Queens' College Cambridge

Further Java

Alistair O'Brien

Department of Computer Science

April 6, 2021

Contents

1 Networking

1.1 Sockets

- TCP connections are abstracted using UNIX *sockets*.
- `Socket` implemented `Closeable` interface $\implies$ Socket is closed on exception w/ try-with-resources.
- Client sockets `Socket`:
  - Socket(String *host*, **int** *port*):

```
try (final Socket s = new Socket(host, port)) {
    ...
} catch (UnknownHostException e) {
    // The IP address of the host could not be determined
    // using DNS.
    ...
} catch (IllegalArgumentException e) {
    // Argument port doesn't satisfy 0 ≤ port ≤ 65535
    ...
} catch (IOException e) {
    // An I/O error occurs when creating the socket
    ...
} catch (SecurityException e) {
    // Security manager exists and has insufficient
    // permissions
    ...
}
```

  - Socket(InetAddress *address*, **int** *port*):

```
try (final Socket s = new Socket(address, port)) {
    ...
} catch (NullPointerException e) {
    // The argument address is null
    ...
} catch (IllegalArgumentException e) {
    // Argument port doesn't satisfy 0 ≤ port ≤ 65535
    ...
} catch (SecurityException e) {
    // Security manager exists and has insufficient
    // permissions
```

```java
        ...
} catch (IOException e) {
    // An I/O error occurs when creating the socket
    ...
}
```

- Server-side ServerSocket:

```java
try (final ServerSocket serverSocket = new ServerSocket(port)) {
    ...
    while (true) {
        // Blocking listening call: accept
        // Exceptions:
        //  - IOException | SecurityException
        Socket s = serverSocket.accept();
        ...
    }
} catch (IllegalArgumentException e) {
    // Argument port doesn't satisfy 0 ≤ port ≤ 65535
    ...
} catch (SecurityException e) {
    // Security manager exists and has insufficient
    // permissions
    ...
} catch (IOException e) {
    // An I/O error occurs when openeing the socket
    ...
}
```

- Input Streams:

```java
try (InputStream in = s.getInputStream()) {
    byte[] buffer = new byte[BUFFER_SIZE];
    int length;

    // Blocking read method
    // Exceptions:
    //  - IOException: bytes cannot be read from socket e.g. closed
    //  - NullPointerException: buffer is null
    while ((length != in.read(buffer)) != -1) {
        // length bytes in array buffer
        ...
    }
} catch (IOException e) {
    // Socket s is closed, or not connected
}
```

- Output Streams:

```java
try (OutputStream out = s.getOutputStream()) {
    // bytes is a byte[]

    // Blocking write method e.g. full sliding window
    // Exceptions:
```

```java
    // - IOException: bytes cannot be written to socket buffer e.g. closed
    out.write(bytes);

    // Forces any buffered bytes in out to be written to socket. (blocking)
    out.flush();
} catch (IOException e) {
    // Socket s is closed, or not connected
}
```

- Stream Decorators (and *Cheat Sheet*):
  - InputStreamReader: Decorates a input stream, reading bytes and decoding them into characters (using a charset).

    ```java
    InputStreamReader in = new InputStreamReader(System.in);
    ```

  - BufferedReader: Reads streams of characters, buffering them into strings / arrays. Improves performance for block-based IO (e.g. disk):

    ```java
    BufferedReader r = new BufferedReader(new InputStreamReader(System.in));
    String in;

    while ((in = r.readLine()) != null) {
        ...
    }
    ```

  - FileReader: Read contents of a text file:

    ```java
    FileReader fileReader = new FileReader(new File(path));
    ```

    Reading all bytes of a file:

    ```java
    // or Paths.get(path)
    byte[] bytes = Files.readAllBytes(new File(path).toPath());
    ```

1.2 Serialization

- Serialized classes implement the Serializable interface:

  ```java
  public class ChatMessage extends Message implements Serializable { ... }
  ```

- Has no fields / methods. Simply tags class w/ serializable semantics.
- During serialization:

  - Serialize primitives: e.g. `int`, `boolean`, etc.
  - Serialize references to o: Recurse and serialize object o. If o doesn't implement `Serializable`, then `NotSerializableException` is thrown.

- Nonserializable state (e.g. a `Socket`) has the `transient` modifier:

```java
private transient String dontSerializeMe;
```

  `static` fields are also not serialized.

- JVM uses versioning w/ `.class` files. Creating a `UID` for each class by hashing the `.class` file.

- JVM can determine whether two classes have the same version $\implies$ detect whether object has a different version to on-disk `.class` version.

- Declare explicit `UID` using:

```java
private static final long serialVersionUID = ...;
```

  Advantages:

  - Changing `.class` definition no-longer changes version implicitly $\implies$ differing definitions:
    * removed fields are ignored
    * new fields are initialized w/ default value (`0 / null / user-defined`)

- `UID` only needs to be modified when:
  - `static` or `transient` fields are modified
  - Changing type of primitive field.
  - Changing inheritance hierarchy

- `ObjectInputStream` and `ObjectOutputStream` are used to serialize and deserialize objects, respectively.

```java
try (ObjectInputStream in = new ObjectInputStream(s.getInputStream())) {

    // Blocking readObject method
    // Exceptions:
    //  - ClassNotFoundException: Class of serialized object isn't loaded in JVM class loader
    //  - InvalidClassException: Deserialization error (see UID conditions)
```

```
    // - IOException: Error when reading stream
    T o = (T) in.readObject();
    ...
} catch (NullPointerException e) {
    // s.getInputStream() is null
    ...
} catch (IOException e) {
    // Error when reading stream (s is closed or not connected)
}
```

```
try (ObjectOutputStream in = new ObjectOutputStream(s.getOutputStream())) {

    // Blocking writeObject method
    // Exceptions:
    // - NotSerializableException: See above.
    // - IOException: Error when writing to stream
    out.writeObject(o);
    ...
} catch (NullPointerException e) {
    // s.getOutputStream() is null
    ...
} catch (IOException e) {
    // Error when writing to stream (s is closed or not connected)
}
```

1.2.1 Class Loaders

- A Java class loader is responsible for loading Java classes (compiled to bytecode, `.class` definitions) during the runtime of the JVM. Initially the default class loader is executed upon JVM initialization and loads classes from the local filesystem.

- **Problem**: Dynamically load classes during the runtime of the JVM from other locations e.g. socket

- **Solution**: Dynamic class loader. See `DynamicObjectInputStream` (Ticklet-2)

- **Problem**: Cannot guarantee non-byzantine behavior from server $\implies$ send malicious Java bytecode containing static initializers that execute the malicious code (since static initializers are executed when the class is loaded).

1.2.2 Reflection

- Reflection permits inspection of classes at runtime:

```java
Class<?> cls = o.getClass();
```

w/ generic wildcards (See IA OOP notes).

- List of *declared* fields (including private):

```java
// Exceptions:
//  - SecurityException: thrown if security manager doesn't have accessDeclaredMembers
//     permission
Field[] fs = cls.getDeclaredFields();
for (Field f : fs) {
    ...
}
```

- Accessing a declared field:

```java
// Suppresses all access modifier checks for f
// Exceptions:
//  - SecurityException: Security manager doesn't have permission.
f.setAccessible(true);

...

// Gets the value of the field in the reflected object o
// Exceptions:
//  - IllegalAccessException: The underlying field f is inaccessible (private, etc)
//  - IllegalArgumentException: o doesn't have the field f or class cls (to which f belongs)
//  - NullPointerException: o is null and f is an instance field
f.get(o);

...
```

- List of public methods:

```java
Method[] ms = cls.getMethods();
for (Method m : ms) {
    ...
}
```

- Get method from name:

```java
try {
    Method m = cls.getMethod(name);
} catch (NoSuchMethodException e) {
    // Method w/ name name is not found
    ...
} catch (NullPointerException e) {
    // name is null
    ...
}
```

- Invoke method:

```
// Exceptions:
//  - IllegalAccessException: The underlying method m is inaccessible
//  - IllegalArgumentException: o is null, m is not in cls, number of parameters and arguments differ
m.invoke(o, arg_1, ..., arg_n);
```

1.2.3 Annotations

- Annotations add metadata to objects in a way which is accessible to the program at runtime.
- Prefixed w/ `@`:

```
@Override
public method foo() {
    ...
}
```

- Annotations declared using `@interface` keyword:

```
public @interface SomeAnnotation {
    ... // metadata fields
}
```

- Built-in annotations:
  - `@Deprecated`: Annotated class, method, field is deprecated. Compiler warning if program uses deprecated code.
  - `@Override`: Annotated method must override method in superclass.
  - `@SuppressWarnings(`*warning*`)`: Compiler suppresses warnings it should generate. e.g. `@SuppressWarnings("deprecation")`
- Reflection support:
  - Annotation: `@Retention(RetentionPolicy.RUNTIME)`
  - o`.isAnnotationPresent(annotationClass)`: True if object o (class, field, method) has annotation w/ class `annotationClass`.
  - `Annotation[]` $as = o$`.getAnnotations()`: Returns list of runtime annotations present on o.
  - `Annotation` $a = o$`.getAnnotation(annotationClass)`: Returns annotation w/ class `annotationClass` on o if present. Otherwise `null`.

2 Concurrency

2.1 Multithreading

- Threads either extend `Thread` or implement `Runnable`:
  - Extend `Thread` (`A extends Thread`):
    1. Implement the `run()` method.
    2. Instantiate `A` and execute `a.start()`

    e.g. Idle thread: thread that schedules another thread and sleeps:

```java
public class Idle extends Thread {

    @Override
    public void run() {
        while (true) {
            // Schedule new thread
            Scheduler.schedule();

            // Block idle until it's re-run by the scheduler
            sleep(200);
        }
    }

}
```

  - Implement `Runnable`:
    1. Implement `Runnable` interface:

```java
public interface Runnable {
    void run();
}
```

       `Runnable` is a *functional interface* (see IA OOP notes) $\implies$ use a *lambda function*.
    2. Create a new instance of the `Runnable` Class
    3. Instantiate `Thread`, passing newly created `Runnable` instance as argument, then execute `start()`.

```
Thread idleThread = new Thread(() -> {
    while (true) {
        Scheduler.schedule();
        Thread.sleep(200);
    }
});
idleThread.start();
```

  - **Advantages of Runnable**:
    * No multiple inheritance $\implies$ `extends Thread` prevents the class from extending any other classes.
    * `Runnable` yields a *task* abstraction $\implies$ executed using `run()` or `Thread`. Hence reuse. Using `extends Thread` produces tightly coupled code which would benefit from being loosely coupled.
    * extending `Thread` $\implies$ each extended thread will have a unique object w/ `run` implementation associated. implementing `Runnable` $\implies$ many threads can share the same Runnable instance. Reduces overhead.
    * `Runnable` shared instance local resources for a group of threads. Easy state sharing w/ out global state.
  - *Rule of Thumb*: extending a class $\implies$ adding new behavior / state. If we're not modifying the `Thread` class then we should be using `Runnable`.

- Daemon threads: JVM will only exit when all threads running are daemons:

```
thread.setDaemon(true);
```

- Thread interrupts:
  - `Thread` contains a `boolean interrupted` flag. `t.interrupt()` sets the flag to `true`.
  - `Thread.interrupted()` returns `interrupted` of the current thread and sets it to `false`.
  - Blocking thread methods, e.g `Thread.sleep`, check `interrupted` and throw `InterruptedException` if set:

```
public static void sleep(long millis) throws InterruptedException {
    while (/* still sleeping */) {
        if (Thread.interrupted()) throw new InterruptedException();
    }
}
```

2.2 Monitors

- All `Object`s implement an *implicit monitor*

Definition 2.2.1. (**Monitor**) A construct consisting of a lock ℓ and a set of condition variables $cv_1, \ldots, cv_n$.

Definition 2.2.2. (**Condition Variable**) A condition variable cv, or *condition queue*, is a queue Q associated w/ a condition c and lock ℓ w/ operations:

- `wait`(cv, ℓ): Thread t blocks until c is asserted. Atomically: releases ℓ, moves t to Q and blocks. Once t is signaled, t is resumed and ℓ is acquired.
- `signal`(cv): Called to indicate c is asserted. Dequeues thread t from Q and unblocks t.
- `broadcast`(cv): Called to indicate c is asserted. Dequeues *all*threads in Q.

- Java's monitors consist of a lock ℓ and a single *non-blocking* condition variable cv (associated condition is defined by Programmer) w/ operations `wait()`, `notify()`, `notifyAll()`.
- Non-blocking condition variables: Signaling thread doesn't block, signaled thread is unblocked (and scheduled at the scheduler's whim). **Problem**: Condition c may not hold when signaled thread is scheduled.
  Solution: `while (!`c`) wait();`

2.2.1 Synchronized Statement

- Monitor locks are also acquired w/ `synchronized` statements:

```
synchronized (object) {
    // Code w/ mutual exclusion on object
    ...
}
```

Acquires *object*'s lock ℓ and releases it after leaving the block.

- **Syntactic Sugar**:

```
public synchronized void
    lockMeUp() {
        ...
    }
```

$\triangleq$

```
public void lockMeUp() {
    synchronized (this) {
        ...
    }
}
```

- Static methods are associated w/ a *class* lock $cls.\ell$ not object lock $o.\ell$.

2.2.2 Deadlock

Definition 2.2.3. (**Deadlock**) A set of threads $t_1, \ldots, t_n$ are deadlocked if the following conditions hold:

(i) **Mutual Exclusion**: Only a single thread t_i can use a resource R_j at a time.

(ii) **Hold & Wait**: A thread t_i is holding the resources $R^i_a, \ldots, R^i_b$ is waiting to acquire resource R^j_c held by thread t_j.

(iii) **No Preemption**: A resource R^i can only be released by the thread t_i holding it.

(iv) **Circular Wait**: A cycle of thread requests exists: $t_1 \to t_2 \to \ldots \to t_1$

- **Problem**: Preventing Deadlock.
- **Idea**: Eliminate one (or more) conditions required for deadline $\implies$ deadlock can never occur.
- **Solutions**:
  - **Mutual Exclusion**: Not required for shared resources (unsafe :( ). Mutual exclusion must only hold for non-sharable resources.
  - **Hold & Wait**: Require that when threads request resource, they don't hold any other resources. Require threads to request and allocate all it's resource at once. Results in low resource utilization and difficult to know maximal resource set (in advance).
  - **No Preemption**: If thread t_i is holding resources (R^i_j) and requests R_k and cannot acquire R_k, then (R^i_j) are released. Preempted resources added to list of resources that t_i is waiting on. t_i is only scheduled again iff all old and new resources can be acquired.

  - **Circular Wait**: Impose a partial ordering of all resources, requiring each thread t_i to acquire (R_j) in order.

- *Rule of Thumb*: **Circular Wait** solution is used. Define partial ordering $\prec$ on objects.