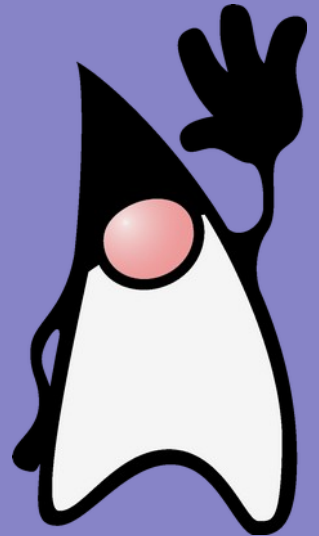


Java

Serialization



Overview

- "saving" complete objects
 - objects “survive” through programs' executions
- persistence
 - lightweight persistence
 - explicit saving and loading
- serialized objects can be transferred via network
- saving a state of objects
 - fields
- code of the class of the object must be available
- possibly dangerous (it is not part of the Java object model)
 - in future might be removed
 - and allowed for **records** only



Usage

- `java.io.Serializable`
 - empty interface
 - serializable objects must implement it
- `ObjectOutputStream`
 - **extends** `OutputStream`
 - **implements** `DataOutput` **and** `ObjectOutput`
 - the method `void writeObject(Object o)`
- `ObjectInputStream`
 - **extends** `InputStream`
 - **implements** `DataInput` **and** `ObjectInput`
 - the method `Object readObject()`



Example

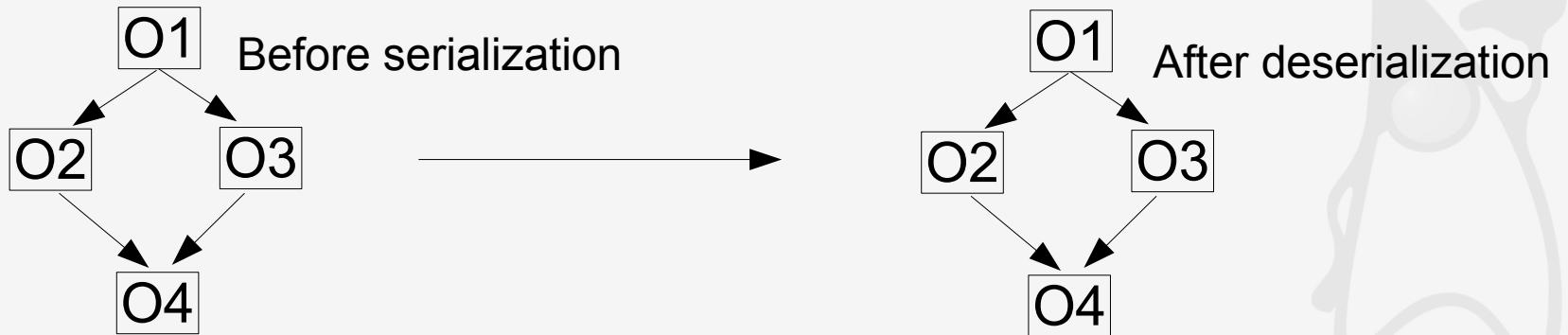
```
public class Data implements Serializable {
    private int d;
    public Data(int d) {this.d = d;}
    public String toString() {
        return super.toString() + ", d=" + d;
    }
}

...
Data data = new Data(1);
...
ObjectOutputStream out = new ObjectOutputStream(
    new FileOutputStream("file.dat"));
out.writeObject(data);
...
ObjectInputStream in = new ObjectInputStream(
    new FileInputStream("file.dat"));

data = (Data) in.readObject();
```

Serialization

- all field (even private ones) are serialized/deserialized
 - the field modifier `transient`
 - the field will not be saved/read
- both primitive and also references are saved
 - recursively are saved all objects from the field
 - during deserialization objects are created “in the same shape”
 - example



Own serialization

- interface Externalizable
 - extends Serializable
 - two methods
 - `void readExternal(ObjectInput in)`
 - `void writeExternal(ObjectOutput out)`
- objects implement Externalizable instead of Serializable
- the rest is the same (almost)
- the `transient` modifier has no meaning
 - saving/reading through the methods `writeExternal` and `readExternal`
- `writeExternal` and `readExternal` are called automatically



