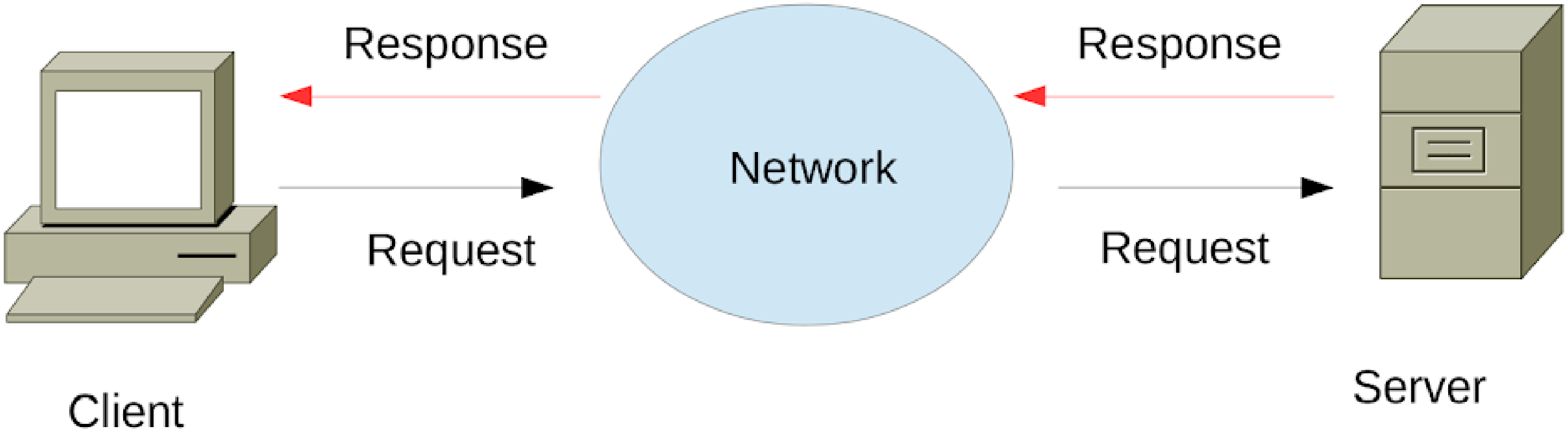
Advanced Java Communication & Safety

Yiwen Wang
y.wang@di.ku.dk
23 Aug, 2018
DIKU, University of Copenhagen

Table of Content

Table of Content

Request-Response Model



Request-Response Model

- Server responsibilities
  - Needs to listen for client requests.
  - On receipt of client request, process the request.
  - Send a response back to the client when the processing completes.
- Client responsibilities
  - Generate and send the request.
  - Be prepared for a response.

Table of Content

HTTP and Request-Response Model

Hyper Text Transfer Protocol:

- An application protocol which implements the request-response client server model.
- Uses TCP (transport layer) for reliable message delivery on an IP network.
- Messages sent in HTTP request and response messages.
- HTTP is stateless.
- Designed to allow scalability in intermediate hardware and software components (proxies).
- Use HTTP for communication instead of building messaging protocol.

HTTP Requests

- HTTP requests are specified using URLs.

  URL = domain:port/path?query_string
- http://www.google.dk/search?q=search+engine+optimisation&ie=utf-8

  query_string : field1=value1&field2=value2&......
- HTTP request methods
  - **GET**
  - HEAD
  - **POST**
  - PUT
  - DELETE
  - OPTIONS

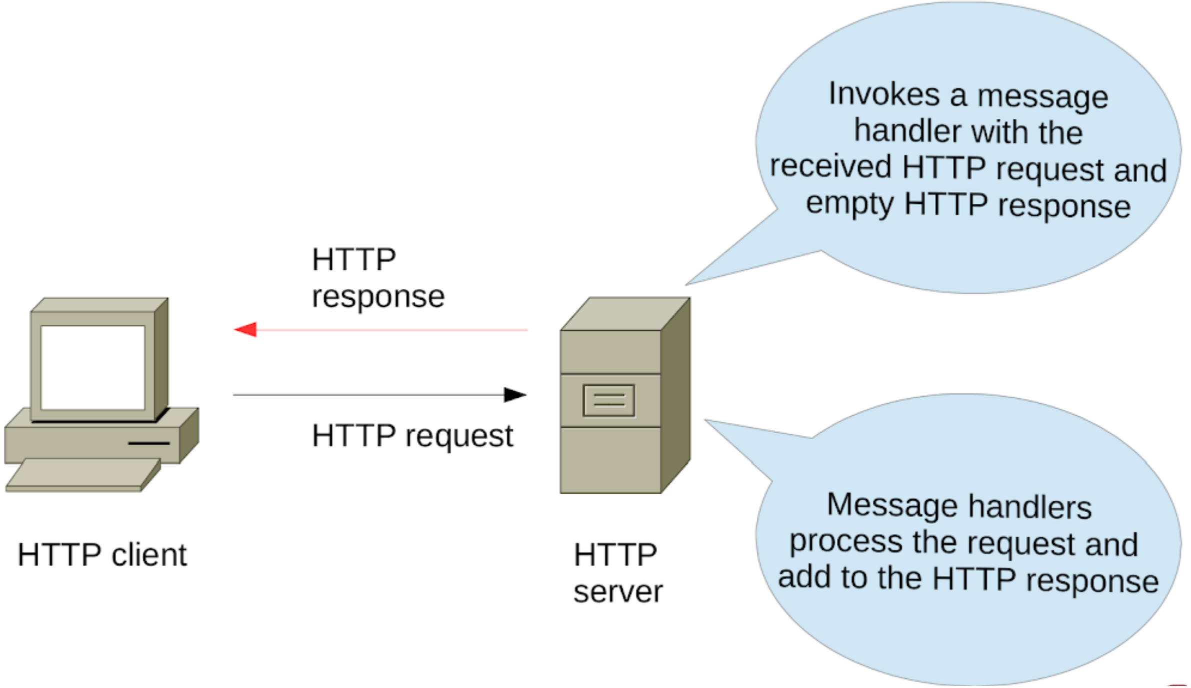
  ...

