
Workbook 2

Introduction

In this workbook you will explore the basic types used for storing data in Java. You will also make use of the associated functions or *operators*. In the assessed exercise, you will create a new data structure which will be used in your implementation of Conway's Game of Life next week. Please make sure you pick up the associated starred exercise for this week as well as the questionnaire on the work last week. *Please complete the questionnaire before starting this workbook so that your ticker can collect it from you.*

Important

The recommended text book for this course is *Thinking in Java* by Bruce Eckel. You can download a copy of the 3rd Edition for free from Bruce's website:

<http://www.mindview.net/Books/TIJ/>

Remember to check the course website regularly for announcements and errata:

<http://www.cl.cam.ac.uk/teaching/0809/ProgJava>

If you finish this exercise early, you may also wish to check the website for a list of additional projects.

Primitive types and operators

In Java we use *variables* to allocate computer memory to store either: (i) a value of a basic, or *primitive*, type or (ii) a reference to an Object. This workbook focuses on how to create and manipulate variables of a primitive type; you will investigate Objects and references to Objects in future weeks. By the end of this term, you should be able to list and use all the primitive types and explain the behaviour of the operators which act upon them.

Variables in Java are different to values in ML. Values in ML are normally immutable, in other words, once you create them you cannot modify them. In Java it is much more common to program with mutable "values" than immutable ones. A mutable "value" in Java can be modified, possibly many times, after it has been created. Consequently such mutable "values" in Java are called *variables* in deference to the fact that Java programs commonly modify the contents of a variable during execution.

There are eight primitive storage types in Java which can be used to store values representing truth or falsity, integers of varying size, or floating point numbers. Table 1, "The primitive types" shows the names, sizes and storage capacity of these types. There is no need to memorise the contents of the table at this stage, simply get an idea of what's there and refer back to it as you need to. The meaning of each of the columns in the table is as follows:

Name	The sequence of characters in a Java source file used to refer to the type.
Size (bits)	The amount of computer memory allocated to store the data in a variable.
Minimum and maximum	The larger the size (bits) of the type, the greater the number of unique values it can store. The Floating-Point Computation course will give you more details about <code>float</code> and <code>double</code> types. These are defined by the international standard IEEE 754.
Example literals	A literal expresses a value which may be stored in a variable. Some examples are given in the table and we'll go into more detail over the next few pages.

name	size (bits)	minimum	maximum	example literals
byte	8	-128	127	
char	16	0	$2^{16}-1$	'A'
short	16	-2^{15}	$2^{15}-1$	
int	32	-2^{31}	$2^{31}-1$	52, 0x34, or 064
float	32	IEEE 754	IEEE 754	0.0F, -1e4F
long	64	-2^{63}	$2^{63}-1$	4294967296L
double	64	IEEE 754	IEEE 754	3.14e-1D, 2.7, or 1e4
boolean	N/A	N/A	N/A	false or true

Table 1. The primitive types

We can create a variable of a particular primitive type by writing the name of the type followed by a memorable name. This can be done either within a class definition (in which case it is also called a *field*) or within a method such as the main method we saw last week. Below is a simple example which declares two variables, both of type `int`:

```
class VariableExample {
    public static int fieldVar;
    public static void main(String[] args) {
        int methodVar;
    }
}
```

You should choose the names of variables carefully, partly because a good name will improve readability (and therefore is strongly encouraged from a software engineering perspective), but also because some names cannot be used to name variables in Java. The following names are *reserved* words in Java:

```
abstract    assert    boolean    break    byte    case
catch       char      class      const    continue default
do          double   else       enum     extends  false
final       finally   float      for       goto     if
implements  import    instanceof int      interface long
native      new       null       package  private  protected
public      return    short      static   strictfp super
switch     synchronized  this      throw    throws   transient
true       try       void       volatile while
```

using any of these words as a name for a variable will result in a compile error. If you encounter this problem, the solution is to change the name of the variable. You will learn what many of these reserved words do in this course, but not all. Some of the advanced features of Java will not be taught until the second year.

