

**CIS \* 1650**

**Fall 2003**

J. Morton

## **Lecture Notes**

Compiled by:

**Dave Heppenstall**  
**dheppens@uoguelph.ca**

```
public class Beginnings
{
    public static void main(String[] args)
    {
        System.out.println("The White Rabbit put on his spectacles.");
        System.out.println("'Where shall I begin, please your Majesty?");
        System.out.println("he asked.");
    }
}
```

- Every Java program consists of one (or more) classes.
  - The class is the main organizing instrument of the Java language.
  - Class Name: Program must be stored in a file called `ClassName.java`
  - Often, a class will contain one (or more) methods.
  - A method is a group of program statements that are given a single name.
  - All statements in Java end in a semicolon.
  - The reason the other lines in the program do not end in a semicolon is that they are not, technically, statements. A Template for Creating Simple Java Programs
- For now, use this template when creating your own Java programs:

```
public class ClassName
{
    public static void main(String[] args)
    {
        YOUR CODE GOES HERE!
    }
}
```

- Save your program in a file called `ClassName.java`
- The code executed when the `println` method is invoked is not defined in the program, but is part of the object out defined in Java's `System` class. The `System.out` object has two methods we can use for printing text to the screen: `println()` and `print()`.

```
public class Countdown
{
    public static void main(String[] args)
    {
        System.out.print("Three...");
        System.out.print("Two...");
        System.out.print("One...");
        System.out.println("Zero");
        System.out.println("Houston, we have liftoff!");
    }
}
```

- Filename: `Countdown.java`
  - Compile Command: `javac Countdown.java`
  - Execute Command: `java Countdown`
- Result on console: `Three...Two...One...Zero`  
`Houston, we have liftoff!`

- What if we wanted to print a double quote character?

```
System.out.println(" "Where shall I begin, please your Majesty?" ");
```

- An escape sequence starts with a backslash character (\), which indicates that the characters should be treated in a special way

```
System.out.println("\"Where shall I begin, please your Majesty?\"");  
"Where shall I begin, please your Majesty?"
```

### **Java Escape Sequences**

<u>Code</u>	<u>Escape Sequence</u>
-------------	------------------------

<code>\b</code>	backspace
<code>\t</code>	tab
<code>\n</code>	newline
<code>\r</code>	carriage return
<code>\"</code>	double quote
<code>'</code>	single quote
<code>\\</code>	backslash

```
public class TwoJokes  
{  
    public static void main(String[] args)  
    {  
        System.out.print("Two jokes by Steven Wright:\n1.\t");  
        System.out.print("I have an answering machine in my ");  
        System.out.print("car.\n\t It says, \"I'm home now, ");  
        System.out.print("but leave a message and I'll call when ");  
        System.out.print("I'm out.\n2.\tI went to a restaurant ");  
        System.out.print("that serves \"breakfast at any time.\"");  
        System.out.print("\n\tSo I ordered French Toast during ");  
        System.out.print("the Renaissance.\n");  
    }  
}
```

Two jokes by Steven Wright:

1. I have an answering machine in my car.

It says, "I'm home now, but leave a message and I'll call when I'm out."

2. I went to a restaurant that serves "breakfast at any time."

So I ordered French Toast during the Renaissance.

### **String Concatenation**

- Can a string (literal) span more than one line?
- No! But the string concatenation (+) operator can be used to append one string to the end of another.

```
System.out.print("One joke by Steven Wright:\n1.\t" +  
    "I have an answering machine in my " +  
    "car.\n\t It says, \"I'm home now, " +  
    "but leave a message and I'll call when " +  
    "I'm out.\n");
```

“One joke by Steven Wright:\n1.\tI have an answering machine...”

- Take Care When Using String Concatenation
- The plus (+) operator is also used for arithmetic addition
- The function the operator performs depends on the type of information on which it operates.

• **Rules:**

1. Operator performs string concatenation → if one or more of the operands is a string
2. Operator performs addition → if both operands are numeric

*Note:* The plus operator is evaluated left to right.

```
public class Plus
{
    public static void main(String[] args)
    {
        System.out.println(10 + 15);
        System.out.println(10 + 15 + "= 10 + 15");
        System.out.println("10 + 15 = " + 10 + 15);
    }
}
```

25

25 = 10 + 15

10 + 15 = 1015

## Lecture 03 – Style and Identifiers

12 Sept 2003

### Comments

- Comments should be included in a program to explain the purpose of the program and to describe the processing steps.
- Comments are not to affect how a program works
- Java comments can take the following two forms:

```
// This is a comment that runs to the end of the line
/* This comment runs to the terminating symbol,
even across line breaks */
```

- Spaces, blank lines, and tabs are collectively called white space
- White space is used to separate words and symbols in a program
- Extra white space is ignored.
- Programs should judiciously employ white space to enhance readability. This includes using consistent indentation
- See *BeginningsWithStyle.java*

## White Space and Indentation Styles

- There are two valid indentation styles

### Block indentation style...

```
public class Beginnings
{
    public static void main(String[] args)
    {
        System.out.println("The White Rabbit put on his spectacles.");
        System.out.println("'Where shall I begin, please your Majesty?");
        System.out.println("he asked.");
    }
}
```

### K&R indentation style...

```
public class Beginnings {
    public static void main(String[] args){
        System.out.println("The White Rabbit put on his spectacles.");
        System.out.println("'Where shall I begin, please your Majesty?");
        System.out.println("he asked.");
    }
}
```

## Evaluation of Method Arguments

- Method arguments are evaluated first, then passed to the method

```
System.out.println("The answer is " + (25 + 3 - 8));
                                     ^ This is evaluated first
                                     The result, 20, is placed back in the equation & the evaluation continues
System.out.println("The answer is " + 20);
System.out.println("The answer is 20");
```

## Arithmetic Operators

- There are five basic arithmetic operations:

<u>Op</u>	<u>Use</u>	<u>Description</u>
+	op1 + op2	Adds op1 and op2
-	op1 - op2	Subtracts op2 from op1
*	op1 * op2	Multiplies op1 by op2
/	op1 / op2	Divides op1 by op2
%	op1 % op2	Divides op1 by op2 and returns the remainder

- The operands of the arithmetic operators must be of a numeric type
- The order of operations is the same as it is in algebra:
  - parenthesis()
  - multiplication / division / remainder
  - addition / subtraction
- Within a level, the terms are evaluated left to right
- Identifiers are the words a programmer uses in a program.
- An identifier can be made up of any combination of the following: A-Z, a-z, 0-9, \_, \$
- They cannot begin with a digit
- Java is case sensitive so Beginnings and beginnings are different identifiers
  - Valid Variables... Label, nextStockItem, NUM\_CARS, \$amount
  - Illegal Variables... 4th\_value, coint#value, all-for-one

- Reserved words have predefined meaning in the language and cannot be used as names.

abstract	continue	future	native	short	var
boolean	default	generic	new	static	void
break	do	goto	null	super	volatile
byte	double	if	operator	switch	while
byvalue	else	implements	outer	synchronized	
case	extends	import	package	this	
cast	false	inner	private	throw	
catch	final	instanceof	protected	throws	
char	finally	int	public	transient	
class	float	interface	rest	true	
const	for	long	return	try	

---

## Lecture 04 – More on Identifiers

15 Sept 2003

1. The computation is more easily understood if broken into smaller sub-parts
2. Intervening steps need to be done before the computation can continue
3. The program can be made abstract by using variable instead of actual numbers (i.e generalized)

Similar to the use of variables in mathematics:

$$n + n = 2n$$

$$1 + 1 = 2$$

$$2 + 2 = 4$$

$$3 + 3 = 6$$

- An alternate view: a variable is a name that represents a location in memory where a value is stored
- All variables must be declared
  - specifies the variable's name
  - Also gives the type of information that will be held there

**int**      The variable holds an integer value

**double**    The variable holds a 'real' number

**String**    A variable holds a string of text

- A variable can be given an initial value by using the assignment (=) operator.

```
int total=10;
String label="Price";
double price=1.2, result=10.0, temp;
temp = -3.1;
```

total	10
price	1.2
label	"Price"
result	10.0
temp	-3.1

- When a variable is referenced in a program, its current value is used
- Look at *TimeExample.java* example
- Look at *Count.java* example

- There are other assignment operators other than =

```

++x      x = x + 1
--x      x = x - 1
x += value  x = x + value
x -= value  x = x - value
x *= value  x = x * value
x /= value  x = x / value
x %= value  x = x % value
    
```

- The ++ and -- operators can be used within any arithmetic evaluation

```

x = 1; x
y = 2 * ++x + 5;
    
```

**Comparison Operators**

- A comparison operator compares two numbers and returns true or false depending on the results of the comparison.