Example

```
public class Data2 implements Externalizable {
    public Data2() { System.out.println("Data2"); }
    public void writeExternal(ObjectOutput out) throws IOException {
        System.out.println("Data2.writeExternal");
    }
    public void readExternal(ObjectInput in) throws IOException,
                                                                    ClassNotFoundException {
        System.out.println("Data2.readExternal");
    }
}

...
Data2 d = new Data2();
ObjectOutputStream o = ....
o.writeObject(d);
...
ObjectInputStream i = ....
d = (Data2) o.readObject();
```



Wrong example

```
public class Data2 implements Externalizable {
    Data2() { System.out.println("Data2"); }
    public void writeExternal(ObjectOutput out) throws IOException {
        System.out.println("Data2.writeExternal");
    }
    public void readExternal(ObjectInput in) throws IOException,
                                                                    ClassNotFoundException {
        System.out.println("Data2.readExternal");
    }
}

...
Data2 d = new Data2();
ObjectOutputStream o = ....
o.writeObject(d);
...
ObjectInputStream i = ....
d = (Data2) o.readObject();
```



Loading objects

- implicit serialization (implementing `Serializable`)
 - during loading no constructor is called
 - objects are created directly
- own serialization (implementing `Externalizable`)
 - first, a constructor is called
 - the default constructor without parameters
 - must be available
 - then, the `readExternal()` is called on the object

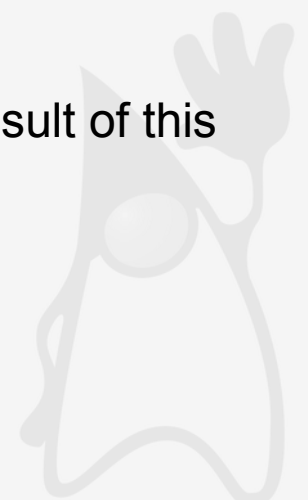


Another approach

- implement the interface `Serializable`
- and add 2 „magic“ methods
 - `private void writeObject(ObjectOutputStream stream) throws IOException;`
 - `private void readObject(ObjectInputStream stream) throws IOException, ClassNotFoundException`
- both methods must have exactly the given signature
 - must be private
- in `readObject()` and `writeObject()`, default loading/saving can be called by the methods `defaultReadObject()` and `defaultWriteObject()`

Other „magic“ methods

- `private void readObjectNoData()` throws `ObjectStreamException`
 - called during loading an object if some of its classes (the class or superclasses) are not stored in a stream
 - usage – when class hierarchy is changed between storing/loading
 - ex: saving an object of the class `Monkey`, which extends `Animal` and loading the object of the class `Monkey`, which extends `Mammal` and it extends `Animal` (the method is used on the class `Mammal`)
- *anything* `Object readResolve()` throws `ObjectStreamException`
 - if the method exists, deserialization of an object of the class returns the result of this method
- *anything* `Object writeReplace()` throws `ObjectStreamException`
 - if exists, its result is serialized



serialVersionUID

- *anything* static final long serialVersionUID = *value*
 - if during deserialization the saved value is different from the value in the class, the `InvalidClassException` is thrown
 - not necessary to use
 - created automatically during serialization
 - but its explicit declaration is strongly recommended



Serialization and std library

- many classes in the std. library implement Serializable
- warning – serialization may not work for all classes between different Java version
 - typically a warning in the documentation

Warning: Serialized objects of this class will not be compatible with future Swing releases. The current serialization support is appropriate for short term storage or RMI between applications ...



