Variables and Expressions

1. Overview

The learning objectives for this lab are:

- To understand Java variables and use them in expressions
- To understand the specifics of operators in Java
- To acquire hands-on experience with variables and expressions by developing and running small programs

2. Variables

Variables are places in memory to store values. There are different kinds of variables, and every language offers slightly different characteristics.

- **Name**.
- **Data Type** specifies the kinds of data a variable an store. Java has two general kinds of data types.
  - 8 basic or *primitive* types (`byte`, `short`, `int`, `long`, `float`, `double`, `char`, `boolean`).
  - An unlimited number of *object* types (`String`, `Color`, `JButton`, ...). Java object variables hold a *reference* (pointer) *to* the object, not the object, which is always stored on the heap.
- **Scope** of a variable is who can see it. The scope of a variable is related program structure: eg, block, method, class, package, child class.
- **Lifetime** is the interval between the creation and destruction of a variable. The following is basically how things work in Java. Local variables and parameters are created when a method is entered and destroyed when the method returns. Instance variables are created by `new` and destroyed when there are no more references to them. Class (static) variables are created when the class is loaded and destroyed when the program terminates.
- **Initial Value**. What value does a variable have when it is created? There are several possibilities.
  - No initial value. Java **local variables have no initial value**. However Java compilers perform a simple flow analysis to ensure that every local variable is assigned a value before it is used. These error messages are usually correct, but the analysis is simple-minded, so sometimes you will have to assign an initial value even though you know that it isn't necessary.
  - User specified initial value. Java allows an assignment of initial values in the declaration of a variable.
  - Instance and static variables are given default initial values: zero for numbers, `null` for objects, and `false` for booleans.
- **Declarations are required**. Java, like many languages, requires you to *declare* variables -- tell the compiler the data type, etc. Declarations are good because they help the programmer build more reliable and efficient programs.
  - Declarations allow the compiler to find places where variables are misused, e.g., parameters of the wrong type. What is especially good is that these errors are detected at compile time. Bugs that make it past the compiler are harder to find, and may not be discovered until the program has been released to customers. This fits the *fail early, fail often* philosophy.
  - A declaration is also the perfect place to write comments describing the variable and how it is used.
  - Because declarations give the compiler more information, it can generate better code.

1.1. Local/Instance/Class Variables

There are three kinds of Java variables:

- **Local variables** are declared in a method, constructor, or block. When a method is entered, an area is pushed onto the *call stack*. This area contains slots for each local variable and parameter. When the method is called, the parameter slots are initialized to the parameter values. When the method exits, this area is popped off the stack and the memory becomes available for the next called method. Parameters are essentially local variables which are initialized from the actual parameters. Local variables are not visible outside the method.
- **Instance variables** are declared in a class, but outside a method. They are also called *member* or *field variables*. When an object is allocated in the *heap*, there is a slot in it for each instance variable value. Therefore an instance variable is created when an object is created and destroyed when the object is destroyed. Visible in all methods and constructors of the defining class, should generally be declared private, but may be given greater visibility.
- **Class/static variables** are declared with the `static` keyword in a class, but outside a method. There is only one copy per class, regardless of how many objects are created from it. They are stored in static memory. It is rare to use static variables other than declared `final` and used as either public or private constants.

Characteristic	**Local variable**	**Instance variable**	**Class variable**
Where declared	In a method, constructor, or block.	In a class, but outside a method. Typically `private`.	In a class, but outside a method. Must be declared `static`. Typically also `final`.
Use	Local variables hold values used in computations in a method.	Instance variables hold values that must be referenced by more than one method (for example, components that hold values like text fields, variables that control drawing, etc), or that are essential parts of an object's state that must exist from one method invocation to another.	Class variables are mostly used for constants, variables that never change from their initial value.
Lifetime	Created when method or constructor is entered. Destroyed on exit.	Created when instance of class is created with `new`. Destroyed when there are no more references to enclosing object (made available for garbage collection).	Created when the program starts. Destroyed when the program stops.
Scope/Visibility	Local variables (including formal parameters) are visible only in the method, constructor, or block in which they are declared. Access modifiers (`private`, `public`, ...) can not be used with local variables. All local variables are effectively private to the block in which they are declared.	Instance (field) variables can been seen by all methods in the class. Which other classes can see them is determined by their declared access: `private` should be your default choice in declaring them. No other class can see private instance variables. This is regarded as the best choice. Define	Same as instance variable, but are often declared `public` to make constants available to users of the class.

