Java-Success.com

Industrial strength Java/JEE Career Companion for those who want to go places

01: Java data types interview Q&A

Posted on October 1, 2014 by Arulkumaran Kumaraswamipillai — No Comments ↓

Q1. How would you go about choosing the right data types for your application?
A1. Java is what is known as a strongly typed language. This means Java only accepts specific values within specific variables or parameters. Some languages, such as JavaScript, PHP, and Perl are weakly typed languages.

1. Know the data limits to prevent any data overflow

600+ Full Stack Java/JEE Interview Q&As ♥Free ♦FAQs

```
  byte   →   short   →   char   →   int   →   long   →   float   →   double
(1 byte)   (2 bytes)   (2 bytes)  (4 bytes)  (8 bytes)  (4 bytes)    (8 bytes)
```

Java primitive data types

```
int bad = 2100000000;     // Close to int max valu
long good = 2100000000L;

long badAgain = 90000000000000000000L;     // Close
BigInteger goodAgain = BigInteger .valueOf(900000

```

2. Prefer immutable wrapper objects to primitives.

Each primitive data type has a corresponding wrapper class like *Integer*, *Long*, *Character*, *Float*, *Double*, etc. There are 8 primitive variables and as many wrapper objects. In Java 5, additional wrapper classes like *AtomicInteger*, *AtomicLong*, *AtomicBoolean* and *AtomicReference* were introduced to provide atomic operations for addition, increment, and assignment. These additional classes are mutable and cannot be used as a replacement for regular immutable wrapper classes.

Q2. What are wrapper classes, and why do you need them?
A2. The **wrapper classes** are a good example of the **decorator design pattern**. They decorate the primitive values, and enhance their behavior by providing immutability, atomicity, null checking, etc.

1) Wrapper objects can be **initialized to null**. This can't be done with primitives. Many programmers initialize numbers to 0 or -1 to signify default values, but depending on the scenario, this may be incorrect or misleading.

2) Wrapper objects are very useful for optional data. Databases almost always have a significant fraction of their fields as optional (that is, as possibly NULL). In addition, the

forms submitted in Web applications can contain optional parameters. Both NULL database fields and missing request parameters naturally map to null object references. With primitives, there is no such natural mapping.

3) Wrapper objects will also set the scene for a *NullPointerException* when something is being used incorrectly, which is much more programmer-friendly as it fails fast than some arbitrary exception buried down the line. Preferably, check for null early on in the method and report it immediately where applicable to adhere to the fail fast principle.

4) The wrapper objects are immutable, hence **inherently thread-safe**. Other threads can only read the values set by the thread that initialized this object.

5) When you create wrapper objects, use the valueOf( ) static factory method for efficiency.

```
Integer i2 = new Integer(5);            //first a
Integer i1 = Integer.valueOf(5);        //2nd app

```

The second approach is in fact an implementation of the flyweight design pattern.

Q. When to prefer primitives?
A. Primitives are faster to create and use than wrapper objects. Wrapper objects need to be auto-unboxed before use. Thus there is an extra step for the JVM to perform. For example, in order to perform arithmetic on an *Integer*, it must first be converted to an int before the arithmetic can be performed. In many business applications this rarely matters unless you were writing something very number-crunching or profiling indicates that the auto-boxing is a performance or memory issue in a particular part of your code as it is executed very frequently.

Anti-pattern: Watch out for premature-optimization anti-pattern where you are tempted to code for a perceived performance gain and sacrificing good design and maintainability.

Q3. When working with floating-point data types, what are some of the key considerations?
A3.