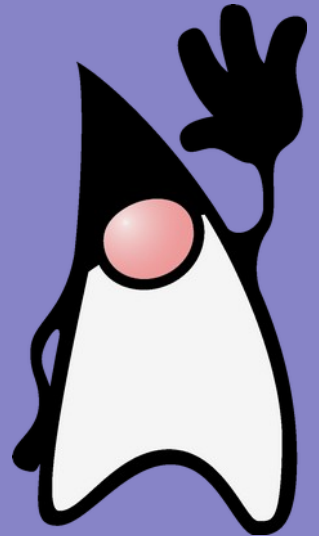
Serialization of records

- also implement Serializable
- but
 - all “magic” methods are ignored
 - also readExternal() and writeExternal()
 - for deserialization, the regular constructor is used
 - if a value does not exist during deserialization, the default value is used
- cyclic references are not deserialized



Java

Preferences



Overview

- the package `java.util.prefs`
- for storing/loading a configuration of programs
- automatically stored/loaded
 - exact place depends on OS
 - separately per user
- only primitive types and strings (max. 8 KB long)
- tuples
 - key – value
 - does not implement the interface `Map`
- hierarchical structure (tree)
 - usually just a single node



Usage

- static methods of the class `Preferences`
- `Preferences.userNodeForPackage(Class c)`
 - returns a node of preferences associated with the package of the given class
- `Preferences.systemNodeForPackage(Class c)`
 - as the previous method
 - a node common for all users
- **ex:**
 - `p = Preferences.userNodeForPackage(Foo.class)`
- name of the node ~ full name of the package
 - dots are replaced by slashes "/"



Example

```
public class Prefs {  
    public static void main(String[] args) {  
        Preferences prefs =  
            Preferences.userNodeForPackage(Prefs.class);  
  
        prefs.put("url", "http://somewhere/");  
        prefs.putInt("port", 1234);  
        prefs.putBoolean("connected", true);  
        int port = prefs.getInt("port", 1234);  
  
        String[] keys = prefs.keys();  
        Arrays.stream(keys).forEach(key -> System.out.println(key +  
                                                                ": " + prefs.get(key, null)));  
    }  
}
```

Methods

- `String get(String key, String def)`
 - returns a value of the key
 - the implicit value must be set
- `int getInt(String key, int def)`
 - as `get`
 - defined for all the primitive types
- `void put(String key, String val)`
 - assigns a value to the key
 - defined also for all the primitive types
- `String[] keys()`
 - return all keys
- `void flush()`
 - writes the changes



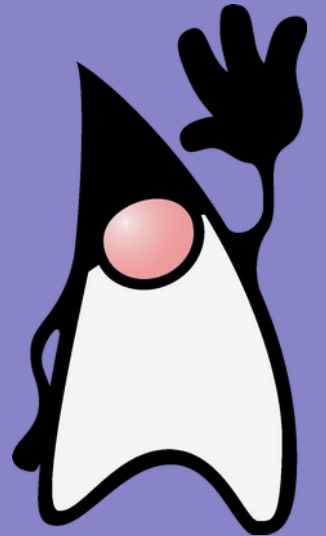
Methods

- `void clear()`
 - clears all the preferences in the node
- `String name()`
 - a name of the node
- `String absolutePath()`
 - an absolute name of the node
- all methods are thread safe
- can be safely used from multiple JVMs at the same time



Java

Communication over network

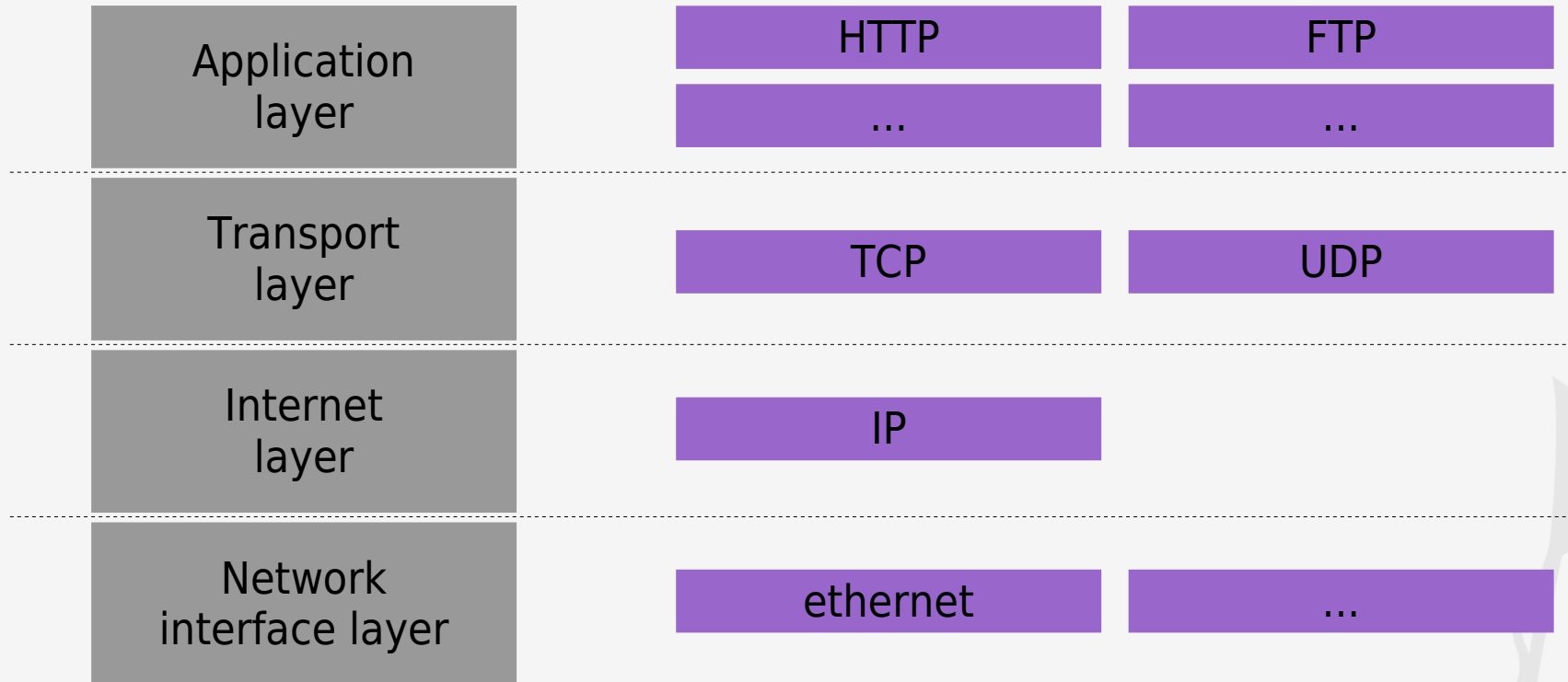


Overview

- the `java.net` package
- easy communication over network
- almost as using files
 - streams over network
- protocols TCP and UDP
 - Internet

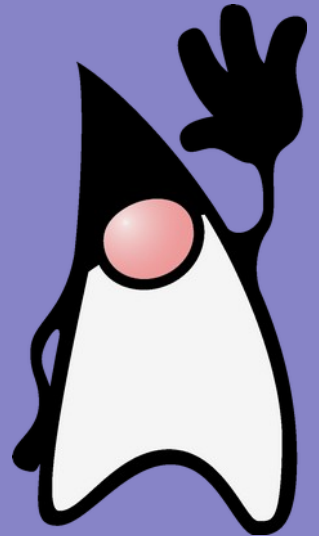


TCP/IP model



java.net

URI and URL



java.net.URI

- representation of URI
 - unique resource identifier (RFC 2396)
- structure URI
 - [scheme:]scheme-specific-part[#fragment]
- absolute URI – has a schema
 - relative URI – has not a schema
- "opaque" URI – the specific part does not start with the slash
 - ex: `mailto:java-net@java.sun.com`
`news:comp.lang.java`
- hierarchical URI – either an absolute URI starting with the slash or relative URI
 - př: `http://java.sun.com/j2se/1.3/`
`../../demo/jfc/SwingSet2/src/SwingSet2.java`



java.net.URI

- hierarchical URI – structure
 - [scheme:][//authority][path][?query][#fragment]
 - authority
 - [user-info@]host[:port]
- all parts of URI are Strings, except the port, which is int
- normalization of URI
 - removing and replacing "." and ".."



java.net.URI: methods

- `String getScheme()`
- `String getSchemeSpecificPart()`
- `String getPath()`
- `String getHost()`
-
- `boolean isAbsolute()`
- `boolean isOpaque()`
- `void normalize()`
- `URL toURL()`
 - creates URL from URI
 - an exception thrown if cannot be created



java.net.URL

- URL is a special case of URI
- unique resource locator
- specifying resources in the web
 - `http://www.mff.cuni.cz/`
- similar methods like URI
 - `get...`
- `InputStream openStream()`
 - opens a stream for reading a file specified by the URL
- `URLConnection openConnection()`
 - creates a connection to the URL object