- Simplest comparison is: equality ==

```

5 == 5    true
2 == 4    false
    
```

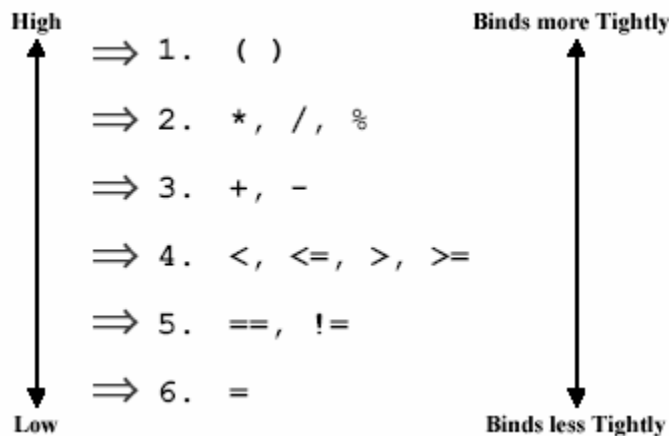
- Don't confuse equality (==) with assignment (=)
- Very common mistake: accidentally typing = instead of ==
- Assume x stores the value 7

```

x == 3    false
x = 3     assign 3 to x
    
```

<u>Operator</u>	<u>Meaning</u>
x == y	x is equal to y
x < y	x is (strictly) less than y
x <= y	x is less than or equal to y
x > y	x is (strictly) greater than y
x >= y	x is greater than or equal to y
x != y	x is not equal to y

- Order of Precedence:



### Syntax 1:

```
if (condition)
    statement;
```

### Syntax 2:

```
if (condition)
{
    statement_1;
    statement_2;
    statement_n;
    ...
}
```

- condition can be any Boolean expression...

```
value1 < value2
value1 > value2
value1 == value2
value1 >= value2
value1 <= value2
value1 != value2
```

### **Braces {}**

- Braces identify a block of statements that act like a single unit

```
{
    versus statement;
}
```

- Where Java allows a single statement, it allows multiple statements between braces
- End of brace *equals* end of statement
- No need for a ;

### **If – Else Statement Pairs**

#### Syntax 1:

```
if (condition)
    statement;
else
    statement;
```

#### Syntax 2:

```
if (condition)
{
    statement_1;
    statement_2;
    statement_n;
}
else
{
    statement_1;
    statement_2;
    statement_n;
}
```

- See *IfDemo.java*
- See *TimeExample2.java*

Variants on the TimeExample

- See *TimeExample2a.java*
  - Code as written in class
- See *TimeExample2b.java*
  - Handles 12hr problem
  - Writes 4:3pm as 4:03pm
  - Uses Cascading and Nested Ifs
- See *TimeExample2c.java*
  - Rewrite of TimeExample2b
  - uses extra variables and a different algorithm to achieve 'cleaner' more easily understood code
  - Uses only simple ifs and if-else's

Nested Ifs

- Any statements can be placed inside an **if** block or **else** block
- This can include other if statements!

E.g. 1. Let's say we want to

- double a if  $a < b$ 
  - print the new a if we are in 'display' mode ( $display == 1$ )
- double b if  $b < a$ 
  - print the new b if we are in 'display' mode ( $display == 1$ )

```
if(a < b)
{
    a *= 2;
    if (display == 1)
        System.out.println("a = " + a);
}
else
{
    b *= 2;
    if (display == 1)
        System.out.println("b = " + b);
}
```

enters when $b < = a$
--------------------------

E.g. 2. Try again:

- double a if  $a < b$ 
  - print the new a if we are in 'display' mode ( $display == 1$ )
- else,  $a \geq b$ 
  - if  $a \neq b$ 
    - double b if  $b < a$
    - print the new b if we are in 'display' mode ( $display == 1$ )

```
if(a < b)
{
    a *= 2;
    if (display == 1)
        System.out.println("a = " + a);
}
else
{
    if (a != b)
    {
        b *= 2;
        if (display == 1)
            System.out.println("b = " + b);
    }
}
```

**Only one  
statement**

OR

```
if(a < b)
{
    a *= 2;
    if (display == 1)
        System.out.println("a = " + a);
}
else if (a != b)
{
    b *= 2;
    if (display == 1)
        System.out.println("b = " + b);
}
```

### Cascading If

- The cascade can continue for as many levels as needed...

```
if(grade < 50)
    System.out.println("Failed");
else if (grade < 60)
    System.out.println("D");
else if (grade < 70)
    System.out.println("C");
else if (grade < 80)
    System.out.println("B");
else if (grade < 90)
    System.out.println("A");
else
    System.out.println("A+");
```

Leap Year Example: The rules for leap year are as follows:

- If a year is divisible by 4 it is a leap year
- Unless the year is also divisible by 100
  - Then although divisible by 4, it is not a leap year
- Unless the year is also divisible by 400
  - Then, although divisible by 100, it is a leap year

2000	yes	(divisible by 400)
2002	no	(not divisible by 4)
2004	yes	(divisible by 4 & not by 100)
2100	no	(divisible by 100 & not by 400)

Let's create *LeapYear.java*

### Boolean Variables

- Boolean values are simply 'true' and 'false'
- They are represented in Java by the literals: **false** **true**

<code>3 &lt; 4 != true</code>	<code>⇒</code>	<b>false</b>
<code>true == true</code>	<code>⇒</code>	<b>true</b>
<code>true != true</code>	<code>⇒</code>	<b>false</b>
<code>false != true</code>	<code>⇒</code>	<b>true</b>
<code>false == false</code>	<code>⇒</code>	<b>true</b>
<code>false &lt; true</code>	<code>⇒</code>	<b>doesn't compile</b> <b>&lt; only handles numbers</b>

```
if (true)
    System.out.println("If block has run");
else
    System.out.println("Else block has run");
```

→ If block has run

```
if (false)
    System.out.println("If block has run");
else
    System.out.println("Else block has run");
```

→ Else block has run

- There are variables that can hold true/false values
- They are said to have a 'Boolean' data type

```
double value = Math.random() * 100;
boolean topHalf = value >= 50.0;
if (topHalf)
    System.out.println(value + " is >= 50%");
else
    System.out.println(value + " is < 50%");
```

**Boolean Operators**

- Just as we have arithmetic operators such as + and \*, there are Boolean operators

!     |     ||     ^     ?     :     &     &&

**!** Not

- ! (read 'not' or 'bang') is a unary operator
- Returns the opposite result

```
!true            false
!false           true
!(3 == 3)        false
!(3 != 5)        false
```

**&, &&** And

x	y	x && y
false	false	false
false	true	false
true	false	false
true	true	true

```
true && false     → false
false && false    → false
(3 < 5) && (10 > 2) → true

(5 < 1) & (10 > 2)
= false & (10 > 2)
= false & true
= false

(5 < 1) && (10 > 2)
= false && (10 > 2)
= false
```

- Binary operator (takes 2 operands)
- Returns 'true' only if both operands are true, otherwise returns 'false'
- && stops at first false and returns false otherwise returns true
- & evaluates all operands then returns the result

## |, || Inclusive Or

x	y	x    y
false	false	false
false	true	false
true	false	false
true	true	true

```

true || false → true
false || false → false
(3 < 5) && (10 > 2) → true
(3 > 5) && (10 > 2) → true

(10 > 2) | (5 < 1)
= true | (5 > 1)
= true | false
= true

(10 > 2) | (5 < 1)
= true | (5 > 1)
= true

```

- Binary operator (takes 2 operands)
- Returns 'true' only if either (or both) operands are true, otherwise returns 'false'
- || stops at first true and returns true, otherwise returns false
- | evaluates all operands then returns result

## ^ Exclusive Or

x	y	x ^ y
false	false	false
false	true	true
true	false	true
true	true	true

```

true ^ false → true
false ^ false → false
(3 < 5) && (10 > 2) → false
(3 > 5) && (10 > 2) → true

```

- Binary operator (takes 2 operands)
- Returns 'true' only if either (but not both) operands are true, otherwise returns 'false'

## If-Then-Else Operator

- Ternary operator (3 operands)
- (condition)? then-expression : else-expression

*e.g. set z to the smaller of the two values, x or y*

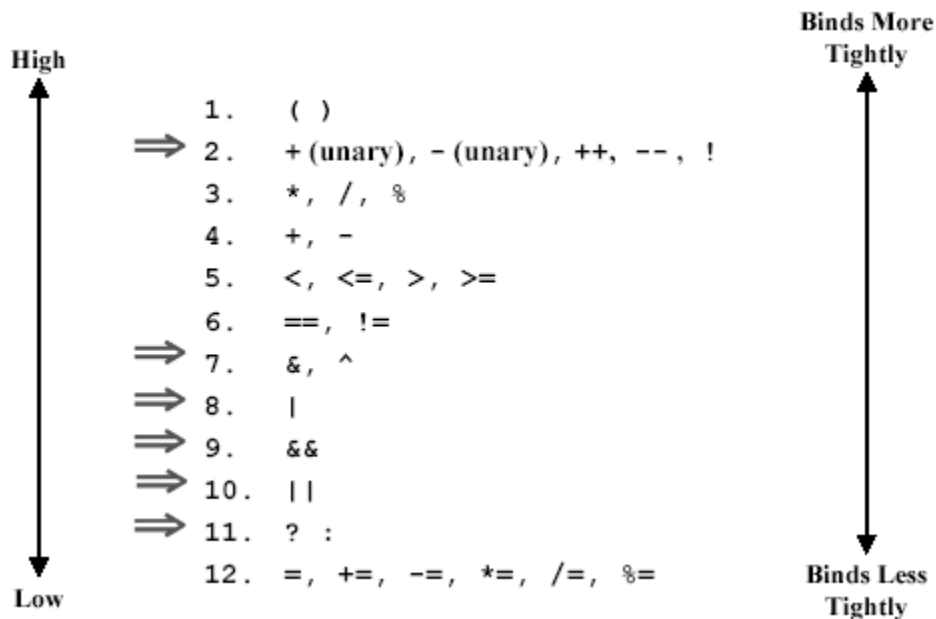
```
if (x < y)
    z = x;
else
    z = y;

z = (x < y) ? x : y;
```

*e.g. Print out a monthly due date*

- the month is randomly generated
- Examples of the printing format:
  1. "The book is due in 5 months"
  2. "The book is due in 1 month"

## Order of Precedence



## Lecture 10 – The While Loop

29 Sept 2003

### **While:**

- The while statement is Java's most fundamental looping statement.
- It repeats a statement (or block of statements) while its controlling expression is true.
- The while statement has the following syntax (identical to if): Syntax for single statement

```
while (condition)
{
    statements;
}
```

- Statement performed over and over again until condition becomes false can be any Boolean expression

### Purposes for Using Loops

- Two different ways of using a loop
  1. Repeat code over and over again (with modification)
    - e.g. 100 bottles of beer song lyrics
    - mathematical sequences
    - iterate over arrays, or user input
  2. Use a variable to cumulate values
    - averages, standard deviations etc.
    - mathematical series, cumulative interest, etc.

