

# Variables, Types, Operators & Expressions

Software Development 1 (F27SA)

Michael Lones

Week 2, lecture 1

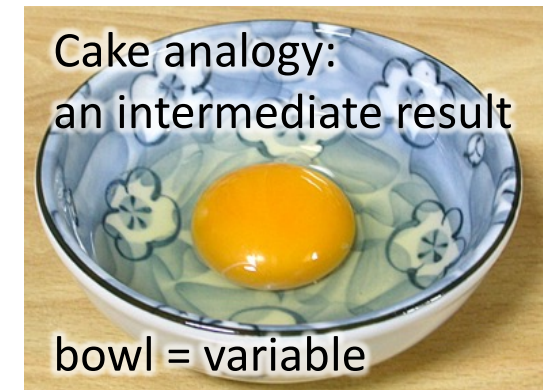
# Today's Lecture

- What are variables, types and operators?
- Primitive types in Java
- Arithmetic operators
- Expressions and precedence rules
- Strings
- Simple input and output

# What is a Variable?

It is a **named memory location** where you can store something

- Each variable can store exactly one thing
- Which might be a number, a string of text, or something more complex
- They are often used to store inputs, intermediate results, and outputs of a computation
- But can also be used for more permanent storage of important data



# What is a Type?

A type defines the **kind of thing** that a variable can store

- Different languages support different types
- Most languages have various types for different kinds of numbers, and for characters and character strings
- Some types are more efficient than others, meaning they are faster to process in the CPU and read/write to/from memory

# Java Variable Types

Java has two broad categories of types:

- **Primitive types.** Numbers, characters and Booleans (which contain true or false values)
- **Object types.** These are more complicated data structures held in memory. You'll come across many of these later in the course. For now, the only one you need to know about is the **String** type, which represents a string of characters.

# Numbers

Java has a selection of different primitive types for representing numbers. Most common are:

- **int** : This represents an integer (a whole number) in the range  $-2^{31}$  to  $2^{31}-1$ . It is relatively efficient to use.
- **double** : This represents a floating-point (decimal), number. It can represent numbers very accurately, but is less efficient than using an `int`.

# Numbers

Other numeric types you might come across are:

- **byte** : This is used when you only need to represent small integers in the range -128 to 127
- **short** : This is used when you only need to represent integers in the range  $\pm 32,768$
- **long** : This is used when an int is not big enough. It can represent integers in the range  $-2^{63}$  to  $2^{63}-1$
- **float** : This can be used when you need to save memory when representing floating point numbers. It has lower precision than a `double`.

# Numbers

In general, use the most efficient type that can represent the numbers you are working with

- If you're using whole numbers, don't use a floating point type. If you're using floating point numbers, don't use an integer type.
- If you know the minimum/maximum number that a variable will contain, then consider `byte`, `short` or `long`. Otherwise, `int` is usually a safe bet.



# Numbers

The efficiency of a type is related to how much memory it uses, e.g.

- A `short` is 16-bit, so it uses 2 bytes of memory
- An `int` is 32-bit, so it uses 4 bytes of memory
- A `double` is 64-bit, so uses 8 bytes of memory

However, memory is cheap these days, so you don't have to worry about this unless you're using loads of data, your code needs to run very fast, or your program is running on an embedded system.

# Declaration

In Java, variables must be declared before they are first used. A declaration has the form:

*type name;*

Some example declarations:

```
int anInteger;
```


```
int age;
```

```
double decimalValue;
```

```
double amountOwed;
```

# Declaration

In Java, variables must be declared before they are first used. A declaration has the form:

*type name*; A semi-colon needs to be placed  
after every statement in Java

Some example declarations:

```
int anInteger;
```

```
int age;
```

```
double decimalValue;
```

```
double amountOwed;
```

# Declaration

In Java, variables must be declared before they are first used. A declaration has the form:

*type name;*

Some example declarations:

`int anInteger;` ✗

`int age;` ✓

`double decimalValue;` ✗

`double amountOwed;` ✓

A variable can be called anything (with some exceptions). However, you should choose something informative whenever possible.

# Declaration

In Java, variables must be declared before they are first used. A declaration has the form:

*type name;*

Some example declarations:

```
int anInteger;
```

```
int age;
```

```
double decimalValue;
```

```
double amountOwed;
```

Most languages have *naming conventions*. In Java, variable names begin with a lower-case letter, and multiple words are combined using "**camelCase**"

# Assignment

You can assign a value to a variable using =

```
age = 18; // assigns the value 18
```

You can also assign an initial value within the variable declaration:

```
int age = 18;
```

```
double amountOwed = 100.0;
```

# Numbers

```
public class NumberDemo {  
    public static void main(String[] args) {  
        int age = 18;  
        double amountOwed = 100.0;  
        System.out.println(age);  
        System.out.println(amountOwed);  
    }  
}
```

