COMP-202
Unit 2: Java Basics

CONTENTS:

Assignment 1

- Assignment 1 posted on WebCt and course website. It is due May 18thst at 23:30

- Worth 6%
-
- Part programming, part binary/number conversions
- Start early!

Question:

- What if I have 2 variables x and y and I want to swap the contents of them? In other words, I want to write something so that afterwards x has the old value of y and y has the old value of x.

Answer 1

- The simplest way to do this is by making a third variable. We could call the variable temp

For example to swap x and y:

```
int temp = x;
x = y;
y = temp;
//make sure the “right” of one line matches
//with the “left” of next line
```

Answer 2

- What if I wanted to swap *without* using a 3rd variable?

```
x = x+y;
y = x – y;
x = x - y;
```

Answer 2

- What if I wanted to swap *without* using a 3rd variable?

Suppose x starts out with value A and y has value B

x = x+y;

x now equals A+B
y now equals B

Answer 2

- What if I wanted to swap *without* using a 3rd variable?

Suppose x starts out with value A and y has value B

```
y = x-y;
```

x now equals A+B
y now equals A

Answer 2

What if I wanted to swap *without* using a 3rd variable?

Suppose x starts out with original value A and y has value B

```
x = x-y;
```

x now equals B
y now equals A

Base- 13 Math

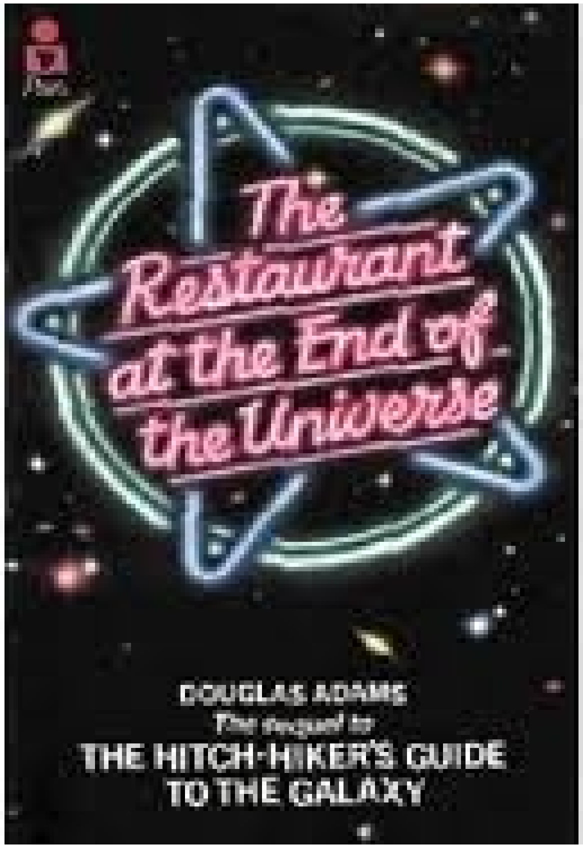
In base 13, what is 9*6 ?

To do this, we first calculate 9*6 in base-10

9*6 = 54

Now, remember in base-13, we can count up to “12” in one digit.

So each digit is worth 13^0, 13^1, etc.


The Restaurant at the End of the Universe
DOUGLAS ADAMS
The sequel to
THE HITCH-HIKER'S GUIDE TO THE GALAXY

Meaning of Life

42

= 4 *13^1 + 2 * 13^0

Solution: Store the results into a variable

If we have a case like this, we can use a variable to store the results of a computation.

To make a variable, you have to do 2 things:

1)Decide what kind of thing or type you want to store.

-If you want to store an integer, this is called int in Java

-If you want to store a number with a fractional part, this is normally called double in Java (also could be float)

-If you want to store letters, this is usually done with something called a String in Java.

Solution: Store the results into a variable

2)Decide on a name for your variable.