1. Never compare float or double with "==" or != operator

```
//endless loop -- don't compare float or double f
for (float f = 5f; f != 10.0; f += 0.1) {
     System.out.println(f);
}
```

2. Use long, int, or *BigDecimal* for storing money, and performing monetary calculations.

Floating point data types like float, double, Float, or Double can result in inaccurate results. use either the BigDecimal or int/long representing the value in its lowest units like cents.

```
private void calculateTotalAccurately1(float uni
        BigDecimal total = BigDecimal.ZERO;
        //use the right constructor
        BigDecimal uc = new BigDecimal(String.va
        BigDecimal ic= new BigDecimal(String.val
        total = uc.multiply(ic);

        total = total.setScale(2, RoundingMode.H
        System.out.println("Total3 --> " + total
}
```

Q4. What is your understanding of widening versus narrowing conversions of primitive data types?
A4. Left to right (e.g. byte to short) is a widening conversion and considered safe because there is no chance for data loss. For example, byte has a range between -128 and 127 and short has a wider range between -32768 and 32767. So

when you go from left to right, the data types are implicitly cast by the compiler since it is safe to do so.

byte → **short** → **char** → **int** → **long** → **float** → **double**
(1 byte) (2 bytes) (2 bytes) (4 bytes) (8 bytes) (4 bytes) (8 bytes)

Java primitive data types

Right to left (e.g. short to byte) is a narrowing conversion and considered unsafe because there is a chance for data loss. So when you go from right to left, the compiler expects you to explicitly cast the data to clearly state that it is safe to do so. If you do not cast explicitly, you will get a compile-time error. For example,

```
byte b = 0;              // valid values are -128 to
short s = 0;             // valid values are -32768
int i = 0;               // valid values are -214748
long l = 0L;             // valid values are  -92233

float f = 0.0F;          // valid values are 1.4E-45
double d = 0.0;          //  valid values are 4.9E-3

b = 30;                 // okay (30 is of type int,
b = 128;                // Not okay (128 is of type

s = b;                  // okay: short  is wider tha
i = s;                  // okay: int is wider than s
l = i;                  // okay: long is wider than

/**
   * compile-time errors
  **/
c = s;                  // Not okay: type char is un
b = i;                  // Not okay: type int is wid
i = l;                  // Not okay: type long is wi
l = f;                  // Not okay: type float is w
f = d;                  // Not okay: type double is

/**
  * fix above compile-time errors with explicit c
**/

c = (char)s;

b = (byte) i;
i = (int) l;
l = (long) f;
f = (float) d;

```

Note: byte and short are signed data types and they cannot be implicitly cast to unsigned char data type even though it is a widening conversion.

Q5. What are the dangers of explicit casting?
A5. Not knowing the MIN and MAX values can result in unexpected results due to loss of data during narrowing.

Trap #1 Be careful when casting explicitly.

```
int iWithinByteRange = 125;
int iOutsideByteRangeMax = 129;

byte bGood = (byte) iWithinByteRange;
System.out.println("bOkay=" + bGood);

byte bBad = (byte) iOutsideByteRangeMax;
System.out.println("Trap #1 - bBad=" + bBad);

```

Trap #2: Simple binary operations apply ‘binary numeric promotions’

The **binary numeric promotion** rule automatically casts each operand to the size of the larger operand type. If neither operand is larger, then both are cast to the same type. In byte b = b + 10, the value 10 is of type int and is the larger operand type compared to b, which is of type byte, hence will be evaluated as follows:

```
byte b = (int)b + 10;
```

Above code throws a compile time error because b+10 evaluates to an int. You need an explicit cast to convert an int to byte as it is a **narrowing conversion**.

Trap #3: Compound Operators, such as +=, -=, etc contain an explicit cast

In byte b+=10 compound binary operation, you may be thinking that b+=10 will be expanded as b = b + 10. But it is really not correct. This is because the compound operations will have an **explicit cast** in the converted byte code.

```
byte b = (byte) (b + 10);
```

performing a compound assignment operation (e.g. +=, -=, *=, etc). You may be thinking that b+=10 will be expanded as b = b + 10. But it is really not correct. This is because the compound operations will have an explicit cast even though it is not shown in the code.

Q6. Do you think the following code will throw a compile-time exception? If yes, how will you fix it?

```
float myVal = (float)3.0/2.0;
```

A6. Yes. It is cast first and then divided as casting operator has precedence over division operator as per the precedence table. So the above code is equivalent to

```
float myVal = 3.0f/2.0;     // float divided by  d
                            //as per the  "binary
```

To fix it, you need to get the division operator to evaluate prior to casting. You can achieve this by introducing a parenthesis around the division as parenthesis has higher precedence (in fact highest) than casting as per the **precedence** table.

```
float myVal = (float)(3.0/2.0);               // dou
```

```
                                        // doub
                                        // cast

```

Q7. What is the output of the following code snippet?

```
public class PrePostOperators {

    public static void main(String[ ] args) {
        int x = 5;
        int y = ++x;
        int z = y++;

        System.out.println("x=" + x + ", y=" + 
    }
}
```

A7. x=6, y=7, z=6

line 1: x=5, y=0, z=0
line 2: x=6, y=6, z=0;
line 3: x=6, y=7, z=6

You need to understand the pre and post increment operators to get this right. ++x is a pre-increment and x++ is a post increment. ++x means x is incremented before being used and x++ means x is incremented after being used. So line2 increments x by one and then assign it to y. Whereas in line 3, z is assigned the old y value (i.e. prior to incrementing) of 6 and then y value is incremented to 7.

As you may rightly ask that as per the precedence table, both pre-increment (i.e. ++expr) and post-increment (i.e. expr++) operators do have precedence over the assignment operator (i.e "="). The line 3 int z = y++; is roughly evaluated as follows:

```
int oldY = 6;               // current value
y = y+ 1;                   // increment y b
z = oldY;                   // z is set to t

```

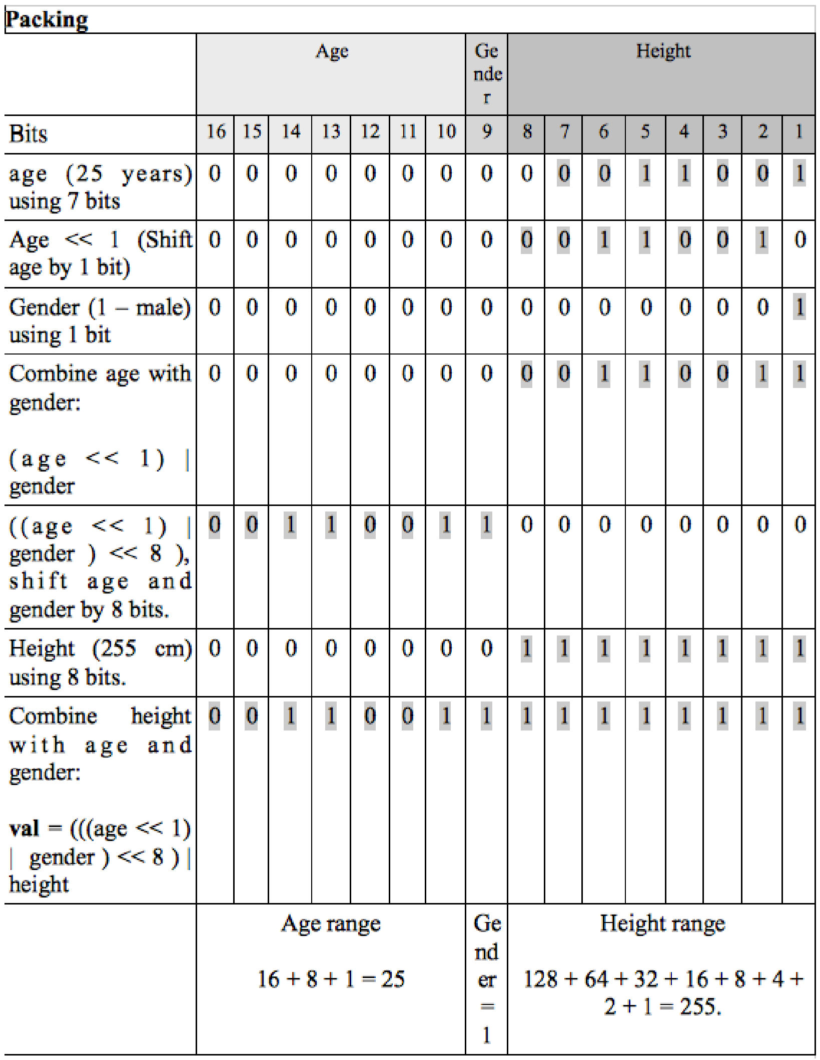
Q8. Can you list some practical applications where the bitwise operations can be applied?
A8.

