VŠB TECHNICKÁ UNIVERZITA OSTRAVA

www.vsb.cz

Programming in Java 2

Jan Kožusznik, David Ježek
david.jezek@vsb.cz

Tel: 597 325 874
Místnost: EA406

1th Lecture

- Maven
- Modules

Architecture

The software industry delights in taking words and stretching them into a myriad of subtly contradictory meanings. One of the biggest sufferers is "architecture." I tend to look at "architecture" as one of those impressive-sounding words, used primarily to indicate that we're talking something that's important. But I'm pragmatic enough not to let my cynicism get in the way of attracting people to my book. :-)

"Architecture" is a term that lots of people try to define, with little agreement. There are two common elements: One is the highest-level breakdown of a system into its parts; the other, decisions that are hard to change. It's also increasingly realized that there isn't just one way to state a system's architecture; rather, there are multiple architectures in a system, and the view of what is architecturally significant is one that can change over a system's lifetime.

From time to time Ralph Johnson has a truly remarkable posting on a mailing list, and he did one on architecture just as I was finishing the draft of this book. In this posting he brought out the point that architecture is a subjective thing, a shared understanding of a system's design by the expert developers on a project. Commonly this shared understanding is in the form of the major components of the system and how they interact. It's also about decisions, in that it's the decisions that developers wish they could get right early on because they're perceived as hard to change. The subjectivity comes in here as well because, if you find that something is easier to change than you once thought, then it's no longer architectural. In the end architecture boils down to the important stuff—whatever that is.

In this book I present my perception of the major parts of an enterprise application and of the decisions I wish I could get right early on. The architectural pattern I like the most is that of layers, which I describe more in Chapter 1. This book is thus about how you decompose an enterprise application into layers and how these layers work together. Most nontrivial enterprise applications use a layered architecture of some form, but in some situations other approaches, such as pipes and filters, are valuable. I don't go into those situations, focusing instead on the context of a layered architecture because it's the most widely useful.

Some of the patterns in this book can reasonably be called architectural, in that they represent significant decisions about these parts; others are more about design and help you to realize that architecture. I don't make any strong attempt to separate the two, since what is architectural or not is so subjective.

Maven overview

- Can generate deployable artifacts from source code
- Compile, pack, test and distribute your source code
- Manage dependencies on external libraries
- Scalable – for small projects but also for big & complex projects

https://en.wikipedia.org/wiki/Enterprise_software

Services provided by enterprise software are typically business-oriented tools such as online shopping and online payment processing, interactive product catalogue, automated billing systems, security, enterprise content management, IT service management, customer relationship management, enterprise resource planning, business intelligence, project management, collaboration, human resource management, manufacturing, enterprise application

enterprise software a software suite with common business applications, tools for modeling how the entire organization works, and development tools for building applications unique to the organizationintegration, and enterprise forms automation.

Convention over configuration

src	
├─main	
│ ├─java	Application/Library sources
│ ├─resources	Application/Library resources
│ ├─filters	Resource filter files
│ └─webapp	Web application sources
├─test	
│ ├─java	Test sources
│ ├─resources	Test resources
│ └─filters	Test resource filter files
├─it	Integration Tests (primarily for plugins)
├─assembly	Assembly descriptors
├─site	Site
├─LICENSE.txt	Project's license
├─NOTICE.txt	Notices and attributions required by libraries that the project depends on
├─README.txt	Project's readme
└─pom.xml	file descriptive of the project

https://en.wikipedia.org/wiki/Enterprise_software

Services provided by enterprise software are typically business-oriented tools such as online shopping and online payment processing, interactive product catalogue, automated billing systems, security, enterprise content management, IT service management, customer relationship management, enterprise resource planning, business intelligence, project management, collaboration, human resource management, manufacturing, enterprise application

enterprise software a software suite with common business applications, tools for modeling how the entire organization works, and development tools for building applications unique to the organizationintegration, and enterprise forms automation.

Enterprise Applications

Lots of people write computer software, and we call all of it software development. However, there are distinct kinds of software out there, each of which has its own challenges and complexities. This comes out when I talk with some of my friends in the telecom field. In some ways enterprise applications are much easier than telecoms software—we don't have very hard multithreading problems, and we don't have hardware and software integration. But in other ways it's much tougher. Enterprise applications often have complex data—and lots of it—to work on, together with business rules that fail all tests of logical reasoning. Although some techniques and patterns are relevant for all kinds of software, many are relevant for only one particular branch.

In my career I've concentrated on enterprise applications, so my patterns here are all about that. (Other terms for enterprise applications include "information systems" or, for those with a long memory, "data processing.") But what do I mean by the term "enterprise application"? I can't give a precise definition, but I can give some indication of my meaning.

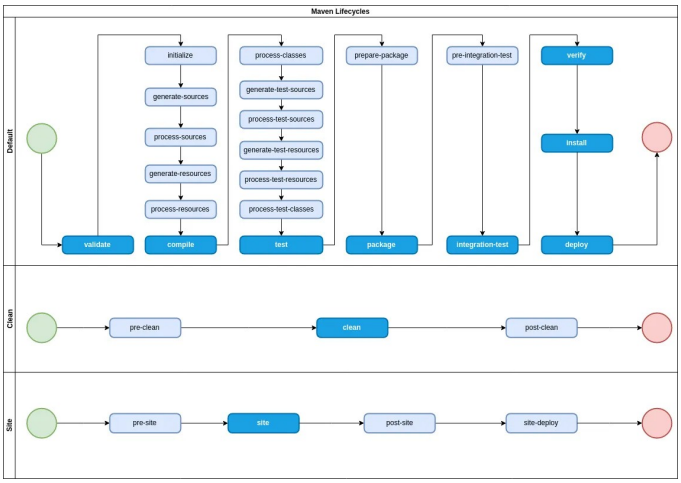
I'll start with examples. Enterprise applications include payroll, patient records, shipping tracking, cost analysis, credit scoring, insurance, supply chain, accounting, customer service, and foreign exchange trading. Enterprise applications don't include automobile fuel injection, word processors, elevator controllers, chemical plant controllers, telephone switches, operating systems, compilers, and games.

Enterprise applications usually involve persistent data. The data is persistent because it needs to be around between multiple runs of the program—indeed, it usually needs to persist for several years. Also during this time there will be many changes in the programs that use it. It will often outlast the hardware that originally created much of it, and outlast operating systems and compilers. During that time there'll be many changes to the structure of the data in order to store new pieces of information without disturbing the old pieces. Even if there's a fundamental change and the company installs a completely new application to handle a job, the data has to be migrated to the new application.

There's usually a lot of data—a moderate system will have over 1 GB of data organized in tens of millions of records—so much that managing it is a major part of the system. Older systems used indexed file structures such as IBM's VSAM and ISAM. Modern systems usually use databases, mostly relational databases. The design and feeding of these databases has turned into a subprofession of its own.

Usually many people access data concurrently. For many systems this may be less than a hundred people, but for Web-based systems that talk over the Internet this goes up by orders of magnitude. With so many people there are definite issues in ensuring that all of them can access the system properly. But even without that many people, there are still problems in making sure that two people don't access the same data at the same time in a way that causes errors.

Maven build lifecycles



https://en.wikipedia.org/wiki/Enterprise_software

Services provided by enterprise software are typically business-oriented tools such as online shopping and online payment processing, interactive product catalogue, automated billing systems, security, enterprise content management, IT service management, customer relationship management, enterprise resource planning, business intelligence, project management, collaboration, human resource management, manufacturing, enterprise application integration, and enterprise forms automation.

Default Lifecycle

- **validate** - validate the project is correct and all necessary information is available.
- **initialize** - initialize build state, e.g. set properties or create directories.
- **generate-sources** - generate any source code for inclusion in compilation.
- **process-sources** - process the source code, for example to filter any values.
- **generate-resources** - generate resources for inclusion in the package.
- **process-resources** - copy and process the resources into the destination directory, ready for packaging.
- **compile** - compile the source code of the project.
- **process-classes** - post-process the generated files from compilation, for example to do bytecode enhancement on Java classes.

Default Lifecycle

- **generate-test-sources** - generate any test source code for inclusion in compilation.
- **process-test-sources** - process the test source code, for example to filter any values.
- **generate-test-resources** - create resources for testing.
- **process-test-resources** - copy and process the resources into the test destination directory.
- **test-compile** - compile the test source code into the test destination directory
- **process-test-classes** - post-process the generated files from test compilation, for example to do bytecode enhancement on Java classes.
- **test** - run tests using a suitable unit testing framework. These tests should not require the code be packaged or deployed.

Default Lifecycle

- **prepare-package** - perform any operations necessary to prepare a package before the actual packaging. This often results in an unpacked, processed version of the package.
- **package** - take the compiled code and package it in its distributable format, such as a JAR.
- **pre-integration-test** - perform actions required before integration tests are executed. This may involve things such as setting up the required environment.
- **integration-test** - process and deploy the package if necessary into an environment where integration tests can be run.
- **post-integration-test** - perform actions required after integration tests have been executed. This may including cleaning up the environment.
- **verify** - run any checks to verify the package is valid and meets quality criteria.
- **install** - install the package into the local repository, for use as a dependency in other projects locally.
- **deploy** - done in an integration or release environment, copies the final package to the remote repository for sharing with other developers and projects.

Clean Lifecycle

- **pre-clean** - execute processes needed prior to the actual project cleaning
- **clean** - remove all files generated by the previous build
- **post-clean** - execute processes needed to finalize the project cleaning

Site Lifecycle

- **pre-site** - execute processes needed prior to the actual project site generation
- **site** - generate the project's site documentation
- **post-site** - execute processes needed to finalize the site generation, and to prepare for site deployment
- **site-deploy** - deploy the generated site documentation to the specified web server

Running lifecycle

mvn <phase>

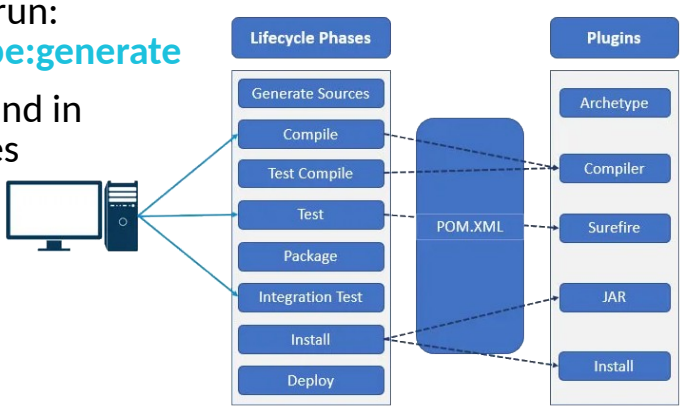
- runs lifecycle containing given phase
- stops in the phase but runs every previous phase
-
-

https://en.wikipedia.org/wiki/Enterprise_software

Services provided by enterprise software are typically business-oriented tools such as online shopping and online payment processing, interactive product catalogue, automated billing systems, security, enterprise content management, IT service management, customer relationship management, enterprise resource planning, business intelligence, project management, collaboration, human resource management, manufacturing, enterprise application integration, and enterprise forms automation.

Plugins

- Provide goals
- Goals can be run:
  mvn archetype:generate
- Goals are bound in specific phases



Kinds of Enterprise Application

When we discuss how to design enterprise applications, and what patterns to use, it's important to realize that enterprise applications are all different and that different problems lead to different ways of doing things. I have a set of alarm bells that go off when people say, "Always do this." For me much of the challenge (and interest) in design is in knowing about alternatives and judging the trade-offs of using one alternative over another. There is a large space of alternatives to choose from, but here I'll pick three points on this very big plane.

Consider a B2C (business to customer) online retailer: People browse and—with luck and a shopping cart—buy. For such a system we need to be able to handle a very high volume of users, so our solution needs to be not only reasonably efficient in terms of resources used but also scalable so that you can increase the load by adding more hardware. The domain logic for such an application can be pretty straightforward: order capturing, some relatively simple pricing and shipping calculations, and shipment notification. We want anyone to be able access the system easily, so that implies a pretty generic Web presentation that can be used with the widest possible range of browsers. Data source includes a database for holding orders and perhaps some communication with an inventory system to help with availability and delivery information.

Contrast this with a system that automates the processing of leasing agreements. In some ways this is a much simpler system than the B2C retailer's because there are many fewer users—no more than a hundred or so at one time. Where it's more complicated is in the business logic. Calculating monthly bills on a lease, handling events such as early returns and late payments, and validating data as a lease is booked are all complicated tasks, since much of the leasing industry's competition comes in the form of little variations over deals done in the past. A complex business domain such as this is challenging because the rules are so arbitrary.

Such a system also has more complexity in the user interface (UI). At the least this means a much more involved HTML interface with more, and more complex, screens. Often these systems have UI demands that lead users to want a more sophisticated presentation than a HTML front end allows, so a more conventional rich-client interface is needed. A more complex user interaction also leads to more complicated transaction behavior: Booking a lease may take an hour or two, during which time the user is in a logical transaction. We also see a complex database schema with perhaps two hundred tables and connections to packages for asset valuation and pricing.