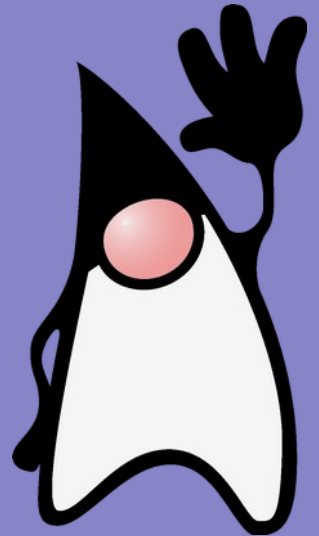
URLConnection

- representation of a connection between the application and URL
- usage
 1. obtaining a connection (`openConnection()`)
 2. setting parameters
e.g. `setUseCaches()`
 3. creating the connection (`connect()`)
the remote object is available then
 4. obtaining content and information
 - content – `getContent()`
 - headers – `getHeaderField()`
 - streams – `getInputStream()`, `getOutputStream()`
 - other – `getContentType()`, `getDate()`, ...



java.net

Adresses (DNS)



InetAddress

- represents an IP address
- obtaining an address
 - static methods of InetAddress
 - `InetAddress getByName(String host)`
 - IP address of the given name of a node
 - returns localhost for null
 - `InetAddress getByAddress(byte[] addr)`
 - IP address for the given address
 - length of the addr array – 4 for IPv4, 16 for IPv6
 - `InetAddress getLocalHost()`
 - address of localhost (127.0.0.1)



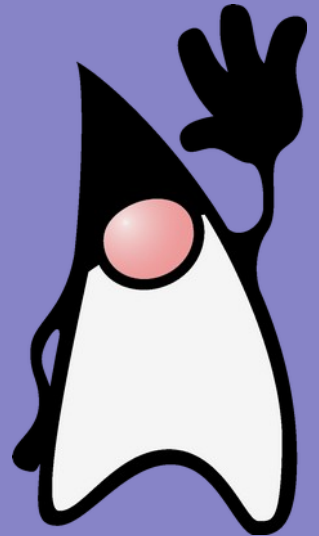
Example

```
public class InetName {  
    public static void main(String[] args) throws Exception {  
        InetAddress a = InetAddress.getByName(args[0]);  
        System.out.println(a);  
    }  
}
```

```
public class Localhost {  
    public static void main(String[] args) throws Exception {  
        System.out.println(InetAddress.getByName(null));  
        System.out.println(InetAddress.getLocalHost());  
    }  
}
```


java.net

Sockets



Overview

- socket = endpoint of a connection
- TCP
 - reliable communication
- connections in both directions
 - both `InputStream` and `OutputStream` can be obtained
- the `ServerSocket` class
 - creates a "listening" socket
 - the `accept()` method
 - waits for an incoming connection
 - returns a socket for communication
- the `Socket` class
 - a socket for communication



Example: simple server

```
try (ServerSocket s = new ServerSocket(6666)) {  
    System.out.println("Server ready");  
    try (Socket socket = s.accept()) {  
        InputStream in = socket.getInputStream();  
        OutputStream out = socket.getOutputStream();  
        while (true) {  
            ...  
            in.read();  
            ...  
            out.write(...);  
            ...  
        }  
    }  
}
```



Example: simple client

```
InetAddress addr = InetAddress.getByName(null);
Socket socket = new Socket(addr, 6666);
try (InputStream in = socket.getInputStream();
     OutputStream out = socket.getOutputStream()) {
    while (...) {
        ...
        out.write(...);
        ...
        in.read();
        ...
    }
}
```



Serving incoming requests

- the previous example – simple server
 - serves only one connections
 - serving multiple connections
 - a new thread for each incoming connection
 - resp. a new task and submit it to an executor
- or
- channels and the Selector class
 - serving multiple requests in a single thread
 - the selector holds a set of sockets
 - the select() method waits until at least one socket is ready to be used
 - similar to the select() function in UNIX systems



