yz709-FJ-sup1

Question 1

1.1 (B) In this course we adopted the Google Java Coding Standard. Describe why a coding standard is important.

- When developers standardise their codes according to the same coding standard, that can help make the codes more readable and ensure code quality.
- Boost programmer efficiency and ease for debugging since programmers can understand each other's codes more effortlessly with coding standards.
- The software could be much safer, reliable and reduce maintenance costs.

💡 Comments:

Question 2

> See wrong.java on Github

1.2 (S) List at least 10 examples of poor style in the following Java code and describe *why* they are problematic and *how* they could be fixed:

```
import java.util.*;

public class wrong {
  public static final boolean t = true;
  public static final boolean f = false;

  public static void oldness(Integer o)
  {
    boolean b = o > 0;
    if (b == f)
    System.out.println("Input must be positive");
    else
         System.out.print("Age in days = ")
          System.out.print(o * 365.25);
  }
```

```

  static public void main(String args[]) {
    Integer i = Integer.parseInt(args[0]);
    oldness(i);
  }
}
```

- Do not import all the packages in the `java.util` library because only a subset of them are used, importing all packages would drastically increase the binary size and be wasteful.
- The value `true` and `false` are inherently static and final; there is no need to create two distinct static last boolean variables `t` and `f`, which are precisely `true` and `false`.
- Use primitive `int` rather than its wrapper class object type `Integer` if we do not need to work with object-oriented programming features or collection classes. We prefer `int` because of performance issues.
- Use `if (o<=0)` to replace `if(b == f)`
- Use error handling with throw-catch rather than only printing out the error messages.
- Enclose the if-else structure with brackets if there is more than one statement.
- Use formatted string rather than separate prints.
- Define the constant at the start of the program, do not use numbers inside the main loop without defining them because that makes the program unreadable.

```
final int DAYINYEAR = 365.24;
// ...
if (b){
  System.out.printf("Age in days = %d \n", o * DAYINYEAR)
}else{
  throw new IllegalArgumentException("Input must be positive");
}
```

- Use `public static void main(String[] args)` rather than `static public void main(String[] args)`
- Only one static method in the class is `wrong`, so we will not use the object-oriented programming paradigm; it is better to define the static function outside the class `wrong`.

```
// My implementation of the wrong class and its functions oldness and main
public class wrong {
    public static void oldness(int o) {
        try {
            if (o <= 0) {
                throw new IllegalArgumentException("Input must be positive.");
            }
            System.out.printf("Age in days = %f", o * 365.25);
        } catch (IllegalArgumentException e) {
            System.out.println(e);
        }
    }

    public static void main(String[] args) {
        oldness(Integer.parseInt(args[0]));
    }
}
```

💡 Comments:

Question 3

1.3 (S) Write two simple Java programs which exhibit the two common patterns for creating and executing a new Thread in Java. Compare and contrast the advantages and disadvantages of each approach.

- Threads can be created by either inheriting from `java.lang.Thread` or implementing the interface `java.lang.Runnable`, but in either cases, we would override the `run()` method

  ```
  // Runnable interface
  class ExpThread1 implements Runnable {
      public void run() {
          System.out.println("expThread1 runs");
      }
  }

  // Thread class
  class ExpThread2 extends Thread {
      public void run() {
          System.out.println("expThread2 runs");
      }

      public static void main(String[] args) {
          Thread exp1 = new Thread(new ExpThread1());
          exp1.start();
          ExpThread2 exp2 = new ExpThread2();
          exp2.start();
      }
  }
  ```

- Runnable interface over Thread class:
  - Single inheritance in java implies only one parent class for each child class, so the class which inherits the Thread class cannot extend other classes. But there are no restrictions on the number of interfaces implemented, so the class that implements the Runnable interface can implement other interfaces and extend one other class.
  - Composition is better than inheritance because a loose coupling between the class implements the Runnable interface and the interface itself.
  - Runnable makes more flexible classes because you can run it in a thread or pass it to some other kind of executor service.
- Thread class over Runnable interface:
  - After inheriting the thread class, every thread creates unique objects associated with them, but all threads share the same objects for threads implementing the Runnable interface.