A third example point is a simple expense-tracking system for a small company. Such a system has few users and simple logic and can easily be made accessible across the company with an HTML presentation. The only data source is a few tables in a database. As simple as it is, a system like this is not devoid of a challenge. You have to build it very quickly and you have to bear in mind that

Artifacts

- File resulting from packaging
- Definition file pom.xml
- Unique ID – artifacts coordinates:
  - artifactId
  - groupId
  - Version

Minimal pom.xml

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.mycompany.app</groupId>
  <artifactId>my-app</artifactId>
  <version>1.0.0</version>
</project>
```


Thinking About Performance

Many architectural decisions are about performance. For most performance issues I prefer to get a system up and running, instrument it, and then use a disciplined optimization process based on measurement. However, some architectural decisions affect performance in a way that's difficult to fix with later optimization. And even when it is easy to fix, people involved in the project worry about these decisions early.

It's always difficult to talk about performance in a book such as this. The reason that it's so difficult is that any advice about performance should not be treated as fact until it's measured on your configuration. Too often I've seen designs used or rejected because of performance considerations, which turn out to be bogus once somebody actually does some measurements on the real setup used for the application.

I give a few guidelines in this book, including minimizing remote calls, which has been good performance advice for quite a while. Even so, you should verify every tip by measuring on your application. Similarly there are several occasions where code examples in this book sacrifice performance for understandability. Again it's up to you to apply the optimizations for your environment. Whenever you do a performance optimization, however, you must measure both before and after, otherwise, you may just be making your code harder to read.

There's an important corollary to this: A significant change in configuration may invalidate any facts about performance. Thus, if you upgrade to a new version of your virtual machine, hardware, database, or almost anything else, you must redo your performance optimizations and make sure they're still helping. In many cases a new configuration can change things. Indeed, you may find that an optimization you did in the past to improve performance actually hurts performance in the new environment.

Another problem with talking about performance is the fact that many terms are used in an inconsistent way. The most noted victim of this is "scalability," which is regularly used to mean half a dozen different things. Here are the terms I use.

Response time is the amount of time it takes for the system to process a request from the outside. This may be a UI action, such as pressing a button, or a server API call.

Responsiveness is about how quickly the system acknowledges a request as opposed to processing it. This is important in many systems because users may become frustrated if a system has low responsiveness, even if its response time is good. If your system waits during the whole request, then your responsiveness and response time are the same. However, if you indicate that you've received the request before you complete, then your responsiveness is better. Providing a progress bar during a file copy improves the responsiveness of your user interface, even though it doesn't improve response time.

Latency is the minimum time required to get any form of response, even if the work to be done is nonexistent. It's usually the big issue in remote systems. If I ask a program to do nothing, but to tell me when it's done doing nothing, then I should get an almost instantaneous response if the program runs on my laptop. However, if the program runs on a remote computer, I may get a few seconds just because of the time taken for the request and response to make their way across the wire. As an application developer, I can usually do nothing to improve latency. Latency is also the reason why you should minimize remote calls.

Throughput is how much stuff you can do in a given amount of time. If you're timing the copying of a file, throughput might be measured in bytes per second. For enterprise applications a typical measure is transactions per second (tps), but the problem is that this depends on the complexity of your transaction. For your particular system you should pick a common set of transactions.

In this terminology performance is either throughput or response time—whichever matters more to you. It can sometimes be difficult to talk about performance when a technique improves throughput but decreases response time, so it's best to use the more precise term. From a user's perspective responsiveness may be more important than response time, so improving responsiveness at a cost of response time or throughput will increase performance.

Load is a statement of how much stress a system is under, which might be measured in how many users are currently connected to it. The load is usually a context for some other measurement, such as a response time. Thus, you may say that the response time for some request is 0.5 seconds with 10 users and 2 seconds with 20 users.

Load sensitivity is an expression of how the response time varies with the load. Let's say that system A has a response time of 0.5 seconds for 10 through 20 users and system B has a response time of 0.2 seconds for 10 users that rises to 2 seconds for 20 users. In this case system A has a lower load sensitivity than system B. We might also use the term degradation to say that system B degrades more than system A.

Efficiency is performance divided by resources. A system that gets 30 tps on two CPUs is more efficient than a system that gets 40 tps on four identical CPUs.

The capacity of a system is an indication of maximum effective throughput or load. This might be an absolute maximum or a point at which the performance dips below an acceptable threshold.

Scalability is a measure of how adding resources (usually hardware) affects performance. A scalable system is one that allows you to add hardware and get a commensurate performance improvement, such as doubling how many servers you have to double your throughput. Vertical scalability, or scaling up, means adding more power to a single server, such as more memory. Horizontal scalability, or scaling out, means adding more servers.

The problem here is that design decisions don't affect all of these performance factors equally. Say we have two software systems running on a server: Swordfish's capacity is 20 tps while Camel's capacity is 40 tps. Which has better performance? Which is more scalable? We can't answer the scalability question from this data, and we can only say that Camel is more efficient on a single server. If we add another server, we notice that swordfish now handles 35 tps and camel handles 50 tps. Camel's capacity is still better, but Swordfish looks like it may scale out better. If we continue adding servers we'll discover that Swordfish gets 15 tps per extra server and Camel gets 10. Given this data we can say that Swordfish has better horizontal scalability, even though Camel is more efficient for less than five servers.

When building enterprise systems, it often makes sense to build for hardware scalability rather than capacity or even efficiency. Scalability gives you the option of better performance if you need it. Scalability can also be easier to do. Often designers do complicated things that improve the capacity on a particular hardware platform when it might actually be cheaper to buy more hardware. If Camel has a greater cost than Swordfish, and that greater cost is equivalent to a couple of servers, then Swordfish ends up being cheaper even if you only need 40 tps. It's fashionable to complain about having to rely on better hardware to make our software run properly, and I join this choir whenever I have to upgrade my laptop just to handle the latest version of Word. But newer hardware is often cheaper than making software run on less powerful systems. Similarly, adding more servers is often cheaper than adding more programmers—providing that a system is scalable.

Maven Repository

- Holds build arteficts
- **Remote**
  - Accessed byt http://, file://, ftp:// or other
  - Provided by a third party
  - Provided by the company to distribute private artifacts/dependencies
- **Local**
  - Cache of dependencies and build artifacts used or produced by your project
  - By default <HOME>/.m2/repository

Chapter 1. Layering

Layering is one of the most common techniques that software designers use to break apart a complicated software system. You see it in machine architectures, where layers descend from a programming language with operating system calls into device drivers and CPU instruction sets, and into logic gates inside chips. Networking has FTP layered on top of TCP, which is on top of IP, which is on top of ethernet.

When thinking of a system in terms of layers, you imagine the principal subsystems in the software arranged in some form of layer cake, where each layer rests on a lower layer. In this scheme the higher layer uses various services defined by the lower layer, but the lower layer is unaware of the higher layer. Furthermore, each layer usually hides its lower layers from the layers above, so layer 4 uses the services of layer 3, which uses the services of layer 2, but layer 4 is unaware of layer 2. (Not all layering architectures are opaque like this, but most are—or rather most are mostly opaque.

Breaking down a system into layers has a number of important benefits.

You can understand a single layer as a coherent whole without knowing much about the other layers. You can understand how to build an FTP service on top of TCP without knowing the details of how ethernet works.

You can substitute layers with alternative implementations of the same basic services. An FTP service can run without change over ethernet, PPP, or whatever a cable company uses.

You minimize dependencies between layers. If the cable company changes its physical transmission system, providing they make IP work, we don't have to alter our FTP service.

Layers make good places for standardization. TCP and IP are standards because they define how their layers should operate.

Once you have a layer built, you can use it for many higher-level services. Thus, TCP/IP is used by FTP, telnet, SSH, and HTTP. Otherwise, all of these higher-level protocols would have to write their own lower-level protocols.

Layering is an important technique, but there are downsides.

Layers encapsulate some, but not all, things well. As a result you sometimes get cascading changes. The classic example of this in a layered enterprise application is adding a field that needs to display on the UI, must be in the database, and thus must be added to every layer in between.

Extra layers can harm performance. At every layer things typically need to be transformed from one representation to another. However, the encapsulation of an underlying function often gives you efficiency gains that more than compensate. A layer that controls transactions can be optimized and will then make everything faster.

But the hardest part of a layered architecture is deciding what layers to have and what the responsibility of each layer should be.

VŠB TECHNICKÁ UNIVERZITA OSTRAVA | FAKULTA ELEKTROTECHNIKY A INFORMATIKY | KATEDRA INFORMATIKY

Remote repository

- Can be used other then default

```
<repositories>
  <repository>
    <id>vsb-education</id>
      <url>https://
       artifactory.cs.vsb.cz
       /repository/education-releases/
      </url>
  </repository>
  <repository>
    <id>sci-java-public</id>
    <url>https://
     maven.scijava.org/
     content/groups/public/
    </url>
  </repository>
</repositories>
```

- Publish builded artifact to own repository: **mvn deploy**

```
<distributionManagement>
  <snapshotRepository>
    <id>vsb-archetypes-snapshots</id>
    <url>https://
     artifactory.cs.vsb.cz/
     repository/archetype-snapshots/</url>
  </snapshotRepository>
  <repository>
    <id>vsb-archetypes-releases</id>
    <url>https://
     artifactory.cs.vsb.cz/
     repository/archetype-releases/</url>
   </repository>
</distributionManagement>
```


When people discuss layering, there's often some confusion over the terms layer and tier. Often the two are used as synonyms, but most people see tier as implying a physical separation. Client–server systems are often described as two-tier systems, and the separation is physical: The client is a desktop and the server is a server. I use layer to stress that you don't have to run the layers on different machines. A distinct layer of domain logic often runs on either a desktop or the database server. In this situation you have two nodes but three distinct layers. With a local database I can run all three layers on a single laptop, but there will still be three distinct layers.

Dependencies

- In pom.xml
- referenced with coordinates of the artifact
- cached in the local repository

```
<dependencies>
  <dependency>
    <groupId>org.openjfx</groupId>
    <artifactId>javafx-controls</artifactId>
    <version>${JavaFX.version}</version>
  </dependency>
  <dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter</artifactId>
    <scope>test</scope>
  </dependency>
</dependencies>
```

Dependency scope

- Compile,
- Provided,
- Runtime,
- Test,
- System,

```
<dependency>
  <groupId>org.junit.jupiter</groupId>
  <artifactId>junit-jupiter</artifactId>
  <scope>test</scope>
</dependency>
```


The Java EE platform is built on top of the Java SE platform. The Java EE platform provides:

- Huge API with support for internet communication, HTTP connectivity, Web development, data persistency, …
- Runtime environment often called JavaEE container that provide server side runtime environment often included as part of web server.

We discuss this platform later in that course.

Compile dependency

- Default for dependencies not specifying scope
- Available to all classpaths of build lifecycle (compile, test-compile, test, package)
- Packaged into final artifact

As soon as the web began to be used for delivering services, service providers recognized the need for dynamic content. Applets, one of the earliest attempts toward this goal, focused on using the client platform to deliver dynamic user experiences. At the same time, developers also investigated using the server platform for this purpose. Initially, Common Gateway Interface (CGI) scripts were the main technology used to generate dynamic content. Although widely used, CGI scripting technology has a number of shortcomings, including platform dependence and lack of scalability. To address these limitations, Java Servlet technology was created as a portable way to provide dynamic, user-oriented content.

What Is a Servlet?
A *servlet is a Java programming language class that is used to extend the capabilities of servers* that host applications accessed by means of a request-response programming model. Although servlets can respond to any type of request, they are commonly used to extend the applications hosted by web servers. For such applications, Java Servlet technology defines HTTP-specific servlet classes.
The javax.servlet and javax.servlet.http packages provide interfaces and classes for writing servlets. All servlets must implement the Servlet interface, which defines life-cycle methods.
When implementing a generic service, you can use or extend the GenericServlet class provided with the Java Servlet API. TheHttpServlet class provides methods, such as doGet and doPost, for handling HTTP-specific services.
This chapter focuses on writing servlets that generate responses to HTTP requests.

Provided dependency

- Available to compile ant test classpaths
- Not packaged as part of your artifact
- It's expected the container, where the artifact will be used, to provide the dependency

As soon as the web began to be used for delivering services, service providers recognized the need for dynamic content. Applets, one of the earliest attempts toward this goal, focused on using the client platform to deliver dynamic user experiences. At the same time, developers also investigated using the server platform for this purpose. Initially, Common Gateway Interface (CGI) scripts were the main technology used to generate dynamic content. Although widely used, CGI scripting technology has a number of shortcomings, including platform dependence and lack of scalability. To address these limitations, Java Servlet technology was created as a portable way to provide dynamic, user-oriented content.

What Is a Servlet?
A *servlet is a Java programming language class that is used to extend the capabilities of servers* that host applications accessed by means of a request-response programming model. Although servlets can respond to any type of request, they are commonly used to extend the applications hosted by web servers. For such applications, Java Servlet technology defines HTTP-specific servlet classes.
The javax.servlet and javax.servlet.http packages provide interfaces and classes for writing servlets. All servlets must implement the Servlet interface, which defines life-cycle methods.
When implementing a generic service, you can use or extend the GenericServlet class provided with the Java Servlet API. TheHttpServlet class provides methods, such as doGet and doPost, for handling HTTP-specific services.
This chapter focuses on writing servlets that generate responses to HTTP requests.

Runtime dependency

- This dependency is not required for compiling your project
- Is required at runtime, when your application run
- Iso required when testing because tests will execute main code

As soon as the web began to be used for delivering services, service providers recognized the need for dynamic content. Applets, one of the earliest attempts toward this goal, focused on using the client platform to deliver dynamic user experiences. At the same time, developers also investigated using the server platform for this purpose. Initially, Common Gateway Interface (CGI) scripts were the main technology used to generate dynamic content. Although widely used, CGI scripting technology has a number of shortcomings, including platform dependence and lack of scalability. To address these limitations, Java Servlet technology was created as a portable way to provide dynamic, user-oriented content.

What Is a Servlet?
A *servlet is a Java programming language class that is used to extend the capabilities of servers* that host applications accessed by means of a request-response programming model. Although servlets can respond to any type of request, they are commonly used to extend the applications hosted by web servers. For such applications, Java Servlet technology defines HTTP-specific servlet classes.
The javax.servlet and javax.servlet.http packages provide interfaces and classes for writing servlets. All servlets must implement the Servlet interface, which defines life-cycle methods.
When implementing a generic service, you can use or extend the GenericServlet class provided with the Java Servlet API. TheHttpServlet class provides methods, such as doGet and doPost, for handling HTTP-specific services.
This chapter focuses on writing servlets that generate responses to HTTP requests.

Test dependency

- This dependency is only required to compile and run test
- Will not be packaged into final assembly (jar, war, ear, etc)

As soon as the web began to be used for delivering services, service providers recognized the need for dynamic content. Applets, one of the earliest attempts toward this goal, focused on using the client platform to deliver dynamic user experiences. At the same time, developers also investigated using the server platform for this purpose. Initially, Common Gateway Interface (CGI) scripts were the main technology used to generate dynamic content. Although widely used, CGI scripting technology has a number of shortcomings, including platform dependence and lack of scalability. To address these limitations, Java Servlet technology was created as a portable way to provide dynamic, user-oriented content.

What Is a Servlet?
A *servlet is a Java programming language class that is used to extend the capabilities of servers* that host applications accessed by means of a request-response programming model. Although servlets can respond to any type of request, they are commonly used to extend the applications hosted by web servers. For such applications, Java Servlet technology defines HTTP-specific servlet classes.
The javax.servlet and javax.servlet.http packages provide interfaces and classes for writing servlets. All servlets must implement the Servlet interface, which defines life-cycle methods.
When implementing a generic service, you can use or extend the GenericServlet class provided with the Java Servlet API. TheHttpServlet class provides methods, such as doGet and doPost, for handling HTTP-specific services.
This chapter focuses on writing servlets that generate responses to HTTP requests.

System dependency

- Similar to Provided Dependency
- Not looked up in repository
- Expected to exists in your development machine

As soon as the web began to be used for delivering services, service providers recognized the need for dynamic content. Applets, one of the earliest attempts toward this goal, focused on using the client platform to deliver dynamic user experiences. At the same time, developers also investigated using the server platform for this purpose. Initially, Common Gateway Interface (CGI) scripts were the main technology used to generate dynamic content. Although widely used, CGI scripting technology has a number of shortcomings, including platform dependence and lack of scalability. To address these limitations, Java Servlet technology was created as a portable way to provide dynamic, user-oriented content.

What Is a Servlet?
A *servlet is a Java programming language class that is used to extend the capabilities of servers* that host applications accessed by means of a request-response programming model. Although servlets can respond to any type of request, they are commonly used to extend the applications hosted by web servers. For such applications, Java Servlet technology defines HTTP-specific servlet classes.
The javax.servlet and javax.servlet.http packages provide interfaces and classes for writing servlets. All servlets must implement the Servlet interface, which defines life-cycle methods.
When implementing a generic service, you can use or extend the GenericServlet class provided with the Java Servlet API. TheHttpServlet class provides methods, such as doGet and doPost, for handling HTTP-specific services.
This chapter focuses on writing servlets that generate responses to HTTP requests.

Exceluded dependency

- Dependency are transitional in default
- Can by excluded