---

## Lecture 11 – Do-While and For Loops

1 Oct 2003

### **Do-While:**

- condition tested at end of each repetition
- unlike while, allows 1st execution in all cases similar to the while statement

```
do
{
    statements;
}
while (condition);
```

### **For:**

- An extension of the While loop which comprises:
  - initialization phase
  - while loop
  - update phase

```
for(initialize ; condition ; update)
{
    statements;
}
```

**The Comma Operator:**

```
for(init_1, init_2, ... , init_M; condition ; update_a, update_b ... update_N)
{
    statement_1;
    statement_2;
    statement_3;
}
```

- Note: type declaration can only be used in init\_1. See *ForLoopComma.java*

**Specializing Iterations:**

- ‘if’ statements can be written within loops to control the behavior of individual iterations. See *SqrtTable.java*

**Nested Loops:**

- Just as with ‘if’ statements, loops can be written within other loops
- There are various reasons to use nested loops
  - Code that needs repeating has a loop in it
  - Need all possible combinations of two or more variables (Cartesian cross)
- Just as with ‘if’ statements, loops can be written within other loops
- Nested loops may or may not interact with the outer loop:
  - Inner loop may be independent of the outer loop
  - or*
  - Inner loop’s body may use outer loops updating variable
  - or*
  - Inner loop’s condition may use outer loops updating variable

- *Box.java*
- *MultiplicationTable*
- *FactorialTable.java*

- Java stores integers in binary form:

<u>Decimal</u>	<u>Binary</u>
5801	1101
$5 \times 10^3 = 5 \times 1000$	$1 \times 2^3 = 1 \times 8$
$+ 8 \times 10^2 = 8 \times 100$	$+ 1 \times 2^2 = 1 \times 4$
$+ 0 \times 10^1 = 0 \times 10$	$+ 0 \times 2^1 = 0 \times 2$
$+ 1 \times 10^0 = 1 \times 1$	$+ 1 \times 2^0 = 1 \times 1$
	<u>13 decimal</u>
Each digit can be 0-9	Each digit can be 0-1

- Negative numbers also stored in binary format
  - There is a sign bit on far left
- 1 means negative, 0 means positive
  - Rest of the number stored in two's compliment form
- Similar to positive binary numbers
- modified to allow for the easy addition of positive and negative numbers
- The more bits used, the larger the numbers that can be stored
- A computer has only a limited number of wires that can carry bits (called word size)
  - 32 bit machine has 32 wires
  - 64 bit machine has 64 wires
- Consequently, there are various sized integer data types to 'match' the word size of various machines
- There are 4 different integer types
  - Each has a different size:

<b>type</b>	<b>size</b>	<b>minimum value</b>	<b>maximum value</b>
byte	8 bits = 1 byte	-128	127
short	16 bits = 2 bytes	-32768	32767
int	32 bits = 4 bytes	-2147483648	2147483647
long	64 bits = 8 bytes	-9223372036854775808	9223372036854775807

## Integer Literals:

- Literals that you specify are of type `int` unless suffixed with an `l` or `L` denoting they are type `long`.

```
int biggestValue = 2147483647; fine Largest int literal can be stored in an int
int biggestValue = 2147483648; error 1 larger than an int
long biggerThanInt = 2147483648; error Literal still type int & so too small
long biggerThanInt = 2147483648L; fine Literal now type long & is stored in a long
```

## • Octal

- Base 8
- Each digit can be 0 – 7
- In Java, octal literals start with a leading 0

05701

$$\begin{array}{r} 5 \times 8^3 = 5 \times 512 = 2560 \\ + 7 \times 8^2 = 7 \times 64 = 448 \\ + 0 \times 8^1 = 0 \times 8 = 0 \\ + 1 \times 8^0 = 1 \times 1 = 1 \\ \hline 3009 \end{array}$$

## • Hexadecimal (Hex)

- Base 16
- Each digit can be 0 – 9, A-F (A=10, B=11, ... ,F=15)
- In Java, hexadecimal literals start with a leading 0x

0x2B5F

$$\begin{array}{r} 2 \times 16^3 = 2 \times 4096 = 8192 \\ + B \times 16^2 = B \times 256 = 2816 \\ + 5 \times 16^1 = 5 \times 16 = 80 \\ + F \times 16^0 = F \times 1 = 15 \\ \hline 11103 \end{array}$$

## Large Value Computations

- Compare two programs
  - *IntPowerOfTen.java*
  - *LongPowerOfTen.java*

### Overflow problem:

- Overflow occurs when the result of an operation on two large values exceeds the size of the data type. Let all values be of size unsigned Byte (not a Java data type).

$$\begin{array}{r} 1111\ 1111 \\ 255 \end{array} + \begin{array}{r} 1111\ 1111 \\ 255 \end{array} = \begin{array}{r} 1\ 1111\ 1110 \\ 510 \end{array}$$

Larger than type: byte

$$\begin{array}{r} 1111\ 1111 \\ 255 \end{array} + \begin{array}{r} 1111\ 1111 \\ 255 \end{array} = \neq \begin{array}{r} 1111\ 1110 \\ 126 \end{array}$$

So we must cut some off

- But  $255 + 255$  does not equal 126. This is the problem.

---

## Lecture 14 – Floating Point Literals and Numbers

8 Oct 2003

### Floating Point Numbers:

- Floating-point (real) numbers have decimal places
- They are stored as a more complex structure

$$\begin{array}{l} 4318.92341 = 0.431892341 \times 10^4 \quad 0\ 431892341000\ 004 \\ -3.1415 = -0.31415 \times 10^1 \quad 1\ 314150000000\ 001 \end{array}$$

- Notice that the decimal point floats
- The number of decimal places allowed depends on total number of digits in number

- There are 2 different sizes of floating point numbers:

<u>type</u>	<u>size</u>	<u>sign bit</u>	<u>Mantissa</u>	<u>exponent</u>
<b>float</b>	32 bits	1	23	8
<b>double</b>	64 bits	1	52	11

<u>type</u>	<u>min value</u>	<u>max value</u>	<u>value closest to 0</u>	<u>accuracy</u>
<b>float</b>	$-3.4 \times 10^{-38}$	$3.4 \times 10^{38}$	$-3.4 \times 10^{-38}$	7 digits
<b>double</b>	$-1.7 \times 10^{308}$	$1.7 \times 10^{308}$	$-1.7 \times 10^{-308}$	16 digits

- FP Literals come in 2 forms
  - Decimal notation      129.135
  - Scientific notation    1.29135e2      ( $1.29135 \times 10^2$ )

- Literals are of type double by default
  - if suffixed with an f or F then they are of type float
  - Opposite of L suffix for int/long, default is the larger type

```
double a = 0.00314;
double a = 3.14E-3;
float a = 0.00314F;
float a = 3.14E-3f;
```

### Roundoff errors:

- A finite number of digits after the decimal point has consequences

$$1/3 = 0.3333333333333333... \infty$$

Since Java has at most 16 digits accuracy...

$$1.0 / 3.0 = 0.3333333333333333$$

and so  $1.0 / 3.0 * 3.0 = 0.9999999999999999$  not 1

- Roundoff error in conjunction with binary representation of numbers can cause unexpected behaviour when using real numbers

$$(0.65)_{10} = (0.101001)_2$$

SB	Mantissa	Exponent
0	10100110011001100110011	11111111

+ 1.0100110011001100110010 x 2<sup>-1</sup>  
which is 0.649609089 in decimal form

- i.e. just because you can write it with few digits in your code doesn't mean that it is stored with just that number of digits in the machine. *See AppleTable.java*
- Because of the precision problem, when counting using floats or doubles, do not write the terminating condition using equality statement:

```
for(double count = 0.0; count == 1.0; count += 0.1)
```

'count' may never exactly equal 1.0. The 'for' statement may never terminate! Instead:

```
for(double count = 0.0; count < 1.1; count += 0.1)
```

## Lecture 15 – Arithmetic Promotion and Assignment

10 Oct 2003

- Arithmetic promotion occurs automatically when an operator needs to modify the type of its operands to perform the operation.
- Smaller types get converted into larger types because they 'fit'

```
short + long    → long
int + byte     → int
double + float → double
float + int    → float
```

`long + double → double`

- The final example was a bit misleading (`long + double → double`)
- You can form much larger numbers with double than with long (e.g.  $1.2e92 = 1.2 \times 10^{92}$  cannot be represented as a long number)
- However, because of double's limited precision, you might get Roundoff errors in the number:  
e.g.  $1234567890123456789 + 0.0 \rightarrow 1.234567890123456e18$   
'789' dropped from number

- An arithmetic expression can have very different results because of arithmetic promotion:

`12 / 5 → 2`

`12 / 5.0 → 2.4`

`int / long + float - int / double`  
`21 / 6L + 3.6F - 21 / 6.0`

= `long + float - int / double`  
`3L + 3.6F - 21 / 6.0`

= `float - int / double`  
`6.6F - 21 / 6.0`

= `float - double`  
`6.6F - 3.5`

= `3.1 double`

- Surprisingly, all arithmetic operators convert any integer smaller than **int** to type **int**

### Assignment Conversion:

- Type conversion is also done automatically when assigning values to variables
  - As long as the value is a smaller type than the variable
  - This is called a widening conversion
  - You get a compiler error if the value is of a larger type than the variable

`long x = 23;`     **23 is converted into type long**

`double x = 23;`     **23 is converted into 23.0 (type double)**

`float x = 23.4;`     **error a double cannot be stored in a float**

`float x = 23.4f;`     **fine, no conversion necessary**

`int x = 1L;`     **error a long cannot be stored in an int**

*No Lecture Today. Lab and seminars have been cancelled for this week.*

---

```
int x = 1L;
```

*error a long cannot be stored in an int*

- Notice that the above value could easily be stored as an int
  - It is the type that is too big not the value
  - This is called a narrowing conversion & it can be done
- Just cast the value from type long to type int before storing
- To cast just write ‘(the new type)’ in front of the value

```
int x = (int) 1L;
```

*now it works*

- More Examples...

```
byte x = (byte) 36;
```

*this works*

```
byte x = (byte) 200;
```

*this also works even though value (127)*

*will store - 56*

```
float x = 5 + (float) 2.4;
```

*this works*

### **Integer Operators & Small Data Types:**

- Look at the following example:

```
L1 short x = (short) 5;
```

```
L2 short y = (short) 3;
```

```
L3 short z;
```

```
L4
```

```
L5 z = x + y;
```

- This produces a compiler error!
- Says: in L5 we are storing an int into a var of type short
- The reason: all arithmetic operators turn small data types (below int) into an int
- Now correct:

```
L1 short x = (short) 5;
```

```
L2 short y = (short) 3;
```

```
L3 short z;
```

```
L4
```

```
L5 z = (short)(x + y);
```

## Random Numbers:

Define variables: **int** y, n, a, b; **double** x;

- To generate a “random” number  $0.0 \leq x < 1.0$

```
x = Math.random();
```

- To generate a random number  $0.0 \leq x < n$

```
x = Math.random() * n;
```

- To generate a random number  $0 \leq y \leq n-1$

```
x = (int) (Math.random() * n);
```

- To generate a “random” integer  $1 \leq y \leq n$

```
y = (int) (Math.random() * n) + 1;
```



- To generate a random number between  $a \leq y \leq b$ , where  $a < b$

```
y = (int) (Math.random() * (b - a + 1)) + a;
```

---

## Lecture 18 – Numerical Methods

17 Oct 2003

- Java’s math class provides a variety of methods for performing advanced mathematical operations.

