

## Familiarization with Dr. Java

=====

// = comment

> = prompt

// Expression : something that evaluates to a value.

Enter an expression in the interactions pane and will see the evaluated value.

someone give me an expression to enter.

## Primitive Types

=====

What are the primitive types in Java?

1)int

2)long

3)short

4)double

5)char

6)boolean

What is the difference between int, long, and short ;

short(16bit):  $-2^{15} \rightarrow 2^{15}-1$

int (32bit):  $^{31}$

long (64bit):  $^{63}$

Show what a primitive type is in the memory model

## Operations on primitive types

=====

A "type" is a set of values  $\{\dots,-2,-1,0,1,2,\dots\}$  and a set of operations on those values,

- what are the operations for numeric values

{ +, -, \*, /, % }

4/5

// whats wrong...it gives a zero because you are doing integer division. The remained can be seen by doing

4%5

and the decimal value by doing

4.0/5.0

### Auto increment and decrement operators (pre and post)

```
int test = 9;
```

```
// pre-increment, adds 1 to the variable and then evaluates  
++test  
test
```

```
// post-increment, evaluates the variable and then adds 1  
test++  
test
```

```
// pre-decrement  
--test  
test
```

```
// post-decrement  
test--  
test
```

### - boolean operations

true, false,  
&&, ||, !

Precedence : !, && , ||

### - Comparison operator

We can use == to compare two primitive values.

```
2==3
4==4
int first = 8;
int second = 8;
first == second
first == second + 1
```

Note the difference between the equality operator == and the assignment operator =. They are easy to confuse and cause many bugs for novice programmers.

example:

For each of the following, give a smaller expression:

```
a == true
a == false
a != true
a != false
```

Give an equivalent expression to:

```
!(a && b)
```

[char & the ASCII character table:](#)

char holds a single character of text. In truth, computers can't actually store or process text - they fake it, by encoding characters as numbers. For example, the char literal 'd' is actually the value 100. A long time ago, a table was built to map each character to a number value. The table is called the "ASCII Table", and can be found online:

[www.asciitable.com](http://www.asciitable.com)

Whenever you store a value in a char, what Java really stores is the number that encodes that character. Each time you retrieve its value, Java looks it up in the ASCII table and shows you the symbol associated with that character.

You can tell Java not to do the lookup by casting the char to an int. For example:

```
char test = 'd';
System.out.println(test); // outputs 'd'
System.out.println((int)test); // outputs '100'
```

You can also assign chars value by using their int equivalent:

```
char test = 100;
System.out.println(test); // outputs 'd'
```

And, of course, this Boolean statement has the value 'true':

```
'd' == (char)100
```

### Interjection : String type

```
String myName = "Alifiya";
System.out.println(myName);
```

We can concatenate two Strings together:

```
myName = myName + " Hussain";
System.out.println(myName);
```

... Lets go back to chars now

so "d" and 'd' are not the same thing. "d" is a String object and 'd' is a char.

try

```
'd' + 'd'
```

The output is not "dd" it is the char equivalent of 200  
Because chars are really just ints, you can do the usual arithmetic operations on them.

Try all of these operators: -, +, \*, /, =, ==.

What happens when we use + to combine a char and a String? Try it:

```
'c' + "s"
```

Look at what happens when we combine an int and a String:

```
68 + "test"
```

## Special characters: escape sequences

We use single and double quotation marks to indicate a char and a String, respectively.

How do we write a character ' in a char? Or a double quote in our String?  
In order to use these characters, we need to use the escape character, which is a backslash: \

When Java sees the escape character, it knows to treat the next character is a special character:

```
\'\' // works!  
"The title is \"ProgramLive\"" // works!
```

There are also escape sequences for carriage return, tab, and other useful characters:

Carriage return: \n  
Tab: \t

Remember - \t is a single character, even though it requires two separate symbols to represent it. Need proof? Try casting the literal '\t' to an int. Note that the value it returns is the ASCII code for the tab character.

## Variables

=====

I have a formula for converting from 28 degrees celsius to fahrenheit.