Integral types

Java has five primitive types which support storage of a range of integral values: `byte`, `char`, `short`, `int` and `long`.

To explore the primitive integral types you will need to write simple Java programs. You will also find it helpful to refer back to the workbook from the previous week; if you have not brought the book with you then you can view it on the course website. To get you started, follow the instructions below, making sure you replace `crsid` with your username. For example, if your username is `arb33` you should use `arb33` in place of `crsid`.

- Create a new directory called `uk/ac/cam/crsid/primitive` in a sensible location alongside your other Java practical work.
- Create a new Java source file called `PrimitiveInt.java` inside the new directory you created in the previous step.
- Enter the following Java program into the file called `PrimitiveInt.java`:

```
package uk.ac.cam.crsid.primitive;

class PrimitiveInt {
    public static void main(String[] args) {
        int i = 1;
        i = i + 1;
        System.out.println(i);
    }
}
```

- Compile your program using the `javac` program.
- Run your program using the `java` program.

If you have followed the steps correctly, your program should print out the number 2. Make sure you work through the above even if it seems obvious! You will need to modify this program later.

Numeric operators

The first set of operators on primitive types we will consider are the numeric operators. These perform numeric or mathematical operations with values of a primitive type and are written as follows: `+` (addition), `-` (subtraction), `*` (multiplication), `/` (division), `%` (modulus or remainder). These are *infix binary* operators. In this context, the word binary means that the operator takes two *operands* and infix means that the operator is written between the arguments. For example, `a + b` uses the infix binary operator `+` to add the values stored in the variables `a` and `b` together; `a` and `b` are the left-hand and right-hand operands to the operator `+`.

The modulus operator (`%`) might be unfamiliar to you. The operation `a % b` calculates the remainder after `a` has been divided by `b`. This can be restated as follows: `a % b` finds a number `x` such that `a - x` is exactly divisible by `b`. An alternative way of thinking about the modulus operation uses clock arithmetic: the position of the hour hand twenty-six hours after midnight can be calculated as `26 % 12`, which of course yields the answer two because twenty-six hours after midnight, the hour hand will be located at position two on the clock face. The answer to `26 % 5` can be found by considering a clock which completes a full revolution every 5 hours (rather than the usual 12) and consequently has only five hours marked on the face. Figure 1, "Clock arithmetic" contains a graphical illustration of `26 % 12` and `26 % 5`.

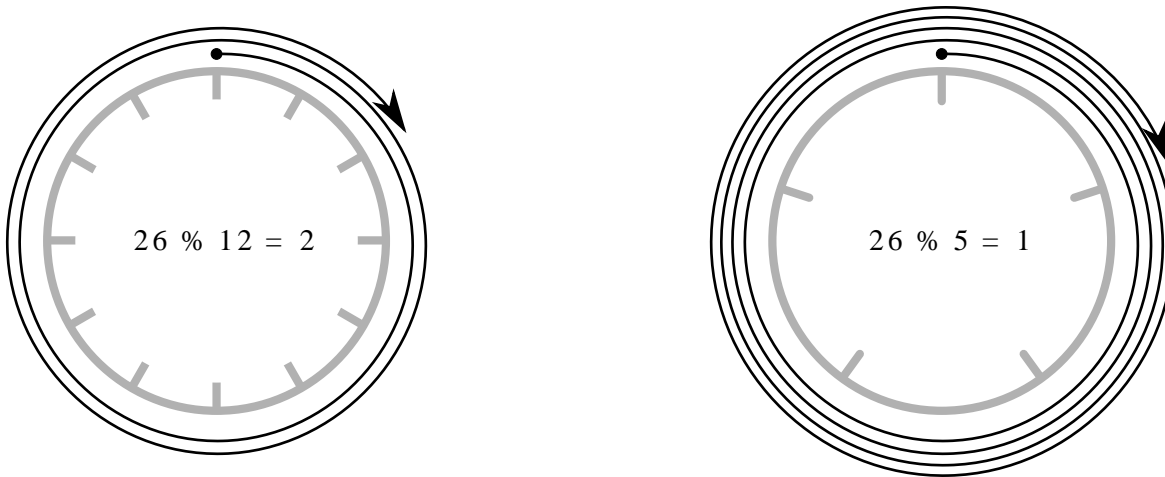


Figure 1. Clock arithmetic

A unary operator is an operator which takes a single argument. The unary operator `-` (minus) is used to negate a literal or the value stored in a variable. For example, `-a` changes the sign of the value stored in the variable `a`.