💡 Comments:

Question 4

> See Finger.java on Github

1.4 (D) Write a client for the *finger* protocol (RFC 742) which is able to remotely retrieve information about users of a Unix system. Your solution should take a user and domain as a single argument, for example "`java Finger arb33@hermes.cam.ac.uk`" and print out details of the user `arb33` on `hermes.cam.ac.uk`.

Note: Your computer will need to be on the University network to successfully connect to `hermes.cam.ac.uk`. You can test whether your computer is able to connect by using the `finger` command-line app found on Mac OS and many Linux systems.

```
package supervision_1;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.net.InetSocketAddress;
import java.net.Socket;
import java.net.SocketAddress;
import java.util.stream.Collectors;

public class Finger {
    static final int DEFAULT_PORT = 79;
    static int timeout = 2000;

    public static void main(String[] args) throws IOException {
        try {
            if (args.length != 1) {
                throw new IllegalArgumentException("Wrong number of arguments.");
            }
            String[] strargs = args[0].split("@");

            SocketAddress socketAddress = new InetSocketAddress(strargs[1], DEFAULT_PO
RT);
            Socket socket = new Socket();
            socket.connect(socketAddress, timeout);

            // continuously receive arguments PrintWriter out = new
            PrintWriter out = new PrintWriter(socket.getOutputStream(), true);
            BufferedReader in = new BufferedReader(new InputStreamReader(socket.getInp
utStream()));
            out.println(strargs[0] + "\n");
            System.out.println(in.lines().collect(Collectors.joining()));
            socket.close();
        } catch (IllegalArgumentException e) {
            System.out.println(e);
        } catch (IOException e) {
            System.out.println(e);
        }
    }
}

/* output:

Login name:       arb33Registered name:   Prof. A.R. Beresford Directory:  /home_8/ar
b33Personal name:     Alastair Beresford  Shell:      /bin/MSshellAffiliation(s):    D
epartment of Computer Science and TechnologyNever logged in.Mail last read Sun Oct 31
 15:35 2021 (GMT)No Plan.

*/
```

💡 Comments:

Question 5

> See wrong.java on Github

1.5 (S) Write down a list of all the ways in which your implementation for Question 1.2 above might throw a Java Exception. Describe, in words, the necessary steps required to ensure your solution is robust to failure.

- If the argument is not an integer type or cannot be parsed as an integer, then a NumberFormatException (i.e., runtime exception) will be thrown in `Integer.parseInt(args[0])`. So in the `main` function, use a try-catch block to handle the exception.

  ```
  public static void main(String[] args) {
          try {
              oldness(Integer.parseInt(args[0]));
          } catch (NumberFormatException e) {
              System.out.println("Error parsing arguments");
          }
      }
  ```

- If the argument is a non-positive number, then an `IllegalArgumentException` would be thrown. We should wrap it inside a try-catch block to prevent further progress with the negative values.

  ```
  public static void oldness(int o) {
          try {
              if (o <= 0) {
                  throw new IllegalArgumentException("Input must be positive.");
              }
              System.out.printf("Age in days = %2f", o * 365.25);
          } catch (IllegalArgumentException e) {
              System.out.println(e);
          }
      }
  ```

💡 Comments:

Question 6

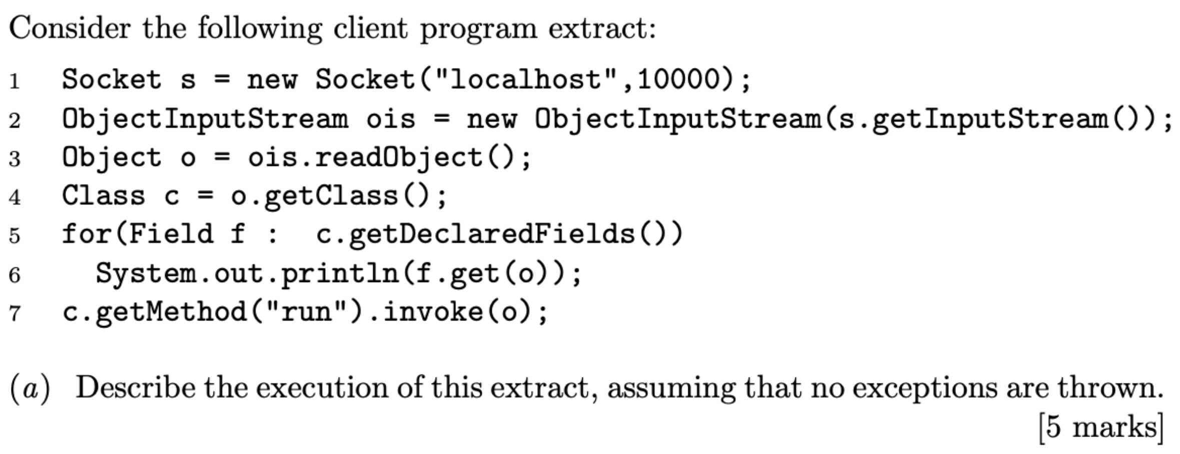
1.6 (S) Swap your solution for Ticklet 1 with your supervision partner. Highlight all the ways in which your partner's solution for Ticklet 1 does not adhere to the BAE coding standard. In what ways could the legibility, robustness and correctness be improved?