$9.0/5.0 * 28 + 32$

Now convert 20 degrees to fahrenheit

Declare a variable to be able to do the conversion for any number.

```
// variable: a name that has a value  
// declaration: an announcement that you are going to use a name as a variable.  
// has the form: type variable ;
```

What is the difference between a "statement" and an "expression".

```
// a declaration is a statement
// statement: an instruction in the java programming language
// a statement is NOT the same as expression
// statements do not produce values
// statements end in semi-colons
```

```
// assignment: give a variable a value.
a = 6;
```

```
// initialization: giving a variable its first value
int a = 6;
```

## Constants

=====

Often a program will demand that a fixed value be used throughout.

Example: GST 7% would be written in Java as

```
public final double GST = 0.07;
```

**final:** The effect of keyword 'final' is that *GST* can only be initialized and cannot be changed after initialization.

You cannot, for example do this:

```
GST = 539.666; // ERROR!
```

## public:

We don't have to worry about anyone altering the value of *GST* so it is okay to make it publicly accessible.

Often it is desirable to have a publicly accessible fixed value.

Example: `Integer.MAX_VALUE`

## style:

Constants are named using all uppercase letters with underscores to separate words.

## Assigning with different types

=====

I have two variables x and y

```
int x = 5
double y = 7
```

How can I assign the value of y to x (it has no fractional part)

```
// type cast: converts the type of an expression
// cast has the form: var = (type) expression ;
```

We'll see more about what is allowed when we do inheritance

## Non-Primitive Types : Classes

=====

Primitive type: stores only a single value.

So if I want to create a variable that stores Temperature, I can use the primitive type double. What if I want to create a variable that represents a Book in a collection?

What characteristics does a book have that will let you describe it?

Here are a few:

- o title
- o author(s)
- o publication date
- o due date (when borrowed)
- o ISBN
- o call number

What do you want to do with books?

checkout, checkin, etc.

In order to make the Book a type, we will write a Class and call it BookRecord.

1) import statements and Class Header:

=====

## The Java API

=====

The Java language comes with a very large library of utility classes. These are common classes that are already programmed and have things you can use like Strings, Date for telling time, lists etc. You only need to know about the parts of this library that you really need, and which package it is in

### Example

java.lang contains things like character-strings, that are essentially "built in" to the language.

java.io contains support for input and output

java.util contains some handy data structures such as lists and hash tables.

For example if you want to use the built in type Date which exists in Java.util, you would have to use it by:

```
java.util.Date myDate = new java.util.Date()
```

In order to avoid writing such long class names, we just import it on top like:

```
import java.util.Date;  
OR  
import java.util.*;
```

then you can just use it by saying

```
Date mydate = new Date();  
import java.util.*;
```

```
/**  
 * This class represents a book in a library system.  
 */
```

```
public class BookRecord{  
  
}
```

## Very important: Commenting

=====

Include comments that describe all the elements

Programmers rely on the class and method comments to figure out what they do.

Examples of good and bad comments:

```
/**  
 * Move a window to the new location and resize it.  
 */  
public void moveResize(int a, int b, int x, int y){  
    // statements that make this happen  
}
```

Which window?

Where is the "new location"?

Resize how? How wide? How high?

A more informative comment:

```
/**  
 * Move the upper left corner of this window to location (x,y)  
 * and make this window w pixels wide and h pixels high.  
 */  
public void moveResize(int w, int h, int x, int y){  
    // statements that make this happen  
}
```

Make sure you mention everything about the return value, the parameters and what it does

WITHOUT revealing HOW it is being done.

Javadoc is another way of commenting your code. More about it later.

## 2) Instance Variables

=====

Sometimes an object needs to remember things.  
Keep track of some information.  
We add data to a class by adding instance variables.

Instance variables look a lot like ordinary variables except they live inside the body of a class (its braces) and the class can control who gets to see them.