Your variable can be named anything you like with a few exceptions:

1)It can only contain letters, numbers, and _ (no ; for example)
2)It must start with a letter
3)There must not be another variable with the same name in scope
4)Variable names are case sensitive so Foo is different than foo
5)There are a few words in Java that are reserved. You can't call your variables these (for example “public”)

Solution: Store the results into a variable

Once you decide on a name and type, you can do what is known as declaring a variable by writing first the type and then the name and then a ;

For example:

```
int mean;
```

would declare a variable which will store an integer. It will be called mean in further computations.

At the beginning the variable mean has no value and is called uninitialized

Solution: Store the results into a variable

To store a value into a variable, you write:

variablename = *expression*

What this means is "assign the value of the variable called variablename to be the value of *expression*"

The equals in Java is very different from the = in math.

1) It is not symmetric. a = b is not the same as b = a
2) It is a one time assignment. All that Java does is evaluate the expression and assign its value.
3) The types on the left and right side of the equal have to be the same. For example, you can't store letters into a number.

Solution: Store the results into a variable

Once you have a variable initialized, you can use it in any other computation:

```
int mean;

mean = ( 1 + 2 + 3 + 4 + 5) / 5;

System.out.println(1 – mean);
System.out.println(2- mean);
......
```

What if you don't declare a variable?

If you write

```
x = 5;
```

without declaring x, you will get a compiler error.

The error will complain that the compiler does not recognize the symbol x.

Some basic types

int : stores an integer number

String : stores letters. For example “Hello World”

double : stores real numbers (fractions)

long : stores a long integer (up to 9 quintillion!)

float : like double, can store numbers

boolean : stores either true or false

char : stores one character

Mismatching types

If you try to store something of type X in something of type Y, the Java compiler will complain.

For example,

int x;
x = “Hello”
What are the types of “Hello” and x

Why does Java care anyway?

Why does the Java compiler care that you are trying to store a string in an int?

Why does Java care anyway?

Answer: The problem is when you write

```
int x;
```

Java is setting aside enough memory to store 1 integer value.

If you try to store a String in it, it doesn't know whether it will fit or not!

Why does Java care anyway?

In addition, every different type has a different conversion to binary.

For example, the code 111001111001 will store a number if it represents an int

The same code would represent some particular letter if it was a String.

Java Program: Hello World

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello World!");
    }
}
```

The first thing is that every program in java MUST be inside of a class. We'll go into more detail on what this means, but you can think of a class as grouping things together

“public class NAMEOFFILE { “ in it.
(Note: we'll see later that this isn't always on the first line though!)

Java Program: Hello World

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello World!");
    }
}
```

Inside a class, there are (usually) 1 or more methods. A method is simply a group of instructions to Java that can have both an input and an output. Conceptually, it is like a function in math.

In this case, the method is main()

Java Program: Hello World

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello World!");
    }
}
```

Every Java program you ever write has to have a main method.

Not every class that you ever write in Java will have a main method.

If you don't have a main method, you can compile your class, but you can't run it.

Java Program: Hello World

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello World!");
    }
}
```

Experiment: See what happens if you change

```
public static void main(String[] args)
```
to

```
public static void Main(String[] args)
```

Try

1)`javac HelloWorld.java`

2)`java HelloWorld`

Java Program: Hello World

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello World!");
    }
}
```

This method is called main because main is written before (String[] args).

We often will call this “the main method”

Multiple methods in a class

A class can consist of many methods. Here is a class with many methods. They are called elmer, bugs, Bugs, daffy, and yosemitesam

```
public class LoonieToons {
    public static void bugs() { }
   public void Bugs() { } //no link to bugs
     public int elmer() { }

     private static double daffy() {
   }     public void yosemitesam(int wile, double coyote){}

}
```

Multiple methods in a class

Because the class LoonieToons does not have a main method you can not run this class