### Example Methods

Return the sine of argument

```
double Math.sin(double)
```

Return the largest integer less than or equal to argument

```
double Math.floor(double)
```

Return the nearest integer to the argument

```
double Math rint(double)
```

- *The argument that you put between the parentheses following the method name can be any expression that produces a value of the required type.*

### Example Constants

p: Return 3.14159276...      `double Math.PI`

e: Returns 2.71828182...      `double Math.E`

- The Math class is a part of Java’s library of methods
  - called Java’s Application Programming Interface (API)
  - List of all methods available is at Sun’s website: <http://java.sun.com/j2se/1.3/docs/api/index.html>

### Rounding to a Chosen Decimal Place:

- Converting an ugly looking double value to a “nice” real number with n decimal digits:
  - Multiply by  $10^n$
  - Round to the nearest integer
  - Divide by  $10^n$

e.g. round 5.2871513234 to 2 decimal places...

```
5.2871513234 * 100      = 528.71513234
Math rint(528.71513234 * 100) = 529.0
Math rint(529.71513234 * 100) / 100 = 5.29
```

- Java automatically drops trailing 0s after the decimal place
  - So \$12.30 would print as \$12.3
- There is nothing we can do about this yet
  - Java does have sophisticated number formatting facilities
  - We will learn about them later when we become more familiar with the use of Objects
- For now to force the second decimal place
  - check to see if there is only one decimal digit  
 $x * 100 \% 10 == 0$
  - Add a “0” to the number through string concatenation
  - Handle the first decimal place (in the case of \$12.00) similarly

### Converting Strings to Numbers:

- We know that “512” is not the same as 512
  - “512” is a string
  - 512 is an int
- If we get a number in the form of a string it would be nice to turn it into a number so we can do arithmetic on it
- Java provides methods to do just that Strings to Integers

#### ...Integers

- there are various ‘number’ classes that hold useful methods when dealing with numbers
  - same as Math class that holds useful methods for arithmetic
  - called wrapper classes
- class called Integer has a method called `parseInt`
  - `Integer.parseInt(String numAsString)`
  - takes a string and turns it into an integer

```
String number = "123";
int n = Integer.parseInt(number);
System.out.println("n = " + n);
System.out.println("2n = " + 2*n);
```

Screen Output:

```
n = 123
2n = 246
```

## ...Doubles

- class Double has a method called parseDouble
  - `Double.parseDouble(String numAsString)`
  - takes a string and turns it into an double

```
String number = "123.45";  
double n = Double.parseDouble(number);  
System.out.println("n = " + n);  
System.out.println("2n = " + 2*n);
```

Screen Output: n = 123.45 2n = 246.9
--

- All that computers ‘understand’ is electrical current down wires:  
no flow = off = 0 flow = on = 1
- Sequence of wires allow representation of more than 1 bit
  - We have used this to represent numbers
  - Computer has circuits that perform ‘addition’, ‘multiplication’ and ‘logic’ on binary numbers
- Therefore numbers are the basic building blocks of everything known to the computer
  - Yet we know that computers handle letters, words, paragraphs, etc. in text editors and word processors
  - How? **All letter, punctuation, etc. are all NUMBERS**
- In fact to Java they are all integer of size int
  - Although `println()` converts them to size short
- Each number is treated as if it were a letter
  - When displayed to the screen the number is converted to pixels in the shape of the letter it represents
  - When checking a dictionary the ‘letters’ in the ‘word’ are numerically compared to the ‘letters’ of a word in a dictionary etc
- The number to letter pairing is called a ‘code’
- Of course more than letters have to be represented
  - Punctuation
  - Symbols such as @ # \$ % & + etc.
  - The space in between words!
  - Numerical digits (0 1 2 3 4 5 6 7 8 9)
  - and of course letters, both UPPER and lower case
- All of the above are called Characters
- Their numerical representation is called a Character Code
- There have been many character codes created over the years
  - EPCIDIC (developed by IBM)
  - **ASCII** (developed by ANSI)
- American National Standard Code for Information Interchange
  - **UNICODE** (developed by ISO)
- an extension of ASCII to international alphabets
- ASCII and UNICODE now almost universal

# The ASCII Character Set (Bottom Byte)

0	16	32	48	64	80	96	112
1	17	33	49	65	81	97	113
2	18	34	50	66	82	98	114
3	19	35	51	67	83	99	115
4	20	36	52	68	84	100	116
5	21	37	53	69	85	101	117
6	22	38	54	70	86	102	118
7	23	39	55	71	87	103	119
8	24	40	56	72	88	104	120
9	25	41	57	73	89	105	121
10	26	42	58	74	90	106	122
11	27	43	59	75	91	107	123
12	28	44	60	76	92	108	124
13	29	45	61	77	93	109	125
14	30	46	62	78	94	110	126
15	31	47	63	79	95	111	127

NUL	DLE	SPACE	0	@	P	'	p
SOH	DC1	!	1	A	Q	a	q
STX	DC2	"	2	B	R	b	r
ETX	DC3	#	3	C	S	c	s
EOT	DC4	\$	4	D	T	d	t
ENQ	NAK	%	5	E	U	e	u
ACK	SYN	&	6	F	V	f	v
BEL	ETB	'	7	G	W	g	w
BS	CAN	(	8	H	X	h	x
HT	EM	)	9	I	Y	i	y
LF	SUB	*	:	J	Z	j	z
VT	ESC	+	;	K	[	k	{
FF	FS	'	<	L	\	l	
CR	GS	-	=	M	]	m	}
SO	RS	.	>	N	^	n	~
SI	US	/	?	O	_	o	DEL

Control Characters

NUL	DLE	SPACE	0	@	P	'	p
SOH	DC1	!	1	A	Q	a	q
STX	DC2	"	2	B	R	b	r
ETX	DC3	#	3	C	S	c	s
EOT	DC4	\$	4	D	T	d	t
ENQ	NAK	%	5	E	U	e	u
ACK	SYN	&	6	F	V	f	v
BEL	ETB	'	7	G	W	g	w
BS	CAN	(	8	H	X	h	x
HT	EM	)	9	I	Y	i	y
LF	SUB	*	:	J	Z	j	z
VT	ESC	+	;	K	[	k	{
FF	FS	'	<	L	\	l	
CR	GS	-	=	M	]	m	}
SO	RS	.	>	N	^	n	~
SI	US	/	?	O	_	o	DEL

Alternate Characters

NUL	?	SPACE	0	@	P	'	p
?	?	!	1	A	Q	a	q
?	?	"	2	B	R	b	r
?	?	#	3	C	S	c	s
?	?	\$	4	D	T	d	t
?	?	%	5	E	U	e	u
?	?	&	6	F	V	f	v
BEL	?	'	7	G	W	g	w
BS	?	(	8	H	X	h	x
HT	?	)	9	I	Y	i	y
LF	?	*	:	J	Z	j	z
?	?	+	;	K	[	k	{
?	?	'	<	L	\	l	
CR	?	-	=	M	]	m	}
?	?	.	>	N	^	n	~
?	?	/	?	O	_	o	DEL

- ASCII character ‘numbers’ are stored in a data type called char
  - char is an integer data type the same size as int
- Variables of type char can be treated like an integer
  - They can be added, subtracted, multiplied, etc.
  - The only difference is in the behaviour of `System.println()`
- Let x be of type int, and c be of type char
- `println(x)` takes x and converts the number into a sequence of ASCII values that represents the number
  - 239 is converted to 2 3 9 and then to 50 51 56 and those numbers are sent to the output
  - 52.8 is converted to 5 2 . 8 which is 53 50 46 56 and those numbers are sent to the output
- `println(c)` takes c and just prints its number as is
  - 66, which represent B, is sent to the output as 66
- DOS or TextPad or Notepad assumes the numbers given to it are ASCII and displays them as such (they always do that)

Character Literals

- Just as int has corresponding literals, so too does char
- Character literals is the character between single quotes
  - 'A', '6', '?', '&', 'c' are character literals
  - 65, 54, 3, 38, 99 are really what they are
- You can do arithmetic using character literals
  - They get converted to type int so you have to cast to (char)
  - 'A' + 5 produces (int) 70 not (char) 70
  - (char) ('A' + 5) produces (char) 70, i.e. 'F'
- See *ASCII\_As\_Numbers.java*

Converting ASCII Digits to Numbers

<u>Character</u>	<u>ASCII</u>	
'0'	48	<i>Just subtract the character '0' from the character digit to get the actual integer number.</i>
'1'	49	
'2'	50	<i>See <b>DigitToNumber.java</b></i>
'3'	51	
'4'	52	
'5'	53	
'6'	54	
'7'	55	
'8'	56	
'9'	57	

## The Character Class

- The class Character is just like the classes Integer and Double...