What instance variables would the Book class need?

```
// pick your data: translate into instance variables

String title; // the title of the book
String author; // the author of the book
Date datePublished; // date of publication
Date dateDue; // date the book is to be returned
String callNumber; // filing number
```

// note the naming conventions in the above 3 variable names

## Information Hiding

=====

It is normally the case that you want to protect your member variables, so that people using your class can't change them directly. Why?

If you wanted there to be a minimum and maximum for something, you can make sure the methods don't let you assign values that are too big or too small.

For instance, you don't want to allow someone to make a mistake and set the `datePublished` to be later than the current date!

When you write classes, you want to protect your fields from harm, too. To do this, you simply tell the compiler that your fields are "private". This is done by putting the word "private" in front of the declaration:

```
private String author; // author of the book
private String title; // title of the book
```

By not putting any access modifiers next to the name, you are making it public.

We're now ready to add some operations!

The operations in a class should always be those that make sense for the concept you are representing.

So, now, ask yourself:

What operations do people perform on books and book records at a library?

Immediately, there's some that makes sense:

- o Create a book
- o Getting the name
- o Changing the price
- o Check out: check this book out, set its due date.
- o Check in: check in this book, reset its due date to nothing.

**Methods:**

=====

Let's add some methods.

There are 3 kinds of methods:

- procedures
- functions
- constructors

**Constructor:**

=====

First kind of method: the constructor.

Use the constructor to initialize instance variables.

Constructors are the methods that are called with the "new" keyword. They produce a value that is the unique address of an object.

A default constructor is made for you by Java, but you can replace it. A good place to initialize your instance variables is in the constructor.

**How do we initialize a Date?**

## Look in the API.

Note: no return type in the constructor's header, not even "void".

You should write a constructor that allows someone using your class to initialize key bits of data. Let's add one to the class.

```
public BookRecord(
    String author, String title, Date datePublished,
    Date dateDue, String callNumber, String isbn, boolean
    isHardCover, String edition){

    this.author = author;
    this.title = title;
    this.datePublished = datePublished;
    this.dateDue = dateDue;
    this.callNumber = callNumber;
    this.isbn = isbn;
    this.isHardCover = isHardCover;
    this.edition = edition;
}
```

Note: you can write more than one constructor for a class.

for example, another sensible constructor would omit the dueDate, since the book wouldn't have a due date until it is checked out. This is called method Overloading

```
public BookRecord(String author, String title, Date
datePublished,
                    String callNumber, String isbn, boolean
                    isHardCover, String edition){

    this.author = author;
    this.title = title;
    this.datePublished = datePublished;
    this.callNumber = callNumber;
    this.isbn = isbn;
    this.isHardCover = isHardCover;
    this.edition = edition;
    this.dateDue = null;
}
```

Compile.

Procedures

=====

Procedure: a method that performs a task and does not produce a value.

Give me a procedure we can write for the BookRecord class.

```
public void setTitle (String name) {  
    }  
}
```

public: is called an "access modifier"

It indicates that every class can call this method.

void: Return type "void" means that it does not produce a value.

So the method can be used in a statement, but not in an expression.

Recall: that a statement performs an action but does not evaluate anything.

Parameter: is a variable that is declared within the parentheses of a method header.

this describes the information that will get passed to this method when it is called.

Compile.

Add statements to the body of the method:

```
this.name = name;
```

this: an object's reference to itself.

Note that from the outside we called methods using f.

We use a reference variable when calling a method from outside its class.

From inside its class, we use "this".

Try the method out in the interactions pane.

```
=====  
return;  
=====
```

Sometimes you want to stop the execution of statements in a procedure after a point.

Example:

```
public void Printme() {  
    System.out.println("me");  
    System.out.println("notme");  
}
```

Change this method so that only "me" is printed and "not me" is not printed.

add the statement `return;` in between the two lines and the second one is not executed.

We've now seen procedures and constructors.  
The third kind of method is the function.

## Functions

=====

Function: a method that produces a value.

Examples of function calls:

```
Date d = new Date();  
d.getDate();
```

where did the value go...how do I see it?

