# Questions and Exercises

*These questions and exercises is an opportunity to see what you've learnt from the lecture as well as practice the new things we've been talking about. In other words, these questions and exercises are completely optional but it's recomended to do them. In the end of the document you will find the answers to the questions as well as possible solutions to the exercises, note that one can solve an exercise in different ways. There will also be some suggestions about what one could code if one want to continue with some more advanced things. These suggestions will not come with a possible solution and might include things that haven't been covered in the lecture.*

**Question 1**
What's the result of the following bitwise operator? Calculate it manually.

$10001100_2$ & $01101011_2$

**Question 2**
What's the result of the following bitwise operator? Calculate it manually.

$40_{10} \mid 12_{10}$

**Question 3**
What the result of the following bitwise operators and bit shifts? Calculate it manually.

$(00000001_2 << 4_{10}) \wedge (01101100_2 >> 2_{10})$

**Question 4**
What's the difference between the right bit shift >> and the right bit shift >>>?

**Exercise**
Create a class that stores 8 different bytes (with values -128 to 127) in one long. One should be able to access and modify all the bytes.

**Further explorations**
Do the same thing as in the exercise but allow the user to define how many bits each integer should store. For instance the user might set it to use 2 bits and therefore you'll be able to store 32 integers. Or the user might set it to 16 and therefore you'll be able to store 4.

# Answers and solutions

*By **V**idar **Swe**nning*

**Answer to Question 1**
$10001100_2$ & $01101011_2 = 0001000_3$

**Answer to Question 2**
$40_{10} = 00101000_2$
$12_{10} = 00001100_2$
$00101000_2 \mid 00001100_2 = 00101100_2$

**Answer to Question 3**
$00000001_2 << 4_{10} = 00010000_2$
$01101100_2 >> 2_{10} = 00011011_2$
$00010000_2 \wedge 00011011_2 = 00001011_2$

**Answer to Question 4**
When shifting bit one has to ”refill” the value with new bits. When you do so with left shifts you just add zeroes to fill the empty spots. When you use the right shift >>> it will do the same, add zeroes. However if you use the right shift >> it will add bits depending on the signed bit. So if the signed bit is a 1 it will add ones, if it is a 0 it will add zeroes.

**Possible solution to Exercise**

```
public class ByteGroup {

   public static void main(String[] args) {
      //just for testing

      ByteGroup group = new ByteGroup();
      group.setByte(0, (byte)-128);
      group.setByte(1, (byte)3);
      group.setByte(2, (byte)-15);
      group.setByte(3, (byte)26);
      group.setByte(6, (byte)2);
      group.setByte(7, (byte)3);
      group.setByte(6, (byte)-4);

      for (int i = 0; i < 8; i++) {
         System.out.println(i + ": " + group.getByte(i));
      }
   }

   private long data;

   public byte getByte(int id) {
      return (byte)((data & (255L << (id * 8L))) >> (id * 8L));
   }
```

```
    public void setByte(int id, byte val) {
        data &= ~(255L << (id * 8L));
        data |= ((long)val & 255) << (id * 8L);
    }

    public long getData() {
        return data;
    }

    public void setData(long data) {
        this.data = data;
    }

}
```