	No part of the program outside of the method / block can see them. A special case is that local variables declared in the initializer part of a `for` statement have a scope of the `for` statement.	getter and setter methods if the value has to be gotten or set from outside so that data consistency can be enforced, and to preserve internal representation flexibility. Default (also called package visibility) allows a variable to be seen by any class in the same package. `private` is preferable. `public`. Can be seen from any class. Generally a bad idea. `protected` variables are only visible from any descendant classes. Uncommon, and probably a bad choice.	
Declaration	Declare before use anywhere in a method or block.	Declare anywhere at class level (before or after use).	Declare anywhere at class level with `static`.
Initial value	None. Must be assigned a value before the first use.	Zero for numbers, false for Booleans, or null for object references. May be assigned value at declaration or in constructor.	Same as instance variable, and it addition can be assigned value in special static initializer block.
Access from outside	Impossible. Local variable names are known only within the method.	Instance variables should be declared `private` to promote information hiding, so should not be accessed from outside a class. However, in the few cases where there are accessed from outside the class, they must be qualified by an object (eg, myPoint.x).	Class variables are qualified with the class name (e.g., Color.BLUE). They can also be qualified with an object, but this is a deceptive style.
Name syntax	Standard rules.	Standard rules, but are often prefixed to clarify difference from local variables, eg with `my`, `m`, or `m_` (for member) as in `myLength`, or `this` as in `this.length`.	`static public final` variables (constants) are all uppercase, otherwise normal naming conventions. Alternatively prefix the variable with "c_" (for class) or something similar.

3. Expressions

Expressions are the basic way to create values. Expressions are created by combining literals (constants), variables, and method calls by using operators. Parentheses can be used to control the order of evaluation.

Types.
Every variable and value has a *type*. There are two kinds of types:
- **Primitive** types: byte, short, char, int, long, float, double, boolean.
- **Object** types: String and array types are builtin, but every class that is defined creates a new object type.

Literals - constants
There is a way to write values of many types (3, 3.0, true, 'a', "abc").

Variables
Every variable must be declared with a type. There are basically three different kinds of variables:
- Local variables in methods.
- Instance variables (often called fields) in objects.
- Class (static) variables in classes.

4. Operators

Operators are used to combine literals, variables, methods calls, and other expressions. Operators can be put into several conceptual groups.
- Arithmetic operators (+, -, *, /, %, ++, --)
- Comparison Operators (<, <=, ==, >=, >, !=)
- Boolean operators (&&, ||, !, &, !, ^, |, &)
- Bitwise operators (&, |, ^, ~, <<, >>, >>>)
- String concatenation operator (+)
- Other (`instanceof`, ?:)
- Assignment operators (=, +=, -=, *=, ...)

4.1. Order of evaluation

The order in which expressions are evaluated is basically left to right, with the exception of the assignment operators. It may be changed by the use of parentheses. See the expression summary for the precedence.

4.2. Comparison Operators

All the standard comparison operators work for *primitive values* (int, double, char, ...). The == and != operators can be used to compare object references, but see Comparing Objects for how to compare object *values*.

