EXPRESSIONS AND ASSIGNMENT

CITS1001
Scope of this lecture

• Assignment statements

• Expressions
ASSIGNMENT STATEMENT
Assignment Statements

mark = 50;

Read as “the value in variable on the LHS becomes equal to the value of expression on the RHS”

Remember: a running program follows the RULES of the language. It does not interpret what you meant to do.
Assignment

• Values are stored into fields (and other variables) via assignment statements:

    variable = expression;
    price = cost;

• A variable stores a single value, so any previous value is lost

• A sequence of statements are performed one after the other. Each statement uses the context provided by the previous statements.
Sequence and Assignment

```c
int x, y, z;
x = 10;
y = 4;
x = y;
z = y + 2;
```

<table>
<thead>
<tr>
<th>time (execution order)</th>
<th>computer memory</th>
<th>final state</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>y</td>
<td>z</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>6</td>
</tr>
</tbody>
</table>
EXPRESSIONS
Expressions

- An *expression* is a portion of Java code that is *evaluated* to yield a *value*
  - any expression has a *type* and a *value*

```
    total = price + 10;
```

- Whenever a statement is executed, each expression will be *evaluated* and *replaced* with its value
- The process of evaluating expressions and doing something with the resulting value is the fundamental process of computing
Evaluation of Expressions

• A simple example

```java
int x;
x = 10 + 15;
```

• The first line creates a variable called `x` of type `int`, while the second line will evaluate the expression `10+15`, replace it with its value (which is `25`) and then assign the resulting value to `x`

• Evaluation of an expression may take several steps and require the cooperation of several objects

```java
double y;
y = (b1.getBalance()*0.01) + (b2.getBalance()*0.05);
```
Types of Expression

- There are several types of expression
  - literals and names
  - method invocations
  - object creations
  - compound expressions built up with operators
    - arithmetic operators
    - relational operators
    - logical operators
Literals

- **Literals** are values that are "hard wired" into the code
  - 23, 10009, 'a', -23.456, 10e-5, 100e2
- In Java, the types `int` and `double` are dominant, in that any literal that does not use a decimal point is assumed to be of type `int`, and any literal uses a decimal point (or scientific notation) is assumed to be a `double`
- The Java compiler will complain if you try to assign a value to a "smaller" type

```java
float f;
f = 23.5;
```
Casting (an aside)

- If the programmer *really* wants to assign a `double` value to a `float` variable, then the compiler can be prevented from complaining by using a cast
- The number is simply *truncated* to fit the new type

```c
float f;
f = (float) 23.5;
```

This says to the compiler “I know this looks dumb, but I *do* know what I’m doing so please just treat this number as a `float` rather than a `double`”
Names

- *Names* are the declared *variables* that are in scope; the expression has the value and the type currently associated with that variable
  - the *value* in the shoebox if it is a primitive type
  - the *reference* in the shoebox if it is a reference type

- For example,

  *circle, price, total, mark, studentname, i*
Method invocations

• Another form of expression is the calling of a non-void method of some object
  • The *value* of the expression is the returned value
  • The *type* of the expression is the return type of the method
• The method call (here b1 is a *BankAccount*)
  • `b1.getBalance()` is an expression
    • Its *type* is *int*, because the return type of the method is *int*
    • Its *value* is the current value of b1’s balance
• You can use such an expression anywhere that an *int* is allowed – the compiler will complain if you violate this
  • *type checking* is a valuable aid to debugging!
Object Creations

• The construction of an object with `new` returns a reference to the newly created object, so this is also an expression.

• The expression

```java
new SimpleCanvas()
```

returns a reference to a `SimpleCanvas` object and so can be used anywhere that such a reference is valid.
Compound Expressions

- A *compound expression* is obtained by combining one or more simple expressions with an *operator*

  \[ 5 + 3 \]

  is a compound expression, obtained by combining the literals 5 and 3 with the operator +
  
  - The *value* of this expression is 8
  - The *type* of this expression is `int`

- We have already used one compound expression
  
  - `balance` is a simple expression
  - `amount` is a simple expression
  - `balance + amount` is a compound expression
Classes of Operators

• Java has a large number of operators
• Arithmetic Operators
  • take numeric expressions and perform arithmetic on them, producing numeric answers
• Relational Operators
  • take numeric, or other, expressions and make logical tests of equality, inequality etc, producing boolean answers
• Logical Operators
  • take boolean expressions and combine them using the rules of logic, producing boolean answers
• Miscellaneous Operators
  • other operators, acting on Strings, bit patterns etc.
## Arithmetic Operators

- We will concentrate on the *arithmetic* operators first
- They are as follows:

<table>
<thead>
<tr>
<th>Precedence</th>
<th>Operator</th>
<th>Operation</th>
<th>Association</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>+</td>
<td>Unary plus</td>
<td>Right-to-left</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>Unary minus</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>*</td>
<td>Multiplication</td>
<td>Left-to-right</td>
</tr>
<tr>
<td></td>
<td>/</td>
<td>Division</td>
<td></td>
</tr>
<tr>
<td></td>
<td>%</td>
<td>Remainder</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>+</td>
<td>Addition</td>
<td>Left-to-right</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>Subtraction</td>
<td></td>
</tr>
</tbody>
</table>
Association

- The association rules explain how to combine a sequence of operators of the same precedence.
- Consider evaluating the expression $100 / 10 / 5$.
- The / operator *associates* left-to-right so the value is $(100 / 10) / 5$.
- If it associated right-to-left, it would be $100 / (10 / 5)$.
- The association rules just follow normal mathematical usage, so there are no surprises.
Arithmetic Operators