HTTP Requests Methods - REST

- One can map HTTP request methods to some actions
  Example: REST-style API for CRUD
- REST - REpresentational State Transfer
- CRUD – Create, Read, Update and Delete

Method	Meaning
POST	Create
GET	Read
PUT	Update
DELETE	Delete

HTTP Communication Workflow



Example: HTTP Requests and Response

- HTTP request:

```
GET /index.html?query=foo HTTP/1.1
Host: www.example.com
```

- HTTP response:

```
HTTP/1.1 200 OK
Date: Mon, 23 May 2005 22:38:34 GMT
Server: Apache/1.3.3.7 (Unix) (Red-Hat/Linux)
Last-Modified: Wed, 08 Jan 2003 23:11:55 GMT
ETag: "3f80f-1b6-3e1cb03b"
Content-Type: text/html; charset=UTF-8
Content-Length: 131
Connection: close
<html>
<head>
	<title>An Example Page</title>
</head>
<body> Hello World, this is a very simple HTML document. </body>
</html>
```

To Use HTTP for Communication

- Need to implement HTTP request methods on clients.
- Need to implement an HTTP server
- Need to implement HTTP response methods on server.
- Need to build delegation of processing to application handlers.
- Handle response methods on clients.
- Thread pool management.
- Network stack management.
- And a lot of everything else for performance.

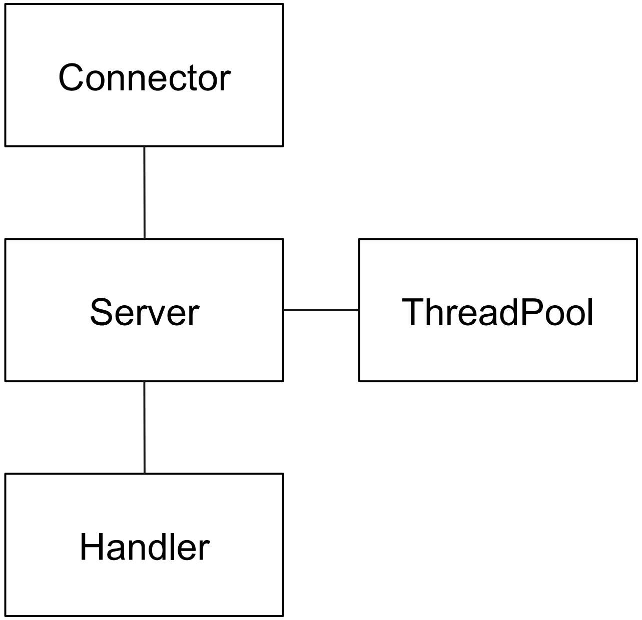
Table of Content

Introducing Jetty

- Jetty provides **HTTP client** libraries to enable HTTP requests and wait for HTTP responses.
- Jetty provides **HTTP server** which listens for HTTP requests and invokes registered handlers to handle requests.
- Jetty provides all the necessary tools (client and server libraries) to enable efficient use of HTTP for communication.
- We will deploy Jetty inside our applications and not the other way around.

Jetty Architecture Overview

- Connector: accepte HTTP connections
- Handler: service requests from the connections and produce responses
- ThreadPool: threads from a thread pool doing the work.



Embedding Jetty Inside Application

- Create a server instance.
- Add/configure connectors.
- Add/configure handlers.
- Start the server.

Now our server ready to handle requests from any HTTP clients (Web browser, Jetty HTTP client etc.)

Table of Content

Starting A Server

```
public class SimpleServer {
	public static void main(String[] args) throws Exception {
		Server server = new Server(8080);
		server.start();
		server.join();
	}
}
```

Let’s create one!

Connectors

- Connectors are links of the server to the outside world which supports the HTTP protocol.
- Each connector element represents a port that the Jetty server will listen on.
- The connectors have just one job → To listen for messages and pass them to the core engine.
- Using connectors, we can control how the HTTP server listens for messages.
- A default connector is created when a server starts if no connectors are specified.
- One server can have multiple connectors.