Operators
The result of every comparison is *`boolean`* (`true` or `false`).

operator	meaning
<	less than
<=	less than or equal to
==	equal to
>=	greater than or equal to
>	greater than
!=	not equal

Common Errors

`0 < x < 100`

Comparison operators can be used with two numbers. Although you can write 0 < x < 100 in mathematics, it is illegal in Java. You must write this as the and of two comparisons:

```
0<x && x<100
```

= instead of ==

Using the assignment operator instead of equality will produce a compiler error, which is easy to fix.

== with floating-point

Because floating-point numbers are not exact, you should always use >= or <= instead of ==. For example, because the decimal number 0.1 can not be represented exactly in binary, (0.1 + 0.1 + 0.1) is *not* equal to 0.3!

For C/C++ Programmers

The Java comparison operators look exactly the same as the C/C++ comparison operators. The difference is that the result type is boolean. Because of this, the common C error of using = instead of == is almost completely eliminated. Java doesn't allow operator overloading however, something that C++ programmers might miss

==, .equals(), and compareTo()

Equality comparison: One way for primitives, Four ways for objects

Comparison	Primitives	Objects
`a == b, a != b`	Equal values	Refer to the same object.
`a.equals(b)`	N/A	Compares values, if it's defined for this class, as it is for most Java core classes. If it's not defined for a (user) class, it behaves the same as ==.
`a.compareTo(b)`	N/A	Compares values. Class must implement the *Comparable<T>* interface. All Java classes that have a natural ordering implement this (String, Double, BigInteger, ...), but BigDecimal may not do what you expect. *Comparable* objects can be used by the Collections *sort()* method and data structures that implicitly sort (eg, TreeSet, TreeMap).
`compareTo(a, b)`	N/A	Compares values. Available only if the *Comparator<T>* interface has been implemented, which is not the case for the Java classes. Typically used to define a comparator object that can be passed to Collections sort() method or ordered data structures.

Comparing Object references with the == and != Operators

The two operators that can be used with object references are comparing for equality (**==**) and inequality (**!=**). These operators compare two values to see if they **refer to the same object**. Although this comparison is very fast, it is often not what you want.

Usually you want to know if the objects have the same *value*, and not whether two objects are a *reference* to the same object. For example,

```
if (name == "Mickey Mouse")   // ALMOST SURELY WRONG
```

This be true if `name` is a reference to the *same object* that `"Mickey Mouse"` refers to. This will probably be false if the String in `name` was read from input or computed (by putting strings together or taking the substring), even though `name` really does have exactly those characters in it.
Many classes (e.g., `String`) define the `equals()` method to compare the *values* of objects.

Comparing Object values with the `equals()` Method

Use the `equals()` method to compare object values. The `equals()` method returns a boolean value. The previous example can be fixed by writing:

```
if (name.equals("Mickey Mouse"))  // Compares values, not referernces.
```

Other comparisons - Comparable<T> interface

The `equals` method and `==` and `!=` operators test for equality/inequality, but do not provide a way to test for the relative values. Some classes (eg, String and other classes with a natural ordering) implement the *Comparable<T>* interface, which defines a `compareTo` method. You will want to implement *Comparable<T>* in your class if you want to use it with Collections.sort() or Arrays.sort() methods. The String class also provides case insensitive comparisons.

Common Errors: using == instead of `equals()` with Objects

When you want to compare objects, you need to know whether you should use `==` to see if they are the *same object*, or `equals()` to see if they may be a different object, but have the *same value*. This kind of error can be very hard to find.

