04: ♦ Top 10 most common Core Java beginner mistakes

Posted on November 28, 2014 by Arulkumaran Kumaraswamipillai — 2 Comments ↓

Mistake #1: Using floating point data types like float or double for monetary calculations. This can lead to rounding issues.

```
1  package com.monetary;
2  import java.math.BigDecimal;
3
4  public class MonetaryCalc {
5
6      public static void main(String[] args) {
7        System.out.println(1.05 - 0.42); //1:  $0.
8         //2: $0.63
9        System.out.println(new BigDecimal("1.05")
10       //3: $0.630000000000000059952043329758453
11       System.out.println(new BigDecimal(1.05) .s
12       //4: $0.63
13       System.out.println(BigDecimal.valueOf(1.05
14       System.out.println(105 - 42); //5: 63 cent
```

```
15     }
16 }
17
```

In the above code, 2, 4, and 5 are correct and 1 and 3 are incorrect usage leading to rounding issues. So, either use ***BigDecimal*** properly as shown in 2 and 4, or use the smallest monetary value like cents with data type long. The cents approach performs better, and handy in applications that have heavy monetary calculations.

Mistake #2: Using floating point variables like float or double in loops to compare for **equality**. This can lead to infinite loops.

```
1  public static void main(String[] args) {
2          float sum = 0;
3          while (sum != 1.0) { //causes infinite l
4              sum += 0.1;
5          }
6          System.out.print("The sum is: "+sum);
7  }
8
```

Fix is to change while (sum != 1.0) to while (sum **<** 1.0). Even then, due to mistake #1, you will get rounding issues (e.g. The sum is: 1.0000001). So, the better fix is

```
1    public static void main(String[] args) {
2          BigDecimal sum = BigDecimal.ZERO;
3          while (sum.compareTo(BigDecimal.ONE)  !=
4              sum =  sum.add(BigDecimal.valueOf(0.
5          }
6          System.out.print("The sum is: "+sum); //
7  }
8
```

Mistake #3: Not properly implementing the ***equals(..)*** and ***hashCode( )*** methods. Can you pick if anything wrong with the following *Person* class.

```
1  public class Person  {
2
3   private String name;
4   private Integer age;
5
```

```
6   public Person(String name, Integer age) {
7    this.name = name;
8    this.age = age;
9   }
10
11  //getters and setters
12
13  @Override
14  public String toString() {
15   return "Person [name=" + name + ", age=" + age
16  }
17 }
18
```

Now, the main class that creates a collection of ***Person***, and then searches for a person.

```
1  import java.util.Collections;
2  import java.util.Set;
3  import java.util.TreeSet;
4  import java.util.concurrent.CopyOnWriteArraySet;
5
6  public class PersonTest {
7
8   public static void main(String[] args) {
9
10   Set<Person> people = new CopyOnWriteArraySet<>
11   people.add(new Person("John", 35));
12   people.add(new Person("John", 32));
13   people.add(new Person("Simon", 30));
14   people.add(new Person("Shawn", 30));
15
16   System.out.println( people);
17
18   Person search = new Person("John", 35);
19
20   if(people.contains(search)){
21    System.out.println("found: " + search);
22   }
23   else {
24    System.out.println("not found: " + search);
25   }
26
27  }
28
29 }
30
```

The **output**

[Person [name=John, age=35], Person [name=John, age=32], Person [name=Simon, age=30], Person [name=Shawn, age=30]]
not found: Person [name=John, age=35]

Q. Why is the person not found even though there in the collection?

A. Every Java object implicitly extends the *Object* class, which has the default implementation of ***equals(…)*** and ***hashCode( )*** methods. The *equals(…)* method is **implicitly** invoked by the *Set* class's *contains(…)* method. In this case the default implementation in the ***Object*** class performs a shallow comparison of the references, and not the actual values like ***name*** and ***age***. The equals and hashCode methods in Java are meant for overridden for POJOs. So, this can be fixed as shown below by overriding the equals and hashCode methods in ***Person***.

```
1  import org.apache.commons.lang.builder.EqualsBui
2  import org.apache.commons.lang.builder.HashCodeB
3
4  public class Person  {
5
6   private String name;
7   private Integer age;
8
9   public Person(String name, Integer age) {
10   this.name = name;
11   this.age = age;
12  }
13
14  //getters and setters
15
16  @Override
17  public int hashCode() {
18   return new HashCodeBuilder().append(name).appe
19  }
20
21  @Override
22  public boolean equals(Object obj) {
23   Person rhs = (Person) obj;
24   return new EqualsBuilder().append(this.name, r
25  }
26
27  @Override
28  public String toString() {
29   return "Person [name=" + name + ", age=" + age
30  }
31
32 }
33
```

Now, output will be

[Person [name=John, age=35], Person [name=John, age=32], Person [name=Simon, age=30], Person

[name=Shawn, age=30]]
found: Person [name=John, age=35]

There are other subtle issues you can face if the *hashCode* and *equals* contracts are not properly adhered to. Refer to the ***Object*** class API for the contract details.

When sorting objects in a collection with compareTo, it is strongly recommended, but not strictly required that **(x.compareTo(y)==0) == (x.equals(y)).** Generally speaking, any class that implements the Comparable interface and violates this condition should clearly indicate this fact.

Mistake #4: Getting a ***ConcurrentModificationException*** when trying to modify (i.e. adding or removing an item) a collection while iterating.

The following code throws a **ConcurrentModificationException.**

```
1 List<T> list = getListOfItems();
2 for (Iterator<T> iter = list.iterator(); iter.has
3   T obj = iter.next();
4   if (obj.someCondition()) {
5     list.remove(0); // ConcurrentModificationExce
6   }
7 }
8
```

To avoid *ConcurrentModificationException* in a **single-threaded environment**, you can remove the object that you are working on with the iterator.

```
1 List<T> list = getListOfItems();
2 for (Iterator<T> iter = list.iterator(); iter.has
3   T obj = iter.next();
4   if (obj.someCondition()) {
5     iter.remove(); //OK to use the iterator
6   }
7 }
8
```

To Avoid ConcurrentModificationException in a **multi-threaded environment**:

If you are using JDK1.5 or higher then you can use ***ConcurrentHashMap*** and ***CopyOnWriteArrayList*** classes. This is the recommended approach compared to other approaches like locking the list with synchronized(list) while iterating, but this approach defeats the purpose of using multi-threading.

Q. Difference between **fail-fast** and **fail-safe** iterators
A. Iterators returned by most of pre JDK1.5 collection classes like *Vector*, *ArrayList*, *HashSet*, etc are **fail-fast** iterators. Iterators returned by JDK 1.5+ *ConcurrentHashMap* and *CopyOnWriteArrayList* classes are **fail-safe** iterators.

Mistake #5: Common exception handling mistakes.

a) Sweeping exceptions under the carpet by doing nothing.

```
1 try{
2
3 }catch(SQLException sqe){
4     // do nothing
5 }
6
```

In few rare scenarios, it is desired to do nothing with an exception, e.g. in finally block, we try to close database connection, and some exception occurs. In this case, exception can be ignored.

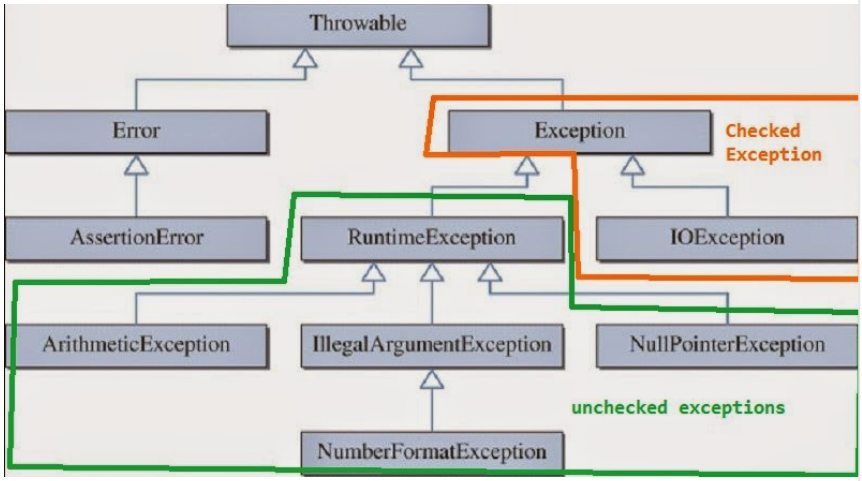
```
1  try{
2
3  }catch(SQLException sqe){
4      ...
5      ...
6  }finally{
7      try{
8          conn.close();
9      }catch(Exception e){
10         //leave it.
11     }
```

```
12 }
13
```

But in general, this is a very bad practice and can hide issues.

b) Inconsistent use of checked and unchecked (i.e. Runtime) exceptions.

Document a consistent exception handling strategy. In general favor unchecked (i.e. Runtime) exceptions, which you don't have to handle with catch and throw clauses. Use checked exceptions in a rare scenarios where you can recover from the exception like deadlock or service retries. In this scenario, you will catch a checked exception like *java.io.IOException* and wait a few seconds as configured with the retry interval, and retry the service configured by the retry counts.



c) Exceptions are polymorphic in nature and more specific exceptions need to be caught before the generic exceptions.

So, it is wrong to catch Exception before *IOException*.

```
1 try{
2     //....
3 } catch(Exception ex){
4     log.error("Error:" + ex)
5 } catch(IOException ex){
6     log.error("Connectivity issue:" + ex); //neve
7 }
```

```
8
```

Fix this by catching the more specific *IOException* first

```
1 try{
2     //....
3 } catch(IOException ex){
4     log.error("Connectivity issue:" + ex);
5 } catch(Exception ex){
6     log.error("Error:" + ex)
7 }
8
```

In Java 7 on wards, you can catch multiple exceptions like

```
1 try{
2     //....
3 }
4 catch (ParseException | IOException exception) {
5     // handle I/O problems.
6 } catch (Exception ex) {
7     //handle all other exceptions
8 }
9
```

d) Wiping out the stack trace

```
1 try{
2     //....
3 }catch(IOException ioe){
4     throw new MyException("Problem in data readin
5 }
6
```

e) Unnecessary Exception Transformation.

In a layered application, many times each layer catches exception and throws new type of exception. Sometimes it is absolutely unnecessary to transform an exception. An unchecked exception can be automatically bubbled all the way upto the GUI Layer, and then handled at the GUI layer with a **log.error(ex)** for the log file and a generic info like “An expected error has occurred, and please contact support on xxxxx” to the user. Internal details like stack trace with host

names, database table names, etc should not be shown to the user as it can be exploited to cause security threats.

Data Access Layer –> Business Service Layer –> GUI Layer

Logging

Mistake #6: Using *System.out.println(…)* statements for debugging without making use of the debugging capabilities provided in IDE tools. If you need to log, use log4j library instead.

```
1 if(log.isDebugEnabled()) {
2     log.debug("..................");
3 }
4 //....
5 log.info("................");
6 //...
7 log.error(ex);
8
```

The log4j library will not only perform better than *System.out.println* statements, but also provides lots of additional features like writing to a file, console, queue, etc, archiving and rolling the log files, controlling the log levels with debug, warn, info, error, etc and many more.

Mistake #7: Reinventing the wheel by writing your own logic when there are already well written and proven APIs and libraries are available. When coding, always have Core Java APIs, Apache APIs, Spring framework APIs, Google Gauva library APIs, and relevant reference documentations handy to reuse them instead of writing your own half baked solutions.

Instead of:

```
1 if(str != null && str.trim().length() > 0) {
2    //.............
3 }
4
```

You can use the **StringUtils** library from the Apache

commons API to simplify your code

```
1 if(!StringUtils.isEmpty(str){
2     //..........
3 }
4
```

EqualsBuilder, *HashCodeBuilder*, *StringBuilder*, etc are Apache commons utility methods.

Mistake #8: Resource leak issues are reported when resources are allocated but not properly disposed (i.e. closed) after use.

The system resources like file handles, sockets, database connections, etc are limited resources that need to be closed once done with them otherwise your applications run the risk of leaking resources and then running out of sockets, file handles, or database connections.