NumberDemo.java

```
$ javac NumberDemo.java  
$ java NumberDemo  
18  
100.0
```

Terminal

# Number Errors

```
public class NumberDemo {  
    public static void main(String[] args) {  
        int age = 18.5;  
        double amountOwed = 100.0;  
        System.out.println(age);  
        System.out.println(amountOwed);  
    }  
}
```

NumberDemo.java

```
$ javac NumberDemo.java
```

Terminal

```
Type mismatch: cannot convert from double to int
```



# Number Errors

```
public class NumberDemo {  
    public static void main(String[] args) {  
        int age;  
        double amountOwed;  
        System.out.println(age);  
        System.out.println(amountOwed);  
    }  
}
```

NumberDemo.java

```
$ javac NumberDemo.java
```

Terminal

The local variable age may not have been initialized

The local variable amountOwed may not have been initialized

# Number Errors

These examples illustrate a couple of things:

- Java won't let you assign a value to a variable if the variable is not accurate enough to fully represent it. In this case, an `int` can not represent the floating point number `18.5`
- Java won't let you use a variable until it has been assigned a value, i.e. until it has been **initialized**

These rules help you to avoid common programming errors. However, other programming languages, such as C, are more lenient than this.

# Casting

You can use casting to change the type of a value. Casting has the syntax:

*(desired\_type) value*

This has various uses when programming. One use is to force a value into a variable when the types are not compatible.

# Casting

```
public class NumberDemo {  
    public static void main(String[] args) {  
        int age = (int) 18.5;  
        double amountOwed = 100.0;  
        System.out.println(age);  
        System.out.println(amountOwed);  
    }  
}
```

NumberDemo.java

```
$ javac NumberDemo.java
```

```
18  
100.0
```

Terminal

Note that the value 18.5 has been truncated as a result of casting

# Expressions

When you assign to a variable, the piece of code on the right of the `=` is known as an **expression**

- An expression can be a value, e.g. a number
- An expression can be another variable
- An expression can involve operators

# Arithmetic Operators

Java has a range of arithmetic operators that can be applied to numbers and numeric variables.

- `*` : multiply
- `/` : divide
- `+` : add
- `-` : subtract
- `%` : modulus

# Assignment

```
public class NumberDemo {  
    public static void main(String[] args) {  
        int age = 18;  
        int agetoday = age;  
        System.out.println(age);  
        System.out.println(agetoday);  
    }  
}
```

NumberDemo.java

```
$ javac NumberDemo.java  
$ java NumberDemo  
18  
18
```

Terminal

This means "copy the value of age  
into the variable agetoday"

# Arithmetic Operators

```
public class NumberDemo {  
    public static void main(String[] args) {  
        int age = 18;  
        int agenextyear = age + 1;  
        System.out.println(age);  
        System.out.println(agenextyear);  
    }  
}
```

NumberDemo.java

```
$ javac NumberDemo.java  
$ java NumberDemo  
18  
19
```

Terminal

This expression uses an arithmetic operator to add one to the value in age



# Arithmetic Operators

```
public class CalcEnergy {  
    public static void main(String[] args) {  
        double mass = 25.457;  
        double c = 3e8; // 3x10^8 in Java  
        double energy = mass * (c * c);  
        System.out.println(energy);  
    }  
}
```

CalcEnergy.java

```
$ java CalcEnergy  
2.29113E18
```

Expressions can contain multiple  
variables and operators

Terminal

# Arithmetic Operators

Java has shortcuts for simple variable operations:

- `i++;` is the same as `i = i+1;`
- `i--;` is the same as `i = i-1;`
- `i+=5;` is the same as `i = i+5;`
- `i-=5;` is the same as `i = i-5;`
- `i*=5;` is the same as `i = i*5;`
- `i/=5;` is the same as `i = i/5;`

▼ In programming language terminology, useful shortcuts like these are known as **syntactic sugar**

# Precedence

When constructing more complex expressions, you have to be aware of precedence rules

- Operators with highest precedence always get applied first, regardless of where they occur within an expression
- If you're not aware of the precedence rules, then your program might have unexpected behaviour


# Precedence

Mathematical operators in order of precedence (high to low):

- `()`
- `++, --`
- `- [minus]`
- `*, /, %`
- `+, - [subtract]`
- `=, +=, -=, *=, /-`

# Precedence

Mathematical operators in order of precedence (high to low):

- $()$   If you want to make sure a sub-expression gets evaluated first, then enclose it in parentheses
- $++$ ,  $--$
- $-$  [minus]
- $*$ ,  $/$ ,  $\%$
- $+$ ,  $-$  [subtract]
- $=$ ,  $+=$ ,  $-=$ ,  $*=$ ,  $/-$

# Precedence

Mathematical operators in order of precedence (high to low):

- $()$
- $++$ ,  $--$
- $-$  [minus]
- $*$ ,  $/$ ,  $\%$
- $+$ ,  $-$  [subtract]
- $=$ ,  $+=$ ,  $-=$ ,  $*=$ ,  $/=$

Multiplication has higher precedence than addition, meaning that:

$a+b*c$  is not the same as  $(a+b)*c$

# Precedence

Mathematical operators in order of precedence (high to low):

- `()`
- `++, --`
- `-` [minus]
- `*`, `/`, `%`
- `+`, `-` [subtract]
- `=`, `+=`, `-=`, `*=`, `/=` ←

Assignment has lowest precedence, so everything to the right of the `=` gets evaluated before the assignment is made

# Exercise

Complete this program so that it correctly calculates the formula  $y = \frac{-1+ax^2+bx}{a(b+x)}$

```
public class EquationExercise {
    public static void main(String[] args) {
        double x = 10.5, a = 2.5, b = 10;
        double y = <add code here>
        System.out.println("y="+y);
    }
}
```

EquationExercise.java



# Be Aware of Types

Be careful when your expression involves types of a lower precision:

```
public class ProblemEquation {  
    public static void main(String[] args) {  
        int a = 1, b = 2;  
        double y = a/b;  
        System.out.println(y);  
    }  
}
```

ProblemEquation.java

```
$ java ProblemEquation
```

Terminal

What do you think the output will be?

# Be Aware of Types

Be careful when your expression involves types of a lower precision:

```
public class ProblemEquation {  
    public static void main(String[] args) {  
        int a = 1, b = 2;  
        double y = a/b;  
        System.out.println(y);  
    }  
}
```

ProblemEquation.java

```
$ java ProblemEquation  
0.0
```

Terminal

# Be Aware of Types

Be careful when your expression involves types of a lower precision:

```
public class ProblemEquation {  
    public static void main(String[] args) {  
        int a = 1, b = 2;  
        double y = a/b;  
        System.out.println(y);  
    }  
}
```

**If you carry out a division (or other operation) between integers in Java, it applies integer division. In this case, this means  $1/2 = 0$**

ProblemEquation.java

```
$ java ProblemEquation  
0.0
```

Terminal

# Be Aware of Types

Be careful when your expression involves types of a lower precision:

```
public class ProblemEquation {  
    public static void main(String[] args) {  
        int a = 1, b = 2;  
        double y = (double) a/b;  
        System.out.println(y);  
    }  
}
```

ProblemEquation.java

In this case, you can solve the problem by casting at least one of the operands to a double: this will result in a floating point division

```
$ java ProblemEquation  
0.5
```

Terminal

# Be Aware of Types

Also be careful when you use numeric literals  
(that is, numbers that you type in your code)

```
public class ProblemEquation {  
    public static void main(String[] args) {  
        double x = 10;  
        double y = 1/2 * x;  
        System.out.println(y);  
    }  
}
```

ProblemEquation.java

```
$ java ProblemEquation  
0.0
```

Terminal

# Be Aware of Types

Also be careful when you use numeric literals (that is, numbers that you type in your code)

```
public class ProblemEquation {  
    public static void main(String[] args) {  
        double x = 10;  
        double y = 1.0/2.0 * x;  
        System.out.println(y);  
    }  
}
```

The solution is to make sure that at least one of the operands is not an integer. You can do this here by adding a decimal point.

ProblemEquation.java

```
$ java ProblemEquation  
0.5
```

Terminal

# Other Primitive Types

Java's two non-numeric primitive types are:

- **char** : This represents a single Unicode character. Unicode is a standard for representing the 136,755 characters found in different languages.
- **boolean** : This has only two possible values: *true* or *false*. It is used in logic expressions. You'll come across Boolean values soon when I introduce conditional execution.

# chars

A **char** represents a single character.

```
char myInitial = 'M';  
char smileyFace = '\u263A'; // ☺
```

Note the **single** quotes around the letter.

In practice, you won't come across chars very often in Java programs. It's easier to use a String.



# Strings

A **String** represents a sequence of characters.

```
String myName = "Michael";
```

Note the **double** quotes around the letters.

A String is an object type, not a primitive type. This is indicated by the initial capital letter in its type.