Every operator in Java produces a result which can then be used as a left-hand or right-hand operand with another operator. The order in which operands are grouped depends on operator precedence which is reasonably intuitive, but does have the potential to catch you out occasionally. You can override the default operator precedence by using round brackets. For example, the expression `1 + 2 * 3` produces the result 7. If you want the addition to take place before the multiplication you can write `(1 + 2) * 3`, giving the result 9.

Numeric operators

Write answers to all of the questions found in this workbook into a plain text file called `answers.txt`. You will need to include your `answers.txt` file in your submission for your second Java Tick. Further information about the submission of the tick can be found at the end of this workbook.

1. In the context of operators in Java what do the words *unary*, *binary*, *infix* and *operand* mean?

You should make modifications to your copy of `PrimitiveInt` answer the following questions.

2. What does `2147483624 + 40` produce? Why?

3. The result of `a / b` is always rounded towards zero if `b` does not exactly divide into `a`. Give example values for both `a` and `b` which demonstrate that the result of an integer division in Java rounds towards zero.

4. What does the modulus operator do if the first, second or both arguments are negative numbers?

Numeric operators with assignment

In the `PrimitiveInt` program, the assignment operator (`=`) is used to assign a new value to the variable `i`. The assignment operator is another example of a binary infix operator in which the left-hand operand is the name of a variable, and the right-hand operand is a value which the variable is updated to store. As mentioned before, every operator in Java produces a result, and the result of evaluating the assignment operator is the value of the right-hand operand. For example, the expression `"i = 1"` has the value `1` as its result.

Assignment operator

5. Modify `PrimitiveInt` to demonstrate that `a = b` returns `b` as its result. Include your modified or additional line(s) in `answers.txt`.

The next operators to consider are the unary increment and decrement operators written as `++` and `--` respectively. They combine together both an assignment and an addition operation, and the position of the increment and decrement operators with respect to the variable name determines the value returned as a result.

If the increment operator is written before the name of the variable, then it is more accurately called the *pre-increment* operator. For example, given a variable `i`, the pre-increment operator is written `++i`. The pre-increment operator is equivalent to writing the expression `i = i + 1`, and therefore `++i` increases the value stored in `i` by one and also returns the increased value as its result.

In contrast, if the increment operator is written after the name of the variable, then this operator is more accurately called the *post-increment* operator. For example, for the variable `i`, the post-increment operator is written `i++`. The post-increment operator increments the value stored in `i` by one, but the result returned by the operator is the value stored in `i` *before* it was incremented.

The pre-decrement and post-decrement operators behave similarly to the pre- and post-increment operators except that the value stored in the variable is reduced by one rather than increased by one.

Add the following code to the `main` function in your copy of `PrimitiveInt`

```
int j = 10;
System.out.println(j--);
System.out.println(j);
```

6. What does the code snippet above print out? Why?

7. If you replace `j--` with `--j` in the above code snippet, what does the program print out?

Java also provides shorthand forms for all numeric operators when combined with assignment. For example, `i = i + 10` can be shortened to `i += 10` and `j = j * k` can be shortened to `j *= k`. A summary of the numeric and assignment operators is shown in Table 2, "Numeric operators and assignment".

<code>+ - / * %</code>	Binary infix numeric operations
<code>-</code>	Unary negation
<code>++ --</code>	Post-increment, pre-increment, post-decrement, pre-decrement
<code>+= -= *= /= %=</code>	Shorthands for variable updating

Table 2. Numeric operators and assignment

Bitwise operators

In order to understand the next set of operators you need to understand how integer numbers are stored in memory as binary numbers. If you have not seen binary numbers before, this section might take you some time to work through; paying careful attention to binary numbers now will pay dividends later.

Counting with decimal, binary and hexadecimal

Binary numbers can be explained by analogy to "normal" numbers. The number system used on a daily basis today is called the decimal number system, sometimes called *base 10 arithmetic*. The decimal system requires 10 unique digits (0, 1, 2, 3, 4, 5, 6, 7, 8 and 9) to write down a number. For example, the number 6409 has 6 thousands (10^3), 4 hundreds (10^2), no tens (10^1) and 9 ones (10^0); hence 6409 can be written as $6 \times 10^3 + 4 \times 10^2 + 0 \times 10^1 + 9 \times 10^0$.

Binary numbers use base 2 arithmetic and consequently, there are only two digits in binary (0 and 1). Binary arithmetic is often used in computers because it simplifies the hardware design. The binary number 1011 has one eight (2^3), no fours (2^2), one two (2^1) and one one (2^0); hence the number 1011 can be written as $1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$, or 11 in decimal. In the preceding example, the binary number was written so that the leftmost position represented the binary value with the highest power of two and the rightmost position represented the least. This is identical to our "normal" decimal number system. For binary numbers we refer the leftmost position as the *most significant bit* and the rightmost position as the *least significant bit*.

A binary number can be converted into its decimal equivalent by adding up all the powers of 2 where there is a corresponding 1 in the binary representation. In other words, if there is a 1 at position i in the number then you include 2^i in the summation used to calculate the equivalent decimal number. By convention, the least significant bit is found at position 0. Therefore the most significant bit position in an n -bit binary number is $n-1$.

8. What is the decimal equivalent of the following binary numbers: 1011, 1, 100001
9. What is the largest decimal value that can be represented with n binary bits?

Hexadecimal notation is used to represent counting in base 16. In this notation we have 16 different digits (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E and F). In Java, a literal value is written in hexadecimal by prefixing the number with the characters `0x` (a zero and lowercase x) in order to avoid confusion with decimal numbers. For example, the hexadecimal number `0x1FAD` has a single four thousand and ninety six ($4096 = 16^3$), fifteen (written as `F`) two hundred and fifty sixes ($256 = 16^2$), ten (written as `A`) sixteens (16^1) and thirteen (written as `D`) ones (16^0). In other words, `0x1FAD` can be written as $1 \times 16^3 + 15 \times 16^2 + 10 \times 16^1 + 13 \times 16^0$.

Hexadecimal numbers are popular with programmers because and they are easily converted to binary numbers with simple mental arithmetic and offer a more compact representation than writing a binary number directly. They also allow us to write numbers which spell out (rude) words. Java also supports octal numbers (base 8) and a literal value can be written in octal by prefixing the number with a leading zero. For example, the octal number `077` has the value 63 in decimal.

10. Produce a table which contains the decimal numbers from 0 to 15 in the first column with their binary, octal and hexadecimal equivalents in the second and third columns.
11. What is the value of the hexadecimal number `0x1FAD` in decimal, octal and binary?
12. Describe in words how to convert a hexadecimal number into a binary number.

The bitwise logical operators

The bitwise logical operators perform logical operations on the binary representation of integers. For each of the n bits in the number, the bit at position i in the left-hand operand is combined with the bit in position i in the right-hand operand; these two bits are used to produce the bit at position i in the result. The operators are as follows

& (bitwise AND)	The output bit is set only if the corresponding bits in the right-hand operand <i>and</i> in the left-hand operand are set;
(bitwise OR)	The output bit is set if the corresponding bits in left-hand operand <i>or</i> in the right-hand operand are set;
^ (bitwise XOR)	The output bit is set if the corresponding bit in the left-hand operand is <i>different</i> to the bit in right-hand operand.

For example, 23 & 24 gives the result 16, or in binary, 10111 *AND* 11000 equals 10000.

13. What are the values of the following expressions? You may give answers in decimal or hexadecimal.

i. $85 \ \& \ 0x2B$

ii. $0x55 \ | \ 0x2B$

iii. $0105 \ \wedge \ 0x2B$

The shift operators

The shift operators are used to move the bits in a number left (towards the most significant bit) or right (towards the least significant bit). They are written as << and >> for left and right shift respectively. For example, the expression $45 \gg 3$ shifts the bits representing the value 45 (written in binary as 101101) right three places, resulting in the value 5 (written in binary as 101).

14. Find the value of x and y such that

i. $0x55 \gg x$ equals 5,

ii. $0x55 \ll y$ equals 10880.

15. Explain why $1 \ll n$ gives the value 2^n .

16. Combine the right shift and bitwise-and operators to give an expression which evaluates to 1 if a variable x has its fifth bit set, and 0 otherwise.

Negative numbers

The binary number system described so far cannot represent negative numbers. Most modern computers use a method called *2's complement* which is able to represent both positive and negative numbers. In this scheme the most significant bit in an n -bit binary number represents $-2^{(n-1)}$ whilst the remaining bits at position i (where $0 \leq i < n-1$) represent the positive values 2^i as in earlier sections. For example, the 8-bit binary number 10001100 is decoded as minus one hundred and twenty eight ($-128 = -2^7$) plus eight ($8 = 2^3$), plus four ($4 = 2^2$); in other words $-2^7 + 2^3 + 2^2 = -116$. This representation might seem a little odd, but it is appealing from the hardware design point of view (understanding why this is the case is beyond the scope of this course).

The right-shift operator >> performs *sign extension*, which means that the value of the most significant bits added during the right-shift operation are filled with the value of the most significant bit rather than with zeros. The alternative right-shift operator >>> performs a right-shift *without* sign extension, and therefore the top bits are always filled with zeros.

There is only one unary bitwise operator in Java, tilde (~), which negates every bit found in its argument; in other words it, swaps 1 for 0 and 0 for 1. A table summarising the bitwise operators is shown in Table 3, "Numeric operators and assignment"

17. What are the bit patterns for the most positive and most negative numbers that can be represented in 2's complement with n bits?

18. What is the result of calculating the following values? Why?

i. `-1>>1`

ii. `-1>>>1`

iii. `~0x55`

19. Describe how to use the xor operator (^) to achieve the same effect as the unary bitwise operator (~).

<code>& ^</code>	Bitwise binary infix operators and, or and xor
<code>~</code>	Bitwise unary negation
<code><< >> >>></code>	Bitwise left and right shift with and without sign-extension
<code>&= = ^= <<= >>= >>>=</code>	Shorthand notation for updating the value held in a variable

Table 3. Numeric operators and assignment

Literal values

As you have seen earlier in the workbook, literal values for `int` can be written in one of three ways:

- in decimal as you might expect (e.g. 52),
- in octal by prefixing the literal with 0 (e.g. 064), or
- in hexadecial by prefixing the literal with 0x (e.g. 0x34).

You can write literal values for `long` similarly, except that you must postfix the literal with an `L` (i.e. 52L, 064L and 0x34L). It's not possible to write a literal value of type `byte` or `short`. If you want to store a literal into a variable of type `short` or `byte` you can do so by assigning an `int` literal which will be converted to a value of the correct type automatically. For example,

```
short myShort = 12;
```

The `char` type is 16-bits wide and is normally used to store values representing characters, but can also be used to store and retrieve integral values. Java supports unicode,¹ making programs which support non-latin character sets easy to write. You can embed unicode characters in literal text strings inside Java source code, and even use unicode characters for variable, field, method or class names. You can specify literal values for `char` variables by:

- placing the character that you want inside single quotes (e.g. 'A'),
- providing a unicode value inside single quotes (e.g. '\u0065'), or
- providing an integer value (e.g. 65).

¹<http://en.wikipedia.org/wiki/Unicode>

We will explore characters and unicode further when we explore Java Strings in more detail later in the course.