• These behave (almost) as you would expect

<table>
<thead>
<tr>
<th>Expression</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>23+53</td>
<td>76</td>
</tr>
<tr>
<td>13*5</td>
<td>65</td>
</tr>
<tr>
<td>30/5</td>
<td>6</td>
</tr>
<tr>
<td>2.37 * 2</td>
<td>4.74</td>
</tr>
<tr>
<td>12 – 15</td>
<td>-3</td>
</tr>
</tbody>
</table>

• But what is $5 * 6 + 3 * 2 - 4 + 6 * 11$?
Precedence

• The compiler uses the precedence and association rules to determine the order of evaluation

\[ 5 \times 6 + 3 \times 2 - 4 + 6 \times 11 \]

becomes

\[ (5 \times 6) + (3 \times 2) - 4 + (6 \times 11) \]

because * has a higher precedence than + or –

• The resulting expression is calculated \textit{left to right}

\[ (((30 + 6) - 4) + 66) \]

• The programmer can \textit{explicitly} use parentheses if a particular order is required
Watch your types!

- Every expression has a *type*, which depends on the operators involved and the types of the *operands*
- In particular, the division operator `/` returns an `int` if its operands are `ints`
- So,
  - `7/5` yields `1`
  - `12/4` yields `3`, but so does `13/4` and `14/4`
  - `7.0/5` yields `1.4`, as does `7/5.0` or `7.0/5.0` because the expression is a floating point number
- The integer is obtained by *truncating* the expression
  - `100/51` yields `1`
Integer Division

- Java’s integer division is always a fertile source of difficult-to-trace bugs
  
  ```java
  double d = 8 / 5;
  ```

- After this statement the value of `d` is `1.0` and not `1.6`!

- Why?
  
  - *first* the expression `8 / 5` is evaluated, and as both the arguments are of type `int`, the answer is truncated to an int
  
  - *then* the value `1` is assigned to the `double` variable, and hence `d` has the value `1.0`

- This leads to some potentially confusing situations
  
  ```java
  2.0/3 * 6 is not equal to 2/3 * 6.0
  ```
The remainder operator

- The operator % returns the *remainder* in a division with two integer arguments

<table>
<thead>
<tr>
<th>Expression</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>10 % 3</td>
<td>1</td>
</tr>
<tr>
<td>12 % 4</td>
<td>0</td>
</tr>
<tr>
<td>13 % 7 % 4</td>
<td>2</td>
</tr>
<tr>
<td>13 % (7 % 4)</td>
<td>1</td>
</tr>
</tbody>
</table>

- An important Java *idiom* or *cliché* is the use of / and % together
  - If someone is 7000 days old, then how old are they?
    7000 / 365 years, plus 7000 % 365 days
The CodePad

Turn on the CodePad by using the View menu and selecting Show Code Pad

This is the CodePad which allows you to enter either expressions or statements and have them performed immediately.
Evaluate an expression

- Enter any expression in order to have it immediately evaluated.
- Here we discover that 15 % 11 has the value 4 and the type int.
- Notice that the CodePad knows you want to evaluate an expression because it has no semicolon.
Relational operators

- Relational operators are crucial in allowing a program to choose among different courses of action depending on the results of calculations.
- They are often used to determine a program’s flow of control, using `if` statements (and others).
  - “If the balance is less than zero, then print out an overdraft statement”

```java
if (balance < 0) {
    // print out the overdraft statement
}
```
Logical Operators

- Boolean expressions can be combined into compound expressions using operators for NOT, AND and OR

<table>
<thead>
<tr>
<th>Operator</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>a &amp;&amp; b</td>
<td>a AND b</td>
</tr>
<tr>
<td>a</td>
<td></td>
</tr>
<tr>
<td>!a</td>
<td>NOT a</td>
</tr>
</tbody>
</table>
Use of the logical operators

- The logical operators combine boolean expressions in the obvious way
- Suppose that `<bool1>` and `<bool2>` represent arbitrary boolean expressions

```
<bool1> && <bool2>
  is true if both `<bool1>` and `<bool2>` are true
<bool1> || <bool2>
  is true if either `<bool1>` or `<bool2>` are true
! <bool1>
  is true if `<bool1>` is false, and false if `<bool1>` is true
```
Relational Operators

- The six relational operators compare numeric values and return a boolean

<table>
<thead>
<tr>
<th>Operator</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>a == b</td>
<td>Is a equal to b</td>
</tr>
<tr>
<td>a != b</td>
<td>Is a not equal to b</td>
</tr>
<tr>
<td>a &gt; b</td>
<td>Is a greater than b</td>
</tr>
<tr>
<td>a &lt; b</td>
<td>Is a less than b</td>
</tr>
<tr>
<td>a &gt;= b</td>
<td>Is a greater than or equal to b</td>
</tr>
<tr>
<td>a &lt;= b</td>
<td>Is a less than or equal to b</td>
</tr>
</tbody>
</table>
Shortcut Evaluation

• Java uses *shortcut* evaluation of the logical operators
  • as soon as it knows the final answer it stops calculating

```java
if (balance > 2000000 || accountName.equals("Kerry Packer")) {
    // send grovelling letter
}
```

• There are two relational tests, one on `balance` and the other on `accountName`, combined with an *OR*
  • if the first test is true, then the entire boolean expression will be true regardless of the second test
  • thus, Java saves time by not even bothering to do the second test
Summary: Expression and Assignment Rules

- Assignment statement means “the value in variable on the LHS becomes equal to the value of expression on the RHS”
- An expression is a portion of Java code that is evaluated to yield a value
- However complicated the expression, it is evaluated from a tree of operations and values to give a final value

```
interest = Math.pow(bal, monthlyRate);
bal = bal + interest - monthlyPayment;
```