Connectors

- Connectors allow us to control
  - ServerHost
  - ServerPort
  - IdleTimeout
  - MaxRequestSize
  - MaxRequestParameters
  - MaxResponseSize
  - ThreadPoolSize
  - Security scheme (one can configure a HTTPS connector)

Connectors

```
public class SimpleConnector {
	public static void main(String[] args) throws Exception {
		Server server = new Server();
		ServerConnector http = new ServerConnector(server);
		http.setHost("localhost");
		http.setPort(8080);
		http.setIdleTimeout(30000);
		server.addConnector(http);
		server.start();
		server.join();
	}
}
```

Request Handlers

- In order to handle incoming requests, Jetty requires registration of request handlers. A handler may
  - Examine/modify HTTP requests.
  - Generate the complete HTTP response.
  - Call another handler.
  - Select one or many handlers for invocation.
- To implement handler:
  - Create handler by extending AbstractHandler class
  - Implement the handle method. (remember, it can be executed by multiple threads)

Request Handlers

```
public class SimpleHandler extends AbstractHandler {
	public void handle(String target, Request baseRequest, HttpServletRequest	request,HttpServletResponse response) throws IOException, ServletException {
		response.setContentType("text/html;charset=utf-8");
		response.setStatus(HttpServletResponse.SC_OK);
		baseRequest.setHandled(true);
		response.getWriter().println("<h1>Hello World</h1>");
	}
}
```

Let’s see it in action!

Parameters to Request Handler

The parameters passed to the handle method are:
● *String target (arg0)* – the target of the request, which is either a URI or a name from a named dispatcher.
● *Request baseRequest (arg1)* – the Jetty mutable request object.
● *HttpServletRequest request (arg2)* – the immutable request object.
● *HttpServletResponse response (arg3)*– the response object
● The handler sets the response status, content-type, and marks the request as handled before it generates the body of the response using a writer.

Sophisticated Request Handlers

The parameters passed to the handle method are:
● A **Handler Collection** holds a collection of other handlers and calls each handler in order.
● A **Handler List** is a Handler Collection that calls each handler in turn until either an exception is thrown or request.isHandled() returns true.
● A **Handler Wrapper** is a handler base class that can be used to daisy chain handlers together in the style of aspect-oriented programming.

Table of Content

Passing Arguments in HTTP Request

- http://localhost:8080/increment
- http://localhost:8080/decrement
- http://localhost:8080/getcount
- http://localhost:8080/addby?value=10
- http://localhost:8080/counter?type=increment
- http://localhost:8080/counter?type=decrement
- http://localhost:8080/counter?type=getcount
- http://localhost:8080/counter?type=addby&value=10

Retrieving Arguments

- ***getParameter("parameter-name")*** is available in HTTPServlet class to retrieve URL parameter.
- ***getParameterMap()*** is available in HTTPServlet class to retrieve "all" parameters and their values.
- Jetty supports duplicate parameters with different values, part of HTTPServlet specification but not HTTP specification.
- Encode and decode parameter values and parameter names using URLEncoder and URLDecoder.
- Be careful to encode and decode your query strings (if needed)!

Do we need POST Requests?

- Does “GET” requests suffice for all scenarios that a “POST” request can be used for ?
- One can send complicated data-structures using “GET” encodes in the query string.
  - http://localhost:8080/runProgram?input=a&input=b&input=c
  - http://localhost:8080/runProgram?n=2&input1=a&input2=b

HTTP POST

- transfer complex data structures (not limited by max URL length - 2048 characters)
- transfer binary data
- semantically different from GET: server state may change after POST request

Accessing Data from POST Requests

```
public String extractPost(HttpServletRequest req) {
        int len = req.getContentLength();
        BufferedReader reqReader = req.getReader();
        char[] cbuf = new char[len];
        reqReader.read(cbuf);
        reqReader.close();
        return new String(cbuf);
}
```

- Generic mechanism to read character encoded data.
- You can use ***req.getInputStream*** to access binary data as byte stream.

Accessing Data from POST Requests

- Once you read from the request, the read content is removed from the request.
- Make sure you close the reader to avoid leaks.
- getParameter is supported in POST requests as well only if the client encodes the data as parameters (content type: application/x-www-form-urlencoded).
- Better to use the generic method for POST.

Building Clients (HTTPClient)

- We used browser as a client → Enabled us to test server side methods.
- Let us build a client now (using HTTP for communication remember).
- **HttpClient** is the Jetty component that allows us to make requests and interpret responses to HTTP servers.
- HttpClient has two kind of APIs:
  - Blocking APIs (send a request and wait for response)
  - Non-Blocking (asynchronous) APIs (send a request and use callbacks to handle response)
- Use **ContentResponse** to get access to the contents of response (blocking APIs) or listeners like **BufferingResponseListener** or **FutureResponseListener** (non-blocking APIs)

Starting an HTTPClient

```
client = new HttpClient();
client.setMaxConnectionsPerDestination(300);
client.setExecutor(new QueuedThreadPool(20));
client.setConnectTimeout(30000);
client.start();
```

- Client can be viewed as a multi-threaded service.
- Once a client is setup, one can use ***client.GET()***, ***client.POST()*** and ***client.newRequest()*** methods to send HTTP requests.
- All set methods are optional. Default is automatically assigned.

Using HTTPClient

- The design of HTTPClient allows concurrent thread-safe exchanges.
- You want to re-use the same client for all HTTP communication, you would not want to start a new client for each HTTP exchange.
- Invoking *stop()* method on the client stops it.

Exchanging Synchronous Messages with Server (GET)

```
public static void sendGetRequest()  throws InterruptedException, ExecutionException,
TimeoutException {
        ContentResponse resp =  client.GET("http://localhost:8080/foo?bar=2");
        System.out.println("Exchange completed");
        System.out.println(resp.getStatus());
        System.out.println(resp.getContentAsString());
}
```

Exchanging Synchronous Messages with Server (GET)

- Use ***GET()*** method of HttpClient to perform *synchronous (blocking)* HTTP GET request.
- The *GET()* method returns ***ContentResponse*** object
- Use *ContentResponse* methods to access contents and status of the response:
  - ***getContentAsString()***: content as string
  - ***getContent()***: content array of bytes
  - ***getStatus()***: response HTTP status
- For this tutorial, we are only using synchronous exchanges.
- For asynchronous requests use generic request builder *newRequest()* and use callbacks for more fine-grained control at various points of request-response lifecycle.

Exchanging Synchronous Messages with Server (POST)

```
public static void sendPostRequest()  throws InterruptedException,
TimeoutException,ExecutionException {
        Request req = client .POST("http://localhost:8080/foo") .param("bar", "2") .content(new
StringContentProvider("Hello \n WWWorld"));
        ContentResponse resp = req.send();
        System.out.println(resp.getStatus());
        System.out.println(resp.getContentAsString());
}
```

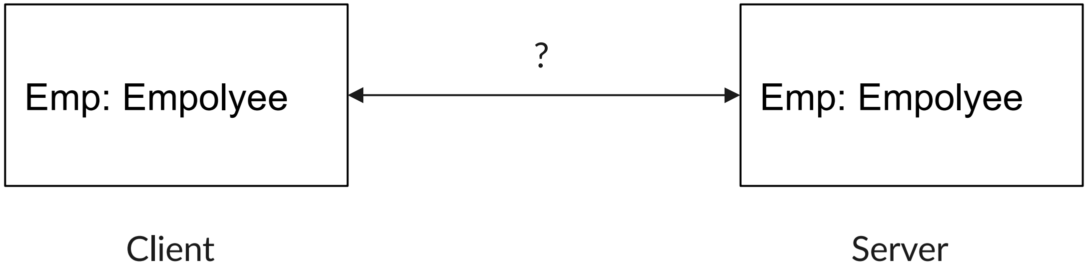
Exchanging Synchronous Messages with Server (POST)

- Use fluent request building interface (chain of method invocations) to add parameters and content.
- Use ***content()*** method along with wrappers like *StringContentProvider*, *BytesContentProvider* etc. to add response body.
- There are more content providers available – see Jetty documentation.
- Use ***send()*** method to send the request.

Table of Content

Discussion: Sending Objects over A Network

How we can transfer instance of some Java class (say, Employee) from a client to a server using HTTP?
Can it be an arbitrary object?



The Need for Serialization and Deserialization

- How do we send data-structures in text/binary form?
- We need to serialize and de-serialize.
- Possible representations → XML, JSON, binary encoding

  ....
- For this course we will use:
  - XML → **XStream** library
  - binary format → **Kryo** library
- There are other possibilities for binary serialization: Java serialization, protocol buffers, other libraries.

Basic Methods

- Xstream (http://x-stream.github.io/):
  toXML() / fromXML()
- Kryo (https://github.com/EsotericSoftware/kryo):
  writeObject() / readObject()

Serialization/Deserialization Examples

let's look at code

Advantages of Using HTTP

- Connect our applications with other HTTP servers. Allows layering.
- There are a lot of mature HTTP server/client implementations with good performance characteristics
- Use proxy servers.
- Leverage HTTP mechanisms like caching (ETags, TTL).
- Use system stack of the Internet.
- Add security layer using HTTPS.
- Most often we end up building some form of reliable message delivery protocol using sockets, why not use HTTP instead?

Disadvantages of Using HTTP

- Latency sensitive applications.
- Limitations of TCP.
- Request/response model does not fit communication pattern in all applications
  - Stream based systems
  - Online games.

Table of Content

Avoiding Excessive Synchronization

- Insufficient synchronization leads to unpredictable behavior. On the other hand, excessive synchronization can lead to
    - Reduced performance.
    - Deadlocks.
    - Broken invariant.
- **To avoid liveness and safety failures, never cede control to the client within a synchronized method or block.** [Joshua Bloch, Effective Java, Third Edition, Item 79]
- Includes: methods provided by the calling code and methods that may have been overridden.

Example: An Observable Counter

```
interface CounterObserver {
	public void counterChanged(ObservableCounter counter);
}
class ObservableCounter {
	private int c = 0;
	private final List<CounterObserver> observers = new ArrayList<CounterObserver>();
	public int getValue() {
		synchronized ( this ) { return c; }
	}
	public void incr() {
		synchronized ( this ) { c++; }
		notifyValueChanged(); /* def on next slide */
	}
/* ... */
}
```

Example: An Observable Counter (Cont'd)

```
class ObservableCounter {
    /* ... */
    public void addObserver ( CounterObserver obs ) {
        synchronized ( observers ) { observers.add ( obs ); }
    }
    public void removeObserver ( CounterObserver obs ) {
        synchronized ( observers ) { observers.remove ( obs ); }
    }
    private void notifyValueChanged () {
        synchronized ( observers ) {
            for ( CounterObserver obs : observers )
            obs.counterChanged ( this );
        }
    }
}
```

Usage Scenario

```
final ObservableCounter counter = new ObservableCounter();
counter.addObserver ( new CounterObserver() {
        public void counterChanged (ObservableCounter counter) {
                System.out.println ( counter.getValue() );
        }});
for (int i = 0; i < 100; i++)
        counter.incr();
```

What happens?

