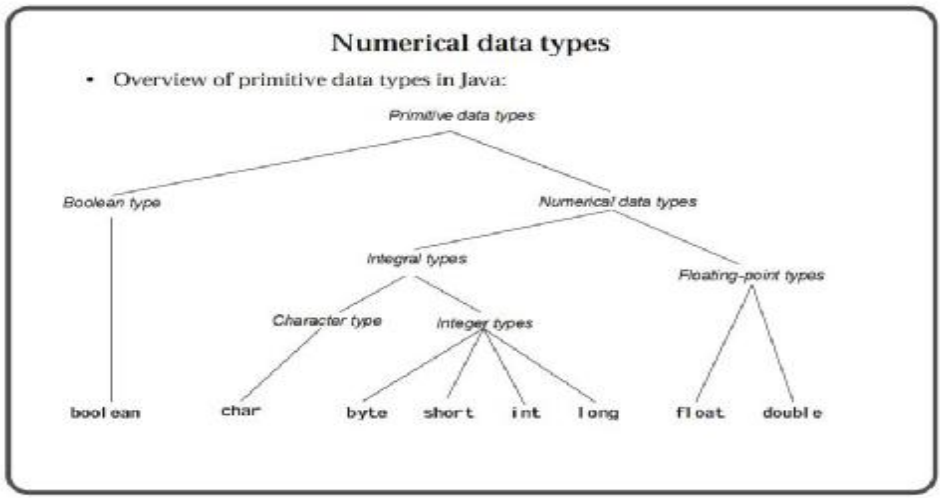
Introduction to Programming

3. Values, Operators, Expressions and Statements

Java's Primitive Types

- Java has eight "primitive" types
- These are used to represent numbers, characters (letters), and true/false values
- The names of the primitive types are all Java keywords – note that they are written entirely in lower case
- The four primitive types we will use most are
  - **int**, **double**, **char**, **boolean**
- The other four are numeric types with different ranges of values
  - **byte**, **short**, **long**, **float**

Primitive Types (figure from Mughal, Hamre & Rasmussen)



Numerical Type value ranges

Type	Range of values	Size of values
byte	-128 to 127 (-2^7 to 2^7-1)	8 bits
short	-32768 to 32767 (-2^{15} to $2^{15}-1$)	16 bits
int	-214783648 to 214783647 (-2^{31} to $2^{31}-1$)	32 bits
long	-9223372036854775808 to 9223372036854775807 (-2^{63} to $2^{63}-1$)	64 bits
float	±3.4028234663852886e+38 ($\pm(2-2^{-23})\times(2^{127})$)	32 bits
double	±1.7976931348623157e+308 ($\pm(2-2^{-52})\times(2^{1063})$)	64 bits

Java – Compound Data

- Non-primitive data
  - Values of primitive types are single atomic values
  - Data often has several components:
    - A date has a day, month and year
    - A name has a surname and forenames, and each one of these is a sequence of letters (chars)
- An example of compound data - String
  - A String is sequence of characters
  - In Java, compound data is represented using a **class** – String is an example of a Java class
    - Note that the class name, String, starts with a capital letter – this is the convention for all Java classes

Types revisited

- A *type* is a set of values, and a set of operations that can be applied to those values
- For the primitive types, and for the type, String, the values can be written directly using literals
  - `"This is a String literal"`
  - `7.01` // a **double** literal
  - `false` // a **boolean** literal

Expressions as values

- An *expression*
  - is a construct made up of literals, variables, operators and method calls (where the method call returns a value)
  - evaluates to a single value
- Examples

  a **double** expression:

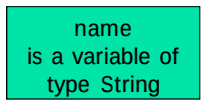
  ```
  (d/2.0 + e*Math.sqrt(++a))
  ```

  a **boolean** expression:

  ```
  x == 2
  ```

  a String expression:

  ```
  "Hello " + name
  ```



Operators

- An *operator* is a symbol such as `"+"`, `"<="`, or `"++"` that represents an operation that can be applied to one or more values in an expression *(quoted from the textbook glossary)*
- In this module we will look at
  - Assignment operators
    - Assign a value to a variable (`=, *=, /=, +=, -=`)
  - Arithmetic operators
    - Add, subtract, multiply, divide and remainder operations on primitive numeric values (`+, -, ++, --, *, /, %`)
  - Relational and equality operators
    - Equality: compare values for equality (`==, !=`)
    - Relational: compare primitive numeric values (`<, <=, >, >=`)
  - Logical operators
    - AND, OR and NOT operations for **`boolean`** values (`&&, ||, !`)

Evaluating expressions

- Sometimes, the result of an expression is ambiguous...
  - e.g. result = 4 * 1 + 3;
  - What gets assigned to result?
    - 7 or 16?
    - depends on the order the operators work in
    - Java does multiplication and division before addition and subtraction (so result = 7)
- Use brackets to force a different order of evaluation (or simply to make order less ambiguous)
  - 4 * (1+3) - obviously 16!

Assignment Operator

- The "=" operator is used to assign the value on the right hand side of the operator to the variable named on the left hand side of the operator
  - The value assigned replaces any previous value the variable had

```
name = "Albert Einstein";
area = PI*radius*radius;
```

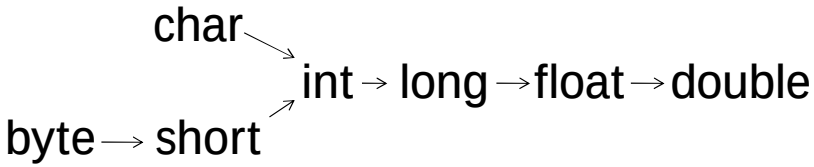

Assignment continued

- Java is strongly typed
  - that is, the value on the right hand side of the "=" operator should be the same type as, or a value of a type compatible with, the type of the variable

```
String name;
name = 7;  /* won't compile, 7
              is not a String */
```


Automatic type conversions

- Assignment Operators
  - Numeric type compatibility
  - A value of any type to the left can be assigned to a variable whose type is to the right (this is called a *widening* conversion)



Widening conversions

- For numeric types, can assign a value of a type with a narrower range of values to a variable of a type with a wider range of values

```
double d = 7; /* OK, int has a
    narrower range of values
    than double */
int i = 7.0; /* Not OK, double
    has a wider range of values
    than int (won't compile) */
```


Type casts

- If you are sure that a variable of a narrower type can hold the value of the wider type (that is, the value is within the range of the narrower type), you can use a *type cast* to convert the value to the narrower type

type cast

```
int i = (int)7.0; // OK now
```

- We did this in the lab exercise that simulated rolling dice, when we cast the double value obtained by multiplying Math.random() by 6 to an int

```
(int)(Math.random()*6) + 1
```


Automatic type conversions

- Arithmetic Operators
  - Automatic type conversion occurs
    - if the two arguments are of different types (the narrower type value is converted to the wider type value)

double literal — 67.0 * 2 — **int** literal

int is converted to **double** and the result is 134.0 – a value of type **double**

- if either argument is a **short**, **byte** or **char** (converted to **int**)

Division & Remainder

- In this module we are going to use **double** for real numbers and **int** for integer numbers so should generally not have to worry about type conversions except for division

10 / 4 = 2	10 / 4.0 = 2.5
10.0 / 4 = 2.5	10.0 / 4.0 = 2.5

- Remainder operator, %, gives remainder on integer division
  - 10 % 4 = 2 (as 10 divided by 4 gives 2 remainder 2)
  - 10 % 3 = 1 (as 10 divided by 3 gives 3 remainder 1)

Increment and Decrement

- Java, like C, has ++ and -- operators for numeric values
- **v++** returns value of v, then adds 1 to v
- **++v** adds 1 to value of v, then returns the value
- same for **v--** and **--v** except subtracts 1

```
int a = 10;
int b = a++; // a equals 11, b equals 10
int c = ++b; // b and c both equal 11
c++;   // expression statement (returned value of 11 is discarded), c = 12
int d = --a + b++ + a++; // result: d = 31, a = 11, b = 12
```

- Will follow textbook advice and only use these operators in stand-alone statements, not as expressions

Assignment operators

- Java, like C, has variations on the assignment operator that combines arithmetic operators with assignment

```
v += e;  // same as, v = v + e;
v -= e;  // same as, v = v – e;
v *= e;  // same as, v = v * e;
v /= e;  // same as, v = v / e;
```