4.3. Bitwise Operators

Java's *bitwise* operators operate on individual bits of **integer (int and long) values**. If an operand is **shorter than** an **int**, it is **promoted to int** before doing the operations.
It helps to know how integers are represented in binary. For example the decimal number 3 is represented as 11 in binary and the decimal number 5 is represented as 101 in binary. Negative integers are store in *two's complement* form. For example, -4 is 1111 1111 1111 1111 1111 1111 1111 1100.
The bitwise operators

Operator	Name	Example	Result	Description
a & b	and	3 & 5	1	1 if both bits are 1.
a | b	or	3 | 5	7	1 if either bit is 1.
a ^ b	xor	3 ^ 5	6	1 if both bits are different.
~a	not	~3	-4	Inverts the bits.
n << p	left shift	3 << 2	12	Shifts the bits of *n* left *p* positions. Zero bits are shifted into the low-order positions.
n >> p	right shift	5 >> 2	1	Shifts the bits of *n* right *p* positions. If *n* is a 2's complement signed number, the sign bit is shifted into the high-order positions.
n >>> p	right shift	-4 >>> 28	15	Shifts the bits of *n* right *p* positions. Zeros are shifted into the high-order positions.

Use: Packing and Unpacking

A common use of the bitwise operators (shifts with *ands* to extract values and *ors* to add values) is to work with multiple values that have been encoded in one int. Bit-fields are another way to do this. For example, let's say you have the following integer variables: age (range 0-127), gender (range 0-1), height (range 0-128). These can be packed and unpacked into/from one short (two-byte integer) like this (or many similar variations).

```
int   age, gender, height;
short packed_info;
. . .
// packing
packed_info = (short)((((age << 1) | gender) << 7) | height);
. . .
// unpacking
height = packed_info & 0x7f;
gender = (packed_info >>> 7) & 1;
age    = (packed_info >>> 8);
```

Use: Setting flag bits
Some library functions take an int that contains bits, each of which represents a true/false (boolean) value. This saves a lot of space and can be fast to process.

Use: Shift left multiplies by 2; shift right divides by 2
On some older computers it was faster to use shift instead of multiply or divide.

```
y = x << 3;        // Assigns 8*x to y.
y = (x << 2) + x;  // Assigns 5*x to y.
```

Use: Flipping between on and off with xor
Sometimes *xor* is used to flip between 1 and 0.

```
x = x ^ 1;      // Or the more cryptic x ^= 1;
```

In a loop that will change x alternately between 0 and 1.

Obscure use: Exchanging values with xor
Here's some weird code. It uses *xor* to exchange two values (x and y). This is translated to Java from an assembly code program, where there was no available storage for a temporary. Never use it; this is just a curiosity from the museum of bizarre code.

```
x = x ^ y;
y = x ^ y;
x = x ^ y;
```

Don't confuse && and &
Don't confuse `&&`, which is the short-circuit *logical and*, with `&`, which is the uncommon *bitwise and*. Although the bitwise *and* can also be used with boolean operands, this is extremely rare and is almost always a programming error.

5. Summary for expressions

Parentheses () have three uses:

1. **Grouping to control order of evaluation, or for clarity.**
2. **E.g., (a + b) * (c - d)**
3. **After a method name to enclose parameters. E.g., x = sum(a, b);**
4. **Around a type name to form a cast. E.g., i = (int)x;**

Order of evaluation

- Higher precedence are done before lower precedence.
- Left to right among equal precedence except: unary, assignment, conditional operators.

Abbreviations

i, j - integer (int, long, short, byte, char) values.
m, n - numeric values (integers, double, or float).
b, c - boolean; x, y - any primitive or object type.
s, t - String; a - array; o - object; co - class or object

Operator Precedence	

`. [] (args) post ++ --`
`! ~ unary + - pre ++ --`
`(type) new`
`* / %`
`+ -`
`<< >> >>>`
`< <= > >= instanceof`
`== !=`
`&`
`^`
`|`
`&&`
`||`
`?:`
`= += -= etc`
	Remember only unary operators
	* / %
	+ -
	comparisons
	&& ||
	= *assignments*
	Use () for all others

Arithmetic Operators	
The result of arithmetic operators is double if either operand is double, else float if either operand is float, else long if either operand is long, else int.	
++i	Add 1 to i *before* using the value in the current expression
--i	As above for subtraction
i++	Add 1 to i *after* using the value in the current expression
i--	As above for subtraction
n + m	Addition. *E.g.* 7+5 is 12, 3 + 0.14 is 3.14
n - m	Subtraction
n * m	Multiplication. *E.g.* 3 * 6 is 18
n / m	Division. *E.g.* 3.0 / 2 is 1.5 , 3 / 2 is 1
n % m	Remainder (Mod) after division of n by m. *E.g.* 7 % 3 is 1

Comparing Primitive Values
The result of all comparisons is boolean (true or false).
== != < <= > >=

Logical Operators	
The operands must be boolean. The result is boolean.	
b && c	Conditional "and". true if both operands are true, otherwise false. Short circuit evaluation. *E.g.* (false && anything) is false.
b || c	Conditional "or". true if either operand is true, otherwise false. Short circuit evaluation. *E.g.* (true || anything) is true.
!b	true if b is false, false if b is true.
b & c	"And" which always evaluate both operands (*not* short circuit).
b | c	"Or" which always evaluate both operands (*not* short circuit).
b ^ c	"Xor" Same as b != c

Conditional Operator

b?x:y	if b is true, the value is x, else y. x and y must be the same type.

Assignment Operators	
=	Left-hand-side must be an lvalue.
+= -= *= …	All binary operators (except && and ||) can be combined with assignment. *E.g.* a += 1 is the same as a = a + 1

Bitwise Operators	
Bitwise operators operate on bits of ints. Result is int.	
i & j	Bits are "anded" - 1 if both bits are 1. 5 & 3 is 1.
i | j	Bits are "ored" - 1 if either bit is 1. 5 | 3 is 7.
i ^ j	Bits are "xored" - 1 if bits are different. 5 ^ 3 is 6.
~i	Bits are complemented (0 -> 1, 1 -> 0)
i << j	Bits in i are shifted j bits to the left, zeros inserted on right. 5 << 2 is 20.
i >> j	Bits in i are shifted j bits to the right. Sign bits inserted on left. 5 >> 2 is 1.
i >>> j	Bits in i are shifted j bits to the right. Zeros inserted on left.

Casts	
Use casts when "narrowing" the range of a value. From narrowest to widest the primitive types are: byte, short, char, int, long, float, double. Objects can be assigned without casting up the inheritance hierarchy. Casting is required to move down the inheritance hierarchy (downcasting).	
(*t*)x	Casts x to type t

Object Operators	
co.f	Member. The f field or method of object or class co.
x instanceof co	true if the object x is an instance of class co, or is an instance of the class of co.
a[i]	Array element access.
s + t	String concatentation if one or both operands are Strings.
x == y	true if x and y refer to the same object, otherwise false (even if values of the objects are the same).
x != y	As above for inequality.
comparison	Compare object **values** with .equals() or .compareTo()
x = y	Assignment copies the *reference*, not the object.

6. Lab Tasks

Write short Java Programs to:

6.1. Compute the odds of a lottery win (6 out of 49, 5 out of 40).

6.2. Simulate the lottery draw (to get the random numbers needed use Math.random)

6.3. Print the numbers drawn in ascending/descending order, without using sorting or arrays. (Hint:: use packing/unpacking/extraction of bits stored in a long)

6.4. Encode a phrase given as a String, where the alphabet used is only Latin letters, spaces (blanks), commas and dots (i.e. less than 64 different symbols) as an arbitrary precision integer. Coding should preserve Unicode order (e.g. space will be coded as a zero, a comma as a 1, a dot as a 2, uppercase A as a 3, etc. – see the table in the previous Laboratory Guide Session.). Print the resulting integer.

6.5. Solve the dual problem: a number is given, print the phrase, using the same encoding.

6.6. Any even natural number, greater than 2 ca be written as a sum of two prime numbers – this is Goldbach's conjecture. Write a program to check this conjecture for numbers in the range from *m* to *n*. The parameters will be given as arguments to a stand-aloe application (arguments for the method **main**).