Floating-point types

The storage of floating point values in `float` and `double` types is based on the IEEE 754 standard. Internally, floating point values are represented by a sign bit, a mantissa and an exponent, which means these types can store a much larger range of values at the expense of reduced precision. Therefore it is natural to express values in scientific notation such as 1.23×10^{-42} , which can be written in Java as `1.23e-42`. In Java all literal floating point values are of type `double` unless explicitly postfix with `F` in which case they are of type `float`.

Because the precision of values stored in variables of type `float` is poor, you should always use a `double` type unless you have undertaken a careful analysis to ascertain that a `float` will provide sufficient accuracy. Such floating point issues are beyond the scope of this course and will be covered in more detail in the Floating Point Computation course.

Boolean type

The `boolean` type does not have any explicit storage size assigned to it: it can only store one of two literal values, `true` or `false`.

Boolean operators

There are six binary comparison operators in Java which take two primitive types `a` and `b` and produce a value with a `boolean` type as a result. The value (`false` or `true`) depends on the values of `a` or `b` and the choice of operator:

- `a` equals `b` (written `a==b`),
- `a` does not equal `b` (`a!=b`),
- `a` is less than `b` (`a<b`)
- `a` is less than or equal to `b` (`a<=b`)
- `a` is greater than `b` (`a>b`)
- `a` is greater than or equal to `b` (`a>=b`)

Expressions or values of type `boolean` can be combined together using the binary logical operators for `or` (`|`) and `and` (`&`) as well as the unary operator for `not` (`!`). Remember that whilst the operators `&` and `|` look identical to the bitwise operators for AND and OR, they take values of a different type, and perform completely different actions!

Here are some examples of expressions using a variety of operators mentioned so far:

```
int a = (1+2)*3;           //9
int b = 1+2*3;            //7
boolean t1 = a == b;      //false
boolean t2 = b >= a+2;    //true
boolean t3 = a != b & !(a % 4 > 1); //true
boolean t4 = t1 | t2 & t3; //false
```

Java also has two short-circuit boolean operators written as `&&` and `||` which behave identically to `&` and `|` respectively except as outlined in the following text. If the left operand of `&&` evaluates to `false` then the right operand is *not evaluated*; similarly if the left operand of `||` evaluates to `true` then the right operand is *not evaluated*. This is generally sensible, since it does not affect the result of the expression and saves the computer from evaluating the contents of the right operand. The only exception to this is if the right operand has a *side-effect* such as updating the contents of a variable, in which case you should use the normal boolean operator not the short-circuit one.

Expressions

20. Which of the following expressions evaluate to `true`?

- i. `1 == 2`
- ii. `true != (!(12 == 3))`
- iii. `('A' != 'a') & (10 == 10L)`
- iv. `(7 << 3) > 21`

Comments

You will often want to provide comments in your code to document how a particular piece of code works, or provide extra information to other programmers who might read or modify your code. You can write comments in Java in one of two ways:

```
/* A comment which can span
   multiple lines */
```

```
//A comment which continues until the end of the line
```

A multi-line comment starts with `/*` and finishes with `*/`; multi-line comments cannot be nested (in other words, a comment cannot contain the characters `/*` or `*/` within it. A single line comment starts with `//` and continues until the end of the line.

If you are writing a large application or library, you will probably want to write more extensive documentation to help other software developers use the library or maintain your software. Java supports *javadoc* to do this. Javadoc is a command line tool which examines your source code, looking for multi-line comments which start with `/**` and uses the information held within the comment as well as its location in the source file to generate HTML documentation for the software project. Javadoc is beyond the scope of this course, although the interested reader who wants to know more may like to take a look at Sun's documentation.²