Character method	Responsibility
<code>isLetter(c)</code>	Returns true if <code>c</code> is a letter (upper or lower case)
<code>isDigit(c)</code>	Returns true if <code>c</code> is a digit
<code>isLetterOrDigit(c)</code>	Returns true if <code>c</code> is a letter or digit
<code>isLowerCase(c)</code>	Returns true if <code>c</code> is a lower-case letter
<code>isWhitespace(c)</code>	Returns true if <code>c</code> is a whitespace character
<code>isUpperCase(c)</code>	Returns true if <code>c</code> is an upper-case letter
<code>toLowerCase(c)</code>	Returns the lower-case representation of <code>c</code>
<code>toUpperCase(c)</code>	Returns the upper-case representation of <code>c</code>
<code>digit(c,10)</code>	Returns the numerical value of the digit (-1 if not a digit)

- See `CharacterClassExample.java`

---

## **Lecture 21 – Simple Keyboard Input**

**24 October 2003**

- Java has various methods for getting characters from the keyboard
- Simplest type takes in a single character at a time: `System.in.read()` ;
- Looks similar to `System.out.print()` ;

### Similarities:

- Both deal with characters
- Both are a part of the System class

### Differences:

- `print()` prints multiple characters
- `read()` reads only 1 character
- `print()` converts numbers to strings;
- `read()` reads only 1 character (no conversion done)
- `print()` takes `\n` and prints out CR LF
- i.e. (char) 13 (char) 10
- `read()` reads only 1 character
- The first Execution gets the CR, the next gets the LF

### Details:

- receive int ASCII values of individual characters
  - must cast to char (char)
- receive only one int per execution of `System.in.read()` ;

- no characters read until 'Enter Key' pressed
  - 'Enter Key' adds all characters on the line onto the 'input buffer' all at once
  - This includes the CR LF characters
  - `System.in.read()` looks at each character in the buffer one at a time
- When `System.in.read()` is executed, the system will not continue the program until the 'Enter Key' is pressed. See *ReadExample.java*

```
import java.io.*;

public class ReadExample
{
    public static void main (String[] args) throws IOException
    {
        char c = ' ';
        final char LF = (char) 10;
        final char CR = (char) 13;

        for (int i = 0; i < 3; i++)
        {
            System.out.print("Type in line: ");
            while((c = (char)System.in.read()) != LF)
            {
                if (c != CR)
                    System.out.print(c);
            }

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

## Lecture 22 – Arrays and Array Variables

27 October 2003

- Do not confuse the array that holds the values with the variable that holds the array

```
int[] arrayVar = new int[6]
    array variable      the array being stored
```

```
int[] arrayVar =
```

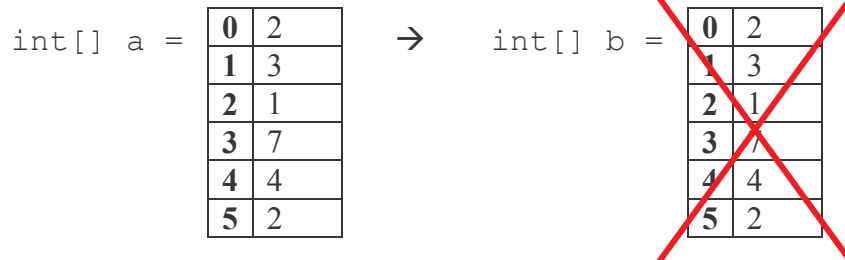
0	
1	
2	
3	
4	
5	

- **new** creates an array (*independent of the array variable*) and the array variable points to the array.

### Shallow Copy:

- Setting an array variable to another array variable does **not** create a new copy of the array!

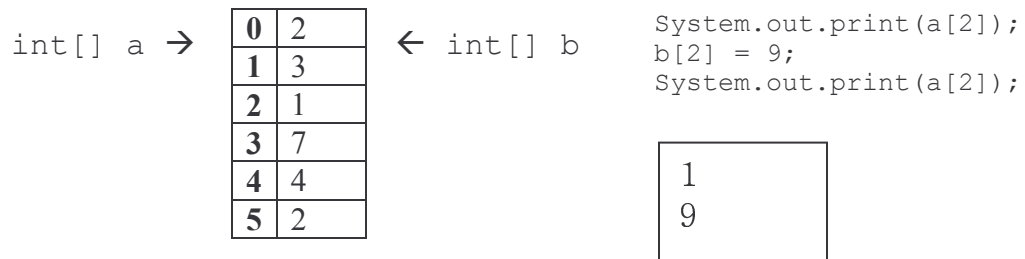
```
int[] a = {2, 3, 1, 7, 4, 2};  
int[] b = a;
```



- Instead it points to what the original variable points to i.e. the same array

`int[] b = a`

- This means that any change in **b** will be 'seen' by **a** because it is a change to the same underlying array!



### Deep Copy:

- Question: How do you create a brand new array with the same values as an original array?
- Answer: Create an array of the same size, and copy **all** the elements!

```
int[] a = {10, 20, 30, 40, 30, 20, 10};
int[] b;
b = new int[a.length];
for (int i = 0; i < a.length; i++)
    b[i] = a[i];
```

### Runtime Determined Array Length:

- An array can be created at runtime with a computed or user defined size
  - just use a variable in the [] in the new statement
  - the variable can hold any value and so the array can be any size

```
int size = CIS1650Input.readInt();
int[]myarray = new int[size];
```

---

## Lecture 23 – Short Hand Code and Methods

29 October 2003

### Code 'Short Hand'

- As the code gets larger and more complex, it becomes difficult to understand & modify
  - There are many times when the same code is needed in different parts of the program
  - This 'repeated' code usually can be thought of as a single 'function' or 'method'
  - When a large program is broken down into meaningful building blocks, the code becomes much more easily understood.

**Before:**

```
action1;
action2;
:
while(...)
{
    action1; //again
    action2; //again
    action3;
    action4;
    action5;
}
:
action4; //again
action5; //again
```

```
void doA()
{
    action1;
    action2;
}
```

**After:**

```
doA()
:
while(...)
{
    doA() //again
    action3;
    action4;
    action5;
}
:
action4; //again
action5; //again
```

```
void doA()
{
    action1;
    action2;
}
```

### Method Definition:

- always within a class
- provides a name for body of code
- may provide parameters to accept values from caller
- may return a value (as result) to caller
- header or signature describes how to interface to outside (caller)

```
public static return-type name(type parm1, type parm2, ...)  
{  
    //executable code goes here  
}
```

*e.g.*

```
public static void doA()  
{  
    action1;  
    action2;  
}
```

- For now static is a necessary keyword. Later we will learn what it means and when to use it and not use it

### The Return Statement

- All methods (except of type 'void') return a value.
- You tell a method what value to return by using a return statement

```
public static double sin45()  
{  
    double PiBy4 = Math.PI / 4.0;  
    return Math.sin(PiBy4);  
}
```

### Return Types

- The return type must match the type of the value returned
- The return type 'void' means that the method does not return a value
- `Math.sqrt(25.0)` returns the value of 5.0
  - It's return type is 'double'
- `System.out.print("Hi")` does not return a value
  - It's return type is 'void'

### Parameters

- Parameters are variables that take in a value (called an argument)
  - Value (argument) is supplied when the method is called
  - The parameter variable is then used just as any other local variable in the method

- You can read it or modify it as often as possible
  - You do not initialize it; that is done when the function is called.

```
public class ParmExample
{
    public static void main (String[] args)
    {
        int x = 5;
        System.out.println(triple(x));
        System.out.println(triple(triple(x)));
    }

    public static int triple(int originalValue)
    {
        int newValue = 3 * originalValue;
        return newValue;
    }
}
```

		value returned?	
		no	yes
parameters?	no	System.in.println();	x = Math.random();
	yes	System.in.print("hi");	if(Math.power(5,2) > 1)
		stand-alone statements	used in context

- All methods must belong to a class
- Usually, you place the class name before the method
  - a dot separates them
  - This tells Java which class holds the method.

```
Math.sqrt()
Math.sin()
Character.toLowerCase()
```
- However if the method is in the same class, you can skip the method name

```

public class ParmEx
{
    Public static void main (String[] args)
    {
        int x = 5;
        System.out.println(ParmEx.triple(x));
        System.out.println(triple(triple(x)));
    }

    public static int triple(int originalValue)
    {
        int newValue = 3 * originalValue;
        return newValue;
    }
}

```

### Method Call

- A method can be invoked from the main (or any other method)
  - this is known as "calling the method"
  - the method must be referred to by name
  - Each argument type in the call must match the parameter type
  - The return value must be used in the calling method as any value of that type is use

```

public static void main(String[] args)
{
    float result = mean(5.0, 10);    WORKS
    float result = mean(5, 10);     DOESN'T WORK
}

public static float mean(double value1, int value2)
{
    float average = (float) ((value1 + value2) / 2.0);
    return average;
}

```

## Lecture 24 – Method Calls and Parameter Passing

31 October 2003

### Reason “5” doesn’t work:

- The method is determined by the type of its parameters
- aMethod(int, int) is not the same as aMethod(double, int) or aMethod(double, double)
- The types of the parameters is called the method's 'signature'
- if you call a method using the wrong signature, Java will say that it can't find that method (even though the method name matches)

- This is what happened in the previous example:
  - mean(5, 10) has a signature mean(int, int)
  - The actual method has the signature mean(double, int)
  - The signatures don't match, so no method can be found & Java complains

### Method Call

- All arguments are evaluated before the method is invoked and the values are passed to the parameters.

```

public static void main(String[] args)
{
    int b = 4;
    int a = YbyXsqrd( 2 , 8 );
}

public static int YbyXsqrd(int x, int y)
{
    return 32 ;
}

```

### Scoping

- The variables in one method are not 'seen' in another method, even if they have the same name.

```

public static void main(String[] args)
{
    int b = 4;
    int a = YbyXsqrd(b - 2, b * 2);
}

public static int YbyXsqrd(int x, int y)
{
    return y * x * b;      DOESN'T WORK
    return y * x * x;     WORKS
}

```

### Pass by Value

- That 'x' remained unaffected is because Java uses pass by value
- The parameter 'x' just gets a copy of the value calculated within the arguments position
  - even if the argument is just a variable, it is evaluated (accessing the value), and it is the value that is returned
  - all variables (whether used as arguments or not) remain unchanged in the main method (or the calling method)

*See PassByValue.java*

## Method Overloading

- The same method with different signatures can be written and used within the same class.
- When the method is invoked in the main (or some other method), the signature is checked and the appropriate method is found & used

*See **MethodOverloading.java***

## Passing Arrays as Parameters

- At one level, arrays are passed as parameters just as every other parameter is passed
- However the array itself is not duplicated
  - What is passed is a 'pointer' to the array
  - This is also true of variables that hold arrays
- See `ArrayExample1.java`
  - All objects behave this way, as we will see later
- An array is just an 'object' (as is a `String`)
- This means that if you change the content of an array inside a method, the array outside will also be changed
  - This is because it is the same array

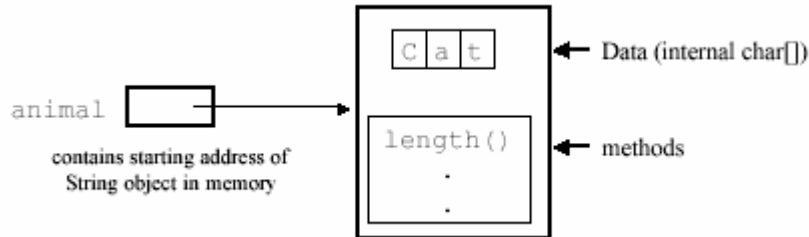
*See **ArrayAsParameter.java***

- Finally, Java is still using pass-by-value
  - Only the value being passed is the 'pointer' to the array
  - e.g. let the parameter variable has a new array assigned to it
  - then the members of the old array haven't been affected (the variable just is pointing to another array)
  - so the array in the calling method will be unchanged since the variable there is still pointing to the old array

*See **ArrayExample2.java***

- String is an object and not a primitive type such as int or double
- Objects contain both data (like arrays) and methods
  - they can only be accessed through the object
- String objects are immutable
  - once created the contents of a string cannot be changed

Creating a String



Using Methods on Strings

- Since strings are objects, we can call string operators on strings using the dot operator
  - It is the only way to use string operators on strings
- Strings are not passed to String methods as arguments
  - Unlike the Character methods, which take in the character they are acting on

```
Character.toUpperCase('h')    'H'
String.toUpperCase("hello")  Compiler Error
"hello".toUpperCase()       "HELLO"
```

```
String animal = "Cat"
"Cat".length    Compiler Error (Unlike arrays, length() is a
method)
"Cat".length()    3
animal.length()   3
```

Creating a New String from a char[]

- "" delimit string literals and is really a shorthand for
 

```
char[] cArray = {'a', 'b', 'c'};
String s = new String(cArray);
```
- The above is identical to "abc"
- Notice the **new** operator
  - It is the same as its use for arrays
  - It means 'create a new object of type String'
- "" delimit string literals and is really a shorthand for
 

```
char[] cArray = {'a', 'b', 'c'};
String s = new String(cArray);
```

- The argument beside the object type holds initialization values
  - In this case a character array (cArray)
  - If there are no initialization arguments, empty parentheses are used  
e.g. `new String()` is the same as `""`

### The Null String

- All object variables can store no object at all
  - or rather it can store a value (pointer) that represents no object at all
  - the value is called null
  - since a String is an object, String variables can hold the value null  
`String theText = null;`

### Empty String vs. Null String

- There is a difference between the empty string `""` and null
  - The empty string is a String with no letters in it  
**but it is a string**
  - null says that the String variable does not hold a string  
**i.e. it is not currently storing a string**

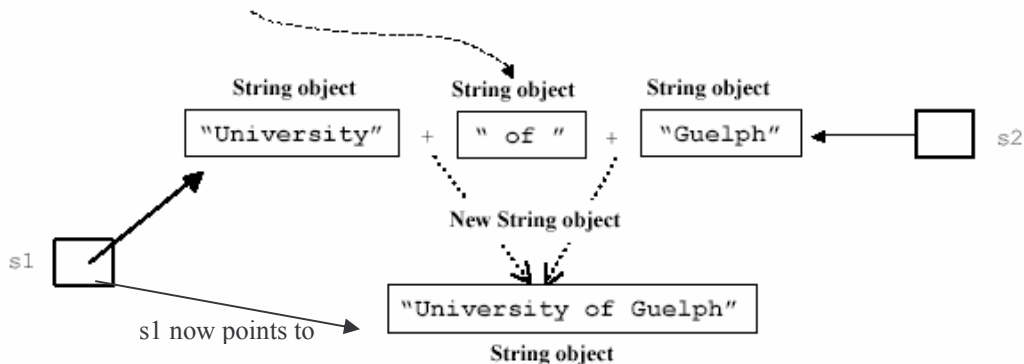
```
"" == new String() != null
```

## Lecture 26 – Joining Strings and String Characters

5 November 2003

- When joining strings a new String object is formed

```
String s1 = "University";
String s2 = "Guelph"
s1 = s1 + " of " + s2;
```



```
String s1,s2,s3,s4;
s1 = "Alfredo";
s2 = "Al";
s3 = s2 + "fredo";
s4 = s1;
```



```
true      s1 == s4
false     s1 == s3
true      s1.equals(s3)
true      s3.equals(s1)
true      s1.equals("Alfredo")
true/false s2 == "Al"
true      s2.equals("Al")
```

*See StringEquality.java*

### Accessing String Characters

- Use the charAt(int position) method
- Returns the character at the given position in the string
  - returns type char
- Like arrays, uses 0 based counting
  - first element in the string is a position 0
- Remember spaces & tabs are characters!

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
M i s s i s s i p p i   R i v e r
length() => 20
```

```
String name = " Mississippi River ";
name.charAt(8)           => i
name.charAt(21)         => exception
name.startsWith(" M")  => True
name.endsWith("r ")    => True
name.replace('s','z')  => Mizzissippi River
name.toLowerCase()     => mississippi river
name.toUpperCase()     => MISSISSIPPI RIVER
name.trim()            => Mississippi River
```

*See StringDeconstruction.java*

- Sometimes we do not know where a character occurs in a string
  - so we can't use charAt() directly
  - we could look at all characters from the beginning until we find the one we are looking for
  - good approach, so String method already exists indexOf(char)

Searching for Characters & Substrings

- The method `indexOf(char)` returns the first location of the character in the string
  - if the character doesn't exist, it returns -1
- Similarly `indexOf(String)` will return the location of the first letter of the target string in the source string
  - usage → `source.indexOf(target)`
  - the target string if it exists in the source string is called a substring (part of the string)
- Finally, a 2nd argument can be added that hold the position in the string from which you want to start (default is 0)
  - usage → `object.indexOf(target, startPosition)`

```

0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7
T o   B e   O r   N o t   T o   B e   .length() => 18

```

```

String text = "To Be Or Not To Be";

text.indexOf('r')           => 7
text.indexOf('q')           => -1
text.indexOf('o', 7)        => 10
text.indexOf("Be")          => 3
text.indexOf("To", 2)       => 13
text.lastIndexOf('e')       => 17
text.lastIndexOf('e', 16)   => 4
text.lastIndexOf("To")     => 13
text.lastIndexOf("To", 12) => 0

```

Dot Operators

- A dot operator
  - First evaluates its left operand (usually a variable) to obtain an object
  - Then finds the method in the object sent by its right operand and has the object run it (using the arguments given)
  - The result of the method is then returned

Concatenating Methods

- You can use the dot operator to successively apply methods to the returned value of a previous dot operator,
  - The return value of the first dot operator must be an object
    - " Mississippi River ".length() → 20
    - "Mississippi River".length() → 17

- The return object of a dot operator does not have to be the same object

```
"Hello".length()           → 5
{'H', 'e', 'l', 'l', 'o'}.length() → 5
```

### Extracting Substrings

- First find the beginning of the substring using
 

```
beginPos = text.indexOf(beginningChar)
```
- Then find the position one character past the end of the substring using
 

```
endPos = text.indexOf(afterEndChar, beginPos)
      or
      endPos = text.indexOf(endChar, beginPos) + 1
```

  - You need to start from `beginPos` because otherwise you may find an earlier occurrence of the end character
- Use the `substring(beginPos, endPos)`
  - where `beginPos` is the index of the beginning character in the original string
  - `endPos` is the index just after the end character in the original string
 

```
resultString = text.substring(beginPos, endPos)
```

*See [OneWordAtATime.java](#)*

## Lecture 28 – Objects

10 Nov 2003

- Methods and information are grouped together into 'objects'
- The 'object' contains information that has behaviour
  - The information is stored inside the object
    - just like the characters of a string are stored inside the string
    - This information is called the object's 'state'
  - The information cannot be accessed directly; only through the object's methods
    - These methods are the object's 'behaviour'
    - some methods only pass the information (possibly modified) from the object to the user
    - others only update the information
    - some methods do both

### Class & Instances

- The information held by the object is the object's 'state'
- You can have many different object that have the same type of state & the same behaviour
  - it is said that all of those objects belong to the same class

- the individual object of a given class (which hold the actual data) is called an instance of the class
- The Car class might be a complete description of a car, including its behaviour (driving, breaking, etc.)
- The actual car in your driveway is an instance of a car
- You may have a second car, this is another instance of the same Car class
  - the term 'instance' is usually used synonymously with the word 'object' (not to be confused with the class Object)

### Class & Type

- A given class, i.e. the generalization of an object is considered by Java to be a data type, like 'int' or 'double'
- By convention, all class names start with an uppercase letter
- Class types are used in the same way as primitive data types
  - they are used to declare variables
  - they are used to define parameters in methods (& help make up the method's signature)