- Not adhere to BAE coding standard:
  - All code files must have a header with copyright and protective markings
  - All code files must have a description
  - Always use finally to clean up after exceptions (e.g., close the socket connection)
  - Logical units within a block should be separated by one blank line and commented correctly
- Improve legibility, robustness and correctness:
  - Legibility:
    - Include copyright and protective markings
    - JavaDoc should be included in all non-private methods, so other classes accessing this method can clearly understand its purposes.

- Robustness & Correctness:
    - Checking all input to ensure they are valid (not null values) and in the appropriate range
    - Be specific when catching exceptions, and don't swallow exceptions. Top-level exception handler must log coherent error message.
    - Access control: limited access to classes, methods and variables.
    - Be aware of code performance, use lazy initialisation where appropriate and avoid excessive object creation.
    - Adhere to a consistent layout and consistent naming convention that offers readability.

Question 6 (y2010p5q9)

1.6 Complete sections (a) and (b) from 2010 Paper 5 Question 9.



Consider the following client program extract:

```
Socket s = new Socket("localhost",10000);
ObjectInputStream ois = new ObjectInputStream(s.getInputStream());
Object o = ois.readObject();
Class c = o.getClass();
for(Field f :  c.getDeclaredFields())
  System.out.println(f.get(o));
c.getMethod("run").invoke(o);
```

(*a*) Describe the execution of this extract, assuming that no exceptions are thrown. [5 marks]

- A socket object is connected to server address "localhost" and port number 10000.
- `ObjectInputStream` helps deserialise bytes into an object, taking the input stream from the socket object; the `readobject ()` method would turn the input stream bytes into an object.
- We get the object's class and iterate through all the fields in the class; for each field, print out the value associated with the object `o`.
- Then get the method `run()` from class `c`, invoking the method on the object `o`.

(*b*) Identify **five** distinct exceptions that may occur during execution of the client program extract. Your answers should include the line number at which the exception would be thrown and a brief description of the problem which would cause it. General virtual machine errors such as OutOfMemoryError or StackOverflowError should not be included in your answer. [2 marks each]

- Line 1 - `UnknownHostException` is thrown if the IP address of the host cannot be determined.
- Line 1 - `IllegalArgumentException` is thrown if the port number is outside of the valid range.
- Line 3 - `ClassNotFoundException` is thrown if the class of the serialised object is not found.
- Line 3 - `StreamCorruptedException` is thrown if the control information in the stream is not consistent.
- Line 6 - `IllegalAccessException` is thrown if this object's field is not accessible due to Java language access control.

💡 Comments:

Question 7

1.7* (O) Write a simple Android app which acts as a Chat Client as described in Workbook 1 or 2.

💡 Comments:

Question 8

1.8* (O) Research and describe in words what a *serialization bomb* is. Explore whether you can devise a means of preventing a serialisation bomb from quickly exhausting all available resources of the Java virtual machine.

- Deserialisation bomb: Many apps do not validate the inputs before deserialising the byte streams so that attackers could insert malicious codes into the byte streams, and the malicious codes would later be executed on the apps; one of the common attacks is denial-of-service, which causes the deserialisation taking forever and consumes all the available resources.
- Solution:
    - We could have a list of available socket connections and store all the serialised objects. Before deserialising a byte stream, check whether it is from a valid serialised object; if so, we could securely deserialise the byte stream, otherwise reject deserialise.

💡 Comments: