CS 2110 - Spring 2009

• very quick overview of Java basics
• actually use inheritance and interfaces
• intro to threads
• Grammars and parsing
• GUIs
• recursion/induction, efficiency, sorting
• data structures (lists, queues, stacks, trees, etc)
• graph algorithms
• applications
• LOTS of programming!

Books etc.
- at java.sun.com
  /reference/api
  /docs/books/tutorial
- IDE from www.eclipse.org
- The Java Language Specification (Gosling et al)
- Data Structures and Algorithm Analysis in Java (Weiss) ..... 
- there are many(!!!) books, and most of them are fine; if you need one then find one that’s not too expensive!!

CS 2110 Java overview
• Java uses classes and objects = hyperorganised!

• A class Car is like a manufacturer who only constructs individual new cars.

• A class Bucket will only construct individual new buckets.

• Cars and buckets have natural things which belong to every car or bucket, although of course cars have different colours, etc.!
To build a red car called ferrari, we might write

```java
Car ferrari = new Car(red);
```

- I’m not promising that this will work!!!!

and to build a yellow car called cbb, we might write

```java
Car cbb = new Car(yellow);
```

Of course,

```java
Bucket rollsroyce = new Bucket(huge);
```

will only give you a peculiarly named huge bucket.

If you want to access stuff in your car, then the dot `.` is the “genitive case”, so `ferrari.colour` would be red, and `cbb.colour` would be yellow. This is also like a path; looking into the ferrari or the cbb to find the individual colours.

More on this later .....
• Enough of such generalities! How do we write a simple program in Java?

• First we need to be able to get stuff in to and out from the computer!

    System.out.println ( "Once upon a time ..." ) ;

looks into the System where it finds an out, and looks into System’s out where it finds a method (or function or routine) which can print a String of characters onto a fresh line on the standard output screen.

    System.out.print ( "Golly gosh" ) ;

does exactly the same, except the method print doesn’t finish with a “new line”.

Tuesday, January 20, 2009
• To read in a String of characters from the standard input keyboard,

\[
\text{InputStreamReader nab = new InputStreamReader(System.in);}
\]
\[
\text{BufferedReader grab = new BufferedReader(nab);}
\]

constructs a BufferedReader called grab so that

\[
\text{grab.readLine();}
\]

reads a whole line of input.

• Java is a language of “let’s pretend!”, so grab is a virtual keyboard which has the ability (amongst other skills) of readLine() - all the other stuff is there to establish a connection between “make believe” and “reality”.
• We can do the same thing with files,

    FileReader secret = new FileReader("spy.oops");
    BufferedReader james = new BufferedReader(secret);

    constructs a BufferedReader called james so that
    james.readLine();
reads a whole line of input from spy.oops. As a matter of common
courtesy, you should
    secret.close();
close the ‘file’ when you’ve finished with it! (If you need to specify a path
for your file, you can have

    = new FileReader("c:/money/penny/spy.oops");

or whatever is appropriate for your system.)
• Similarly,

```java
FileOutputStream plop = new FileOutputStream("meow.t");
PrintWriter scribble = new PrintWriter(plop);
```

allows

```java
scribble.println("What big teeth you have!");
```
to write to the file `meow.t`, which again should be closed by

```java
plop.close();
```
when finished with.

• Of course, we could have done the same thing when writing to the screen,

```java
PrintWriter tube = new PrintWriter(System.out, true);
tube.println("How time flies!");
```

Here, `tube` is the name of the make believe computer screen.

• Now that we can get stuff into and out of the computer, let’s actually write a program ...
import java.io.*; // so that i/o stuff is available

public class PlayTime {
    public static void main(String[] args) throws Exception {
        InputStreamReader ca = new InputStreamReader(System.in);
        BufferedReader va = new BufferedReader(ca);
        PrintWriter bon = new PrintWriter(System.out, true);

        int x, y = 2;

        bon.println("Enter an integer.");
        x = Integer.parseInt(va.readLine());
        y = y * x - x / 2;

        bon.println("x was " + x + " and y is " + y);
        ca.close();
    } // end of main method
} // end of class PlayTime

This whole file would be called PlayTime.java and 'compiled' and run as relevant to your computer system.

Now for some routine details ...
Sam Loyd’s 8 Puzzle

Goal: Given an initial configuration of tiles, find a sequence of moves that will lead to the sorted configuration.

A particular configuration is called a state of the puzzle.
State Transition Diagram of 8-Puzzle

State Transition Diagram: picture of adjacent states.
A state Y is adjacent to state X if Y can be reached from X in one move.
State Transition Diagram for a 2x2 Puzzle

Sorted State

Solutions for this state:
- SWN
- WSENWSENW
- SWEWN

......
Graphs

• State Transition Diagram in previous slide is an example of a graph: a mathematical abstraction
  – vertices (or nodes): (e.g., the puzzle states)
  – edges (or arcs): connections between pairs of vertices
  – vertices and edges may be labeled with some information (name, direction, weight, cost, …)

• Other examples of graphs: airline routes, roadmaps, …
  – A common vocabulary for problems
Path Problems in Graphs

• Is there a path from node A to node B?
  – Solve the 8-puzzle

• What is the shortest path from A to B?
  – 8-puzzle (efficiently)
  – MapQuest

• Traveling salesman problem
• Hamiltonian cycles
Simulating the 8-puzzle

• What operations should puzzle objects support?
• How do we represent states?
• How do we specify an initial state?
• What algorithm do we use to solve a given initial configuration?
• How should we present information to the user? (GUI design)
• How to structure the program so it can be understood, maintained, upgraded?
## Primitive Types

<table>
<thead>
<tr>
<th>Type</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte</td>
<td>-128 &lt;= integer &lt;= 127</td>
</tr>
<tr>
<td>short</td>
<td>-32768 &lt;= “ “ &lt;= 32767</td>
</tr>
<tr>
<td>int</td>
<td>-2^31 &lt;= “ “ &lt;= 2^31 - 1</td>
</tr>
<tr>
<td>long</td>
<td>-2^63 &lt;= “ “ &lt;= 2^63 - 1</td>
</tr>
<tr>
<td>float</td>
<td>+/- 2^-149 &lt; decimal &lt; (2^24 - 1)2^104</td>
</tr>
<tr>
<td>double</td>
<td>+/- 2^-1074 &lt; “ “ &lt; (2^53 - 1)2^971</td>
</tr>
<tr>
<td>char</td>
<td>unicode \u0000 to \uffff, i.e. 0 to 65535</td>
</tr>
<tr>
<td>boolean</td>
<td>false, true</td>
</tr>
</tbody>
</table>

(Float and double are described in IEEE 754, cf java.sun language spec 4.2.3)

- The declaration
  ```java
  int boo;
  ```
  makes `boo` an allowable name for an integer. The declaration and initialisation
  ```java
  int boo = 13074;
  ```
  makes `boo` an allowable name for an integer, and before it can be used, initialises its value to 13074.

- Similarly we can have
  ```java
  double whoosh = 9.874;
  char cuckoo = ‘A’;
  boolean ouch = false;
  ```
  awkward characters like `?` or ‘ can be assigned using \ as in
  ```java
  cuckoo = ‘\?’ ;
  cuckoo = ‘\’’ ;
  ```

These have some amusing consequences!
• Arithmetic

<table>
<thead>
<tr>
<th>Operation</th>
<th>Expression</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>3 + 4</td>
<td>7</td>
</tr>
<tr>
<td>-</td>
<td>3 - 4</td>
<td>-1</td>
</tr>
<tr>
<td>*</td>
<td>3 * 4</td>
<td>12</td>
</tr>
<tr>
<td>/</td>
<td>3 / 4</td>
<td>0</td>
</tr>
<tr>
<td>%</td>
<td>3 % 4</td>
<td>3</td>
</tr>
</tbody>
</table>

What do you think \((-3) \% 4\) evaluates to?

As an aside, it’s worth noting that there is a non-primitive type `String` which carries a string of characters, and

```java
String tut = “Methought I was,”;
String um = “ there is no man ...”;
tut = tut + um;
```

gives `tut` the updated value of `Methought I was, there is no man ...`

So for strings, `+` appends the second string to the end of the first string.

• Also, for those who like calculating...
  ```java
  Math.sin(1.78);
  Math.atan(72.4); etc.
  ```
  all do the obvious. `Math` is a repository of lots of useful stuff!

• Now for an example...
public class Multiplier {
    public static void main ( String [ ] args ) throws Exception {
        InputStreamReader isr = new InputStreamReader ( System.in ) ;
        BufferedReader comingIn = new BufferedReader ( isr ) ;
        PrintWriter goingOut = new PrintWriter ( System.out , true ) ;

        int x , y ; // space to store input
        long z = 0 ; // bigger space for answer!
        String ask = “Please enter an integer.” ; // may as well be polite!

        goingOut.println ( ask ) ;
        x = Integer.parseInt ( comingIn.readLine( ) ) ;
        goingOut.println ( ask ) ;
        y = Integer.parseInt ( comingIn.readLine( ) ) ;

        z = x * y ; // actually PERFORM the multiplication!!!
        goingOut.print ( “The value of ” +x+ “ times ” +y ) ;
        goingOut.println ( “ is ” +z ) ;
        goingOut.println ( “Thanks for using ” +“Multiplier\”. Do come again!” ) ;
        comingIn.close( ) ;
    } // end of main method
} // end of class Multiplier

CS 2110 Java overview- arithmetic 17
• There are also various ‘shorthands’ ...  
  ```java
  int a, b;
  a = 17;
  a = a + 12;  --->  a ↔ 29
  a = a - 4;  --->  a ↔ 25
  a = a * 3;  --->  a ↔ 75
  a = a / 5;  --->  a ↔ 15
  a++ .... a = a + 1
  b = 2 * ( a++ );  --->  a ↔ 16 and b ↔ 30
  a-- ... a = a - 1
  a = 4 * ( ++b );  --->  a ↔ 124 and b ↔ 31
  a--;  --->  a ↔ 123
  --b;  --->  b ↔ 30
  ```

• Type conversion is also very handy ...  
  ```java
  int a = 73, b = 10;
  double c;
  c = a / b;  --->  c ↔ 7
  c = ( double ) a / b;  --->  c ↔ 7.3
  ```

• Comparisons  
  ```
  a == b  a < b  a > b
  a != b  a <= b  a >= b
  ```

  These have the obvious meaning for the primitive types.
• Logic

<table>
<thead>
<tr>
<th>A &amp;&amp; B</th>
<th>AND</th>
<th>A &amp; B</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td></td>
<td>B</td>
</tr>
<tr>
<td>! A</td>
<td>NOT</td>
<td></td>
</tr>
</tbody>
</table>

So for example, \( A \neq B \) and \( !(A == B) \) are equivalent. The difference between \& and && (similarly for | and ||) relies on “short-circuiting”.

\((3 == 7) && (2 == 3/0)\)
evaluates comfortably to false, since failure occurred in the first term there was no need to go further, but

\((3 == 7) & (2 == 3/0)\)
is a disaster, since the single ampersand has no short-circuit provision. Typically we use the ‘single’ flavour when we need to ensure that each term of the expression is evaluated, such as when incrementing variables.

• Control

• if ( ............ )
  {  ............  }
• else
  {  ............  }

Branching processes
• switch ( name ) {
  case value : ----- break ;
  case value : ----- break ;
  case value : ----- break ;
  default : ----- 
}

• do
  
  while ( ............ ) ;

We can now broaden our previous example ...
import java.io.*;

public class Arithmetic {
    public static void main ( String [ ] args ) throws Exception {
        InputStreamReader isr = new InputStreamReader ( System.in ) ;
        BufferedReader comingIn = new BufferedReader ( isr ) ;
        PrintWriter goingOut = new PrintWriter ( System.out , true ) ;

        char op ; // space to hold a character representing some arithmetic operator
        int x , y ; // space to store input
        double answer = 0 ; // space for answer, possibly decimal because of division!
        String ask = “Please enter an “ ; // more generic this time – notice the space for legibility

        goingOut.println ( ask + “integer.” ) ;
        x = Integer.parseInt ( comingIn.readLine( ) ) ;
        goingOut.println ( ask + “operator.” ) ;
        op = ( comingIn.readLine( ) ).charAt(0) ;
        goingOut.println ( ask + “integer.” ) ;
        y = Integer.parseInt ( comingIn.readLine( ) ) ;

        switch ( op ) {
            case ‘+’ :  answer = x + y ;  break ;
            case ‘-’ :  answer = x - y  ;  break ;
            case ‘*’ :  answer = x * y  ;  break ;
            case ‘/’ :  answer = y != 0   ?   x / y   :   Double.NaN ;      break ;
            default : goingOut.println( “Sorry, I didn’t understand what you meant ... bye-bye!” ) ;  answer = Double.NaN ;
        } // end switch statement interpreting the arithmetic operation to be performed

        goingOut.println ( “The value of ” +x+“ ” +op+“ ” +y+“ is ” +answer) ;
        goingOut.println ( “Thanks for using "Multiplier". Do come again!" ) ;
        comingIn.close( ) ;
    } // end of main method
} // end of class Arithmetic

CS 2110 Java overview- control 21
Delegation - There are times when it’s sensible to delegate certain tasks within a program. Javaspeak for this is methods (alias functions or subroutines). For example ...

```java
public class Stats {
    public static void main ( String [ ] args ) {
        int a = 7 , b = 12 , c = 24 ;
        double d ;

        d = avg ( a , b ) ;
        System.out.println ( “The average of ” +a+ “ and ” +b+ “ is ” +d+ “.” ) ;
        System.out.println ( “Including” +c+ “ changes this to” + avg( a , b , c ) + “.” ) ;
    } // end of main method

    public static double avg ( int x , int y ) {
        return ( x + y ) / 2.0 ;
    } // end of two-variable avg method

    public static double avg ( int x , int y , int z ) {
        return ( x + y + z ) / 3.0 ;
    } // end of three-variable avg method

} // end of class Stats
```

This is all playing with the thread of control, rather like passing the baton back and forth in a relay race.
• Types of types - For the **primitive types**, actual values are copied and passed. For any other ‘types’, since Java doesn’t really know how big they might be, instead of copying/passing whatever constitutes value, Java copies/passes the *address/reference* to the object, leaving the object wherever it happens to sit in memory. Indeed...

```java
Car ferrari = new Car(red) ;
```

does not make *ferrari* the actual car; it makes instead *ferrari* the *address* of the actual car.

```java
Car bus = new Car(green) ;
```
This leads to various consequences...

```
Hat cheap = new Hat (white) ;
Hat riding = new Hat (black) ;
```

Then

```
Hat tasteless = new Hat (white) ;
```

gives us two white hats, but

```
cheap != tasteless
```

because the addresses are different. However,

```
cheap = riding ;
```

makes the changes in cyan which mean

```
cheap == riding
```

both being the same address, namely the address of the riding hat ... the address of the cheap hat has been lost in the process!
This raises the obvious question of what _address_ a reference holds if it’s not currently referencing any particular object. The answer is

```
null
```

If this situation is not picked up by the compiler, but only noticed at run-time, then we can easily get the common error message

```
NullPointerException
```

if we attempt to access that object’s properties, i.e., a reference had been declared but not assigned the address of an actual existent object.

It’s also worth noting that once an object is no longer being referenced, as in _cheap_ after the assignment _cheap = riding_ ; Java allows the memory allocated to the object _cheap_ to be written over (whenever that may happen) ... this is called automatic garbage collection.

A reference cannot reference a primitive variable (hence the existence of _wrapper classes_ like _Integer_, _Double_, _Character_, _Boolean_, etc.).
• As a final comment in this vein, suppose

\[
yummy ( \text{rhubarb} ) ;
\]

is a call to a method \textit{yummy} with declaration

\[
\text{public int yummy ( \text{----- custard} ) \{} \text{.........................} \text{}}
\]

then if \text{-----} is a primitive type the variable \textit{custard} \textit{copies} the value of \textit{rhubarb}, but if \text{-----} is a reference type then \textit{custard} \textit{points} to the same object that \textit{rhubarb} does.

• We should observe that in

\[
\text{Car ferrari} = \text{new Car (red)} ;
\]

the word \textit{new} \textit{creates} an \textit{anonymous} object according to the manufacturer class \textit{Car} which has been told explicitly to make it \textit{red}; the phrase \textit{Car ferrari} \textit{makes} \textit{ferrari} an allowable \textit{reference} to a Car-like object; finally the \texttt{=} assigns the \texttt{address} of the anonymous car to the name \texttt{ferrari} !!!
• Strings - We’ve seen already that there is a non-primitive class `String` for which the operator `+` is defined by ‘concatenation’. Some other things about the String class are worth listing ...

```
String vide = ""; // no space between the pair of double quotes
```

creates `vide` as an empty String. Be aware that ‘a’ is a single character whereas “a” is a String with length one, and that `char` is a primitive but `String` is a reference type, so they cannot be the same.

Be aware also that using `==` and `!=` to compare Strings will probably* fail as with other reference types ... they refer to addresses not values. To test value, use either

```
trois.equals (two)  \rightarrow  true , false
```

```
trois.compareTo (two)  \rightarrow  trois <  = \rightarrow  two
```

Formally the `compareTo()` method comes from the `Comparable` interface, about which we’ll hear more later.
Other String tricks are ...

myriad.length( ) ;
which if the String myriad is “Did I blink?” returns the integer value 12, and ...

myriad.charAt( 9 ) ;
which returns the character value ‘n’, since the count starts with D being in position 0 not 1 !! Also ...

myriad.substring( 2 , 9 ) ;
returns the String “d I bli”, since the method counts from the first location number (included) to the last one (excluded).

Any primitive type can be converted to a String by, for example,

Integer.toString( 9746 ) ;
which returns the String (really the reference to the String) “9746”, so then if PI were to be defined in Math with the natural meaning, then

Double.toString( Math.PI ) ;
would return the address of a horribly long String!

(It’s worth looking at the API to see the various methods in the wrapper classes for the primitive types.)
Some of the other useful String methods, assuming \( s \) has been initialised to be some String, include ...

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( s.toUpperCase() )</td>
<td>returns the obvious new String</td>
</tr>
<tr>
<td>( s.toLowerCase() )</td>
<td>returns the obvious new String</td>
</tr>
<tr>
<td>( s.charAt(7) )</td>
<td>returns the char at position 7</td>
</tr>
<tr>
<td>( s.indexOf('t') )</td>
<td>returns the position number of the first ‘t’</td>
</tr>
<tr>
<td>( s.length() )</td>
<td>returns the length of ( s )</td>
</tr>
<tr>
<td>( \text{String.valueOf}(91.6) )</td>
<td>returns the input as a String “91.6”. Not just for numbers</td>
</tr>
<tr>
<td>( s.toCharArray() )</td>
<td>returns an array of chars formed from ( s )</td>
</tr>
<tr>
<td>( s.replace('t','o') )</td>
<td>returns a new String by replacing all t’s by o’s in ( s )</td>
</tr>
</tbody>
</table>

Again, you should look up the String class in the API to see other methods and more detail.

Finally, the following ‘conversions’ have equivalent effect ...

\[
\begin{align*}
\text{int} \ n &= \text{Integer.parseInt}(“96”); \\
\text{int} \ n &= \text{Integer.valueOf} (“96”).\text{intValue}(); \\
\text{int} \ n &= \text{new Integer} (“96”).\text{intValue}();
\end{align*}
\]

For primitives other than \text{byte, short, int, long}, only the analogues of the last two forms work.

(Filling in details from the * on slide 21, formally a String is an \text{immutable} object, but there are some oddities. If a String is initialised to a \text{String literal} by \text{String} \( \text{wow} = “gosh” \); then “gosh” has a particular memory location, so \text{String} \( a = “gosh”, \ b = “go”+“sh”, \ tail = “sh”; \) make \( \text{wow} == a \) and \( \text{wow} == b \) both true since as \text{String literals} they will point to the same created \text{String} location. But if \text{run-timing} a String, as \text{String} \( c = \text{new String} (“gosh”), \ d = “go”+tail, \ e = “go”+(true? “sh”: “duh”); \) all of \( \text{wow} == c \) and \( \text{wow} == d \) and \( \text{wow} == e \) will be false, yet \( f = c\.toString() \) gives \( a == f \) to be true. In all these cases the reference flavour of, for example, \( a\.equals(d) \) will be true. So \( == \) can give surprising results!)
Arrays - arrays in Java are reference types, so

```java
int [ ] boxes ; 
or
int boxes [ ] ;
```

declares `boxes` to be the reference for an integer array which doesn’t yet exist (so the value of `boxes` is `null`). To create, we of course use `new` ...

```java
boxes = new int [2010] ;
```

which creates an anonymous array to hold 2010 integers pointed to by `boxes` (i.e., it holds the address of that array), which might be initialised by ...

```java
for ( int i = 0 ; i < boxes.length ; i++ ) 
    // so length is a property, not a method, for arrays
    boxes [ i ] = i * i % 12 ;
    // or any other silly calculation you like!
```

Arrays of reference types are created similarly, though to avoid the dreaded `NullPointerException`, if our `array` were to be built by ...

```java
Rhubarb [ ] holes = new Rhubarb [2010] ;
```

then the initialisation might be ...

```java
for ( int i = 0 ; i < holes.length ; i++ )
    // just as with Strings, arrays start counting from 0
    holes [ i ] = new Rhubarb( i ) ; 
    // now there are some actual Rhubarbs there to access!
```

assuming that the class `Rhubarb` knows how to build with an integer `i`!

Again, since arrays are `references`, be careful with `=`, `==`, and `!=`. Of course, arrays are passed by reference into methods just like any other reference ... another standard source of errors.
Two (and higher) dimensional arrays are handled similarly ...

Rhubarb [] [] croquet;
declares croquet to be the reference of a two dimensional array of Rhubarbs, and then ...

croquet = new Rhubarb [12][24];
creates an array prepared to hold two gross of Rhubarb, which then can be filled with any actually created (using new) actual Rhubarbs.

It’s worth noting that the above two dimensional array croquet is really treated as if it were a one dimensional array holding 12 objects, where each of these objects is a one dimensional array holding 24 actual addresses of Rhubarbs.
So ...
This means we can write ...

```java
croquet = new Rhubarb[10][ ];
```
which makes `croquet` the reference of a two dimensional array of `Rhubarbs` which can be thought of as having 10 rows of as yet undetermined length (note that the computer has no real sense of rows vs columns!). Of course, the length of each row will have to be fixed (using `new`) before the rows can be filled ...

```java
for ( int i = 0 ; i < croquet.length ; i++ ) {
    croquet[i] = new Rhubarb[i*i+1];
    for ( int j = 0 ; j < croquet[i].length ; j++ )
        croquet[i][j] = new Rhubarb(i+j);
} // end outer for loop on i
```

thus forming a *triangular* array!

Using this and similar tricks, we can size our arrays dynamically, though always being careful to remember that we’re working with references. The concrete size of the array must be fixed before it’s used.

You must have wondered what the array `args` is used for in ...

```java
public static void main ( String[] args ) { ......................... }
```

It’s designed to allow the use of `command line arguments`. So ...

Tuesday, January 20, 2009
public class Average {
    public static void main ( String [] args ) throws Exception {
        double avg = 0.0;

        if ( args.length != 0 ) {
            // so this array has some size!
            for ( int i = 0 ; i < args.length ; i++ )
                avg += Integer.parseInt ( args [ i ] );
            avg = avg / args.length ;
            System.out.println ( "The average of the input values is " + avg );
        } // end if args array has any length
        else
            System.out.println ( "You didn’t give me any numbers!" );
    } // end main method
} // end class Average

This could be invoked by running the compiled program from a command line and typing ...

Average 3 17 4 9684 18

forms the args array of Strings

Whilst we’re on the topic of arrays, there’s a nice way, using the split( ) method from the String class, to deconstruct strings into potentially convenient arrays of characters, which is worth a digression ...
Suppose we’ve been given the string “Once upon a time trouble 7 168”, and would like to break it up into ‘words’. We could no doubt code this using a bunch of for loops and some careful checking of spaces (try it for fun!), but alternatively ...

```java
String input = "Once upon a time trouble 7 168";
String[] outArray = input.split(" ");
```

will create the following array of Strings ...

```
Once upon a time trouble 7 168
```

If we wanted instead to separate the string by the letter ‘o’ instead of spaces, then ...

```java
outArray = input.split("o");
```

gives the array ...

```
Once upon a time trouble 7 168
```

Note that it ignored the upper case ‘O’. Even more adventurous is ...

```java
outArray = input.split("[bcir6]");
```

which yields ...

```
Once upon a time trouble 7 168
```

thus separating it by any of the characters within the square brackets!
This [bcir6] is an example of a regular expression. They form a convenient collection of tricks (often rather arcane!) for searching for patterns amongst strings of characters. Although you can read about them easily online, we’ll give a short list of some of the more useful examples ...

<table>
<thead>
<tr>
<th>Regular Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ack7]</td>
<td>any of a, c, k, or 7</td>
</tr>
<tr>
<td>[^tvW]</td>
<td>any char other than t, v, or W</td>
</tr>
<tr>
<td>[a-z&amp;&amp;[^b-p]]</td>
<td>any of a through z except b through p</td>
</tr>
<tr>
<td>\d</td>
<td>any digit (i.e., 0,1,2,3,4,5,6,7,8,9)</td>
</tr>
<tr>
<td>\s</td>
<td>any space (also tab, newline, etc.)</td>
</tr>
<tr>
<td>\w</td>
<td>any word char (a-z, a-Z, 0-9, and _ )</td>
</tr>
<tr>
<td>.</td>
<td>any character! (but not newlines)</td>
</tr>
<tr>
<td>a?</td>
<td>‘a’ exactly once or not at all</td>
</tr>
<tr>
<td>ra*t</td>
<td>“rt” or “rat” or “raat” or “raaat” etc.</td>
</tr>
<tr>
<td>ra{3}t</td>
<td>only “raaat”… and ra{3,5}t allows 3-5 a’s</td>
</tr>
</tbody>
</table>

So, as an example ...

```java
String junk = "this is a loooongish krazy string with gaziiliions of i's in it, gosh!" ;
String test = "[io]{1,2}\w\s.." ; // notice the need for \ to ensure that the ‘ \’ is ‘seen’
String [ ] stuff = junk.split(test) ;
```

```java
th  a l  azy str  th gaz  i's  , gosh!
```

There’s an initial space here.