Usage Scenario

```
final ObservableCounter counter = new ObservableCounter();
counter.addObserver ( new CounterObserver() {
        public void counterChanged (ObservableCounter counter) {
                System.out.println ( counter.getValue() );
        }});
for (int i = 0; i < 100; i++)
        counter.incr();
```

What happens?=⇒ Prints “1” ... “100” and exits.

Usage Scenario

```
final ObservableCounter counter = new ObservableCounter();
counter.addObserver ( new CounterObserver() {
	public void counterChanged(ObservableCounter counter) {
		System.out.println(counter.getValue());
		if (counter.getValue() >= 42)
			counter.removeObserver(this);
	}});
for (int i = 0; i < 100; i++)
	counter.incr();
```

What happens?

Usage Scenario

```
final ObservableCounter counter = new ObservableCounter();
counter.addObserver ( new CounterObserver() {
	public void counterChanged(ObservableCounter counter) {
		System.out.println(counter.getValue());
		if (counter.getValue() >= 42)
			counter.removeObserver(this);
	}});
for (int i = 0; i < 100; i++)
	counter.incr();
```

What happens?=⇒ Prints “1” ... “42” and throws ConcurrentModificationException!

The Problem

- Reentrant behavior via **alien method call**.

```
public void removeObserver(CounterObserver obs) {
        synchronized(observers){ observers.remove(obs); }
}
private void notifyValueChanged() {
        synchronized(observers) {
                for (CounterObserver obs : observers)
                /* obs.counterChanged() alien! */
                obs.counterChanged(this);
        }
}
```

- We synchronized on observers to prevent modification while iterating.
- Intrinsic locks are reentrant.

A Contrived Example

- Call removeObserver() from other thread:

```
counter.addObserver(new CounterObserver() {
        public void counterChanged(final ObservableCounter c) {
                System.out.println(counter.getValue());
                if (counter.getValue() >= 42) {
                /* spawn new thread t,
                    call c.removeObserver(this) from t,
                    wait for t to finish */
                }
}});  (see supplementary material for full code)
```

A Contrived Example

- Call removeObserver() from other thread:

```
counter.addObserver(new CounterObserver() {
        public void counterChanged(final ObservableCounter c) {
                System.out.println(counter.getValue());
                if (counter.getValue() >= 42) {
                /* spawn new thread t,
                   call c.removeObserver(this) from t,
                   wait for t to finish */
                }
}});  (see supplementary material for full code)
```

- Now, deadlock!

The Solution

- How to solve this problem and still provide the desired functionality?

The Solution

- How to solve this problem and still provide the desired functionality?
- Move call to alien code outside critical region.

```
private void notifyValueChanged() {
	List<CounterObserver> snapshot = null;
	synchronized(observers){
		snapshot = new ArrayList<CounterObserver>(observers);
	}
	for (CounterObserver obs : snapshot)
		obs.counterChanged(this);
}
```

- This is also called an ***open call*** – alien code is only called when the system is in a consistent state and no locks are held.
- See CopyOnWriteArrayList for a lock-free approach.

Hazards due to Over-synchronization

Three main hazards:

- Exception or deadlocks (previous examples)
- Broken invariants.
  - Method might temporarily invalidate invariant, call alien method, then reestablish invarient. Alien method might operate on inconsistent state.
- Performance problems.
  - You don't know how much time alien code spends. Holds lock while running.
  - Synchronization has a cost for single-threaded use as well (e.g. precludes optimizations).

Recap

- Do as little work as possible when inside a synchronized region.
- Never call alien methods within synchronized region.
- Don’t make a class thread-safe if it is mostly used in a single-threaded context.
- See Item 79 in [Effective Java, Third Edition] (supplementary material).

Table of Content

Documenting Levels of Thread Safety

- How a class behaves in concurrent context is part of its contract with its clients. If contract is not clear, programmer is forced to make assumptions.
- Thread safety not synonymous with presence of synchronized modifier.
- Thread safety is not an all-or-nothing property.

Levels of Thread Safety

- Rough characterization of thread safety levels (not exhaustive):
  - **immutable.** Instances have no mutable state. Obviously thread-safe. Examples: *String*, *Integer* ...
  - **unconditionally thread-safe.** Instances mutable, but has sufficient internal synchronization. Ex.: *ConcurrentHashMap*, ...
  - **conditionally thread-safe.** Like above, but some methods require external synchronization for safe concurrent use.
  - **not thread-safe.** Only designed for single-threaded usage. Requires external synchronization. Ex.: *ArrayList*, *HashMap*, ...
  - **thread-hostile.** Cannot be used in a concurrent context, no matter what, e.g. due to global state. Always a result of a flawed design—only occurs in legacy code.

Conditional Thread Safety

- Documenting conditional thread safety requires care.
  - Document which invocation sequences require external synchronization.
  - Document which lock(s) must be acquired.