Example 1: To pack and unpack values.

For example, to represent

- age of a person in the range of 0 to 127. Use 7 bits.
- gender of a person 0 or 1 (0 – female and 1 – male). Use 1 bit.
- height of a person in the range of 0 to 255. Use 8 bits.

To pack this info: (((age << 1) | gender) << 8) | height. For example, age = 25, gender = 1, and height = 255cm. Shift the age by 1 bit, and combine it with gender, and then **shift** the age and gender by 8 bits and **combine** it with the height.



Packing

	Age							Gender	Height							
Bits	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1
age (25 years) using 7 bits	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	1
Age << 1 (Shift age by 1 bit)	0	0	0	0	0	0	0	0	0	0	1	1	0	0	1	0
Gender (1 – male) using 1 bit	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
Combine age with gender: (age << 1) | gender	0	0	0	0	0	0	0	0	0	0	1	1	0	0	1	1
((age << 1) | gender) << 8), shift age and gender by 8 bits.	0	0	1	1	0	0	1	1	0	0	0	0	0	0	0	0
Height (255 cm) using 8 bits.	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
Combine height with age and gender: **val** = (((age << 1) | gender) << 8) | height	0	0	1	1	0	0	1	1	1	1	1	1	1	1	1	1
	Age range 16 + 8 + 1 = 25							Gender = 1	Height range 128 + 64 + 32 + 16 + 8 + 4 + 2 + 1 = 255.							

Bitwise operations

```java
public class Binary {

    public static void main(String[ ] args) {

        //packing
        int val = ((((25 << 1) | 1) << 8) | 255)
        System.out.println("packed=" + val);
        System.out.println("packed binary="
                                + Integer.toBinar

        //unpacking
        System.out.println("height=" + (val & 0x
        System.out.println("gender=" + ((val >>>
        System.out.println("age=" + ((val >>> 9)

    }
}
```

Output:

```
packed=13311
packed binary=11001111111111
height=255
gender=1
age=25
```

Unpacking (or extracting) height: Extract the low order 8 bits.

packed **value**:	0	0	1	1	0	0	1	1	1	1	1	1	1	1	1	1
Masking: 0xFF	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
Extract height = value **&** 0xFF:	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
									2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
height:									128 + 64 + 32 + 16 + 8 + 4 + 2 + 1 = **255**							

Unpacking (or extracting) gender: Extract bit 9.

packed **value**:	0	0	1	1	0	0	1	1	1	1	1	1	1	1	1	1
8 lower order bits (i.e. shaded area) are shifted out. value >>> 8	0	0	0	0	0	0	0	0	0	0	1	1	0	0	1	1
Masking: 1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
Extract gender = (value >>> 8) **&**1:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
																2^0
gender:																**1**

Unpacking (or extracting) age: Extract bits 10 – 16 (i.e. higher order bits).

packed **value**:	0	0	1	1	0	0	1	1	1	1	1	1	1	1	1	1
Extract age = value >>> 9:	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	1
										2^6	2^5	2^4	2^3	2^2	2^1	2^0
age:										$2^4+2^3+2^0=$ 16 + 8 + 1 = **25**						

Example 2: To compactly represent a number of attributes like being bold, italics, etc of a character in a text editor.

This is a more practical example.

shadow	blink	subscript	superscript	strikethrough	underline	italics	bold
0	1	0	1	0	0	0	1

```
import java.util.Arrays;

public class Binary6 {
    public static void main(String[ ] args) {
        byte[ ] vals = { 0, 1, 0, 1, 0, 0, 0, 1

        byte value = pack(vals);
        System.out.println("packedValue=" + valu
        System.out.println("unpackedValues="
                + Arrays.toString(unpack(value))
    }

    public static byte pack(byte[ ] vals) {
        byte result = 0;
        for (byte bit : vals) {
            result = (byte) ((result << 1) | (bi
        }
        return result;
    }

    public static byte[ ] unpack(byte val) {
        byte[ ] result = new byte[8];
        for (int i = 0; i < 8; i++) {
            result[i] = (byte) ((val >> (7 - i))
        }
        return result;
    }
}
```

Example 3: If you can think of anything as slots or switches that need to be flagged on or off,

you can think of bitwise operators. For example, if you want to mark some events on a calendar.

6 Saturday	5 Friday	4 Thursday	3 Wednesday	2 Tuesday	1 Monday	0 Sunday
		▨	▨	▨		
				▨	▨	▨

Example 4: To multiply or divide by 2^n

```
public class ShiftOperator {

    //multiply 10 by 2 power n where n = 6
    private static final int  MULTIPLY = 10 << 6

    //Divide 640 by  2 power n where n = 6.
    private static final int  DIVIDE = 640 >> 6;

    public static void main(String[ ] args) {
         System.out.println(MULTIPLY);
         System.out.println(DIVIDE);
    }
}
```

Popular Posts

♦ 11 Spring boot interview questions & answers
852 views

♦ Q11-Q23: Top 50+ Core on Java OOP Interview Questions & Answers
825 views

18 Java scenarios based interview Questions and Answers
447 views

001A: ♦ 7+ Java integration styles & patterns interview questions & answers
400 views

♦ 7 Java debugging interview questions & answers
311 views

♦ 10 ERD (Entity-Relationship Diagrams) Interview Questions and Answers
301 views

01b: ♦ 13 Spring basics Q8 – Q13 interview questions & answers
292 views

01: ♦ 15 Ice breaker questions asked 90% of the time in Java job interviews with hints
286 views

♦ Q24-Q36: Top 50+ Core on Java classes, interfaces and generics interview questions & answers
263 views

8 Git Source control system interview questions & answers
215 views

Bio | **Latest Posts**

Arulkumaran Kumaraswamipillai

Mechanical Eng to freelance Java developer in 3 yrs. Contracting since 2003, and attended 150+ Java job interviews, and often got 4 - 7 job offers to choose from. It pays to prepare. So, published Java interview Q&A books via Amazon.com in

As a Java Architect

Java architecture & design concepts interview Q&As with diagrams | What should be a typical Java EE architecture?

Senior Java developers must have a good handle on

2005, and sold 35,000+ copies. Books are outdated and replaced with this subscription based site.

About Arulkumaran Kumaraswamipillai

Mechanical Eng to freelance Java developer in 3 yrs. Contracting since 2003, and attended 150+ Java job interviews, and often got 4 - 7 job offers to choose from. It pays to prepare. So, published Java interview Q&A books via Amazon.com in 2005, and sold 35,000+ copies. Books are outdated and replaced with this subscription based site.

‹ Java identifiers

♥ 14 Unix interview questions & answers for Java developers ›

Posted in Data types, member-paid

Leave a Reply

Logged in as geethika. Log out?

Comment

Post Comment

open all | close all

80+ step by step Java Tutorials

open all | close all

Preparing for Java written & coding tests

How good are your...to go places?

Empowers you to open more doors, and fast-track

Technical Know Hows

☀ Java generics in no time ☀ Top 6 tips to transforming your thinking from OOP to FP ☀ How does a HashMap internally work? What is a hashing function?
☀ 10+ Java String class interview Q&As ☀ Java auto un/boxing benefits & caveats ☀ Top 11 slacknesses that can come back and bite you as an experienced Java developer or architect

Non-Technical Know Hows

☀ 6 Aspects that can motivate you to fast-track your career & go places ☀ Are you reinventing yourself as a Java developer? ☀ 8 tips to safeguard your Java career against offshoring ☀ My top 5 career mistakes

Prepare to succeed

☀ Turn readers of your Java CV go from “Blah blah” to “Wow”? ☀ How to prepare for Java job interviews? ☀ 16 Technical Key Areas ☀ How to choose from multiple Java job offers?

Select Category ▼

© Disclaimer