Multithread server

```
class ServeConnection extends Thread {
    private Socket socket; private InputStream in; private OutputStream out;
    public ServeConnection(Socket s) throws IOException {
        socket = s; in = ...; out = ...; start();
    }
    public void run() {
        while (true) {
            in.read();
            out.write(...);
        }
    }
}

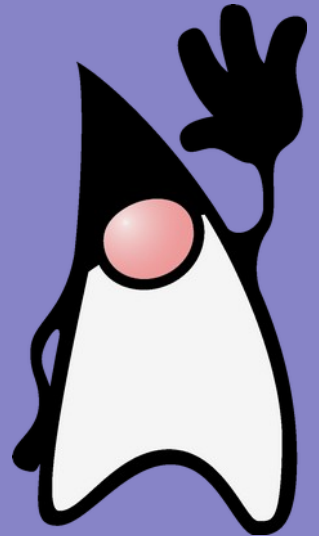
public class Server {
    public static void main(String[] args) throws IOException {
        ServerSocket s = new ServerSocket(6666);
        while(true) {
            Socket socket = s.accept();
            new ServeConnection(socket);
        }
    }
}
```



It is better to use
an executor

java.net

UDP



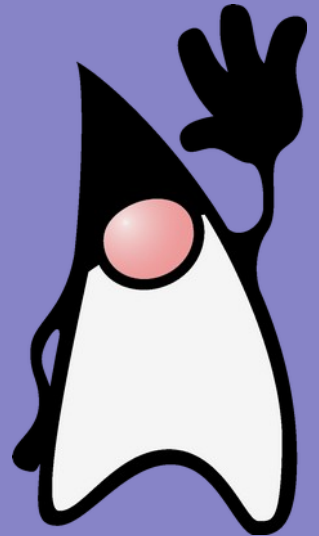
Overview

- unreliable communication
- the DatagramSocket class
 - for both server and client
 - sending/receiving datagrams
 - void send(DatagramPacket d)
 - void receive(DatagramPacket d)
- the DatagramPacket class
 - a datagram
 - void setData(byte[] buf)
 - byte[] getData()
 - sets/returns a buffer for the datagram
 - int getLength()
 - void setLength(int a)
 - length of data in the datagram



java.net

HTTP API (client)



java.net.http

- since Java 11
- supports
 - HTTP 2
 - WebSockets
 - asynchronous calls
 - returns a Future



java.net.http

```
HttpClient client = HttpClient.newBuilder()
    .version(Version.HTTP_1_1)
    .followRedirects(Redirect.NORMAL)
    .connectTimeout(Duration.ofSeconds(20))
    .proxy(ProxySelector.of(new InetSocketAddress("proxy.example.com", 80)))
    .authenticator(Authenticator.getDefault())
    .build();
```

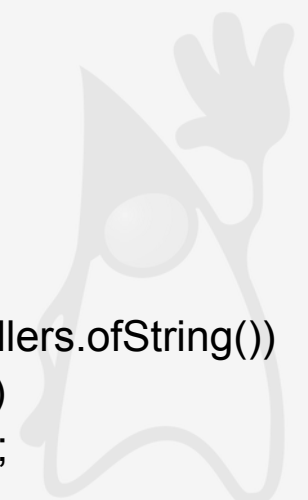
```
HttpRequest request = HttpRequest.newBuilder()
    .uri(URI.create("https://foo.com/"))
    .timeout(Duration.ofMinutes(2))
    .header("Content-Type", "application/json")
    .POST(BodyPublishers.ofFile(Paths.get("file.json")))
    .build();
```

- synchronous call

```
HttpResponse<String> response =
    client.send(request, BodyHandlers.ofString());
System.out.println(response.statusCode());
System.out.println(response.body());
```

- asynchronous call

```
client.sendAsync(request, BodyHandlers.ofString())
    .thenApply(HttpResponse::body)
    .thenAccept(System.out::println);
```





Slides version J10.en.2022.1
This slides are licensed under a [Creative Commons Attribution-NonCommercial 4.0 International License](#).