Conditional Thread Safety

- Example from *Collections.synchronizedMap* documentation:
- It is imperative that the user manually synchronize on the returned map when iterating over any of its collection views:

```
Map m = Collections.synchronizedMap(new HashMap());
        ...
Set s = m.keySet(); // Needn't be in synchronized block
        ...
synchronized (m) { // Synchronizing on m, not s!
        Iterator i = s.iterator(); // Must be in synchronized block
        while (i.hasNext())
                foo(i.next());
}
```

- Failure to follow this advice may result in non-deterministic behavior.

Recap

- Every class should clearly document its thread safety properties.
- Conditionally thread-safe classes must document which usage patterns require external synchronization, and which locks to aquire.
- See **Item 82** in [Effective Java, Third Edition] (supplementary material).

Table of Content

Publication and Escape

- *Publishing* an object: make it available outside its current scope.
  - By storing a reference to it in a field accessible to other code.
  - By passing it to a method in another class.
  - By returning a reference to it.

    …
- We often don't want to publish objects comprising internal state.
  - Ex.: Linked list nodes are internal to a list implementation.
  - Breaks encapsulation; harder to maintain invariants.
  - Unintended publication is called escape.
- Escapes may not lead to errors immediately, but they are very confusing once they happen!

Example: Escape via Public Field

```
public static Set<Secret> secrets;
public void initialize() {
        secrets = new HashSet<Secret>();
}
```

- Everyone can read the public **secrets** field, including alien code.
- For every published **Secret**, we also implicitly publish its non-private references.

Example: Escape via Return

```
class UnsafeChars {
        private Char[] lowerAlphas = new Char[] {
                'a', 'b', ...
        };
        public Char[] getLowerAlphas() { return lowerAlphas; }
}
```

- What’s wrong here?

Example: Escape via Return

```
class UnsafeChars {
	private Char[] lowerAlphas = new Char[] {
		'a', 'b', ...
	};
	public Char[] getLowerAlphas() { return lowerAlphas; }
}
```

• What's wrong here?

- The *lowerAlphas* field is internal to *UnsafeChars* and intended to be immutable.
- By returning it, alien code may modify it (arrays are mutable).

Example: Escape via Inner Classes

```
public class ThisEscape {
	public ThisEscape(EventSource source) {
		source.registerListener(
			new EventListener() {
				public void onEvent(Event e) {doSomething(e);}
		});
	}
	private void doSomething(Event e) { ... }
}
```

- Subtle problem: The *this* reference to *ThisEscape* instance (implicitly) published too early!
- Event source might call *doSomething* before class is fully initialized. Problem if escape is last?

Example: Escape via Inner Classes

```
public class ThisEscape {
	public ThisEscape(EventSource source) {
		source.registerListener(
			new EventListener() {
				public void onEvent(Event e) {doSomething(e);}
		});
	}
	private void doSomething(Event e) { … }
}
```

- Subtle problem: The *this* reference to *ThisEscape* instance (implicitly) published too early!
- Event source might call *doSomething* before class is fully initialized. Problem if escape is last?
- Yes, Subclass constructors run after superclass constructors.

Safe Construction

Do not let reference to this escape during construction.

```
public class SafeListener {
	private final EventListener listener;
	private SafeListener() {
		listener = new EventListener() {
			public void onEvent(Event e) { doSomething(e); }
		};
	}
	private void doSomething(Event e) { … }
	public static SafeListener newInstance(EventSource source) {
		SafeListener safe = new SafeListener();
		source.registerListener(safe.listener);
		return safe;
	}
}
```

Thread Confinement

- Using data confined to a single thread is automatically thread-safe, even if encapsulating class is not.
- Thread confinement idioms are used in many places.
  - Ex.: The Swing GUI toolkit require all state modifications to occur from main thread. **Swing objects are not thread-safe.**
  - Other threads must submit jobs to an event queue.
- Thread confinement simplifies reasoning.

Thread Confinement

Three coding practices for enforcing thread confinement:

- Ad-hoc thread confinement via usage conventions.
  - Ex.: The event queue of the Swing toolkit.
  - Conventions cannot be enforced by compiler.
  - Requires discipline and proper documentation.
- Stack confinement.
  - Declare references to non-thread safe objects on the stack.
  - Don’t let references escape.
- Use **ThreadLocal** to hold values.
  - Current value is local to each thread.

Immutability

- If an object is immutable, then it is automatically safe to share it.
- No formal definition of immutability. Here are five conventions for rock-solid immutable objects:
  - Don’t provide methods that modify object state.
  - Make the class **final**.
    - Subclasses might violate the immutability contract.
  - Make all fields **final**.
    - Documents intent. Also good practice in general.
    - As a bonus: unsynchronized sharing (know what you’re doing).
  - Make all fields **private**.
    - Prevents publishing references to internal mutable state.

  ...

Immutability

- If an object is immutable, then it is automatically safe to share it.
- No formal definition of immutability. Here are five conventions for rock-solid immutable objects:

  ...

  - Ensure exclusive access to internal mutable state.

    Make defensive copies of mutable objects provided by clients

    e.g. internalList = new ArrayList<T>(clientList)).

Sharing Immutable Objects

- Sharing immutable objects require less synchronization than mutable objects.
- **volatile** fields can be used to atomically publish all fields of an immutable object.
- Fields can be read by other threads without synchronization.
  Not the case for mutable objects.