*eg: a user defined method's signature that will compares a date with the current calendar:*

```
myString; String
Calendar myCalendar;
isItToday(Date theDate, Calendar current)
```

### Creating Instances from a Class

- Instances are created from a class by using a Class constructor together with the new operator
  - you are asking the class to create a new instance of that type
  - the constructor consists of the class name (same as the type) followed by arguments inside parentheses
- the arguments are used to 'initialize' the newly created instances i.e give it its initial value (state).

```
myString = new String({'n', 'e', 'w'});
myCalendar = new Calendar(2002, 3, 27);
isItToday(new Date("2002/4/17"), myCalendar);
```

*note:* while Date is a real class in the API, it is used here with a fictitious constructor

### Getting State Info from an Object

- The dot operator can often get state information from an object
- Each piece of info stored in an object is given a name (& a type)
  - the name is called an 'instance variable'
  - you can often get that info directly from the object
- use the instance variable name through the dot operator
- you do not use the () after the variable name

```
char[] chars = {'n', 'e', 'w'};
e.g. System.out.print(chars.length)
```

### Instance Variables & the API Library

- You can only get the information if the instance variable is 'public'
  - it may be declared 'private'
  - the API documentation will tell you whether the variable is accessible or not
- If the instance variable is public, you can use it as a normal variable
  - i.e you can get its value or set it to a value through the = operator
- In the API library, you usually do not have direct access to variable
  - the only cases where you do, the variable has been declared 'final', so you can't change it

### Global Constants – Class Variables

- There are many classes that have constants you can use
- These constants are available from the class itself
  - all instances of the class know these constants
  - you can access them from any instance of that class
  - you can also access them directly from the class itself by using the dot operator on the class name instead of on an instance
- this is why they are called class variables
  - Math.PI
  - Calendar.JANUARY
- you could have used `myCalendar.JANUARY` but by asking the Calendar class, it shows that you are using a class variable.

### Class Variables and the API

- The static keyword: In Java, class variables are denoted using the word 'static'.
- When the word static is used, it tells Java that the variable can be accessed without the creation of an instance
  - this is why we used the word static before all the global constants we have created
- we so far have used the classes we wrote without creating any instances of it
  - consequently, the constants we use must be available even when there are no instances – hence the use of static
- in a class description in the API documentation, you can tell the global constants because they are the variables that have been declared 'static'

---

## **Lecture 29 – Using Methods and Classes**

**12 Nov 2003**

### Command-Line Arguments

```
sort [file [output]]
copy file1 dir\file2
```

```
java BC.java
spell msg | java CS
```

```

public static void Main(String[] args)
{
    for(int i=0; i < args.length; i++)
        System.out.println(args[i]);
}

```

- Use quotes to prevent Java from separating space delimited objects.
- It is common for utilities to allow you to mix and match command line arguments:  
`java sort -i input -o output`

### Using Methods

- Methods are invoked using the dot operator
- You do not need to pass any information already contained in the object
  - the object already knows about the info it contains
  - extra information needed by the method is passed through the method's arguments
  - we have already seen this at length with the String methods

### Class, or Static Methods

- You can have 'class methods' as well
  - class methods are to class variables as 'instance' or regular methods are to instance variables
  - i.e. instead of calculating values based on state information, they calculate values based on global class information
  - we have seen many class methods (denoted 'static' in the API documentation) in the Math class
    - e.g. `Math.sqrt()`, `Math.sin()`, etc.
  - this is why we used the word static before all methods of all the classes we have written
    - we so far have used the classes we wrote without creating any instances of it
    - consequently, the methods we write must be available even when there are no instances
    - hence the use of static

### System.out.print()

- `System.out` is a static variable
  - it returns an instance of the class `PrintStream`
- The `print()` method \ belongs to the class `PrintStream`
  - since it is accessed from a `PrintStream` instance, it is an instance method (not a class method such as `Math.pow()`).

### *Example: The GregorianCalendar Class*

- The `GregorianCalendar` class creates a calendar object
  - this holds a date and optionally a time & location (i.e 5:10pm EST)

- You can enter the date of interest using the `GregorianCalendar` constructor
  - eg. to enter the date 'November 22, 1963' use
 

```
Calendar jfk = new GregorianCalendar(1963, Calendar.NOVEMBER, 22)
```
  - to use the current date you need to use another class as a helper
 

```
TimeZone stz = SimpleTimeZone.getDefault(); // gets current time&date
Calendar today = new GregorianCalendar(stz);
```

#### *Getting Info from a Calendar object*

- use the `get()` method
  - the `get` method takes an argument
- a single integer that tells the method what type of info we want
- those integers can be obtained from the `Calendar` class
  - e.g. `Calendar.YEAR`, `Calendar.MONTH`, `Calendar.DAY_OF_MONTH`
- `get()` then returns the info in the form of an integer

```
Calendar cldr = new GregorianCalendar(2001, Calendar.MAY, 5);
System.out.println(cldr .get (Calendar.DAY_OF_WEEK));
System.out.println(cldr .get (Calendar.MONTH));
System.out.println(cldr .get (Calendar.DAY_OF_MONTH));
System.out.println(cldr .get (Calendar.YEAR));
```

#### *Adding Days & Rolling Days onto a Calendar Date*

- `calendar.add(5, Calendar.MONTH)`
  - adds 5 months onto the date
  - if we are in October, 5 months later will take us to March of the next year
  - we can get information about the new date using the `get()` method
- `calendar.roll(5, Calendar.MONTH)`
  - adds 5 months onto the date
  - if we are in October, 5 months later will take us to March of the same year

## **Lecture 30 – Integrating Classes**

**14 Nov 2003**

```
public class TestProggy
{
    public static void main{}
    {
        Deck mydeck = new Deck(1);
        Card mycard = new Card;
        Card a = new Card (Card.CLUB, s)
        mycard.setSuit (Card.HEART);
        mycard.setVal (3);
        mycard.getSuit = mycard.CLUB;
    }
}
```

```

public class Card
//this class is not a complete program, therefore it does not have a
main method.
{
    private char suit;
    private int val;
    public static final char CLUB = 'c';
    public static final char DIAMOND='d';
    public static final char HEART = 'h';
    public static final char SPADE = 's';

    public suit getSuit()
    {
        return Suit;
    }

    public val getVal()
    {
        return Val;
    }

    public boolean setSuit(char ch)
    {
        if (ch == 'd' || (ch =='h') || (ch =='s') || (ch =='c'))
        {
            suit = ch;
            return true;
        }
        return false;
    }

    public boolean setVal(int v)
    {
        if (v > 0) && (v < 14)
        {
            val = v;
            return true;
        }
        return false;
    }

    public Card(char s; int v)
    //This is the Constructor, always matches the class name. This is
    called when you make a new Card.
    {
        setSuit(s);
        setVal(v);
        //Instead of doing that, you could have randomized them.
        return;
    }
}

```

```
public class Deck
//again, no main here either. More helper code.
{
    private Card[] dc;
    private int top;

    public Shuffle()
    {
        //Simplest thing to do here would be to take one card, replace
it        with another card. Repeat 52 squared times.
        int p1 = _____ * deck.length;
        int p2 = _____ * deck.length;
    }
}
```

- One class can actually be based on another class
  - It is actually the same class, but modified
- methods can be added
- instance variables can be added
- behaviors can be changed slightly
  - This modified class is given a different name, but is said to 'inherit' from the original class
- the class that inherits is called a subclass
- the original class is called the superclass
  - Variables can use the original class's name in their declaration
  
- Many variables and methods that can be or even must be used with a class may not be written about in that class
  - they are to be found in the the class's superclass
- The API documentation always tells you what methods and variables are inherited
- It also tells you the class that is being inherited from
  - Java uses the term 'extends' to mean 'is a subclass of'
  - e.g. `GregorianCalendar` extends `Calendar`
- much of the information is in `Calendar`
  - e.g. all of the constants such as `Calendar.MAY` or `Calendar.MONTH`

### The Import Statement

- Many classes are stored together in the same directory
- Java only gives you access to the classes in the directory `java.lang`
- To get access to other classes you have to tell Java where they are, and which classes you want
  - if you want a single class write: `import directory_location.ClassName;`
  - if you want all classes from a directory write: `import directory_location.*;`
  - the import statement must be written at the top of the file, outside of the class name

- The `Calendar` and `GregorianCalendar` classes are in the directory

`java.util`, we therefore have to add the statements:

```
import java.util.Calendar;
import java.util.*;
```

### The Full GregorianCalendar Example

- See `GregorianCalendarExample.java`

### Example: Reading Streams

- Input and output are similar to strings:
  - they are composed of characters
- There is 1 big difference
  - a string has a fixed length
  - but you can write to the screen forever

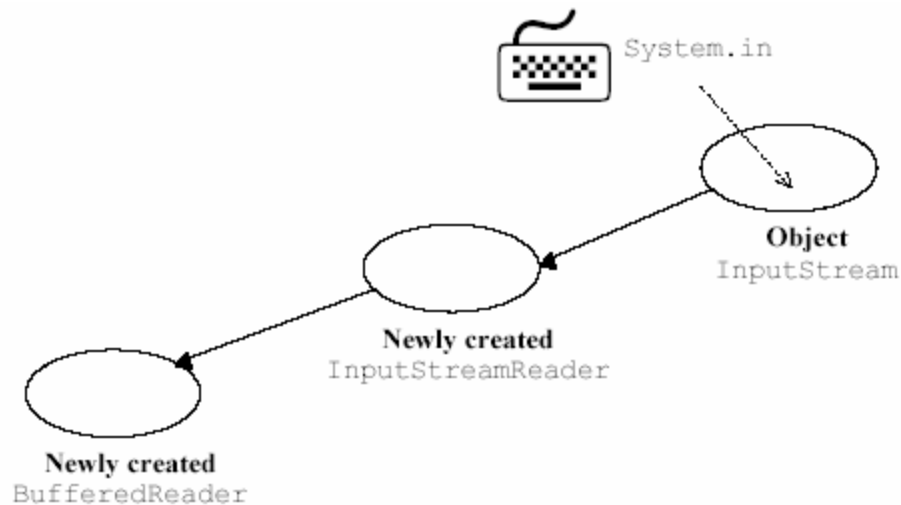
- and you can read from a keyboard forever
- Java calls a collection of characters that does not end a stream
  - Analogy: input is like a stream of characters flowing from the user ...
  - System.in is an example of an InputStream
  - System.out is an example of a PrintStream