Bad code: if an exception is thrown before in.close() is reached, the Scanner that is holding on to the System.in resource will never get closed.

```
1 public void readShapeData() throws IOException {
2     Scanner in = new Scanner(System.in);
3     log.info("Enter salary: ");
4     salary = in.nextDouble();
5     in.close();
6 }
7
```

Good code: The finally block is reached even if an exception is thrown. So, the scanner will be closed.

```
1  public void readShapeData() throws IOException {
2      Scanner in = new Scanner(System.in);
3      try {
4          log.info("Enter salary: ");
5          salary = in.nextDouble();
6      } finally {
7          in.close();  //reached even when an exce
8      }
```

```
 9  }
10
```

Mistake #9: Comparing two objects (== instead of .equals)

When you use the == operator, you are actually comparing two object references, to see if they point to the same object. You cannot compare, for example, two strings or objects for equality, using the == operator. You must instead use the .equals method, which is a method inherited by all classes from java.lang.Object.

Mistake #10: Confusion over passing by value, and passing by reference as Java has both primitives like int, float, etc and objects.

When you pass a primitive data type, such as a char, int, float, or double, to a function then you are passing by value, which means a copy of the data type is duplicated, and passed to the function.If the function chooses to modify that value, it will be modifying the copy only.

When you pass a Java object, such as an array or an Employee object, to a function then you are passing by reference, which means the reference is copied, but both the original and copied references point to the same object. Any changes you make to the object's member variables will be permanent – which can be either good or bad, depending on whether this was what you intended.

5 bonus mistakes:

1. Forgetting that Java is zero-indexed.
2. Not writing thread-safe code with proper synchronization or thread-local objects.
3. Not favoring immutable objects. Immutable objects are inherently thread-safe.
4. Not properly handling null references, and causing the ubiquitous *NullPointerException*. This is a run time exception and the compiler can't warn you.
5. Capitalization errors where the class names and Java file names should start with capital letters and method

and variable names should start with lowercase letters.

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 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.

‹ Top 5 OOPs tips for Java developers

► Java Beginner Interview Questions and Answers on the Object class Video ›

Posted in Java Overview, member-paid

Tags: Core Java FAQs, Novice FAQs, TopX

2 comments on "04: ♦ Top 10 most common Core Java beginner mistakes"

Nikky says:

July 2, 2015 at 2:21 am

What is the #2 way of calculation in Mistake #1

Reply

akumaras says:

July 2, 2015 at 10:48 am

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

Not sure if I understand your question. #2 way is to use the BigDecimal class for monetary values.

```
//2: $0.63
System.out.println(new BigDecimal("1.05")
.subtract(new BigDecimal("0.42")));
```

Reply

Leave a Reply

Logged in as geethika. Log out?

Comment

Post Comment

open all | close all
- Best Practice (6)
- Coding (26)
- Concurrency (6)
- Design Concepts (7)
- Design Patterns (11)
- Exception Handling (3
- Java Debugging (21)
- Judging Experience In
- Low Latency (7)
- Memory Managemen
- Performance (13)
- QoS (8)
- Scalability (4)
- SDLC (6)
- Security (13)
- Transaction Managen

80+ step by step Java Tutorials

open all | close all
- Setting up Tutorial (6)
- Tutorial - Diagnosis (2
- Akka Tutorial (9)
- Core Java Tutorials (2
- Hadoop & Spark Tuto
- JEE Tutorials (19)
- Scala Tutorials (1)
- Spring & HIbernate Tu
- Tools Tutorials (19)
- Other Tutorials (45)

Preparing for Java written & coding tests

open all | close all
- ♦ Complete the given
- Can you write code? (
- Converting from A to I
- Designing your classe
- Java Data Structures
- Passing the unit tests
- What is wrong with th
- Writing Code Home A
- Written Test Core Jav
- Written Test JEE (1)

How good are your…to go places?

open all | close all
- Career Making Know-
- Job Hunting & Resum

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

The contents in this Java-Success are copy righted. The author has the right to correct or enhance the current content without any prior notice.

These are general advice only, and one needs to take his/her own circumstances into consideration. The author will not be held liable for any damages caused or alleged to be caused either directly or indirectly by these materials and resources. Any trademarked names or labels used in this blog remain the property of their respective trademark owners. No guarantees are made regarding the accuracy or usefulness of content, though I do make an effort to be accurate. Links to external sites do not imply endorsement of the linked-to sites.

↑