Safe Publication

```
class Holder {
	private int n;
	public Holder(int n) { this.n = n;}
	public void assertSanity() {
		if (n!= n)
			throw new AssertionError();
	}
}
class Unsafe {
	public Holder holder;
	public void publish() { holder = new Holder(42); }
}
```

Might get exception from *holder.assertSanity()* if runs concurrently with *publish()*!

Safe Publication

- Unsafety of previous example due to inadequate synchronization, as discussed earlier.
- Reordering might cause **Holder** reference to be visible before constructor has finished.
- Safe publication: Object reference **and** object state must be made visible at the same time.
    - Initialization must *happen-before* publication.
- Publishing via concurrent collections is safe due to internal synchronization.

Useful Sharing Policies

Try to categorize object references as following one of the following policies [Goetz et al., 2005, Section 3.5.6]:

- **Thread-confined**. Accessed exclusively by owning thread.
- **Shared read-only**. Shared between threads without additional synchronization since no threads modify its state. Includes immutable objects.
- **Shared thread-safe**. Shared between threads. Has adequate internal synchronization to support safe concurrent modification.
- ***Guarded.*** Shared between threads. Only conditionally thread-safe: modification requires guarding by some lock.

Recap

- Do not allow internal, mutable state to escape.
- Do not allow reference to **this** to escape during construction.
- Separate construction from registration/configuration.
- Limit sharing; confine state when possible.
- No synchronization needed if data confined to single thread.
- Design objects to be immutable when possible.
- See Item 15 in [Bloch, 2008] and Chapter 3 in [Goetz et al., 2005] (supplementary material).

Table of Content

When to Override Equals

- By default, the equals method compares objects by identity.
  - That is, reference equality.
  - You can override equals to provide a coarser, semantic equivalence.
- When do we want a coarser equals?

When to Override Equals

- By default, the equals method compares objects by identity.
  - That is, reference equality.
  - You can override equals to provide a coarser, semantic equivalence.
- When do we want a coarser equals? Often what we want when our classes represent values:
  - Integer, String, Point2D, Email, Hashtag, …
  - Database records.
- When do we NOT want a coarser equals?

When to Override Equals

- By default, the equals method compares objects by identity.
  - That is, reference equality.
  - You can override equals to provide a coarser, semantic equivalence.
- When do we want a coarser equals? Often what we want when our classes represent values:
  - Integer, String, Point2D, Email, Hashtag, ...
  - Database records.
- When do we NOT want a coarser equals? For example:
  - If each instance of a class is inherently unique. E.g. threads, files, game entities, ...
  - You don't care for ``logical equality''. E.g. Random could override equals, but doesn't.
  - An appropriate override exists in superclass. E.g. AbstractSet.

Contract

- Reflexivity
  For x not null, x.equals(x) == true.
- Symmetry
  x.equals(y) == y.equals(x) for x, y not null.
- Transitivity
  For x, y, z not null: If x.equals(y) and y.equals(z) then x.equals(z).
- Consistency
  Repeated invocations give same result unless one or both of the compared objects change.
- Non-nullity
  If x not null, then x.equals(null) == false.

Reflexivity

It's pretty hard to violate reflexivity.

Symmetry

```
final class CaseInsensitiveString {
	private final String s;
	public CaseInsensitiveString(s) {
		if (s == null) throw NullPointerException;
		this.s = s;
	}
	@Override public boolean equals(Object o) {
		if (o instanceof CaseInsensitiveString)	return s.equalsIgnoreCase(((CaseInsensitiveString) o).s);
		if (o instanceof String)	return s.equalsIgnoreCase((String) o);
		return false;
	}
}
```

Symmetry

```
final class CaseInsensitiveString {
	private final String s;
	public CaseInsensitiveString(s) {
		if (s == null) throw NullPointerException;
		this.s = s;
	}
	@Override public boolean equals(Object o) {
		if (o instanceof CaseInsensitiveString)	return s.equalsIgnoreCase(((CaseInsensitiveString) o).s);
		if (o instanceof String)	return s.equalsIgnoreCase((String) o);
		return false;
	}
}
```

We can have cis.equals("abc") but not "abc".equals(cis).
Do not equate objects of distinct implementation types.

Transitivity

- When you add value components to a class by extending it, you may be tempted to override the equals method.
- Unfortunately, this will violate transitivity.
- Even worse: it is impossible to add a value component to an instantiable class while preserving the equals contract.
- Note that it is possible to add value components to abstract classes and preserve the equals contract.

Consistency

- In general: Don't write equals implementations that rely on unreliable resources.
  For example, don't depend on network or file system resources.

Non-nullity

- You usually don't accidentally return true when comparing with null.
- You may, however, accidentally throw a NullPointerException when calling .equals(null).
- This is not allowed---you should return false instead.
- Note that instanceof is adequate, as it includes an implicit null check:

```
@Override public boolean equals(Object o) {
	if (!(o instanceof TheType))
		return false;
	TheType t = (TheType) o;
	...
}
```

General Recipe for Implementing Equals

- Make your class final or abstract.
- Use instanceof to check that argument is non-null and of right type.
- Cast argument to correct type.
- Compare fields according to your equality semantics.
    - Use == (reference/value equality) for primitive types.
    - Use .equals() for objects.
- Verify that your implementation is reflexive, symmetric, transitive and consistent.