### Readers

- Streams have to have 'readers' to look at & interpret the characters that flow through them
- For InputStreams, the reader is class InputStreamReader
  - To know which stream is being read, the InputStreamReader's constructor takes in an input stream (which for us is System.in) `InputStreamReader inStream = new InputStreamReader(System.in);`
- InputStreamReader is in directory java.io
  - Therefore you need to include `import java.io.*;` at the top of the file

### Buffered Readers

- To use readers effectively we must add a buffer to it
- The BufferedReader class takes a Reader class object during its construction,
  - it adds a buffer to it
  - it also adds a useful method `readLine()` which 'chunks' the stream into 'lines'
- `readLine()` reads the stream until it sees a CR/LF
- it then returns what it has seen so far as a string
- it does not include the CR/LF in the string
- you can then parse the string for words and numbers



```

BufferedReader keyboard = new BufferedReader (
    new InputStreamReader(System.in));

String line = keyboard.readLine(); // read a line of text
  
```

Wrapper Classes

- Since primitive data types are not objects Java provides 'object versions' of them
  - These are called Wrapper Classes
  - e.g. Integer class, Double class, Character class, etc.
- Wrapper classes also provide various static methods of use
  - we have already seen many of them
  - e.g. Character.isDigit(), Integer.parseInt(), etc.
- If you want to store a number or character in an object that takes as input only objects, you would transform the primitive data type into an object using the Wrapper Class
  - e.g. the Vector class (similar to arrays, but more powerful) only stores objects
- to add a the number 5.2 to the vector use: `vector.add(new Double(5.2))`

The toString() Method

- All classes have `toString()`
- `toString()` is called on an object when an object is printed, or when it is added to a string "The object is " + object is the same as "The object is" + `object.toString()`  
*See ToStringExample.java*

Exception Handling

- Errors can occur during the running of the program
  - They are not necessarily bugs; they can be user errors
- You can try to implement error checking routines directly in your code
  - This interferes with the flow of the algorithm
  - It can become difficult to see what is happening in the code amidst all of the error checking
- Java therefore provides you with a mechanism that can gather your error checking / correcting together
  - Called Exception Handling
- If something goes wrong during the running of a method, the method can Throw an Exception
- This stops the processing of the method & Java transfers control to the error handling portion
  - If there is no error handling portion, then depending on the error, Java either
- won't compile the code
- or during runtime will stop the program and give an error msg

Try - Catch

- You provide error checking routine through the use of try-catch blocks
  - Your code goes into the 'try' part
  - If an error is detected, Java will see if there is a catchblock for the error below the try-block
- Each type of error is given its own name – called Exceptions

- E.g IOException, try

```
try
{
    // can produce an IOException
    ch = System.in.read()

    // can produce a NumberFormatException
    num = Integer.parseInt(arg[0])
}
catch (IOException e)
{
    // do something about the bad read
}
catch (NumberFormatException e)
{
    // do something about the bad formatting
}
```

- If no Exceptions occur during the running of the try-block
  - all catch-blocks are skipped and the method continues with any code that follows the last catch-block
- If an Exception occurs
  - Java finds the catch clause with the same name of the Exception thrown
  - The catch clause is run
  - Control is then given to the line following the final catch-block
- the program does not return to the try-block
  - If no catch clause can be found with the same name, the method itself throws an exception Try - Catch
- If no Exceptions occur during the running of the try-block
  - all catch-blocks are skipped and the method continues with any code that follows the last catch-block
- If an Exception occurs
  - If no catch clause can be found with the same name, the method itself throws an exception
- the method is stopped where the exception is thrown
- the method returns, but not with a value but with the exception
- the calling method now must deal with the exception
- this continues until it is the main method that throws the exception
- when this happens the program stops, and an error message is printed out

### Exceptions and Classes

- The exception thrown is an instance of class Exception
  - e.g. IOException is a subclass (of a subclass) of class Exception
- Two useful methods of class Exception are:
  - e.getMessage()
- returns as a String the error message stored in the exception e
  - e.printStackTrace()

- returns the method in which the exception was thrown, as well as the method that called that method, as well as the method that called that method, etc.

```
try
{
    // can produce an IOException
    ch = System.in.read()

    // can produce a NumberFormatException
    num = Integer.parseInt(arg[0])
}
catch (IOException e)           parameters of the catch clause
{
    // do something about the bad read
}
catch (NumberFormatException e)
{
    // do something about the bad formatting
}
```

```
try
{
    // can produce an IOException
    ch = System.in.read()

    // can produce a NumberFormatException
    num = Integer.parseInt(arg[0])
}
catch (IOException e)           Type definitions of the parameters
{
    // do something about the bad read
}
catch (NumberFormatException e)
{
    // do something about the bad formatting
}
```

*See Calculator.java*

## Lecture 33 – More on Exceptions

21 Nov 2003

### Throw

- you can throw your own exceptions
- Syntax: `throw exceptionObject;` commonly 'protected' by a condition that prompts the exception

```
if(exception condition occurs)
    throw exceptionObject;
```

- The exception object can be the one in the catchclause's parameter
- 

## Lecture 34 – Objects, Classes and Methods Summary

24 Nov 2003

### User Defined Objects

- Instead of just using the objects provided in the API library, it would be nice to make objects of our own design
  - can store values of differing types to pass back from methods
  - forms a nice package for our code so others can use it
- yet allows us to keep control of the internal workings of the code we wrote
  - good for debugging
- if a bug occurs, we look at what 'object' caused the problem, & we know where in the code the problem must have occurred
  - allows for natural design of large scale problems
- easy to expand
- easy to change

### Creating a User Defined Class

- We have already been doing so
  - each Java program is a class
  - that is why we write: `public class ProgramName`
  - a class that can be used by another program is simply written without a `main()` method

### How do we write the methods?

- Again we have already been doing it!
  - The methods that can be accessed from an object is the method of the same name written in the object's class

```

public class ClassName
{
    public int methodName()
    {
        // does something
        return answer
    }
}

public class ProgramName
{
    public static void main(String[] args)
    {
        int value;
        ClassName cn = new ClassName();
        value = cn.methodName();
    }
}

```

### Class methods vs. Instance Methods

- Notice that we did not use the term 'static in front of the method 'methodName()' '
- The method is accessed from an actual object, not from the class itself
  - It is an instance method
  - It is not a class method i.e. a static method
  - Consequently we do not use the term static in front

### How do we store information?

- Again we have already been doing it!
  - When we created global constants, these are the 'class variables'
- These variables can be access from anywhere inside the class
  - They are global to the class
- They can also be accessed from other methods
  - use the dot operator on an object of the class and call for the variable by name

```

public class ClassName
{
    int myInt = defaultValue;
    public int methodName()
    {
    }
}

```

```

public class ProgramName
{
    public static void main(String[] args)
    {
        int value;
        ClassName cn = new ClassName();
        value = cn.myInt;
    }
}

```

- Notice that we did not use the term 'static' in front of 'global' variable
- The variable is accessed from an actual object, not from the class itself
  - It is an instance variable
  - It is not a class method i.e. a static method
  - Consequently we do not use the term static in front
- Important Difference!!
  - Instance variables will hold different information for different instances of the object.

```

try
{
    line = kb.readLine().trim();
    for(i = 0; !line.equals("quit"); i++)
    {
        value = Integer.parseInt(line);
        store[i] = value;
        line = kb.readLine().trim();
    }
}
catch (NumberFormatException e)
{
    System.out.println("Problem in line " + i);
    throw e;
}

```

- or you can create a new object from an exception class and throw that
  - you can place a message in the new instance using the constructor argument

```

for(i = 0; (ch = (char)System.in.read()) != CR; i++)
{
    if(Character.isLetter(ch) || Character.isWhitespace(ch))
        lineArray[i] = ch;
    else
        throw new IOException("No digits or punctuation allowed");
}

return new String(lineArray);

```

- You can even create your own Exception classes to throw and catch
  - this is done by extending the Exception class (creating a subclass)
  - this will not be covered in the course

## Checked / Unchecked Exceptions

- There are two types of Exceptions
  - checked exceptions
- must be caught immediately or the code will not compile
  - you must handle these exceptions in a catch clause
  - or explicitly re-throw them
- direct subclass of type Exception
  - unchecked exceptions
- the compiler allows these to be automatically re-thrown if not caught
  - i.e they are not check for during compile time
- direct subclass of type RuntimeException
- if thrown during runtime & not caught
  - cascades up the call stack until it is caught or it reaches the main and stops the program & prints out the Exception message & call stack

## Checked Exceptions & Throws Clause

- if a checked exception is thrown (by a method or directly)
  - that method must be placed in a try-block
  - the try-block must have a catch clause that corresponds to the checked exception's type
  - the try-block can be omitted if the method that the code is in is declared to re-throw that Exception

```
public static void fromUser(String[] line, BufferedReader kb, int i)
    throws IOException, NullPointerException
{
    if(line[i].equals("get line from user"));
    {
        line[i] = kb.readLine().trim()
    }
}
```

is the same as

```
public static void fromUser(String[] line, BufferedReader kb, int i)
{
    try
    {
        if(line[i].equals("get line from user"));
        {
            line[i] = kb.readLine().trim()
        }
    }
    catch (IOException e) throw e;
    catch (NullPointerException e) throw e;
}
```

## Exception Examples

- Most Exceptions encountered are unchecked
  - NumberFormatException
  - IndexOutOfBoundsException
  - NullPointerException
- The main exception to the above is
  - IOException