# Strings

You can join Strings together:

```
String fullName = myName + " Lones";
```

And do other useful things:

```
myname.substring(3); // Mic
```

```
myname.length(); // 7
```

```
myname.toUpperCase(); // MICHAEL
```

For more information, see:

<https://docs.oracle.com/javase/7/docs/api/java/lang/String.html>

# Strings

```
public class StringDemo {  
    public static void main(String[] args) {  
        String name = "Bob";  
        int age = 18;  
        String message = "My name is " + name +  
                           ". I am " + age + "."  
        System.out.println(message);  
    }  
}
```

```
$ java StringDemo  
My name is Bob. I am 18.
```

Terminal

# Strings

```
public class StringDemo {  
    public static void main(String[] args) {  
        String name = "Bob";  
        int age = 18;  
        System.out.println(  
            "My name is "+name+". I am "+age+".");  
    }  
}
```

StringDemo.java

```
$ java StringDemo  
My name is Bob. I am 18.
```

Terminal

# Constants

Sometimes you want a variable to maintain a constant value. You can achieve this by declaring a variable to be static final:

```
static final int DAYS_IN_YEAR = 365;  
static final String VERSION = "V2.5";
```

Note that constants are usually given an uppercase name to make them easy to spot in your code.



# Constants

Sometimes you want a variable to maintain a constant value. You can achieve this by declaring a variable to be static final:

```
static final int DAYS_IN_YEAR = 365;  
static final String VERSION = "V2.5";
```

Note that constants are usually given an uppercase name to make them easy to spot in your code. 

You'll get an error if you try to change a constant value, so use it to represent things that will never change.

# Constants

Constants must be declared outside of `main`:

```
public class ConstantDemo {
    static final int DAYS_IN_YEAR = 365;
    public static void main(String[] args) {
        int age = 18;
        int days = age * DAYS_IN_YEAR;
        System.out.println(
            "I am " + days + " days old.");
    }
}
```

ConstantDemo.java

# Constants

These are normally used to represent **fundamental unchanging** values in a particular domain

- There shouldn't be many of these in a program
- In mathematical programs, things like **PI** and **E**
- In a time-based program, things like **24** hours in a day, **60** minutes in an hour, **60** seconds in a minute
- They wouldn't normally be used for derived values like "number of seconds in a year"



# Input

In Java, you can use a `Scanner` to read input that the user types at the command line

- `Scanner` is an example of a utility class, something you'll hear a lot more about in Part 2, but fortunately you don't need to know what it is in order to use it.
- You will be using it in the lab session next week.
- If you want to know everything about it, see <https://docs.oracle.com/javase/7/docs/api/java/util/Scanner.html>

# Input

This is an example of using `Scanner` to read in an integer value from the user:

```
import java.util.Scanner;
public class ScannerDemo {
    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        int input = scan.nextInt();
        System.out.println("Your input: " + input);
    }
}
```

ScannerDemo.java

# Input

This is an example of using `Scanner` to read in an integer value from the user:

```
import java.util.Scanner;  
public class ScannerDemo {  
    public static void main(String[] args) {  
        Scanner scan = new Scanner(System.in);  
        int input = scan.nextInt();  
        System.out.println("Your input: " + input);  
    }  
}
```

ScannerDemo.java

Add this at the top of the file

# Input

This is an example of using `Scanner` to read in an integer value from the user:

```
import java.util.Scanner;
public class ScannerDemo {
    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        int input = scan.nextInt();
        System.out.println("Your input: " + input);
    }
}
```

**Add this line before you start reading from the user**

ScannerDemo.java

# Input

This is an example of using `Scanner` to read in an integer value from the user:

```
import java.util.Scanner;
public class ScannerDemo {
    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        int input = scan.nextInt();
        System.out.println("Your input: " + input);
    }
}
```

**This line reads in the next integer entered by the user, and stores it in a variable**

ScannerDemo.java

# Input

Scanner has different *methods* for different types of input, e.g. for a **double**:

```
import java.util.Scanner;
public class ScannerDemo {
    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        double input = scan.nextDouble();
        System.out.println("Your input: " + input);
    }
}
```

# Input

Scanner has different *methods* for different types of input, e.g. for a **string**:

```
import java.util.Scanner;
public class ScannerDemo {
    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        String input = scan.next();
        System.out.println("Your input: " + input);
    }
}
```

# Input

You can re-use the same `Scanner` if you want to read more than one input from the user:

```
import java.util.Scanner;

public class ScannerDemo {
    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        String input1 = scan.next();
        double input2 = scan.nextDouble();
        System.out.println(input1 + input2);
    }
}
```



# Quiz

Which of these are valid Java expressions?

A. `int count = 55 + (12/4);`

B. `int i = 55.5 * 1000.0;`

C. `int n = 100++;`

D. `String pre = "I " + "ate ";`

E. `String post = count + " pies";`

F. `String sentence = pre + post`

# Summary

- A variable is a named storage location.
- A variable's type determines what it can store.
- Java has primitive types and object types.
- `int` and `double` are used to store numbers.
- Casting can be used to change the type of a value.
- Operators can be used to build more complex expressions, which can be assigned to variables.
- It's important to be aware of operator precedence.
- Constants store values that won't change.

# Next Lecture

- Conditional execution (`if`, `switch...case`)