Vector class.

boolean add(Object ptr)

Add a new pointer/element to the collection. Adds to the "end" for collections that have an ordering. Returns true if the collection is modified (it might not be when adding to a Set type collection).

iterator()

Return a new iterator set up to iterate through the collection and remove elements. Do not add to the collection while iterating.

The iterator responds to... hasNext() true if more elements, next() return the next element, and remove() removes the previous elem returned by next()

Utilities -- these are most likely implemented to just call the basic methods above

boolean isEmpty()

boolean contains(Object o) -- iterative search

boolean remove(Object o) -- iterative remove  
 boolean addAll(Collection c) -- true if receiver changed  
 ... and so on

## ArrayList

Replaces the old "Vector" class -- like an array, but it can grow over time  
 add() -- add pointer to end  
 int size() -- number of elements  
 Object get(int index) -- retrieve the elem pointer, indexed (0..len-1)  
 iterator() -- return an iterator object to iterate over the array list

## ArrayList Demo Code

```
public static void demoArrayList() {
    ArrayList strings = new ArrayList();

    // add things...
    for (int i= 0; i<10; i++) {
        // Make a String object out of the int
        String numString = Integer.toString(i);
        strings.add(numString); // add an elem to the collection
    }

    // access the length
    System.out.println("size:" + strings.size());

    // ArrayList supports a for-loop access style...
    for (int i=0; i<strings.size(); i++) {
        String string = (String) strings.get(i);
        // Note: cast the get() to its actual class
        System.out.println(string);
    }

    // ArrayList also supports the "iterator" style...
    Iterator it = strings.iterator();
    while (it.hasNext()) {
        String string = (String) it.next(); // get pointer to elem
        System.out.println(string);
    }

    // Call toString()
    System.out.println("to string:" + strings.toString());

    // Iterate through and delete
    it = strings.iterator(); // get a new iterator (at the beginnig again)
    while (it.hasNext()) {
        it.next(); // get pointer to elem, and discard
        it.remove(); // remove the above elem
    }
}
```