```
public class LoonieToons {
    public static void bugs() { }
   public void Bugs() { } //no link to bugs
     public int elmer() { }

     private static double daffy() {
   }     public void yosemitesam(int wile, double coyote){}

}
```

The main() method

Any program you ever write in Java will always start the the beginning of the main method.

Remember that the beginning of a method is always marked by the { that immediately follows the name of the method.

Using Methods

If you write code in a method other than the main method, there are only 2 ways that your code will be executed:

1)If one of your commands inside the main method is to execute this method.

2)If one of your commands, which is inside a method called by the main method, is to execute this method (and so on if a command was to execute a method which had a command to execute a method which had a command to execute a method......)

The name of a method

Remember that you can figure out the name of a method by looking at the method header:

```
public static void madness() {
.....method body/definition
}
```

Note that you can only call a method like this when the method madness() is part of the same method. For the time being, we are only using 1 class, so this will always be true unless we use **library** methods (will talk about later)

Question:

```
public class MethodTest {
    public static void madness() {
        System.out.println("I'm in madness!);
    }

    public static void main(String[] args) {
        System.out.println("I'm the main method");
        ???????
        System.out.println("Good-bye");
    }
}
```

What would you write instead of ?????? to call the method madness ? What will the program print?

Question:

```
public class MethodTest {
    public static int madness() {
        System.out.println(“I'm in madness!”);
    }

    public static void main(String[] args) {
        System.out.println(“I'm the main method”);
        madness();
        System.out.println(“Good-bye”);
    }
}
```

Question:

```
I'm the main method
I'm in madness!
Good-bye
```

Watch out!

Any variables you create inside one method are NOT related to variables of the same name in another method!

Every method you ever write, will have it's own set of variables!

Watch out!

```
public static void confusing() {
    x = 0;
    ----->this is an error. x is undeclared!
}

public static void main(String[] args) {
    int x = 5;
    int y = 3;
    confusing();
}
```

Watch out!

```
public static void confusing() {
    int x = 0;
    ----->no error, but changes a different x.
}

public static void main(String[] args) {
    int x = 5;
    int y = 3;
    confusing();
    System.out.println(x);
}
```

Watch out!

```
public static void confusing() {
     x = 0;
     ----->this is an error. x is undeclared!
}

public static void main(String[] args) {
     int x = 5;
     int y = 3;
}
```

Note: Even if we don't call the method confusing(), we still get a compiler error.

But what if I need my method to use the values of variables?

There are 2 ways that you can “share” variables between methods:

1) You can actually declare a variable outside of any method (but inside a class). This has some confusing results (you need to write static before the variables) and is generally bad style. Don't do this on your assignments!
2) You can *pass* the *values* of variable(s) to your method. When you do this, every time your method is called, certain variables will have their values initialized already.

But what if I need my method to use the values of variables?

2)You can *pass* the values of variable(s) to your method.

But what if I need my method to use the values of variables?

To pass the value of a variable(s) to a method, you have to do 2 things:

1)When you write the method header, specify the types and names that all variables initialized at the start of your method. These names can be whatever you want and don't have to be related to any other names in other methods.

2)When you call your method, you need to give values to the method to correspond with each variable listed in 1)

Changing the method header

```
public static void madness(int x) {
}
```

This means the method called madness can take 1 value as input. This value has to be of type int and it will be referred to as x inside the method. When we call the method madness(), we now have to give it one value.

```
public static void main(String[] args) {
   madness(3);  ----> calls madness. inside madness, x=3
   madness(10-5); -->calls madness. inside madness,x=5
}
```

Changing the method header

```
public static void doubleMad(int x, double y) {
}
```

This means the method called doubleMad takes 2 values as input. The first of these values is of type int and will be called x. The second of these values is of type double and will be called y.

```
doubleMad(3,5.0); ----> calls doubleMad, x=3, y=5.0
double foo = 100;
doubleMad(1,foo); ----> calls doubleMad, x=1, y=100
doubleMad(2, foo-1); --->calls doubleMad,x=1, y=99
```

Changing the method header

```
public static void doubleMad(int x, double y) {
}
```

What do you think happens if we wrote:

```
doubleMad(3);   ???
doubleMad(2,1,10); ???
doubleMad(1.5,    3 ) ;     ???
```

Changing the method header

```
public static void doubleMad(int x, double y) {
}
```

What do you think happens if we wrote:

```
doubleMad(3);   too few arguments
doubleMad(2,1,10); too many arguments
doubleMad(1.5,    3 ) ;     arguments of wrong type
```

Trick question

```
public static void swap(int a, int b) {
    int temp = a;
    a = b;
    b = temp;
}

public static void main(String[] args) {
    int x = 3;
    int y = 2;
    swap(x,y);
    System.out.println(“x is “ + x + “  y is “ + y);
}
```

Trick question

Remember that only the VALUE of the variable is given to the method, not the actual variable itself.

This means that when we write swap(x,y) , we are saying:

“call the method swap. The value of a and b inside swap should be 3 and 2 (respectively) inside the method swap.”

Inside the method swap(), we do actually swap the values of a and b. But it has no effect on the main method.

Trick question

```
public static void swap(int x, int y) {
    int temp = x; --->still is no link between x in swap
      and main. Remember each method has its own set
      of variables.
    x = y;
    x = temp;
}
 public static void main(String[] args) {
    int x = 3;
    int y = 2;
    swap(x,y);
    System.out.println("x is " + x + " y is " + y);
}
```

Getting a result from your method

If you want to use a result from a method, you can do what is known as *returning a value* from the method. You can only get ONE (or zero) result from a method.

There are 2 things you have to do:

1)Choose what type you want to get from the method. Once you have chosen that, specify it in the method header.

2)Write *return expression* inside the method at the point you want to give the result back. Whenever a return statement is reached, the method executes the return statement and LEAVES THE METHOD!

Getting a result from your method

```
public static int intReturner() {
     int x = 100;
     return x + 10;
}
```

If you declare a method to return an int, you HAVE to return an int. If we left off the return statement, the method would not compile.

Note that you can have more than one return statement in a method once we add more complicated things like if statements. BUT only the first will be executed.

Getting a result from your method

```
public static int intReturner() {
    return 10;
    System.out.println("I just returned.");
}
```

The above line is not printed because the method already returned at the above line.

In fact, the compiler is smart enough in this case to know you have reached unreachable code and you will get a compiler error!

Using the result of a method

Once you have written a method to return type *t*, you can put a call to the method ANYWHERE that something of type *t* can appear.

For example:

```
int x = intReturner();
int y = intReturner() + 1;
System.out.println(intReturner());
madness(intReturner());
```

Making the OUTPUT of your method depend on the INPUT

A return statement can really be any expression, as long as the type specified in the method header matches the type in the return statement. This means, we can make a method depend on the input:

```
public static int addOne(int x) {
    return x+1;
}

int y = 3;
System.out.println(addOne(y));
System.out.println(addOne(y) + addOne(addOne(y)));
```

Reason for using a method

There are many reasons to use methods and not just put everything in the main method.

In many ways, a method is like a “subprogram” or a program within your program:

1)It has an input and output
2)Each method has its own set of variables

This means that the method is very well self-contained. They help to keep our code well organized.

Reason for using a method

One big benefit of using methods is that you can repeat computations very easily. If you have a complicated computation with many statements in it, you can call the same method many times from your code without lots of copying and pasting.

Another advantage is that.....

What happens in ~~Vegas~~ a method stays in a method

This is good because once we write a method and test it, we never have to change it again

Bug fixing

Suppose you make a mistake in your computations and you never use methods. Because you did the same computation over and over again, now you have to change every single one of these.

If you used a method, you just have to change the 1 place where the mistake happened.

Exercise

Write a method called celciusToFahrenheit

Your method should take as input a double representing a Celsius temperature and return a double, which is the original value but in Fahrenheit.

Note: F = C * 1.8 + 32

Now write how you would call this method.

Important note!

```
public static void CallAFunction(int airline) {
    System.out.println(airline);
}
```

a is just a name I'm calling a variable inside the method. I can replace it with any other valid Java identifier

Important note!

```
public static void CallAFunction(int delta) {
    System.out.println(delta);
}
```

a is just a name I'm calling a variable inside the method. I can replace it with any other valid Java identifier

Important note!

```
public static void CallAFunction(int aircanada) {
    System.out.println(aircanada);
}
```

a is just a name I'm calling a variable inside the method. I can replace it with any other valid Java identifier

Important note!

```
public static void CallAFunction(int united) {
    System.out.println(united);
}
```

a is just a name I'm calling a variable inside the method. I can replace it with any other valid Java identifier

More on Variables and Expressions

What should I call my classes variables and methods?

- Identifiers (what we call our methods or variables or classes) can have:
-
- any alphabetical letters (either case)
- numbers 0-9 (as long as it isn't the first character)
- _   (underscore)
- $
-
- Identifiers are case sensitive, so
- int b;   is not the same as   int B;

Naming Conventions

- We have many conventions that we follow to make naming more consistent.
-
- Variables and Methods should all be lower-case EXCEPT for the FIRST letter of every word other than the first
-
- Method names should start with a verb
-
- Class names follow the same convention as variables except the FIRST letter is also capitalized
- Constants (we'll see later) must be ALL CAPITALS

Examples:

- int iHateComputers; //good!
- boolean ILoveComputers; //bad :(
- float COMPUTERSAREAWESOME; // bad :(
- double x; // bad-- not descriptive
-
- class MyClass; // good!
- class Myclass; // bad!
-
- public static int Computer() ; // bad – no verb and cap C
- public static int turnOnComputer(); // good

Types

- • In Java, all variables and all values have a type
- • A *type* is a category of values that a variable belongs to and
- determines:
- – How to interpret and use the value it contains
- – What are the possible values it can contain
- – How much memory should be reserved

Types

- In a computer, everything is stored as 1s and 0s (or on/off switches)
-
- Knowing the **type** of a variable tells the computer what the **encoding** is.

Recall:

int : stores an integer number

double: stores a “real” number.

float: also stores a “real” number (smaller than a double)

String : stores words

double

If you write .0 after an integer constant, it will be stored as a double.

```
int x = 3.0;
```

double

doubles can store real numbers with fractional values.

They can not store an infinite amount of digits, for example in the number pi. They can only store to a limited precision.

ex: double x = 3.14159 ; // can only store //some of the digits

double

If you write .0 after an integer constant, it will be stored as a double.

```
~~int x = 3.0;~~

int x = 3; OR
double x = 3.0;
```

Types : Example: Storing a float

- A float normally takes up 32 bits in memory.
- It is similar to a double except doubles use 64 bits
- To use a float, write f after the number (e.g. 3.0f)
- 1 bit is for plus or minus
- 5 bits are for the exponent (1 of which is for +/-)
- 10 bits are for the number

Types : Example: Storing a float

- A float normally takes up 32 bits in memory.
- 1 bit is for plus or minus
- 8 bits are for the exponent (1 of which is for +/-)
- 23 bits are for the number

1 0 0000011 00000000000000000000010

sign signexp exponent base

Types : Example: Storing a float

- A float normally takes up 32 bits in memory.
- 1 bit is for plus or minus
- 8 bits are for the exponent (1 of which is for +/-)
- 23 bits are for the number

1 0 0000011
00000000000000000000010

sign signexp exponent base

= 2 * 2^(-3)

Types : Example: Storing an int

- An int takes up 32 bits in memory
- 1 bit is for the sign
- other 31 bits are for the number

1 000000000000000000000000000001010

sign number

Types : Example: Storing an int

- An int takes up 32 bits in memory
- 1 bit is for the sign
- other 31 bits are for the number

1 0000000000000000000000000001010

sign number

= 10

Thought experiment

Suppose Java gave no error when we write:

int x = 6.0f ;
3.0f looks like the following as a float:

1 1 0000001
00000000000000000000011

sign signexp exponent base

i.e. 3.0 * 2^1

Thought experiment

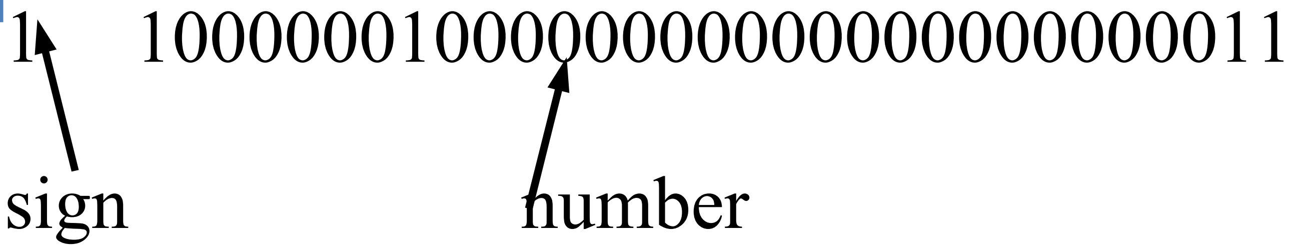
1   1   0000001
  00000000000000000000011

sign   signexp   exponent   base

i.e. 3.0 * 2^1

When Java puts this number into an int, it will be read totally differently!

Thought experiment



i.e. 1082130435

Some languages, such as C, will allow you to do this without giving an error. (What are the pluses and minuses of this?)

Types: Primitive vs. Reference

- There are two types in java: primitive and reference
-
-
- So far we have only seen primitive, except for String
-
- Primitive types represent very basic types (int, double, etc)

Primitive Types

•

There are exactly 8 primitive types in Java

- Positive and negative whole numbers:
  - byte, short, int, long
- Positive and negative numbers with decimal parts ("floating points numbers"):
  - float, double
- Characters (like a & * 6 /)
  - char
- And one of them represents boolean values (true or false):
  - boolean

Real numbers vs floating point

In a computer, we don't have an infinite amount of memory. Thus we can't actually store a number such as PI

It is also difficult for a computer to store something like

5.000000000000000000000000000000000001

Problem is it stores it as “base * 2^(some power)”

Char variables

- A *character set* is an ordered list of characters, and each
- character corresponds to a unique number
- • A char variable stores a single character from the Unicode
- character set
- char gender;
- gender = 'M';
- • Unicode is an international character set, containing symbols
- and characters from many world languages
- • Character values, also called character *literals* are delimited
- by apostrophes:
- 'a' 'X' '7' '$' ',' '\n'

But what about '

- What if we want to store an ' inside of a char?
-
- char x = ''';   // (three single quotes)
-
-
- It will think the 2nd ' marks the end of the char!
-
- Answer: Use an escape sequence. To do this, write a \ before the
- character.
-
- char x = '\'';

Boolean Values

- Boolean values are used to store things that are either "true" or "false"
- For example, we could create the variable.
- boolean isClassAlmostOver;

Arithmetic Expressions

- An *expression* is any combination of *operands* and [optional] *operators*.
  - An *operand* can be a literal value (like `5` or `3.14` or `'a'`), a variable, or the value returned by a method call (like `nextInt()`)
- *Arithmetic expressions* use the following operators
  - Addition: `x + y`
  - Subtraction: `x - y`
  - Multiplication: `x * y`
  - Division: `x / y`
  - Remainder: `x % y`

Integer Division

•

•

- If both operands to the division operator (/) are **integers**, the result is an integer (the fractional part is discarded)
  - 9 / 2 = 4
- The remainder operator (%) returns the remainder after dividing the second operand by the first
  - 10 % 3 = 1
- Division by 0 with integers (e.g. 1 / 0)
- Produces run-time error
- The program has to avoid it, or it will crash

Careful!

You have to be careful. Things that are true in math are not necessarily true in Java.

```
int x = (1 / 2 ) + ( 1 / 2 ) ;
```

What is x?

Careful!

```
double x = 1/2;
```

This does not work either. The problem is that both 1 and 2 are ints. When you divide 1 by 2 as ints you get 0. Then it is converted to a double but it is too late

Better:

```
double x = 1.0/2.0;
```

OR

```
double x = .5;
```

Operator Precedence

•Operators can be combined into complex expressions:

```
result  =  total + count / max - offset;
```

•Operators have a well-defined precedence which determines the order in which they are evaluated

1) Expressions inside parentheses.
2) Multiplication (`*`), division (`/`), and remainder (`%`)
3) Addition (`+`) and subtraction (`-`)

•Parentheses and arithmetic operators with the same precedence are evaluated from **left to right**

Operator Precedence Examples

- What is the order of evaluation in the following expressions?

```
a + b + c + d + e        a / (b + c) - d % e
```

1 2 3 4

```
a + b * c - d / e        a / (b * (c + (d - e)))
```

Assignment Operator Sides

•The left-hand and right-hand sides of an assignment statement can contain the same variable:

```
count = count + 1;
```

First, `1` is added to the original value of `count`; the result is stored in a temporary memory location

Then, the overall result is stored into `count`, overwriting the original value

•The fact that the assignment operator has lower precedence than arithmetic operators allows us to do this

Meaning of life (part deux)

Another explanation for 9*6 = 42 is order of operations

Someone decided to be clever and write 9 as 1+8 and 6 as 5+1

1 + 8 * 5 + 1 = 42

Converting from one type to another

Sometimes we want to convert from one type to another. For example, you might want to put an int into a double or a double into an int (as best you can).

Widening conversion

Converts something that takes up less memory to something that takes up more.

e.g.

int -----> double

Usually no information lost.

Narrowing conversion

Converts something that takes up more memory to something that takes less more.

e.g.

double ----> int

Usually some information lost.

Types: Mixed Expressions

•

•Sometimes expressions will be complicated and have more than 1 type in them
•What if you try to do a mathematical operation on two different types.

Ex:

3.5 * 2

The first operand is a double. The second operand is an int.

Types: Mixed Expressions

When you have mixed types like this, Java will always try to convert the NARROWER type to the WIDER type

For example, if you mix an int and double in an expression, Java will convert the int to a double

```
int x = 3.5 * 2;  // error---> trying to put a double into int
```

Types: Mixed Expressions

What will x equal in the following?

```
double x = 1.0 / 2 ;
```

Types: Mixed Expressions

What will y equal in the following?

```
double y= 4 / 3 + 4.0 / 3;
```

Casting

Sometimes you will want Java to force Java to turn an
expression from one type to another

For example, maybe we want to get the integer part of a number.

Then you can CAST a double to an int.

Casting

Write in parenthesis the type you want to convert to, and place
that before the expression you want to convert.

e.g.

int x = (int) 7.5;

x is now set to be 7.

Note: The conversions always round DOWN

Casting

Casting is very powerful

Casting is very dangerous. You can lose information!

Casting

Casting is temporary: it does not change the type of the value in a variable for the rest of the program, only for the operation in which the value is cast

e.g.
```
double x = 3.5;
   int y = (int) x;
```

x is still a double!

Casting : Integer Division

What happens in the following:

```
double x = (double) 1 / 2;
```

Note: The casting operation is performed BEFORE any
other
operation (unless there are parenthesis saying
otherwise)

Careful again!

What happens in the following:

```
double x = (double) (1 / 2);
```

This is a case where you cast, but it is too late. The integer division has already been performed because of the parentheis

Part 4: `String` Basics

```
System.out.println(“A String is ”
+ “a sequence of characters “
+ “inside quotation marks.”);
```

The `String` Type (1)

- As we saw earlier, a variable of type `char` can only store a single character value

```
char c1 = 'a';
char c2 = '%';
```

- To store an ordered sequence of characters, like a whole sentence, we can use a variable of type `String`

`String` Concatenation (+)

- In Java, + can be used to concatenate strings
  - `“hello” + ”world”` results in `"helloworld"`
  - `“hello” + “ world”` results in `“hello world”`
    - Notice the space before `world`
  - `“number ” + (5 + 2) results in “number 7”`

A `String` literal cannot be broken across two lines of source code

- The following code fragment causes an error:

```
"This is a very long literal
  that spans two lines"
```

- The following code fragment is legal:

```
"These are 4 short concatenated "
+ "literals "
+ "that are on separate source-code " +
"lines"
```

`String` Variables and Values

- Variables of type `String` are declared just like variables of other types

```
String message;
```

- Actual `String` literals are delimited by double quotation marks (")

```
String greeting = "Hello!";
```

Comparing Strings

•Strings are *reference* types. This means that the value in the variable is actually the location in memory that the actual data is stored in (as opposed to just storing the data)

What does this mean? Two Strings can actually store the different values (addresses) but the different addresses have the same contents! This will matter when we discuss comparing variables

Mixed-Type Concatenation

Remember: the plus operator (`+`) is used for *both* arithmetic addition and for string concatenation

The function that the `+` operator performs depends on the type of the values on which it operates

– If both operands are of type `String`, or if one is of type `String` and the other is numeric, the `+` operator performs string concatenation (after "promoting" the numeric operand, if any, to type `String` by generating its textual representation)

Mixed-Type Concatenation

This suggests a useful trick to convert a value whose type is a primitive type to a `String`: concatenate the empty `String` `""` with the value

```
   int i = 42;
   String s = "" + i;
     // s contains the String
"42"
```

Trick questions

System.out.println(“5 + 3 =” + 5 + 3);

System.out.println(5 + 3 + “is the same as 5+3”);

System.out.println(“5+3 =” + (5+3));

Note: The + operator is evaluated from left to

+= assignment

Programmers got lazy and sick of constantly writing statements like

x = x +5;

So as a shortcut, you can write

x += 5 ;

to mean the same thing

+=, -=, *=, /= assignment

You can also do this with -,*, and /

++ operator

They then got even lazier......

The ++ operator can be used to add one to an int.

```
int x = 5;
x++;

//adds one to x
```

++ operator

You can also write it **before** the variable

```
int x = 5;
++x;

//adds one to x
```

++ operator

The difference is the order of operations.
x++ increments AFTER getting the value,
++x increments BEFORE getting the value

```
int x = 5, y=5;
System.out.println(x++); // prints 5, makes
x=6
System.out.println(++y); //prints 6, makes
y=6
```

-- operator

Same as ++ except it decreases the value of the variable.

```
int x = 5;
x--;
```

Recommendation

To avoid any confusion here, it is strongly recommend that you only use increment statements by themselves.

Do not do things like the following!

```
double wtf = (double) (x++ + --x);
```

It will just be confusing

Constants (1)

- A constant is an identifier that is similar to a variable except that it holds one value for its entire existence
- In Java, we use the `final` modifier to declare a constant

```
final double PI = 3.14;
```

- The compiler will issue an error if you try to assign a value to a constant more than once in the program

```
final double PI = 3.14;
// Some more statements...
PI = 2.718;
    // Error: cannot assign a value to a
    // final variable more than once
```

Next Class

- If statements
- Boolean (true/false) expressions
- Loops