Simple Statements

- Similar to a sentence in a natural language
- Generally each statement is written on a separate line (or perhaps over several lines)
- In Java, simple statements end with a semi-colon
  - expression statements (that are both a statement and an expression) are also possible in Java
- Examples of simple statements

```
TextIO.put(d);
result = (d/2.0 + e*Math.sqrt(++a));
++a;  // an expression statement
```

Strings

- String is an example of a class
- String is included as a standard part of the Java language
- String is also a type (*e.g.* you can declare a variable of type String)
- Once a String object is created, it can never be changed
  - the jargon is that Strings are *immutable*
- Parallel to “real world” – think of a String as a word or a sentence that you say

Strings continued

- A String is a sequence of characters
  - Each character is of type **char**
- A literal of type String is written in double quotes

```
"This is a String literal"
```

- A literal of type **char** is a single character written in single quotes

```
'a'
```


String as a class

- One big difference between a class and a primitive type is that a class can define methods
- String defines a number of methods that serve as operations on Strings
- Once you have a String object you can call its methods to operate on the String

String - some useful methods

(see Eck section 2.3.4 for more)

- Consider:

```
String s1 = "Goodbye";
String s2 = "bye";
```

  - `s1.length()`
    - This returns the number of characters in s1
    - in this case, 7
  - `s1.charAt(n)`
    - This returns the **char** at position n (n is an **int**)
    - The first **char** is at position 0
    - E.g.

```
s1.charAt(3) is 'd'
```


String methods continued

```
String s1 = "Goodbye";
String s2 = "bye";
```

- `s1.indexOf(s2)`
  - This returns the index position in s1 of the first occurrence of the String s2
  - in this case, it returns 4 as the String “bye” can be found as a substring beginning at position 4 of “Goodbye”
- `s1.toUpperCase()` returns `"GOODBYE"`

  `s1.toLowerCase()` returns `"goodbye"`
  - Important! These methods return a new String.
  - The original String is *not* changed, it is still `"Goodbye"`
  - Can make s1 refer to the new String by writing, for example:

```
s1 = s1.toUpperCase();
```

Data Structures

- Most pieces of information that we use are part of something bigger
  - A single name could be one entry in a register of names
  - A date of birth is just a date, but is also part of the information about a person
- Programming languages provide structures to collect data together
  - Arrays, Lists – collections of data of the same type (like the register)
  - Records – a collection of data of different types (like the info on a person)
    - in Java a class is used to group the fields of the record

list = new int[10];

Index	Value
0	81
1	23020
2	-1702
3	1200
4	185
5	-17402
6	31048
7	100
8	17
9	64

class Student {
String name;
int mark;
}

name	Fred Smith
mark	55

Java and Objects

- Arrays and “records” in Java are instances of classes, or *objects*
- Unlike for primitive types a variable of the type contains a *reference* to the object and NOT a value of the type
- Instances need to be created using a *constructor* (more on this later)

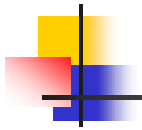
```
Student currentStudent = new Student();
currentStudent.name = "Fred Smith";
```


Structured Programming

- Apply structure to data (information) to simplify organisation
  - Arrays and records (Java classes) keep pieces of data together to maintain the associations in real life – a name, address and phone number, a list of web addresses
- (Next week we will begin to) use structure in executable statements to make the execution adapt to the prevailing circumstances
  - Process data according to conditions
  - Process a lot of data using only a few statements

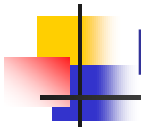
In this lecture....

- Expressions and Statements
  - Expressions compute a value
  - Statements as instructions, end in a semi-colon
- The eight primitive types
  - **boolean, char, int, double, float, byte, short, long**
- String – an example class
  - String is a type, strings are immutable



In this lecture....

- Compound data
  - Data that has more than one component
- Data structures
  - “Records” and arrays
  - In Java these are classes



Reading for Next Week

- **Continue reading chapter 2 of Eck**
  - **sections 2.3, 2.4 and 2.5**
- Next week we will start to look at control structures

End

- Questions?