How do I know what it is returning ... look in the API

it returns an int...

so

```
int date = d.getDate();  
System.out.println(date);
```

So lets write a function for the `BookRecord` class:

I want to checkout a book. Sometimes the book may be lost, already checked out, or something else might go wrong when checking it out. So you want a checkout method to confirm whether it worked or not.

Write a method `checkOut` that returns a boolean. True if the book was checked out successfully, and false if it is not available.

```
/**
 * Check out this book, set it to be due in one month
 */

public void checkOut(){

    // Now! (with millisecond precision)
    this.dateDue = new Date();

    // loan for one month from today
    // Date class has int getMonth() and setMonth(int)
    // methods
    this.dateDue.setMonth(this.dateDue.getMonth() + 1);
    // Will this work if the current month is December? Y

    return true;
}
```

## Overriding `toString()` and `equals()`

=====

There are a couple of methods that every Java class includes automatically. The methods `equals()` and `toString()` are already defined in the `Object` class. Since every Java class is a subclass of class `Object`, they are automatically included.

Every class will have its own way of representing itself as a string  
Every class will have its own way of comparing two objects of its type.

So we write our own versions of these methods.  
We say that we "**override**" the methods in the superclass.

Before you write an "equals" method, you need to decide the basis of comparison for a class. For the book class, let's say that the books are the same ("equal") if they have the same ISBN number, and are both the same in terms of being a soft cover or hard cover. We could choose other

criteria for comparison, but we'll use just these for now.

```
/**
 * Returns true iff this book and other have the same isbn
 * and both are hardcovers.
 */
public boolean equals(Object other){
    BookRecord otherBook = (BookRecord)other;    // type cast
    boolean sameBook = this.isbn.equals(otherBook.isbn) &&
        this.isHardcover == otherBook.isHardcover;
    return sameBook;
}
```

Note: The expressions `this.isbn.equals(otherBook.isbn)` and `otherBook.isbn.equals(this.isbn)` are equivalent.

When you override a method, the methods' "signatures" must match.

Method signature: name, type and number of parameters (return type doesn't count).

Method `equals` in class `BookRecord` has a single parameter of type `Object` because method `equals` in class `Object` has a single parameter of type `Object`.

The next method is `toString`. The `toString` method returns a `String` representation of the object.

What happens if I use the method `toString()` when its not been explicitly defined in my class?

Before writing (implementing) `toString`, you must decide on the format of the string representation.

What information do we want to display?

In what order?

We'll use this representation, though we really could have chosen any one we like that makes sense:

```
/**
 * Returns the string representation of this book
 * record in the format:
 * CallNumber:Title:Author:ISBN
 */
public String toString(){
    String s = callNumber + ":" + title + ":" + author + ":" +
isbn;    // catenation operator
```

```
    return s;
}
```

## MEMORY MODEL

- Primitive types vs objects
- toString and equals...how they work differently for primitive types and objects.

### Getter & Setter Methods:

=====

It is often (but not always!) the case that you want those people using your class to be able to access and change the data within it.

We do this by adding

"Setter" methods let a client change a value, and

"getter" methods let a client retrieve a value.

Here are very basic getter & setter methods for the author field of the BookRecord class:

```
public void setAuthor(String newAuthor) {
    this.author = newAuthor;
}

public String getAuthor(String newAuthor) {
    return this.author; // give this author back to the caller
}

/**
 * Sets the isbn of this book record to the given isbn.
 */
public void setISBN(String isbn){

    // Before storing the value of the parameter in our
instance variable,
    // we might want to check the value of 'isbn'
    // (passed in by the caller of your method )
    // too ensure that it has the correct format to be a
valid ISBN.

    this.isbn = isbn;
}

/**
 * Returns the ISBN of this book record.
 */
```

```
public String getISBN() {  
    return this.isbn;  
}
```

=====  
JAVADOC  
=====

## JavaDoc Comments

API documentation is convenient:

- online
- cross-linked
- standard format, so easy to read

Where does the API documentation come from?

Who wrote it?