Besides: **Always override hashCode if you override equals.**

Table of Content

General Recipe for Implementing Equals

- Why do all object have a hashCode() method?

General Recipe for Implementing Equals

- Why do all object have a hashCode() method?
  To optimize data structures relying on equals (e.g. HashSet).

- Consequence: You must override hashCode() when you override equals().

Contract for Hashcode

- Consistency

  Multiple invocations to hashCode must return same integer provided no equals-relevant information have changed between invocations.
- Compatibility with equals

  If two objects are equals, then they have the same hashCode.

It is NOT a requirement that distinct objects have distinct hashcodes.

However, performance is improved if they generally do.

```
@Override public int hashCode() { return 42; }
```

Implementing Hashcode

- Only fields compared in equals should affect the hashcode.
- The **Objects.hash()** method gives a good scrambling.

```
public class Point {
	private final int x, y;
	…
	@Override public boolean equals(Object o) { … }
	@Override public int hashCode() {
		return Objects.hash(x, y);
	}
}
```

Table of Content

Comparable

- Comparison method not in Object but separate interface:

```
interface Comparable<T> {
	int compareTo(T obj);
}
```

- Similar to equals, but allows order comparisons in addition to equality testing.

a.compareTo(b)	Meaning
<0	a less than b
=0	a equal to b
>0	a greater than b

Why Implement it?

- Implementing Comparable<T> allows your class to be used with generic algorithms and structures depending on this.
- Example: String implements it. This sorts a list of strings:

```
public void m(){
        String[] strings =new String[] { "foo", "baz", "bar" };
        Set<String> s = new TreeSet<String>();
        Collections.addAll(s, strings);
        System.out.println(s);
}
```

prints [bar, baz, foo]

- In general, implement it if you have a value class (as for equals).

Contract

sgn(x): -1, 0, 1 if x negative, zero or positive, respectively.

- Anti-symmetry
  sgn(x.compareTo(y)) == -sgn(y.compareTo(x)).
- Transitivity
  If x.compareTo(y) > 0 and y.compareTo(z) > 0, then x.compareTo(z) > 0.
- Congruence
  If x.compareTo(y) == 0, then for all z, sgn(x.compareTo(z)) == sgn(y.compareTo(z)).
- Consistent with equals
  x.compareTo(y) == 0 if and only if x.equals(y) == true. Not required (but strongly recommended)

Comparable induces Equivalence

- Comparable induces equivalence: Objects a and b equivalent iff a.compareTo(b) == 0.
- Contract implies reflexivity, symmetry, transitivity.
- As with equals: cannot extend classes with value components and preserve contract.

Two Equivalence Relations

- Equivalence induced by natural order may be different from equals. Document this.
  E.g. The natural ordering on MyClass is inconsistent with equals.
- Not necessarily a catastrophe, but might break interfaces. For example:
  - HashSet<T> uses equals and hashCode;
  - TreeSet<T> uses natural order equivalence.
  - Consequence: Changing set implementation can change results.

Implementing Comparable

- Implement by comparing field-by-field, as with equals.
- Unlike equals, type checks are not needed since Comparable<T> is statically typed.
- Unlike equals, compareTo should throw NullPointerException on comparison with null.

Optimize Judiciously

- Don't be too clever.
- It might be tempting to compare two int fields as follows:

```
public class MyClass
        implements Comparable<MyClass> {
        private int x;
        ...
        public int compareTo(MyClass other){
        return x - other.x;
        }
}
```

- Integer arithmetic might overflow! Better:

```
 return Integer.compare(x, other.x);
```

Table of Content

Cloning

```
public class Object {
        ...
        protected Object clone() { ... };
}
```

- The protected Object.clone method returns a field-by-field clone of the object it is called on.
- The cloned object is created without calling any constructors.
  - This cannot be emulated otherwise.
- clone throws an exception unless class implements the empty interface Cloneable
  - This is a pretty non-standard design choice.
  - To be frank, it is a bit of a hack.

Problems with Clone

- No way of telling if a given class has a callable clone method.
  - Since Cloneable is empty and Object.clone is protected.
- Weak contract. Nothing is an absolute requirement.
  - should satisfy x.clone() != x.
  - should satisfy x.clone().getClass() == x.getClass().
  - should satisfy x.clone().equals(x) == true.
  - may copy internal data structures (deep-copy).
  - should not call any constructors.
- Contract can only be satisfied if implementations call super.clone.
- Does not permit final fields referring to mutable objects if they need to be deeply copied.

Alternative Ways of Cloning

- It's not really worth it to deal with all the problems with clone.
- Better to just provide your own means of copying. For example:

Provide a constructor with a single argument whose type is the class containing the constructor.

```
public MyClass(MyClass o) { ... }
```

Provide a static factory method which can create a copy.

```
public static MyClass newInstance(MyClass o) { ... }
```

Conversion Factories

- Copy factories may also be defined to take interfaces as arguments.
- This effectively gives you conversion factories.
- Much more powerful than clone.

Recap

- Don't use clone except, perhaps, to copy arrays.
- Prefer rolling your own methods for copying.