```
<dependency>
  <groupId>org.apache.commons</groupId>
  <artifactId>commons-text</artifactId>
  <version>1.1</version>
  <exclusions>
    <exclusion>
       <groupId>org.apache.commons</groupId>
       <artifactId>commons-lang3</artifactId>
    </exclusion>
  </exclusions>
</dependency>
```

Optional dependencies

- Are automatically excluded

```
<dependency>
  <groupId>org.junit.jupiter</groupId>
  <artifactId>junit-jupiter</artifactId>
  <version>5.10.1</version>
  <scope>test</scope>
  <optional>true</optional>
</dependency>
```

Packaging

- Determines output
  - jar,war, ear

```
<project
xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/
XMLSchema-instance"
xsi:schemaLocation="http://
maven.apache.org/POM/4.0.0
http://maven.apache.org/maven-
v4_0_0.xsd">
<modelVersion>4.0.0</modelVersion>
<groupId>cz.vsb.fei.java2</groupId>
<artifactId>lab01</artifactId>
<version>0.0.1-SNAPSHOT</version>
<name>lab01</name>
<packaging>jar</packaging>
```


The Structure of the Patterns

Every author has to choose his pattern form. Some base their forms on a classic patterns book such as [Alexander et al.], [Gang of Four], or [POSA]. Others make up their own. I've long wrestled with what makes the best form. On the one hand I don't want something as small as the GOF form; on the other hand I need to have sections that support a reference book. So this is what I've used for this book.

The first item is the name of the pattern. Pattern names are crucial, because part of the purpose of patterns is to create a vocabulary that allows designers to communicate more effectively. Thus, if I tell you my Web server is built around a Front Controller (344) and a Transform View (361) and you know these patterns, you have a very clear idea of my web server's architecture.

Next are two items that go together: the intent and the sketch. The intent sums up the pattern in a sentence or two; the sketch is a visual representation of the pattern, often but not always a UML diagram. The idea is to create a brief reminder of what the pattern is about so you can quickly recall it. If you already "have the pattern," meaning that you know the solution even if you don't know the name, then the intent and the sketch should be all you need to know what the pattern is.

The next section describes a motivating problem for the pattern. This may not be the only problem that the pattern solves, but it's one that I think best motivates the pattern.

How It Works describes the solution. In here I put a discussion of implementation issues and variations that I've come across. The discussion is as independent as possible of any particular platform—where there are platform-specific sections I've indented them so you can see them and easily skip over them. Where useful I've put in UML diagrams to help explain them.

When to Use It describes when the pattern should be used. Here I talk about the trade-offs that make you select this solution compared to others. Many of the patterns in this book are alternatives; such Page Controller (333) and Front Controller (344). Few patterns are always the right choice, so whenever I find a pattern I always ask myself, "When would I not use this?" That question often leads me to alternative patterns.

The Further Reading section points you to other discussions of this pattern. This isn't a comprehensive bibliography. I've limited my references to pieces that I think are important in helping you understand the pattern, so I've eliminated any discussion that I don't think adds much to what I've written and of course I've eliminated discussions of patterns I haven't read. I also haven't mentioned items that I think are going to be hard to find, or unstable Web links that I fear may disappear by the time you read this book.

I like to add one or more examples. Each one is a simple example of the pattern in use, illustrated with some code in Java or C#. I chose those languages because they seem to be languages that the largest number of professional programmers can read. It's absolutely essential to understand that the example is not the pattern. When you use the pattern, it won't look exactly like this example so don't treat it as some kind of glorified macro. I've deliberately kept the example as simple as possible so you can see the pattern in as clear a form as I can imagine. All sorts of issues are ignored that will become important when you use it, but these will be particular to your own environment. This is why you always have to tweak the pattern.

One of the consequences of this is that I've worked hard to keep each example as simple as I can, while still illustrating its core message. Thus, I've often chosen an example that's simple and explicit, rather than one that demonstrates how a pattern works with the many wrinkles required in a production system. It's a tricky balance between simple and simplistic, but it's also true that too many realistic yet peripheral issues can make it harder to understand the key points of a pattern.

This is also why I've gone for simple independent examples instead of a connected running examples. Independent examples are easier to understand in isolation, but give less guidance on how you put them together. A connected example shows how things fit together, but it's hard to understand any one pattern without understanding all the others involved in the example. While in theory it's possible to produce examples that are connected yet understandable independently, doing so is very hard—or at least too hard for me—so I chose the independent route.

The code in the examples is written with a focus on making the ideas understandable. As a result several things fall aside—in particular, error handling, which I don't pay much attention to since I haven't developed any patterns in this area yet. They are there purely to illustrate the pattern. They are not intended to show how to model any particular business problem.

For these reasons the code isn't downloadable from my Web site. Each code example in this book is surrounded with too much scaffolding to simplify the basic ideas so they're worth anything in a production setting.

Not all the sections appear in all the patterns. If I couldn't think of a good example or motivation text, I left it out.

Modules - goals

- Reliable configuration
- Strong encapsulation
- Scalable Java platform
- Greater platform integrity
- Improved performance

http://martinfowler.com/ieeeSoftware/patterns.pdf

Modules introduction

- List modules
  java –list-modules
- Module declaration
- Module declaration
- In root of source folder file:
  module-info.java

```
module cz.vsb.fei.java2.jez04lab01 {
    requires transitive javafx.controls;
    requires javafx.fxml;
    requires cz.vsb.fei.java2.lab01text2asciiart;
    requires java.logging;
    opens cz.vsb.fei.java2.jez04lab01 to javafx.fxml;
    exports cz.vsb.fei.java2.jez04lab01;
}
```


Context

The presentation-tier request handling mechanism receives many different types of requests, which require varied types of processing. Some requests are simply forwarded to the appropriate handler component, while other requests must be modified, audited, or uncompressed before being further processed.

Problem

Preprocessing and post-processing of a client Web request and response are required.

When a request enters a Web application, it often must pass several entrance tests prior to the main processing stage. For example,

Has the client been authenticated?
Does the client have a valid session?
Is the client's IP address from a trusted network?
Does the request path violate any constraints?
What encoding does the client use to send the data?
Do we support the browser type of the client?

Some of these checks are tests, resulting in a yes or no answer that determines whether processing will continue. Other checks manipulate the incoming data stream into a form suitable for processing.

The classic solution consists of a series of conditional checks, with any failed check aborting the request. Nested if/else statements are a standard strategy, but this solution leads to code fragility and a copy-and-paste style of programming, because the flow of the filtering and the action of the filters is compiled into the application.

The key to solving this problem in a flexible and unobtrusive manner is to have a simple mechanism for adding and removing processing components, in which each component completes a specific filtering action.

Forces

Common processing, such as checking the data-encoding scheme or logging information about each request, completes per request.
Centralization of common logic is desired.
Services should be easy to add or remove unobtrusively without affecting existing components, so

Running java with modules

java --module-path <directory> --module <module>/<class with main>

java -p <directory> -m <module>/<class with main>

java -p <directory> <module>

Module-info: requires

requires [transitive] [static] <module-name>

Module-info: Export packages by modules

exports <package-name>

exports <package-name> to <module-name>

Module-info: Support for services

uses <interface-name>

provides <interface-name> with <class-name>

...

ServiceLoader.load()

Module-info: Allow runtime access

opens <package-name>

opens <package-name> to <module-name>

open module <module-name> {

}

Packaging as a standalone JRE

jlink –module-path <dirs> --add-module <module-name> --output <out-dir>

Java --module <module-name>/<main-class>

Backward compatibility

Unnamed module

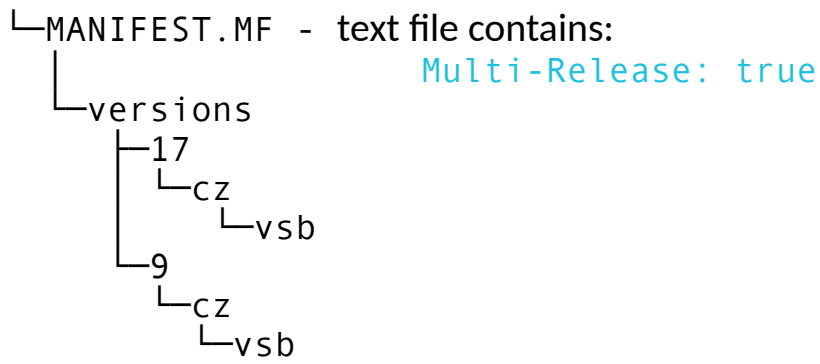
- When a class or JAR is loaded onto the classpath, but not the module path, it’s automatically added to the unnamed module. It’s a catch-all module to maintain backward compatibility with previously-written Java code.

Automatic modules

- We can include unofficial modules by adding existing JAR files to the module path. The name of the module will be derived from the name of the JAR. Automatic modules will have full read access to every other module loaded by the path.

Allow support for multiple java version

META-INF – folder in project/jar file



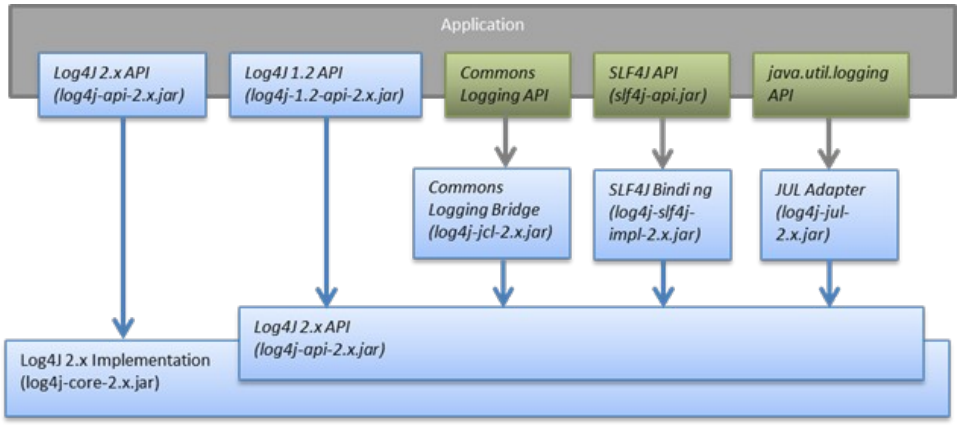
2nd lecture

- Logging
- Assertions
- Profiling
- Effective Java:
  - Static factory methods
  - Builders
  - Correct implementation of equals

Logging

- Write runtime info with System.out.println is inappropriate in a production environment
- Logging framework is used instead of it (Java Util Logging, log4j, logback,log4j2,slf4j)
-

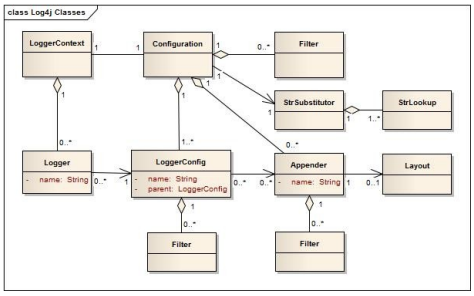
Log4j2 - State of The Art logging framework

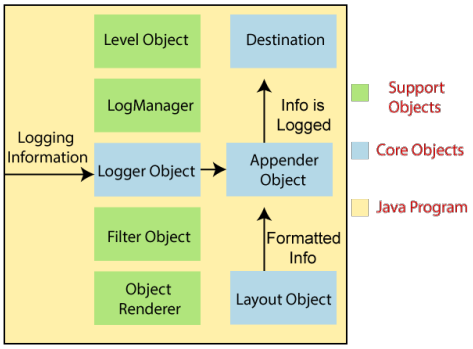


Maven dependecies

```
<dependency>
  <groupId>org.apache.logging.log4j</groupId>
  <artifactId>log4j-core</artifactId>
  <version>2.22.1</version>
</dependency>
<dependency>
  <groupId>org.apache.logging.log4j</groupId>
  <artifactId>log4j-api</artifactId>
  <version>2.22.1</version>
</dependency>
```

Log4j architecture





Logger

```
static Logger log = LogManager.getLogger(App.class);


public static void main(String[] args) {
    log.info("Hello {}", () -> "world");
    log.log(Level.TRACE,"Hello {}",  "world");

    try {
        Files.copy(Paths.get("source"), Paths.get("path"));
    } catch (IOException e) {
        log.fatal("copying", e);
    }

}
```

Trace
Debug
Info
Warn
Error
Fatal

Configuration

- log4j.xml(properties,yaml, json) – put to the classpath or set property **–Dlog4j.configurationFile=<location>**

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Configuration status="WARN">
    <Appenders>
        <ThresholdFilter level="DEBUG"/>
        <Console name="Console" target="SYSTEM_OUT">
            <PatternLayout
                 pattern="%d{HH:mm:ss.SSS} [%t] %-5level %logger{36} - %msg%n" />
        </Console>
    </Appenders>
    <Appenders>
    <RollingFile name="RollingFile" fileName="logs/app.log"
                 filePattern="logs/app-%d{MM-dd-yyyy}.log.gz">
      <PatternLayout>
        <pattern>%d %p %c{1.} [%t] %m%n</pattern>
      </PatternLayout>
      <TimeBasedTriggeringPolicy />
    </RollingFile>
  </Appenders>
    <Loggers>
        <Logger name="javaII.lab01.App" level="trace">
            <AppenderRef ref="Console" />
            <AppenderRef ref="RollingFile" />
        </Logger>
        <Root level="error">
            <AppenderRef ref="Console" />
        </Root>
    </Loggers>
</Configuration>
```

Marker

- Can use filter “MarkerFilter” for appender

```
Marker marker = MarkerManager.getMarker("tp-count");

log.info(marker, "sensitive info");
```