²<http://java.sun.com/j2se/javadoc/writingdoccomments/>

Java Tick 2

A Java `long` is an integer with 64 bits. Your task is to write a class called `PackedLong` which uses the 64 bits inside the java `long` integer to store 64 boolean values. We will make use of the `PackedLong` class next week as the basis for our Game of Life implementation. You should create a source file called `PackedLong.java` inside the directory structure `uk/ac/cam/crsid/tick2/` where `crsid` is your username. Place the following code inside `PackedLong.java`:

```
package uk.ac.cam.crsid.tick2; //TODO: replace crsid with your username

public class PackedLong {

    /*
     * Unpack and return the nth bit from the packed number at index position;
     * position counts from zero (representing the least significant bit)
     * up to 63 (representing the most significant bit).
     */
    public static boolean get(long packed, int position) {
        // set "check" to equal 1 if the "position" bit in "packed" is set to 1
        long check = //TODO: complete this statement

        return (check == 1);
    }

    /*
     * Set the nth bit in the packed number to the value given
     * and return the new packed number
     */
    public static long set(long packed, int position, boolean value) {
        if (value) {
            // TODO: complete this
            // update the value "packed" with the bit at "position" set to 1
        }
        else {
            // TODO: complete this
            // update the value "packed" with the bit a "position" set to 0
        }
        return packed;
    }
}
```

The `PackedLong` class does not contain a special "main" method. Consequently this means that it cannot be executed directly using the `java` command line program. Do not add a main method to this class; instead create two extra Java source files inside the same directory structure. Save the following program into `uk/ac/cam/crsid/tick2/TestBit.java`:

```
package uk.ac.cam.crsid.tick2; //TODO: replace crsid with your username

public class TestBit {
    public static void main(String[] args) throws Exception {
        long currentValue = new Long(args[0]).longValue();
        int position = new Integer(args[1]).intValue();
        boolean value = PackedLong.get(currentValue, position);
        System.out.println(value);
    }
}
```

Save the following program into `uk/ac/cam/crsid/tick2/SetBit.java`:

```
package uk.ac.cam.crsid.tick2; //TODO: replace crsid with your username

public class SetBit {
    public static void main(String [] args) throws Exception {
        long currentValue = Long.parseLong(args[0]);
        int position = Integer.parseInt(args[1]);
        boolean value = Boolean.parseBoolean(args[2]);
        currentValue = PackedLong.set(currentValue,position,value);
        System.out.println(currentValue);
    }
}
```

Once you have written your `PackedLong` implementation, you should be able to test your code using the two supplied programs.

The `TestBit` program accepts two arguments on the command line. The first argument is a `long` value which represents the 64 boolean bits, and the second argument tells the `TestBit` program which boolean value to print out. For example, executing

```
java uk.ac.cam.crsid.tick2.TestBit 85 4
```

should produce the answer `"true"`. If it does not, your implementation of `PackedLong` is incorrect. The `SetBit` program accepts three arguments on the command line. The first is the `long` value which represents the 64 boolean bits, the second argument tells `SetBit` which bit to set, and the third argument tells `SetBit` whether the chosen bit in the new value should be set to 1 (to represent `true`) or 0 (to represent `false`). For example, executing

```
java uk.ac.cam.crsid.tick2.SetBit 85 2 false
```

should produce the answer `"81"`. If it does not, then your implementation of `PackedLong` is incorrect.

Testing your implementation with the two examples above will probably not be sufficient to locate all the errors in your implementation of `PackedLong`. Therefore you should produce a more complete list of test cases, paying particular attention to sign extension. Once you are happy with your program you should submit a jar file named `crsid-tick2.jar` to `ticks1a-java@cl.cam.ac.uk` for testing, remembering to replace `crsid` with your username. Your jar file should contain the following files:

```
META-INF/
META-INF/MANIFEST.MF
uk/ac/cam/crsid/tick2/SetBit.class
uk/ac/cam/crsid/tick2/SetBit.java
uk/ac/cam/crsid/tick2/TestBit.class
uk/ac/cam/crsid/tick2/TestBit.java
uk/ac/cam/crsid/tick2/PackedLong.class
uk/ac/cam/crsid/tick2/PackedLong.java
uk/ac/cam/crsid/tick2/answers.txt
```

Note: in the first workbook you built a jar file with a specified entry point in order to tell the Java runtime program how to execute the program within the jar file. There are two candidates for the entry point in this tick: `TestBit` and `SetBit`. Please make `TestBit` the entry point for your jar file.