They are generated automatically using a tool called JavaDoc. This tool extracts special comments from your code and outputs html pages.

2 steps:

- include documentation comments in your code.
- run JavaDoc on your code.

Documentation comments begin with `/**` rather than `/*` for an ordinary multiline comment.

The method summary is the first sentence terminated by a period followed by a single space.

Any detailed comments following the first sentence are only shown in the detail section, not in the summary.

JavaDoc uses "tags" to decide how to format your web document.

The most common JavaDoc Tags are:

`@param <parameter name> <description of parameter>`

`@return <description of return value>`

`@throws`

`@see`

Can also use HTML formatting tags such as:

<code></code>

<ul> & <li>

<br>

<p>

But keep it to a minimum as it makes the code look ugly.

## NEW EXAMPLE - Player and Team

### Class Interactions

=====

We have been interacting with classes in the interactions pane of DrJava. Now we will build and observe classes that interact with each other.

Classes interact when code in one class calls methods in another class. You have seen this before in class BookRecord which interacted with both String and Date).

We will now build a pair of new classes that interact with each other: Player and Team. We will define these classes in the context of a system used to keep track of player and team performance (wins and losses).

I define what every Player object will be like (data & operations).  
Then I define what every Team will be like (data & operations).

Each team has 2 players.  
Team class "encapsulates" 2 players, team wins, team losses and the operations to manipulate this data.

encapsulate: putting data and the operations to manipulate the data together.

### Player

=====

We will define a player to have the following attributes (also called properties or simply data):

- name
- number of wins (as part of any team, and solo)
- number of losses (as part of any team, and solo)

The operations on a player will be:

- Set/get name
- won (called whenever the player wins a game)
- lost (called whenever the player loses a game)
- Get wins/losses (returns the number of wins / losses for this player)

Team

We will define a team to have the following attributes (also called properties or data):

- team name
- number of team wins
- number of team losses
- player 1
- player 2

(could have more players, but this will suffice for demonstration purposes).

Operations on a team will include:

- Set/get name
- Set/get player 1
- Set/get player 2
- won (called whenever the team wins a game - needs to update the player records)
- lost (called whenever the team loses a game - also needs to update the player records)
- get team winning percentage
- get players' overall winning percentage

Now that we have decided on the attributes and operations for these classes, we can write

the Java code for them. As we saw last time, we write our Java code in the Definitions Pane

(also known as the Class pane) of DrJava. It is like a special text editor that understands

Java syntax (knows how to highlight and indent in a meaningful way). We use it to write our

own data types (classes).

The Player class is easy to write since it has just basic attributes and accessor (get/set) methods.

Begin by writing "skeleton code" with only method "stubs".

Try to compile...it will fail.

Put return statements into the bodies of the get methods.

Try to compile...it should succeed.

Note the effect of the return statement:

1. value of expression to the right of the word 'return' is evaluated.
2. that value is given back to the statement that called the method.
3. execution of the method containing the return stops immediately.

Now implement the rest of the operations for the Player class:

3 constructors

and the methods

won

lost

equals

toString

Make an instance of this class in the interactions pane.

```
Player p = new Player();
```

```
p.toString()
```

```
p // drjava calls toString automatically
```

Make a second instance using a different constructor.

```
Player p2 = new Player("Beckham");
```

Check if the 2 instances are equal.

```
p.equals(p2)
```

```
p.setName("Beckham");
```

```
p.equals(p2)
```

## Static modifier

=====

Static members: data and operations that belong to the class (to 0 or more instances of a class).

If a method does not access any fields in a class nor does it call any methods of that class, then there is no need to make it dependent on an object of that class because it will always be the same for every object. This is when you want to make it a static method.

The modifier (keyword) "static" has 2 important effects when placed before a variable or method name:

1. Don't need to make an object to use static data or call static methods. They exist without ever creating an object.
2. Static data and methods are \*shared\* among all objects of the class where these members are defined.

Constants are often made static because it would be a waste of storage to make separate copies of the same value for each instance of a class. Since the value never changes, it makes sense to share a single copy in static space.

Add a couple of static variables to class player and a couple of static methods.

```
public static int getOverallWins()  
public static int getOverallLosses()
```

We can access the overallWins and overallLosses methods using this line of code in the interactions window or in another program:

```
Player.getOverallWins()  
Player.getOverallLosses()
```

OR

```
Player p = new Player();  
p.getOverallWins()  
p.getOverallLosses()
```

```
p.wins();  
p.wins();  
Player.getOverallWins()
```

```
Player p2 = new Player();
p2.wins();

Player.getOverallWins()
// static variable kept track of wins and losses for
// both instances
```

Try:

```
Player.getName()
// getName is not a static method;
// need to create an instance of the Player class to access it.
```

Try:

```
add static method setOverallWins(int wins) and try using
this.overallWins = wins
```

## The Team class

=====

Now we can implement a class that interacts with our Player class. Class Team has 2 players. These will be implemented as private reference variables of type Player.

Again use the development strategy:

1. start from empty file and begin class definition, declaration of instance variables and write method headers with empty bodies.
2. compile and put in necessary stub code.
3. test and implement the bodies of the methods (review how to write class methods).

## Main program

=====

Above we said that static data and operations belong to "0 or more" instances (objects) of the class in which they are defined.

How do we begin running a program?

Where does execution begin?

A program couldn't start running and creating Objects if we needed to first make an object.

With zero objects, we need another way to start our program running.

That is why every executable program has a main class with a static main method.

Let's write a main class for a program that interacts with the Team class.

```
/**
 * A example of a main class
 * A program that models Venus and Serena Williams
 */
public class Sisters{

    public static void main(String[] arg){ // before execution of
main

        // call the static methods before constructing any
instances
        int wins = Player.getOverallWins(); // 1
        int losses = Player.getOverallLosses(); // 2

        // construct 2 instances
        Player serena = new Player("Serena"); // 3
        Player venus = new Player("Venus"); // 4

        // construct a Team and set its players
        Team team = new Team(); // 5
        team.setPlayer1(serena); // 6
        team.setPlayer2(venus); // 7
    }
}
```

To run this program in the interactions window type:

```
java Sisters
```

or

```
Sisters.main(null); // call the static method using the name of
the class
```

Let's trace the main method of class Sisters in our model of memory.

## MEMORY MODEL

---

We differentiate among 3 separate regions of memory:

- object space
- static space
- The method call stack

Instances of non-primitive types are modelled in the "object space".

There are 2 other types of memory in our model:

- static space
- the method call stack

*Populate the static space before beginning execution of main.*

Then begin sequentially executing the statements in method main.

The method call stack (from the PL glossary):

A list of methods that have been called but whose bodies have not yet terminated, in reverse order in which they were called.

The same term is used for the stack of frames for method calls that have not yet terminated.

Stack: a list of elements that is changed by only 2 operations: push and pop.

Push: add an element to the front (or top) of the list

Pop: remove the first element of the list (pop the top element off the list).

Stack: also called a "last in first out" (LIFO) list

Static members are variables and methods that belong to a class, rather than a particular object.

- static members exist before any instances of the class are constructed and continue to exist as long as the program is running. (non-static variables and methods don't exist until an instance is constructed by calling a constructor with 'new').
- Static members are SHARED among all instances of a class. There is only one copy for the whole class and all instances of that class.

Another couple of useful static methods are

- print and
- println of class PrintStream.

These methods display the characters of a String on the monitor.

```
System.out.print("Hi ");  
System.out.print("there!");
```

```
System.out.println("Hi ");  
System.out.print("there!");
```

Note: DrJava's interactions pane displays return values as a convenience. This doesn't happen in a real program. If you want to display information to the screen, you must provide explicit statements to send data to the screen.

Data sent out of a computer program is called 'output'.

## String class

=====

Java has a class called String.

```
String text = "It is lovely today.";  
text  
String moreText = "It is lovely today.";
```

text and moreText are references to objects of the String class.

Compare the Strings using ==  
text == moreText

Why does this evaluate to false?

== was not comparing the sequence of characters, it was comparing the String objects. Each object has a unique memory address and when the variables are compared, then the objects are found to be different.

To compare the sequence of characters, use the method "equals".

```
text.equals(moreText)
```

## Operations on Strings

A String represents a sequence of characters plus operations that are useful for manipulating them. We don't care how String is implemented, we just need to know how to make one and how to use it.

To find out what they are and what can they do:  
Look up String in the API!

Ways to construct a String object:

```
String s; // declare a variable of type String
String q; // declare a second variable of type String
```

```
s = new String("ess"); // instantiate a string object by calling the constructor
q = "ess"; // or use the shortcut
```

The previous 2 statements are equivalent.  
They each create a String object in memory and store the address of the objects created in the variables 's' and 'q' respectively.

Strings are not exactly part of the Java language, but they also sort of are.  
They are so common that the language provides a shortcut.

[We will use this shortcut in the memory model too.]

"ess" is called a string literal.

literal: direct representation of a constant (fixed) value.  
Differs from a variable: a name associated with a value that can vary (not constant).

Let's look up some more String methods in the API:

**Method length():** [Note: has a good specification comment.]

```
r = "ar";
s = "ess";

int l;
```

```
l = s.length();
l = r.length();
l = "".length();
```

Note: "" is the empty string.

The empty string is NOT the same as the null string.

```
r = null;
r.length();
```

### **Method charAt(int):**

```
s
char c = s.charAt(0);
c
c = s.charAt(2);
c
```

```
c = "".charAt(0); // there is no 0th character
```

### **Method toUpperCase():**

```
s = s.toUpperCase();
s
```

### **Method concat(String):**

and its shorthand '+' (demonstrated in toString method of BookRecord)

```
r = "sillin";
```

```
String t = r.concat(s);
t
```

```
t = r + s; // same thing, Java treats the strings like primitive types.
```

### **Method int indexOf(String str)**

Returns the index within this string of the first occurrence of the specified substring.

We use indexOf to find a particular position in a string where a given substring occurs and then use pass this information to substring, in effect telling the string which portion of it we want a copy of [extract is a bad word cause it implies deletion].

**Method String substring(int) // includes the 1st index through the last**

**- String substring(int, int) // includes 1st index, \*excludes\* the 2nd**

An example using substring and indexOf:

Canadian drivers' licence numbers encode more than you might realize. Consider the licence number "J9854-54467-51223". It must belong to a male born December 23, 1975. Write statements to extract the day, month and year from the String licence.

```
String licence = "J9854-54467-51223";  
String day = licence.substring(15,17);  
String month = licence.substring(13,15);
```

```
licence.substring(10,13)
```

```
"7-5"
```

We don't want to store the hyphen:

```
String year = licence.substring(10,11) + licence.substring(12,13);
```

```
day  
month  
year
```

Now suppose we don't know the index of the year within the string licence, but we DO know that the year straddles the second of 2 hyphens that occur in the string:

Find the 1st hyphen using indexOf:

```
int i = licence.indexOf("-");
```

Discard everything up to and including the first hyphen:

```
licence = licence.substring(i+1);
```

Find the second hyphen:

```
int i = licence.indexOf("-");
```

This tells us where to find the year.

Now extract the year:

```
year = licence.substring(i-1, i) + licence.substring(i+1, i+2);
```

## Wrapper Classes

All primitive types have Wrapper classes Integer, Double, Character, Boolean. They allow objects to be created from primitive types.

### Converting strings into numerical types

```
Integer.parseInt("1")  
Double.parseDouble("4.1")
```

### Getting the primitive type value:

```
Integer.valueOf()
```

or

```
Integer a = ("1")  
a.intValue();
```

## Math class

=====

Math.max

Math.min

Eg: You have three int variables called *currentLoad*, *maxLoad*, and *addedLoad*. Write an expression that returns the sum of *currentLoad* and *addedLoad* if it is less than *maxLoad*, otherwise, return *maxLoad*.