MDC (Mapped Diagnostic Context)

```
MDC.put("transaction.id", tx.getTransactionId());
MDC.put("transaction.owner", tx.getSender());
log4jBusinessService.transfer(tx.getAmount());
MDC.clear();
```

```
log4j.appender.consoleAppender.layout.ConversionPattern=
  %-4r [%t] %5p %c{1} %x - %m - tx.id=%X{transaction.id} tx.owner=%X{transaction.owner}%n
```

```
638  [pool-1-thread-2]  INFO Log4JBusinessService
  - Has transfer of 1104$ completed successfully ? true. - tx.id=2 tx.owner=Marc
638  [pool-1-thread-2]  INFO Log4JBusinessService
  - Preparing to transfer 1685$. - tx.id=4 tx.owner=John
```

Assertions

- They provide mechanism for internal consistency checks.
  - E.g. constraints among values of attributes is ensured.
- They could be removed in production version.
  - E.g. they are ignored during runtime.
- Java provides support with assert keyword.
-

Java Assertion Statement

- Two forms are used:
  - **assert boolean-expression**
  - **assert boolean-expression: description;**
  - **assert conn != null : "Connection is null";**
- The “boolean-expression” expresses something that should be true during its execution.
- An AssertionError is thrown if the assertion is false – it contains “description”

Assert example

```
public void removeRecord(String key)
{
    if(key == null){
        throw new IllegalArgumentException("...");
    }
    if(keyInUse(key)) {
        Record details = book.get(key);
        details.freeData();
        details.removeFromIndex();
        numberOfEntries--;
    }
    assert !keyInUse(key);
    assert isConsistentIndex() :
           "Inconsistent index in removeRecord";
}
```

Guidelines for Assertions

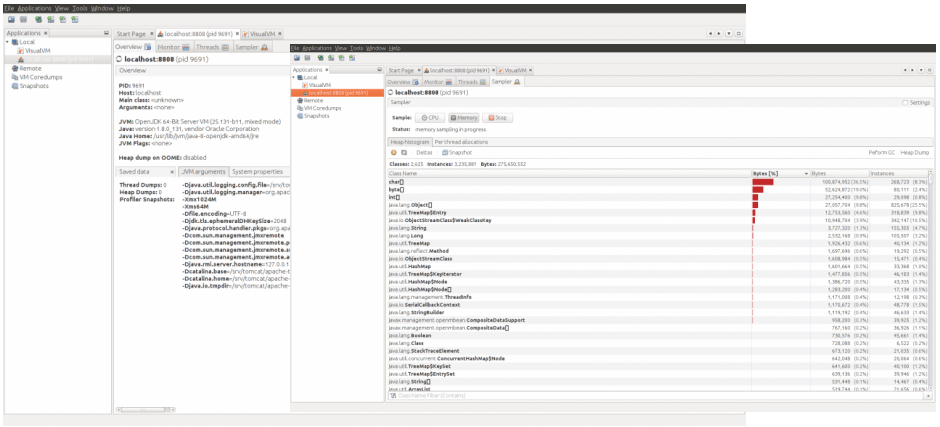
- Use it for internal consistency check.
- (Not?) Remove from production code.
- Don’t include normal functionality:

```
// Incorrect use:
assert book.remove(name) != null;
```

- They has to have no side effect.
- Do not use it for exception throwing – it is not an alternative
-

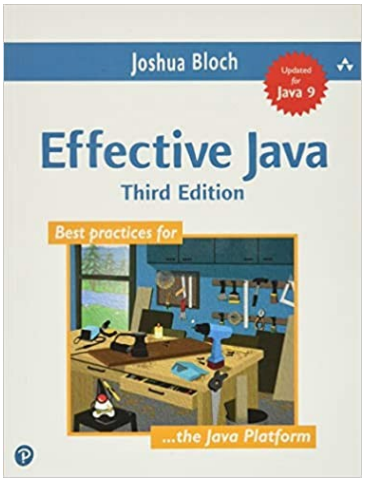
Profiling – Java VisualVM

- https://visualvm.github.io/download.html



Effective Java

- BLOCH, Joshua. Effective Java. 3rd edition. Boston: Addison-Wesley Professional, 2017. ISBN 978-0-13-468599-1.



Consider static factory methods instead of constructors

```
public static Boolean valueOf(boolean b) {
    return b ? Boolean.TRUE : Boolean.FALSE;
}
```

Consider static factory methods instead of constructors - advantages

- unlike constructors, they have names
- unlike constructors, they are not required to create a new object each time they’re invoked
- unlike constructors, they can return an object of any subtype of their return type.
- class of the returned object can vary from call to call as a function of the input parameters
- the class of the returned object need not exist when the class containing the method is written.

Consider static factory methods instead of constructors - limitation

- classes without public or protected constructors cannot be subclassed;
- they are hard for programmers to find

Static factory methods name convention

- from
- of
- valueOf
- instance or getInstance
- create or newInstance
- getType
- newType
- type

```
LocalDate d =
LocalDate.from(LocalDateTime.now());
Set<Rank> faceCards =
EnumSet.of(Rank.JACK, Rank.QUEEN,
Rank.KING);
BigInteger prime =
BigInteger.valueOf(Integer.MAX_VALUE);
StackWalker luke =
StackWalker.getInstance(Option.SHOW_HIDDEN
_FRAMES);
Object newArray =
Array.newInstance(String.class, 10);
FileStore fs =
Files.getFileStore(Paths.get("/home"));
BufferedReader br =
Files.newBufferedReader(Paths.get("/tmp/te
st.txt"));
```

Consider a builder when faced with many constructor parameters -telescoping constructors

- provide a constructor with only the required parameters

```
public NutritionFacts(int servingSize, int servings) {
      this(servingSize, servings, 0);
   }

   public NutritionFacts(int servingSize, int servings,
         int calories) {
      this(servingSize, servings, calories, 0);
   }
NutritionFacts cocaCola = new NutritionFacts(240, 8, 100, 0, 35, 27);

NutritionFacts cocaCola = new NutritionFacts(240, 8, 100, 0, 35, 27);
```

Consider a builder when faced with many constructor parameters – JavaBeans pattern

- JavaBean may be in an inconsistent state partway through its construction.
- the JavaBeans pattern precludes the possibility of making a class immutable

```
NutritionFacts cocaCola = new NutritionFacts();
cocaCola.setServingSize(240);
cocaCola.setServings(8);
cocaCola.setCalories(100);
cocaCola.setSodium(35);
cocaCola.setCarbohydrate(27);
```

Consider a builder when faced with many constructor parameters

- the client calls a constructor (or static factory) with all of the required parameters and gets a builder object.

```
NutritionFacts cocaCola = new
NutritionFacts.Builder(240, 8)
  .calories(100).sodium(35)
  .carbohydrate(27).build();
```

1.1 Component granularity

There are a range of different kinds of JavaBeans components:

1. Some JavaBean components will be used as building blocks in composing applications. So a user may be using some kind of builder tool to connect together and customize a set of JavaBean components s to act as an application. Thus for example, an AWT button would be a Bean.
2. Some JavaBean components will be more like regular applications, which may then be composed together into compound documents. So a spreadsheet Bean might be embedded inside a Web page.

Portability

One of the main goals of the JavaBeans architecture is to provide a platform neutral component architecture. When a Bean is nested inside another Bean then we will provide a full functionality implementation on all platforms. However, at the top level when the root Bean is embedded in some platform specific container (such as Word or Visual Basic or ClarisWorks or Netscape Navigator) then the JavaBeans APIs should be integrated into the platform's local component architecture.

Beans v. Class Libraries

Not all useful software modules should necessarily turn into beans. Beans are appropriate for software components that can be visually manipulated and customized to achieve some effect. Class libraries are an appropriate way of providing functionality that is useful to programmers, but which doesn't benefit from visual manipulation. So for example it makes sense to provide the JDBC database access API as a class library rather than as a bean, because JDBC is essentially a programmatic API and not something that can be directly presented for visual manipulation. However it does make sense to write database access beans on top of JDBC. So for example you might write a "select" bean that at customization time helped a user to compose a select statement, and then when the application is run uses JDBC to run the select statement and display the results

Design time v. run-time

Each Java Bean component has to be capable of running in a range of different environments. There are really a continuum of different possibilities, but two points are particularly worth noting. First a bean must be capable of running inside a builder tool. This is often referred to as the *design environment. Within this design environment it is very important that the bean should* provide design information to the application builder and allow the end-user to *customize the* appearance and behaviour of the bean. Second, each bean must be usable at run-time within the generated application. In this environment there is much less need for design information or customization. The design time information and the design time customization code for a component may potentially be quite large. For example, if a component writer provides a "wizard" style customizer that guides a user through a series of choices, tthe run-time code for the bean. We therefore wanted to make sure that we have a clear split between the design-time aspects of a bean and the run-time aspects, so that it should be possible to deploy a bean at run-time without needing to download all its design time code. So, for example, we allow the design time interfaces (described in chapters 8 and 9) to be supported in a separate class from the run-time interfaces (described in the other chapters).hen the customization code may easily dwarf

Security Issues

Java Beans are subject to the standard Java security model. We have neither extended nor relaxed the standard Java security model for Java Beans. Specifically, when a Java Bean runs as part of an untrusted applet then it will be subject to the standard applet security restrictions and won't be allowed to read or write arbitrary files, or to connect to arbitrary network hosts. However when a Java Bean runs as part of a stand-alone Java application, or as part of a trusted (signed) applet, then it will be treated as a normal Java application and allowed normal access to files and network hosts. In general we advise Java Bean developers to design their beans so that they can be run as part of untrusted applets. The main areas where this shows up in the beans APIs are:

• *Introspection. Bean developers should assume that they have unlimited access to the* high level Introspection APIs (Section 8) and the low-level reflection APIs in the design-time environment, but more limited access in the run-time environment. For

example, the standard JDK security manager will allow trusted applications access to even private field and methods, but will allow untrusted applets access to only public fields and methods. (This shouldn't be too constraining - the high-level Introspection

APIs only expose "public" information anyway.) • *Persistence. Beans should expect to be serialized or deserialized (See Section 5) in both*

the design-time and the run-time environments. However in the run-time environment, the bean should expect the serialization stream to be created and controlled by their parent application and should not assume that they can control where serialized data is read from or written to. Thus a browser might use serialization to read in the initial state for an untrusted applet, but the applet should not assume that it can access random files.

• *GUI Merging. In general untrusted applets will not be permitted to perform any kind of* GUI merging with their parent application. So for example, menubar merging might occur between nested beans inside an untrusted applet, but the top level menubar for the untrusted applet will be kept separate from the browser's menubar.

None of these restrictions apply to beans running as parts of full-fledged Java applications, where the beans will have full unrestricted access to the entire Java platform API.

What should be saved

When a bean is made persistent it should store away appropriate parts of its internal state so that it can be resurrected later with a similar appearance and similar behaviour. Normally a bean will store away persistent state for all its exposed properties. It may also store away additional internal state that is not directly accessible via properties. This might include (for example) additional design choices that were made while running a bean Customizer (see Section 5) or internal state that was created by the bean developer.

A bean may contain other beans, in which case it should store away these beans as part of its internal state.

However a bean should not normally store away pointers to external beans (either peers or a parent container) but should rather expect these connections to be rebuilt by higher-level software. So normally it should use the "transient" keyword to mark pointers to other beans or to event listeners. In general it is a container's responsibility to keep track of any inter-bean wiring it creates and to store and resurrect it as needed.

For the same reasons, normally event adaptors should mark their internal fields as "transient".

Obey the general contract when overriding equals

- No need to override:
  - Each instance of the class is inherently unique.
  - There is no need for the class to provide a “logical equality” test.
  - A superclass has already overridden equals, and the superclass behavior is appropriate for this class.
  - The class is private or package-private, and you are certain that its equals method will never be invoked.

Obey the general contract when overriding equals

- Signature … `public boolean equals(Object other)`
- Reflexive
- Symmetric
- Transitive
- Consistent
- For any non-null reference value x, x.equals(null) must return false

Obey the general contract when overriding equals – violated symmetry

```
public class Point {
  private final int x;
  private final int y;

  public Point(int x, int y) {
    this.x = x;
    this.y = y;
  }

  @Override
  public boolean equals(Object o) {
    if (!(o instanceof Point))
      return false;
    Point p = (Point) o;
    return p.x == x && p.y == y;
  }

// ... // Remainder omitted
}
```

```
public class ColorPoint extends Point {
  private final Color color;

  public ColorPoint(int x, int y, Color
color) {
    super(x, y);
    this.color = color;
  }
// ... // Remainder omitted

//Broken - violates symmetry!
  @Override
  public boolean equals(Object o) {
    if (!(o instanceof ColorPoint))
      return false;
    return super.equals(o) && ((ColorPoint)
o).color == color;
  }
}
Point p = new Point(1, 2);
ColorPoint cp = new ColorPoint(1, 2,
Color.RED);
```

Obey the general contract when overriding equals – violated transitivity

```
// Broken - violates transitivity!
public boolean equals(Object o) {
  if (!(o instanceof Point))
    return false;

// If o is a normal Point, do a color-blind comparison
  if (!(o instanceof ColorPoint))
    return o.equals(this);
// o is a ColorPoint; do a full comparison
  return super.equals(o) && ((ColorPoint) o).color == color;
}

ColorPoint p1 = new ColorPoint(1, 2, Color.RED);
Point p2 = new Point(1, 2);
ColorPoint p3 = new ColorPoint(1, 2, Color.BLUE);
```

The Liskov Substitution Principle in practical software development

- The principle defines that objects of a superclass shall be replaceable with objects of its subclasses without breaking the application. That requires the objects of your subclasses to behave in the same way as the objects of your superclass. You can achieve that by following a few rules, which are pretty similar to the design by contract concept defined by Bertrand Meyer.
- An overridden method of a subclass needs to accept the same input parameter values as the method of the superclass. That means you can implement less restrictive validation rules, but you are not allowed to enforce stricter ones in your subclass. Otherwise, any code that calls this method on an object of the superclass might cause an exception, if it gets called with an object of the subclass.
- Similar rules apply to the return value of the method. The return value of a method of the subclass needs to comply with the same rules as the return value of the method of the superclass. You can only decide to apply even stricter rules by returning a specific subclass of the defined return value, or by returning a subset of the valid return values of the superclass.

Obey the general contract when overriding equals – violated Liskov subst. principle

```
// Broken - violates Liskov
substitution principle (page 43)
@Override
public boolean equals(Object o) {
  if (o == null || o.getClass() !=
      getClass())
    return false;
  Point p = (Point) o;
  return p.x == x && p.y == y;
}
```

```
public class PointWithDesc extends Point
{
  public PointWithDesc(int x, int y) {
    super(x, y);
  }

  public String getDescription() {
    return String.format("[%d, %d]",
        getX(), getY());
  }
}
```

Obey the general contract when overriding equals

```java
public class Point {
  private final int x;
  private final int y;

  public Point(int x, int y) {
    this.x = x;
    this.y = y;
  }

  @Override
  public boolean equals(Object o) {
    if (o instanceof Point p)
      return p.canEqual(this) && …;
    return false;
  }

// ... // Remainder omitted
  public boolean canEqual(Point that) {
    return that instanceof Point;
  }
}
```

```java
public class ColorPoint extends Point {
  private final Point point;
  private final Color color;

  public ColorPoint(int x, int y, Color color) {
    super(x, y);
    this.color = Objects.requireNonNull(color);
  }

  @Override
  public boolean equals(Object o) {
    if (!(o instanceof ColorPoint))
      return false;
    ColorPoint cp = (ColorPoint) o;
    return super.equals(cp) &&
           cp.color.equals(color);
  }
// ... // Remainder omitted
  @Override
  public boolean canEqual(Point that) {
    return that instanceof ColorPoint;
  }
}
```

Always override hashCode when you override equals

1) int result = <c for first significant field computed by 2>
2) For every remaining significant field f in your object, do the following:
   a) Compute an int hash code c for the field:
      I. a primitive type => Type.hashCode(f)
      II. object reference => invoke hashCode on the field or compute a “canonical representationor or use 0 (or some other constant, but 0 is traditional).
      III. an array => a hash code for each significant element; use a constant, preferably not 0 – no significatn element; use Arrays.hashCode – all significant elements.
   b) result = 31 * result + c
3) Return result.

Always override hashCode when you override equals

- Do not be tempted to exclude significant fields from the hash code computation to improve performance.

3rd lecture

- Lombok
- Effective java II

Project Lombok

- Project Lombok is a java library that automatically plugs into your editor and build tools, spicing up your java.
- Never write another getter or equals method again, with one annotation your class has a fully featured builder, Automate your logging variables, and much more.
- **https://projectlombok.org**

```
<dependency>
  <groupId>org.projectlombok</groupId>
  <artifactId>lombok</artifactId>
  <version>1.18.30</version>
  <scope>provided</scope>
</dependency>
```

Lombok features

val – Finally! Hassle-free final local variables.

var – Mutably! Hassle-free local variables.

@NonNull – or: How I learned to stop worrying and love the NullPointerException.

@Cleanup – Automatic resource management: Call your close() methods safely with no hassle.

@Getter/@Setter – Never write public int getFoo() {return foo;} again.

@ToString – No need to start a debugger to see your fields: Just let lombok generate a toString for you!

@EqualsAndHashCode – Equality made easy: Generates hashCode and equals implementations from the fields of your object..

@NoArgsConstructor, @RequiredArgsConstructor and @AllArgsConstructor – Constructors made to order: Generates constructors that take no arguments, one argument per final / non-nullfield, or one argument for every field.

Lombok features

@Data – All together now: A shortcut for **@ToString**, **@EqualsAndHashCode**, **@Getter** on all fields, and **@Setter** on all non-final fields, and **@RequiredArgsConstructor**!

@Value – Immutable classes made very easy.

@Builder – … and Bob's your uncle: No-hassle fancy-pants APIs for object creation!

@SneakyThrows – To boldly throw checked exceptions where no one has thrown them before!

@Synchronized – synchronized done right: Don't expose your locks.

@With – Immutable 'setters' - methods that create a clone but with one changed field.

@Getter(lazy=true) – Laziness is a virtue!

@Log – Captain's Log, stardate 24435.7: "What was that line again?"

Maven compile with Lombok

```
<plugins>
  <plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-compiler-plugin</artifactId>
    <version>3.12.1</version>
    <configuration>
     <annotationProcessorPaths>
       <path>
         <groupId>org.projectlombok</groupId>
         <artifactId>lombok</artifactId>
         <version>1.18.30</version>
       </path>
     </annotationProcessorPaths>
   </configuration>
</plugin>
```

Delombok

- Normally, lombok adds support for all the lombok features directly to your IDE and compiler by plugging into them.
- However, lombok doesn't cover all tools. For example, lombok cannot plug into javadoc, nor can it plug into the Google Widget Toolkit, both of which run on java sources. Delombok still allows you to use lombok with these tools by preprocessing your java code into java code with all of lombok's transformations already applied.
- Delombok can of course also help understand what's happening with your source by letting you look at exactly what lombok is doing 'under the hood'.

```
java -jar lombok.jar delombok src -d src-delomboked
```

Minimize the accessibility of classes and members

- make each class or member as inaccessible as possible
- Instance fields of public classes should rarely be public
- classes with public mutable fields are not generally thread-safe
- it is wrong for a class to have a public static final array field, or an accessor that returns such a field

Minimize mutability

- Don’t provide methods that modify the object’s state (known as mutators).
- Ensure that the class can’t be extended
- Make all fields final.
- Make all fields private.
- Ensure exclusive access to any mutable components.

Record Classes

- Record classes, which are a special kind of class, help to model plain data aggregates with less ceremony than normal classes.
  - For background information about record classes, see JEP 395.

https://docs.oracle.com/en/java/javase/17/language/records.html

Record example

```
record Rectangle(double
length, double width) { }


Rectangle r = new
Rectangle(4,5);
```

“plain“ Java

```
public final class Rectangle {
    private final double length;
    private final double width;

    public Rectangle(double length, double width) {
        this.length = length;
        this.width = width;
    }
    double length() { return this.length; }
    double width()  { return this.width; }
    // Implementation of equals() and hashCode(),
which specify that two record objects are equal if
they are of the same type and contain equal field
values.
    public boolean equals...
    public int hashCode...
    // An implementation of toString() that returns
a string representation of all the record class's
fields, including their names.
    public String toString() {...}
}
```

Record class

A record class declaration consists of a name; optional type parameters (generic record declarations are supported); a header, which lists the "components" of the record; and a body.

A record class declares the following members automatically:

- For each component in the header, the following two members:
  - A private final field with the same name and declared type as the record component. This field is sometimes referred to as a component field.
  - A public accessor method with the same name and type of the component.
- A canonical constructor whose signature is the same as the header. This constructor assigns each argument from the new expression that instantiates the record class to the corresponding component field.
- Implementations of the equals and hashCode methods, which specify that two record classes are equal if they are of the same type and contain equal component values.
- An implementation of the toString method that includes the string representation of all the record class's components, with their names.

As record classes are just special kinds of classes, you create a record object (an instance of a record class) with the new keyword.

The Canonical Constructor of a Record Class

- The following example explicitly declares the canonical constructor for the `Rectangle` record class. It verifies that `length` and `width` are greater than zero. If not, it throws an `IllegalArgumentException`:

```
public Rectangle(double length, double width) {
  if (length <= 0 || width <= 0) {
    throw new java.lang.IllegalArgumentException(
      String.format("Invalid dimensions: %f, %f", length,
        width));
  }
  this.length = length;
  this.width = width;
}
```

Favor composition over inheritance

```
//Broken - Inappropriate use of inheritance!
public class InstrumentedHashSet<E> extends HashSet<E> {
  // The number of attempted element insertions
  private int addCount = 0;
  public InstrumentedHashSet() {}
  public InstrumentedHashSet(int initCap, float loadFactor) {
    super(initCap, loadFactor);
  }
  @Override public boolean add(E e) {
    addCount++;
    return super.add(e);
  }
  @Override public boolean addAll(Collection<? extends E> c) {
    addCount += c.size();
    return super.addAll(c);
  }
  public int getAddCount() {
    return addCount;
  }
}
```

Favor composition over inheritance

```
//Wrapper class - uses composition in place of inheritance
public class InstrumentedSet<E> extends ForwardingSet<E> {
  private int addCount = 0;
  public InstrumentedSet(Set<E> s) {
    super(s);
  }
  @Override public boolean add(E e) {
    addCount++;
    return super.add(e);
  }
  @Override public boolean addAll(Collection<? extends E> c) {
    addCount += c.size();
    return super.addAll(c);
  }
  public int getAddCount() {
    return addCount;
  }
}
```

Favor composition over inheritance

```
//Reusable forwarding class
public class ForwardingSet<E> implements Set<E> {
  private final Set<E> s;
  public ForwardingSet(Set<E> s) { this.s = s; }

  public void clear()               { s.clear();            }
  public boolean contains(Object o) { return s.contains(o); }
  public boolean isEmpty()          { return s.isEmpty();   }
  public int size()                 { return s.size();      }
  public Iterator<E> iterator()     { return s.iterator();  }
  public boolean add(E e)           { return s.add(e);      }
  public boolean remove(Object o)   { return s.remove(o);   }
  public boolean containsAll(Collection<?> c) { return s.containsAll(c); }
  public boolean addAll(Collection<? extends E> c) { return s.addAll(c); }
  public boolean removeAll(Collection<?> c) { return s.removeAll(c);   }
  public boolean retainAll(Collection<?> c) { return s.retainAll(c);   }
  public Object[] toArray()          { return s.toArray();  }
  public <T> T[] toArray(T[] a)      { return s.toArray(a); }
  @Override public boolean equals(Object o) { return s.equals(o);  }
  @Override public int hashCode()    { return s.hashCode(); }
  @Override public String toString() { return s.toString(); }
}
```

Use enums instead of int constants

```
public enum Apple {
  FUJI, PIPPIN, GRANNY_SMITH
}

public enum Orange {
  NAVEL, TEMPLE, BLOOD
}
```

```
public static final int APPLE_FUJI         = 0;
public static final int APPLE_PIPPIN       = 1;
public static final int APPLE_GRANNY_SMITH = 2;

public static final int ORANGE_NAVEL  = 0;
public static final int ORANGE_TEMPLE = 1;
public static final int ORANGE_BLOOD  = 2;
```

Enum type with data and behavior

- To associate data with enum constants, declare instance fields and write a constructor that takes the data and stores it in the fields.
- Java programming language enum types are much more powerful than their counterparts in other languages. The `enum` declaration defines a class (called an `enum type`). The enum class body can include methods and other fields. The compiler automatically adds some special methods when it creates an enum. For example, they have a static values method that returns an array containing all of the `values` of the enum in the order they are declared. This method is commonly used in combination with the for-each construct to iterate over the values of an enum type. For example, this code from the `Planet` class example below iterates over all the planets in the solar system.

Enum type with data and behavior

```
public enum Planet {
  MERCURY(3.302e+23, 2.439e6),
  VENUS(4.869e+24, 6.052e6),
  EARTH(5.975e+24, 6.378e6),
  MARS(6.419e+23, 3.393e6),
  JUPITER(1.899e+27, 7.149e7),
  SATURN(5.685e+26, 6.027e7),
  URANUS(8.683e+25, 2.556e7),
  NEPTUNE(1.024e+26, 2.477e7);
  // In kilograms
  private final double mass;
  // In meters
  private final double radius;
  // In m / s^2
  private final double
surfaceGravity;
  // Universal gravitational
constant in m^3 / kg s^2
  private static final double G
= 6.67300E-11;
```

```
// Constructor
Planet(double mass, double radius){
  this.mass = mass;
  this.radius = radius;
  surfaceGravity = G *
      mass / (radius * radius);
}
public double mass() {
  return mass;
}
public double radius() {
  return radius;
}
public double surfaceGravity() {
  return surfaceGravity;
}
public double surfaceWeight(
                    double mass) {
  // F = ma
  return mass * surfaceGravity;
}}
```

Enum with different behavior

- Enum type that switches on its own value - questionable

```
public enum Operation {
  PLUS, MINUS, TIMES, DIVIDE;

  // Do the arithmetic operation represented
by this constant
  public double apply(double x, double y) {
    switch (this) {
    case PLUS:
      return x + y;
    case MINUS:
      return x - y;
    case TIMES:
      return x * y;
    case DIVIDE:
      return x / y;
    }
    throw new AssertionError(
            "Unknown op: " + this);
  }
}
```

Enum with different behavior II

- Enum type with constant-specific class bodies and data

```
public enum Operation {
  PLUS("+") {
    public double apply(double x, double y)
      { return x + y;  }  },
  MINUS("-") {
    public double apply(double x, double y)
      { return x - y;  }  },
  TIMES("*") {
    public double apply(double x, double y)
      { return x * y;  }  },
  DIVIDE("/") {
    public double apply(double x, double y)
      { return x / y;  }  };
  private final String symbol;
  private Operation(String symbol)
    { this.symbol = symbol; }
  @Override public String toString()
    { return symbol; }
  public abstract double apply(
                      double x, double y);
```

Implementing a fromString method on an enum type

```
private static final Map<String, Operation>
stringToEnum =
  Stream.of(Operation.values()).collect(
    Collectors.toMap(Object::toString, e -> e));

//Returns Operation for string, if any
public static Optional<Operation> fromString(
  String symbol) {
  return Optional.ofNullable(
            stringToEnum.get(symbol));
}
```

Enum that switches on its value to share code – questionable

```
enum PayrollDay {
  MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY;
  private static final int MINS_PER_SHIFT = 8 * 60;

  int pay(int minutesWorked, int payRate) {
    int basePay = minutesWorked * payRate;

    int overtimePay;
    switch (this) {
    case SATURDAY:
    case SUNDAY: // Weekend
      overtimePay = basePay / 2;
      break;
    default: // Weekday
      overtimePay = minutesWorked <= MINS_PER_SHIFT ?
              0 : (minutesWorked - MINS_PER_SHIFT) * payRate / 2;
    }
    return basePay + overtimePay;
  }
}
```

The strategy enum pattern

```
//The strategy enum pattern
enum PayrollDay {
  MONDAY, TUESDAY, WEDNESDAY,
THURSDAY, FRIDAY,
SATURDAY(PayType.WEEKEND),
SUNDAY(PayType.WEEKEND);

  private final PayType payType;

  PayrollDay(PayType payType) {
    this.payType = payType;
  }
  PayrollDay() {
    this(PayType.WEEKDAY);
  } // Default

  int pay(int minutesWorked,
          int payRate) {
    return payType.pay(
          minutesWorked, payRate);
  }
```

```
  // The strategy enum type
  private enum PayType {
    WEEKDAY {
      int overtimePay(
          int minsWorked, int payRate) {
        return
          minsWorked <= MINS_PER_SHIFT
          ? 0 :
          (minsWorked - MINS_PER_SHIFT) *
          payRate / 2;
      } },
    WEEKEND {
      int overtimePay(
           int minsWorked, int payRate) {
        return minsWorked * payRate / 2;
      }     };
    abstract int overtimePay(
                int mins, int payRate);
    private static final int
                MINS_PER_SHIFT = 8 * 60;
    int pay(int minsWorked, int payRate)
{
      int basePay = minsWorked * payRate;
      return basePay +
         overtimePay(minsWorked,
payRate);
    }   }   }
```

Enum with switches

- Switches on enums are good for augmenting enum types with constant-specific behavior.

```
public static Operation
inverse(Operation op) {
  switch (op) {
  case PLUS:
    return Operation.MINUS;
  case MINUS:
    return Operation.PLUS;
  case TIMES:
    return Operation.DIVIDE;
  case DIVIDE:
    return Operation.TIMES;

  default:
    throw new
AssertionError("Unknown op: " + op);
  }
}
```

Enum with switches

- Switches on enums are good for augmenting enum types with constant-specific behavior.

```
public static Operation inverse(Operation op) {
  //Can raise NullPointerException
  return switch (op) {
    case PLUS -> Operation.MINUS;
    case MINUS -> Operation.PLUS;
    case TIMES -> Operation.DIVIDE;
    case DIVIDE ->  Operation.TIMES;
  };
}

public Operation inverse() {
  return switch (this) {
    case PLUS -> Operation.MINUS;
    case MINUS -> Operation.PLUS;
    case TIMES -> Operation.DIVIDE;
    case DIVIDE ->  Operation.TIMES;
  };
}
```

Using enums

- Use enums any time you need a set of constants whose members are known at compile time.
- It is not necessary that the set of constants in an enum type stay fixed for all time.

Prefer lambdas to anonymous classes

```
// Anonymous class instance as a function object - obsolete!
Collections.sort(words, new Comparator<String>() {
  public int compare(String s1, String s2) {
    return Integer.compare(s1.length(), s2.length());
  }
});

// Lambda expression as function object (replaces anonymous
class)
Collections.sort(words, (s1, s2) ->
   Integer.compare(s1.length(), s2.length()));
```

Lambdas

- Omit the types of all lambda parameters unless their presence makes your program clearer.
- if a computation isn’t self-explanatory, or exceeds a few lines, don’t put it in a lambda
- Don’t use anonymous classes for function objects unless you have to create instances of types that aren’t functional interfaces.

Prefer method references to lambdas

Method Ref Type	Example	Lambda Equivalent
Static	`Integer::parseInt`	`str -> Integer.parseInt(str)`
Bound	`Instant.now()::isAfter`	`Instant then = Instant.now();` `t -> then.isAfter(t)`
Unbound	`String::toLowerCase`	`str -> str.toLowerCase()`
Class Constructor	`TreeMap<K,V>::new`	`() -> new TreeMap<K, V>()`
Array Constructor	`int[]::new`	`len -> new int[len]`

Prefer method references to lambdas

- Where method references are shorter and clearer, use them; where they aren't, stick with lambdas.

```
//lambda
map.merge(key, 1, (count, incr) -> count + incr);

//method reference
map.merge(key, 1, Integer::sum);
```

Favor the use of standard functional interfaces

Interface	Function Signature	Example
`UnaryOperator<T>`	`T apply(T t)`	`String::toLowerCase`
`BinaryOperator<T>`	`T apply(T t1, T t2)`	`BigInteger::add`
`Predicate<T>`	`boolean test(T t)`	`Collection::isEmpty`
`Function<T,R>`	`R apply(T t)`	`Arrays::asList`
`Supplier<T>`	`T get()`	`Instant::now`
`Consumer<T>`	`void accept(T t)`	`System.out::println`

Functional interface

- If one of the standard functional interfaces does the job, you should generally use it in preference to a purpose-built functional interface.
- Don’t be tempted to use basic functional interfaces with boxed primitives instead of primitive functional interfaces.
- Always annotate your functional interfaces with the `@FunctionalInterface` annotation.

BiConsumer<T,U>, BiFunction<T,U,R>, BinaryOperator<T>, BiPredicate<T,U>, Consumer<T>, Function<T,R>, ObjDoubleConsumer<T>, ObjIntConsumer<T>, ObjLongConsumer<T>, Predicate<T>, Supplier<T>, ToDoubleBiFunction<T,U>, ToDoubleFunction<T>, ToIntBiFunction<T,U>, ToIntFunction<T>, ToLongBiFunction<T,U>, ToLongFunction<T>, UnaryOperator<T>

BooleanSupplier, DoubleBinaryOperator, DoubleConsumer, DoubleFunction<R>, DoublePredicate, DoubleSupplier, DoubleToIntFunction, DoubleToLongFunction, DoubleUnaryOperator, IntBinaryOperator, IntConsumer, IntFunction<R>, IntPredicate, IntSupplier, IntToDoubleFunction, IntToLongFunction, IntUnaryOperator, LongBinaryOperator, LongConsumer, LongFunction<R>, LongPredicate, LongSupplier, LongToDoubleFunction, LongToIntFunction, LongUnaryOperator

Use streams judiciously

Streams

- easier to read
- Shorter
- Slower (sometimes)

No streams

- easier to read
- easier to debug (for cycle)

- Prefer side-effect-free functions in streams
- Prefer Collection to Stream as a return type
- Use caution when making streams parallel

Check parameters for validity

```
/**
 * Returns a BigInteger whose value is (this mod m).
 * This method differs from the remainder method
 * in that it always returns a non-negative BigInteger.
 *
 * @param m the modulus, which must be positive
 * @return this mod m
 * @throws ArithmeticException if m is less than or equal to 0
 */
public BigInteger mod(BigInteger m) {
  if (m.signum() <= 0)
    throw new ArithmeticException("Modulus <= 0: " + m);
  ... // Do the computation
}
```

Check null, ranges

```
//Inline use of Java's null-checking facility
this.strategy = Objects.requireNonNull(strategy,
"strategy");
```

Another inline `public static` methods of class `java.util.Objects`

```
int checkFromIndexSize(
   int fromIndex, int size, int length)
long checkIndex(long index, long length)
long checkFromToIndex(
    long fromIndex, long toIndex, long length)
```

Make defensive copies when needed

- You must program defensively, with the assumption that clients of your class will do their best to destroy its invariants.

Mutable parameters in constructor

- it is essential to make a defensive copy of each mutable parameter to the constructor
- defensive copies are made before checking the validity of the parameters and the validity check is performed on the copies rather than on the originals

```
//Repaired constructor - makes defensive copies of parameters
public Period(Date start, Date end) {
 this.start = new Date(start.getTime());
 this.end   = new Date(end.getTime());

 if (this.start.compareTo(this.end) > 0)
   throw new IllegalArgumentException(
       this.start + " after " + this.end);
}
```

Mutable return values

- return defensive copies of mutable internal fields

```
public Date start() {
  return new Date(start.getTime());
}

public Date end() {
  return new Date(end.getTime());
}
```

Mutable return values

- Wrap mutable return values with unmutable wrappers

```
public Collection<Objects> getCollections() {
  return Collections.unmodifiableCollection(collections);
}
```

Return empty collections or arrays, not nulls

- never return null in place of an empty array or collection

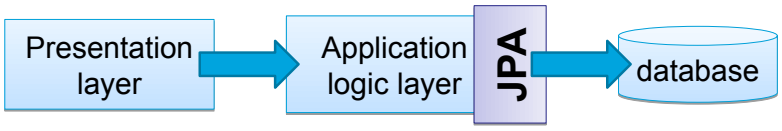
```
List<Cheese> cheeses = shop.getCheeses();
if (cheeses != null && cheeses.contains(Cheese.STILTON)) {
   System.out.println("Jolly good, just the thing.");
}

return Collections.emptyList();
```

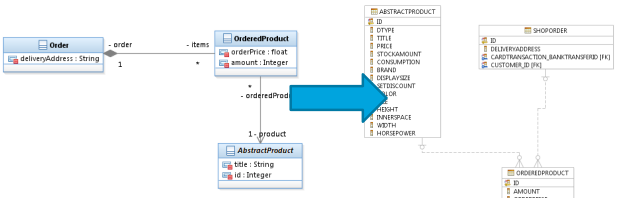
4th lecture

- JPA – Java Persistence API
- CDI – Common Dependency Injection

JPA: overview

- API persistency using ORM
- Only interface – implementation should be connected.





JPA: Entity

- **Entity** – light-wight object from persistence object. Typicaly are related with database table. Each object is related to one record in the database table.
- **Persistent state of entity**: represented by instance variables and class properties. Mapping between database and properties is defined by annotations.
-

JPA – Entity class

- an annotation
- `javax.persistence.Entity`
- Nonparametric public or protected constructor.
- Class nor methods nor instance variables are `final`
- Entity class can be descendant of entity class or non-entity class. Non-entity classes can be descendant of entity class.
- Persistence instance variables have to be declared as private, protected or package-private. They should be accessed through set and get methods.

JPA: example of Entity class

```
@Entity
@Table(name="ShopOrder")
public class Order {
   @Id
   @GeneratedValue(strategy=GenerationType.IDENTITY)
   private int id;
   @OneToOne
   private Transaction cardTransaction;
   @ManyToOne()
   private Customer customer;
   @OneToMany(mappedBy="order")
   private Set<OrderedProduct> items;
   private String deliveryAddress;
   …
 }
```

JPA: persistence properties, instance variables

- Instance variables – persistence provider access directly to them
- Properties – Persistence access properties using get, set method
  - Can be used: Collection, Set, List, Map even generic versions
- override `equals()` `hashcode()`
- Types:
  - Java primitive data types
  - java.lang.String,
  - other serializable types (boxed classes, java.math.BigInteger, java.math.BigDecimal, java.util.Date, java.util.Calendar, java.sql.Date, java.sql.Time, java.sql.TimeStamp, user serializable types, byte[], Byte[], char[], Character[], enum types, other entities,

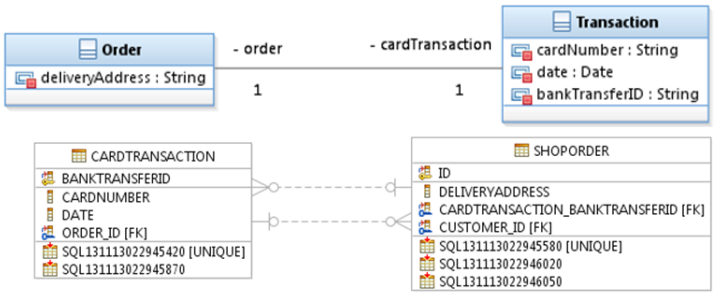
JPA: primary key

- Every entity should contain own key
- `@javax.persistence.Id`
- Composite Primary Key
  - Have to exisit class which define composite key
  - @javax.persistence.EmbeddedId
  - @javax.persistence.IdClass
  - Have to be composed from types:
    - Java Primitive data types (and coresponding embedded classes)
    - java.lang.String
    - java.util.Date (DATE), java.sql.Date
- Float numbers should not be used.

JPA – relation 1-1

```
@Entity
public class Order {
@OneToOne
private Transaction cardTransaction;
…
```

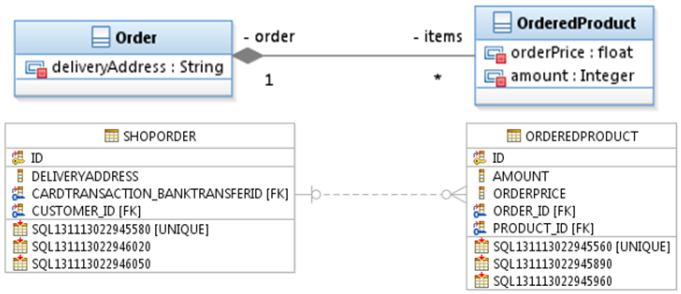
```
@Entity
public class Transaction {
@OneToOne
private Order order;
…
```



JPA – relation 1-N

```
@Entity
public class Order {
@OneToMany(mappedBy="order")
private Set<OrderedProduct>
items;
…
```

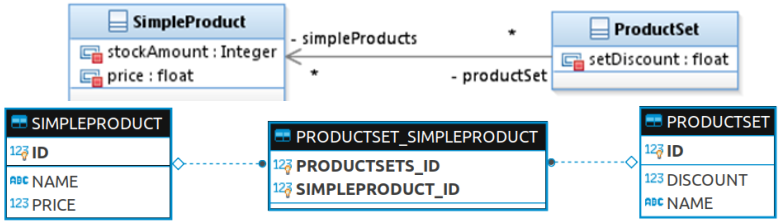
```
@Entity
public class OrderedProduct {
@ManyToOne
private Order order;
…
```



JPA – relation M-N

```
@Entity
public class SimpleProduct
    extends AbstractProduct {
  @ManyToMany(
    mappedBy="simpleProduct")
private List<ProductSet>
  ProductSets;
}
```

```
@Entity
public class ProductSet
    extends AbstractProduct
  @ManyToMany
  private List<SimpleProduct>
    simpleProduct;
  private float setDiscount;
}
```



JPA - inheritence

- Entity can extend non-entity or abstract class

```
@Entity
public abstract class Employee {
  @Id
  protected Integer employeeId;
}
//  ...
@Entity
public class FullTimeEmployee
     extends Employee {
  protected Integer salary;
}
// ...
@Entity
public class PartTimeEmployee
    extends Employee {
  protected Float hourlyWage;
}
```

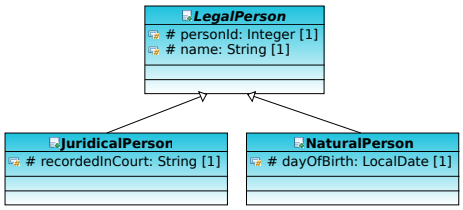
JPA – inheritance mapping strategy

- One table on a class hierarchy
- One table for a particular class
- Join strategy

```
public enum InheritanceType {
  SINGLE_TABLE, TABLE_PER_CLASS, JOINED
}

@Inheritance(strategy = InheritanceType.JOINED)
```

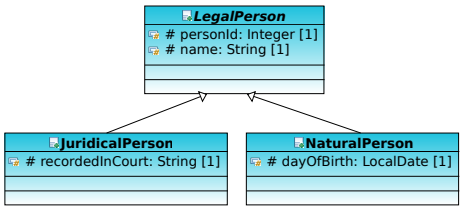
JPA: SINGLE_TABLE

```
@Inheritance(strategy=
  InheritanceType.SINGLE_TABLE)

@DiscriminatorColumn(
  name = "type",
  ColumnDefinition =
     "TINYINT(1)",
  DiscriminatorType =
     DiscriminatorType.INTEGER)

public enum DiscriminatorType
{ STRING, CHAR, INTEGER }

@DiscriminatorValue(
  value = "1")
```



LegalPerson	
PK	personID
	name
	dayOfBirth
	recordInCourt
	discriminator

JPA: One table for particular entity

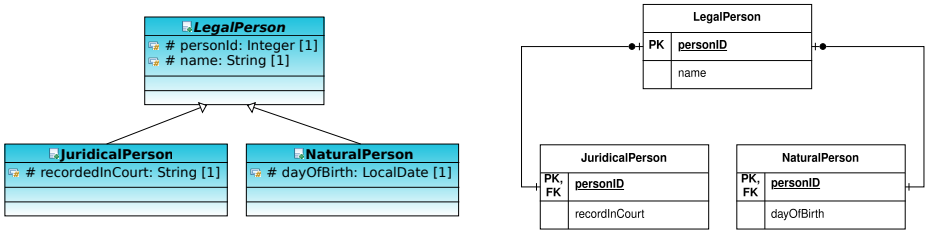


JuridicalPerson	
PK	personID
	name
	recordInCourt

NaturalPerson	
PK	personID
	name
	dayOfBirth

```
@Inheritance(strategy=
  InheritanceType.TABLE_PER_CLASS)
```

JPA: Join strategy




```
@Inheritance(strategy=InheritanceType.JOINED)
```

JPA: MappedSuperclass

```java
@MappedSuperclass
public abstract class LegalPerson {
  @Id
  @GeneratedValue(strategy = GenerationType.AUTO)
  protected Integer personId;
  protected String name;
}

@Entity
public class NaturalPerson extends LegalPerson {
  protected LocalDate dayOfBirth;
}

@Entity
public class JuridicalPerson extends LegalPerson {
  protected String recordedInCourt;
}
```

JPA: entity management

- Persistent context: set of entities existing in a particular datastore
- EntityManager:
  - Creates, deletes, finds, exetues queries

JPA: Application managed EntityManager

```
// fetched from somewhere - CDI for example;
EntityManager em;

// fetched from somewhere - CDI for example;
EntityManagerFactory emf;

//In desktop app without CDI
emf = Persistence
  .createEntityManagerFactory("persistenceUnitName");

EntityManager em = emf.createEntityManager();
// use it in the current thread
```

JPA: find entity

```
@PersistenceContext
EntityManager em;

public void enterInvoice(int personID, Invoice newInvoice) {
  LegalPerson person = em.find(LegalPerson.class, personID);
  person.getInvoices().add(newInvoice);
  newInvoice.setPerson(person);
}
```

JPA: entity lifecycle

- New
- Managed
- Detached
- Removed

```
em.persist(newInvoice);
em.merge(person);
em.flush();
em.detach(person);
em.remove(person);
```

JPA: queries

```
public List<LegalPerson> findWithName(String name) {
  return em.createQuery(
  "SELECT p FROM LegalPerson p WHERE p.name LIKE :personName",
LegalPerson.class)
  .setParameter("personName", name)
  .setMaxResults(10)
  .getResultList();
}

//...
query.setFirstResult(100)
```

JPA: named queries

```
@NamedQuery(//class annotation - entity
    name="findAllPersonsWithName",
    query="SELECT p FROM LegalPerson p WHERE p.name
LIKE :personName"
)

return em.createNamedQuery("findAllCustomersWithName",
LegalPerson.class)
    .setParameter("personName", "Smith")
    .getResultList();
```

JPA: parameters in queris

- Named

```
return em.createQuery(
"SELECT c FROM LegalPerson p WHERE p.name LIKE :personName",
    LegalPerson.class)
.setParameter("personName", name)
.getResultList();
```

- Numbered

```
return em.createQuery(
"SELECT c FROM LegalPerson p WHERE p.name LIKE ?1",
    LegalPerson.class)
.setParameter(1, name)
.getResultList();
```

JPA: Persistence Units

- Package containing all entity class mapped on one datastore
- must contain file
  - `META-INF/persistence.xml`

persistence.xml

```
<?xml version="1.0" encoding="UTF-8"?>
    <persistence version="2.1" xmlns="http://xmlns.jcp.org/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
    http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd">
      <persistence-unit name="Slaids">
        <provider>org.hibernate.ejb.HibernatePersistence</provider>
        <jta-data-source>java:/jdbc/slaids</jta-data-source>
        <properties>
          <property name="jakarta.persistence.schema-generation.database.action"
            value="create"/>
          <property name="hibernate.hbm2ddl.auto" value="create"/>
          <property name="hibernate.dialect"
            value="org.hibernate.dialect.DerbyTenSevenDialect"/>
        </properties>
      </persistence-unit>
    </persistence>
```

JPA – Query Language

- Select Statements
  - SELECT, FROM, WHERE, GROUP BY, HAVING, ORDER BY
- Update, Delete Statement
  - UPDATE Player p SET p.status = ’inactive’ WHERE p.lastPlayed < :inactiveThresholdDate
  - DELETE FROM Player p WHERE p.status = ’inactive’ AND p.teams IS EMPTY

JPA: examples of queries

- SELECT p FROM Player AS p
- SELECT DISTINCT p FROM Player AS p WHERE p.position = ?1
- SELECT DISTINCT t FROM Player AS p JOIN p.teams AS t
- SELECT DISTINCT p FROM Player AS p WHERE p.team IS NOT EMPTY
- SELECT t FROM Team AS t JOIN t.league AS l WHERE l.sport = ’soccer’ OR l.sport =’football’
- SELECT DISTINCT p FROM Player AS p JOIN p.teams AS t WHERE t.city = :city
- SELECT DISTINCT p FROM Player AS p JOIN p.teams AS t WHERE t.league.sport = :sport

JPA: LIKE in query

- SELECT p FROM Player p WHERE p.name LIKE ’Mich%’
- _ - any one character
- % - zero or many any characters
- ESCAPE – defines escape character

- LIKE ’_%’ ESCAPE ’\’
- NOT LIKE

JPA: IS EMPTY, NULL in queries

- SELECT t FROM Team t WHERE t.league IS NULL
- SELECT t FROM Team t WHERE t.league IS NOT NULL
- Cannot use WHERE t.league = NULL

- SELECT p FROM Player p WHERE p.teams IS EMPTY
- SELECT p FROM Player p WHERE p.teams IS NOT EMPTY

JPA – BETWEEN, IN in queries

- SELECT DISTINCT p FROM Player p WHERE p.salary BETWEEN :lowerSalary AND :higherSalary
- p.salary >= :lowerSalary AND p.salary <= :higherSalary

- o.country IN ('UK', 'US', 'France')

JPA Criteria1

- It enables us to write queries without doing raw QL
- Gives us some object-oriented control over the queries
- Enable do easily and reliable dynamical queries

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<LegalPerson> cr =
    cb.createQuery(LegalPerson.class);
Root<LegalPerson> root = cr.from(LegalPerson.class);
cr.where(cb.like(root.get("name"), "Dav%"));
cr.select(root);
TypedQuery<LegalPerson> query = em.createQuery(cr);
List<LegalPerson> results = query.getResultList();
```

JPA criteria – using expression

```
cr.select(root).where(
  cb.gt(root.get("itemPrice"), 1000));

cr.select(root).where(
  cb.like(root.get("itemName"), "%chair%"));

cr.select(root).where(
  cb.between(root.get("itemPrice"), 100, 200));
```

JPA criteria - predicate chaining

```
Predicate greaterThanPrice =
  cb.gt(root.get("itemPrice"), 1000);

Predicate chairItems =
  cb.like(root.get("itemName"), "Chair%");

cr.select(root).where(
  cb.or(greaterThanPrice, chairItems));
```

JPA criteria – using metamodel1

```java
@Entity
@Table(name = "students")
public class Student {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private int id;

    @Column(name = "first_name")
    private String firstName;

    @Column(name = "last_name")
    private String lastName;

    @Column(name = "grad_year")
    private int gradYear;

    // standard getters and setters
}
```

```java
@Generated(value = "org.hibernate.jpamodelgen.JPAMetaModelEntityProcessor")
@StaticMetamodel(Student.class)
public abstract class Student_ {

    public static volatile SingularAttribute<Student, String> firstName;
    public static volatile SingularAttribute<Student, String> lastName;
    public static volatile SingularAttribute<Student, Integer> id;
    public static volatile SingularAttribute<Student, Integer> gradYear;

    public static final String FIRST_NAME = "firstName";
    public static final String LAST_NAME = "lastName";
    public static final String ID = "id";
    public static final String GRAD_YEAR = "gradYear";
}
```

```java
//session set-up code
CriteriaBuilder cb = session.getCriteriaBuilder();
CriteriaQuery<Student> criteriaQuery = cb.createQuery(Student.class);

Root<Student> root = criteriaQuery.from(Student.class);
criteriaQuery.select(root).where(cb.equal(root.get(Student_.gradYear), 2015));

Query<Student> query = session.createQuery(criteriaQuery);
List<Student> results = query.getResultList();
```

[1] https://www.baeldung.com/hibernate-criteria-queries-metamodel

CDI - Contexts and Dependency Injection

- standard dependency injection framework included in Java EE 6 and higher.
- Contexts and Dependency Injection (CDI) enables your objects to have their dependencies provided to them **automatically**, instead of creating them or receiving them as parameters. CDI also **manages the lifecycle** of those dependencies for you.
- Java beans - **CDI bean**. CDI beans are classes that CDI can instantiate, manage, and inject automatically to satisfy the dependencies of other objects.
- Almost any Java class can be managed and injected by CDI - JavaBeans.

CDI – Contexts and Dependency Injection

- **Contexts:** This service enables you to bind the lifecycle and interactions of stateful components to well-defined but extensible lifecycle contexts.
- **Dependency injection**: This service enables you to inject components into an application in a typesafe way and to choose at deployment time which implementation of a particular interface to inject.
- Integration with the Expression Language (EL)
- The ability to decorate injected components
- The ability to associate interceptors with components using typesafe interceptor bindings
- An event-notification model
- A web conversation scope in addition to the three standard scopes (request, session, and application) defined by the Java Servlet specification

CDI – About Beans

A bean has the following attributes:

- A (nonempty) set of bean types
- A (nonempty) set of qualifiers
- A scope
- Optionally, a bean EL name
- A set of interceptor bindings
- A bean implementation

CDI – beans

The following kinds of objects can be injected:

- (Almost) any Java class
- Session beans
- Java EE resources: data sources, Java Message Service topics, queues, connection factories, and the like
- Persistence contexts (Java Persistence API EntityManager objects)
- Producer fields
- Objects returned by producer methods
- Web service references
- Remote enterprise bean references

CDI – Scopes

@RequestScoped

- A user's interaction with a web application in a single HTTP request.

@SessionScoped

- A user's interaction with a web application across multiple HTTP requests.

@ApplicationScoped

- Shared state across all users' interactions with a web application.

@Dependent

- The default scope if none is specified; it means that an object exists to serve exactly one client (bean) and has the same lifecycle as that client (bean).

@ConversationScoped

- A user's interaction with a servlet, including JavaServer Faces applications. The conversation scope exists within developer-controlled boundaries that extend it across multiple requests for long-running conversations. All long-running conversations are scoped to a particular HTTP servlet session and may not cross session boundaries.

@ViewScoped

- Come with JSF 2.2

CDI – Giving Beans EL Names

- @Named
- @Named("AnyName")

5th Lecture

- REST
- HTTP-based RESTful API
- Quarkus
- REST client

REST – REpresentational State Transfer

- Software architectural style
- Fielding, Roy Thomas (2000). "Chapter 5: Representational State Transfer (REST)". Architectural Styles and the Design of Network-based Software Architectures (Ph.D.). University of California, Irvine.
- REST gives a coordinated set of constraints to the design of components in a distributed hypermedia system that can lead to a higher-performing and more maintainable architecture.
- To the extent that systems conform to the constraints of REST they can be called RESTful.

Representational State Transfer (REST)

- Communicate over HTTP with the same HTTP verbs (GET, POST, PUT, DELETE, etc.)
- REST interfaces with external systems using resources identified by URI
- DELETE /people/tom
- Roy Thomas Fielding in his 2000 PhD dissertation "Architectural Styles and the Design of Network-based Software Architectures"

VŠB TECHNICKÁ UNIVERZITA OSTRAVA | FAKULTA ELEKTROTECHNIKY A INFORMATIKY | KATEDRA INFORMATIKY

Representational State Transfer (REST)

Architectural constraints

- **Client–server**
- **Stateless**
- **Cacheable**
- **Layered system**
- **Code on demand (optional)**
- **Uniform interface**
  - Identification of resources
  - Manipulation of resources through these representations
  - Self-descriptive messages
  - Hypermedia as the engine of application state (HATEOAS)

Architectural constraints
The architectural properties of REST are realized by applying specific interaction constraints to components, connectors, and data elements.[4][6] One can characterise applications conforming to the REST constraints described in this section as "RESTful".[2] If a service violates any of the required constraints, it cannot be considered RESTful. Complying with these constraints, and thus conforming to the REST architectural style, enables any kind of distributed hypermedia system to have desirable non-functional properties, such as performance, scalability, simplicity, modifiability, visibility, portability, and reliability.[4]
The formal REST constraints are:
Client–server
See also: Client–server model
A uniform interface separates clients from servers. This separation of concerns means that, for example, clients are not concerned with data storage, which remains internal to each server, so that the portability of client code is improved. Servers are not concerned with the user interface or user state, so that servers can be simpler and more scalable. Servers and clients may also be replaced and developed independently, as long as the interface between them is not altered.
Stateless
See also: Stateless protocol
The client–server communication is further constrained by no client context being stored on the server between requests. Each request from any client contains all the information necessary to service the request, and session state is held in the client. The session state can be transferred by the server to another service such as a database to maintain a persistent state for a period and allow authentication. The client begins sending requests when it is ready to make the transition to a new state. While one or more requests are outstanding, the client is considered to be *in transition*. The representation of each application state contains links that may be used the next time the client chooses to initiate a new state-transition.[8]
Cacheable
See also: Web cache
As on the World Wide Web, clients and intermediaries can cache responses. Responses must therefore, implicitly or explicitly, define themselves as cacheable, or not, to prevent clients from reusing stale or inappropriate data in response to further requests. Well-managed caching partially or completely eliminates some client–server interactions, further improving scalability and performance.
Layered system
See also: Layered system
A client cannot ordinarily tell whether it is connected directly to the end server, or to an intermediary along the way. Intermediary servers may improve system scalability by enabling load balancing and by providing shared caches. They may also enforce security policies.
Code on demand (optional)
See also: Client-side scripting
Servers can temporarily extend or customize the functionality of a client by the transfer of executable code. Examples of this may include compiled components such as Java applets and client-side scripts such as JavaScript. "Code on demand" is the only optional constraint of the REST architecture.
Uniform interface
The uniform interface constraint is fundamental to the design of any REST service.[4] The uniform interface simplifies and decouples the architecture, which enables each part to evolve independently. The four constraints for this uniform interface are:
Identification of resources Individual resources are identified in requests, for example using URIs in web-based REST systems. The resources themselves are conceptually separate from the representations that are returned to the client. For example, the server may send data from its database as HTML, XML or JSON, none of which are the server's internal representation. Manipulation of resources through these representations When a client holds a representation of a resource, including any metadata attached, it has enough information to modify or delete the resource. Self-descriptive messages Each message includes enough information to describe how to process the message. For example, which parser to invoke may be specified by an Internet media type (previously known as a MIME type). Responses also explicitly indicate their cacheability.[4] Hypermedia as the engine of application state (HATEOA

What Are RESTful Web Services?
RESTful web services are built to work best on the Web. Representational State Transfer (REST) is an architectural style that specifies constraints, such as the uniform interface, that if applied to a web service induce desirable properties, such as performance, scalability, and modifiability, that enable services to work best on the Web. In the REST architectural style, data and functionality are considered resources and are accessed using **Uniform Resource Identifiers (URIs)**, typically links on the Web. The resources are acted upon by using a set of simple, well-defined operations. The REST architectural style constrains an architecture to a client/server architecture and is designed to use a stateless communication protocol, typically HTTP. In the REST architecture style, clients and servers exchange representations of resources by using a standardized interface and protocol.
The following principles encourage RESTful applications to be simple, lightweight, and fast:
Resource identification through URI: A RESTful web service exposes a set of resources that identify the targets of the interaction with its clients. Resources are identified by URIs, which provide a global addressing space for resource and service discovery. See The @Path Annotation and URI Path Templates for more information.
Uniform interface: Resources are manipulated using a fixed set of four create, read, update, delete operations: PUT, GET, POST, and DELETE. PUT creates a new resource, which can be then deleted by using DELETE. GET retrieves the current state of a resource in some representation. POST transfers a new state onto a resource. See Responding to HTTP Methods and Requests for more information.
Self-descriptive messages: Resources are decoupled from their representation so that their content can be accessed in a variety of formats, such as HTML, XML, plain text, PDF, JPEG, JSON, and others. Metadata about the resource is available and used, for example, to control caching, detect transmission errors, negotiate the appropriate representation format, and perform authentication or access control. See Responding to HTTP Methods and Requests and Using Entity Providers to Map HTTP Response and Request Entity Bodies for more information.
Stateful interactions through hyperlinks: Every interaction with a resource is stateless; that is, request messages are self-contained. Stateful interactions are based on the concept of explicit state transfer. Several techniques exist to exchange state, such as URI rewriting, cookies, and hidden form fields. State can be embedded in response messages to point to valid future states of the interaction. See Using Entity Providers to Map HTTP Response and Request Entity Bodies and “Building URIs” in the JAX-RS Overview document for more information.

HTTP-based RESTful API

- Web service API
- REST architectural constraints
- Protocol HTTP – most common implementation:
  - URI
  - HTTP methods
  - media type

RESTfull – API rules

- **Hypermedia Controls** - The objective of hypermedia controls is to advise the client of what can be done next and to supply the URIs necessary to perform the next action.
- **Resource Naming** - RESTful APIs are written for clients and should have meaning for the clients of those APIs. When choosing nouns to name the resources, you should be familiar with the structure of the application’s data and how your clients are likely to use them. There are no defined rules as to how you should name your resources, but there are conventions that, if followed, can help you create a set of self-descriptive resource names that others intuitively understand.

RESTfull – API rules

- **Nouns Not Verbs** - You must name the resources after nouns, not verbs or actions. The purpose of the resource name is to represent the resource. The HTTP method describes the action to be performed.
- To represent a single user resource, you would use the noun users to represent all users and the user's ID to identify the specific user, like so:

  users/123456

- An example of a **non REST** and badly formed URI would be

  users/123456/update ,

  users/123456?action=update

RESTfull – API rules

- The nature of data is that it is **hierarchical**. So imagine that you want to represent all the posts of the user with ID 123456 . You would use the noun posts to represent all posts and create the URI

  users/123456/posts

- **different ways** - To represent all posts by a specified user, you can use the URI

  posts/users/123456

RESTfull – API rules

- **Self Descriptive** - As you have seen, the nouns chosen should reflect the resource they represent. Combining these representations with identifiers makes the URI easy to interpret and intuitive to understand. If you read a URI in combination with its HTTP method and it is not immediately obvious what resource it represents, it has failed as a RESTful URI.

RESTfull – API rules

- **Plural Not Singular** - Resource names should be plural because they represent collections of data. The resource name users represents a collection of users, and the resource name posts represents a collection of posts.
- The idea is that plural nouns represent a collection in the service, and the ID refers to one instance within that collection.
- It may be justifiable to use a singular noun if there is only one instance of that data type in the entire application, but this is quite uncommon.

RESTfull – API rules – HTTP methods

- GET - You use this method to get resource representations from the service. You should never use it to update, delete, or create a resource. Calling it once should have the same effect as calling it 100 times.
- If the resource requested is successful, the representation of the resource is returned in the body of the HTTP response in the requested data format, which commonly is either JSON or XML. The HTTP response code returned is 200 (OK) . If the resource is not found, it should return 404 (NOT FOUND) , and if the resource request is badly formed, it should return 400 (BAD REQUEST) .
- A well formed URI that you might use in your forum application could be GET users/123456/ followers , which represents all the followers of the user 123456 .

RESTfull – API rules – HTTP methods

- **POST** - You use the POST method to create a new resource within the given context. For example, to create a new user, you would post to the users resource the data necessary for a new user to be created. The service takes care of creating the new resource, associating it to the context, and assigning an ID.
- On successful creation, the HTTP response is 201 (CREATED) , and a link to the newly created resource is returned either in the Location header of the response or in the JSON payload of the response body. The resource representation may be returned in the response body. This is often preferable to avoid making an additional call to the API to retrieve a representation of the data that had been just created. This reduces the chattiness of the API.
- In addition to the HTTP response codes to a GET request, a POST can return 204 (NO CONTENT) if the body of the request is empty. A well formed URI that you might use in your forum application could be POST users , with a request body containing the new user's details or POST users/123456/ posts to create a new post for the user 123456 from the data in the request body.

RESTfull – API rules – HTTP methods

- **PUT** - The PUT method is most commonly used to update a known resource. The URI includes enough information to identify the resource, such as a context and an identifier. The request body contains the updated version of the resource.
- If the update is successful, it returns the HTTP response code 200 . A URI that updates a user's information is PUT users/123456 . Less commonly, you can use the PUT method to create a resource if the client creates the identifier of the resource. However, this way of creating a resource is a little confusing. Why use a PUT when a POST works just as well and is commonly known?
- An important point to note about updating a resource is that the entire representation of the resource is passed to the service in the HTTP body request, not just the information that has changed.

RESTfull – API rules – HTTP methods

- **DELETE** - Surprisingly, you use this method to delete a resource from a service. The URI contains the context and the identifier of the resource. To delete a user with the ID 123456, you use the URI

  DELETE users/123456

- The response body may include a representation of the deleted resource. A successful deletion results in a 200 (OK) HTTP response code being returned; if the resource is not found, a 400 code is returned.

RESTfull – API rules

- A REST API must not define fixed resource names or hierarchies (an obvious coupling of client and server). Servers must have the freedom to control their own namespace. Instead, allow servers to instruct clients on how to construct appropriate URIs, such as is done in HTML forms and URI templates, by defining those instructions within media types and link relations.
- *[Failure here implies that clients are assuming a resource structure due to out-of band information, such as a domain-specific standard, which is the data-oriented equivalent to RPC's functional coupling].*

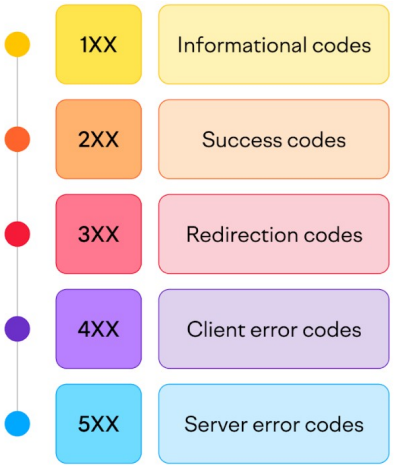
Semantic of HTTP methods

HTTP method	Description	CRUD	
GET	Get a representation of the target resource's state.	Fetch all or any resource	GET /user/ GET /user/1
POST	Let the target resource process the representation enclosed in the reques.	Create a Resources	POST /user?name=user17age=20
PUT	Set the target resurce's state tot the state defined by the representation enclosed in the request.	Update a Resource	PUT /user/1?name=changed-name
DELETE	Delete the target resource's state.	Delete a Resource	DELETE /user/1
HEAD	Fetch metainfo		HEAD /user
OPTIONS	Fetch all verbs allowed		OPTIONS /user

HTTP Headers

Headers	Example
Auth: <session-token>	Auth: 1155dassdasd5-asd5666asd-asdas
Accept: <media Type>	Accept:application/json
Content-Type: <ct>	Content-Type: text/html; charset=UTF-8
Allow:<methods>	Allow: GET, POST, HEAD

HTTP Status Codes



Status Codes	
200 – OK	Successful Return for sync call
307 – Temporarily Moved	Redirection
400 – Bad Request	Invalid URI, header, request param
401 – Un authorized	User not authorized for operation
403 – Forbidden	User not allowed to update
404 – Not Found	URI Path not available
405 – Method not allowed	Method not valid for the path
500 – Internal Server Error	Server Errors
503 – Service unavailable	Server not accessible

RESTfull – API CoD

Code on Demand

- REST allows client functionality to be extended by downloading and executing code in the form of applets or scripts. This simplifies clients by reducing the number of features required to be pre-implemented. Allowing features to be downloaded after deployment improves system extensibility. However, it also reduces visibility, and thus is only an optional constraint within REST.
- At the time this was written, the web was mostly just static documents and the only "web client" was the browser itself. Now it's commonplace for JavaScript-powered web apps to be consuming REST APIs. This is an example of code on demand - the browser grabs an initial HTML document and supports <script> tags inside that document so that an application can be loaded on-demand.

JAX-RS (Java API for RESTful Web Services or Jakarta RESTful Web Services)

URL prefix for whole application:

```
@ApplicationPath("/api")

public class RestApplication
  extends jakarta.ws.rs.
    core.Application {

}
```

Example of class to handle URL requests:

```
@Path("/notifications")
public class NotificationsResource {
  @GET
  @Path("/ping")
  public Response ping() {
    return Response.ok().entity("Service online").build();
  }
  @GET
  @Path("/{id}")
  @Produces(MediaType.APPLICATION_JSON)
  public Response getNotification(@PathParam("id") int id) {
    return Response.ok().entity(new Notification(id, "john",
"test notification")).build();
  }
  @POST
  @Path("/")
  @Consumes(MediaType.APPLICATION_JSON)
  @Produces(MediaType.APPLICATION_JSON)
  public Response postNotif(Notification n) {
    return Response.status(201).entity(n).build();
  }
}
```

JAX-RS: Path and Query parameters

```
@Path("/")
public class DatasetRegisterServiceEndpoint {
  public static final String UUID = "uuid";
  private static final String R_X_PARAM = "rx";
  private static final String R_Y_PARAM = "ry";
  private static final String R_Z_PARAM = "rz";
  private static final String VERSION_PARAM = "version";
  private static final String MODE_PARAM = "model";
  private static final String TIMEOUT_PARAM = "timeout";
  // ...

  @Path("datasets" + "/{" + UUID + "}" + "/{" + R_X_PARAM + "}" + "/{" + R_Y_PARAM + "}" +
     "/{" + R_Z_PARAM + "}" + "/{" + VERSION_PARAM + "}" + "/{" + MODE_PARAM + "}")
  @GET
  public Response start(@PathParam(UUID) String uuid, @PathParam(R_X_PARAM) int rX,
    @PathParam(R_Y_PARAM) int rY, @PathParam(R_Z_PARAM) int rZ,
    @PathParam(VERSION_PARAM) String version,
    @SuppressWarnings("unused") @PathParam(MODE_PARAM) String mode,
    @QueryParam(TIMEOUT_PARAM) Long timeout) {
//EXAMPLE Query: GET  /datasets/445666-5555644-555555/1/1/1/latest/write?timeout=10000
    return /*...*/null;
  }
}
```

JAX-RS: Build HTTP response

```
@GET
public Response start1(@PathParam(UUID) String uuid,
    @PathParam(R_X_PARAM) int rX, @PathParam(R_Y_PARAM) int rY,
    @PathParam(R_Z_PARAM) int rZ, @PathParam(VERSION_PARAM)
    String version, @PathParam(MODE_PARAM) String mode,
    @QueryParam(TIMEOUT_PARAM) Long timeout) {

  log.debugv("start: timeout = {}", timeout);
  Response resp = checkVersionUuidTS.run(uuid, version);
  if (resp != null) {
    return resp;
  }
  return Response.temporaryRedirect(URI.create("/"
    + uuid + "/" + rX + "/" + rY + "/" + rZ + "/" + version)).build();
}
```

JAX-RS: Structured data in API

```
//JSON data are sent in POST request
@POST
@Path("datasets/")
@Consumes(MediaType.APPLICATION_JSON)
public Response
createEmptyDataset(DatasetDTO dataset)
{
```

```
// JSON data are sent in GET response
@GET
@Path("datasets/")
@Produces(MediaType.APPLICATION_JSON)
public DatasetDTO
createDataset(/*...*/) {
```

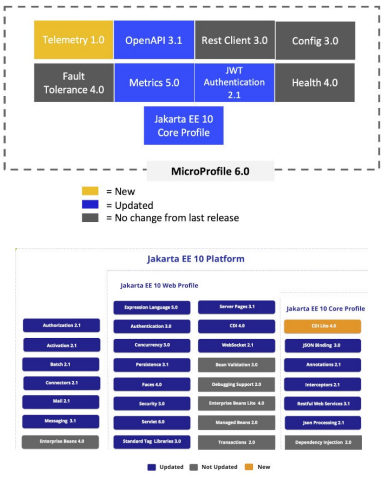
JSON Binding API (JSR 367)

```
public class Person2 {
  private int id;
  @JsonbProperty("person-name")
  private String name;
  @JsonbNillable
  private String email;
  @JsonbTransient
  private int age;
  @JsonbDateFormat("dd-MM-yyyy")
  private LocalDate registeredDate;
  private BigDecimal salary;
  @JsonbNumberFormat(locale =
"en_US",
    value = "#0.0")
  public BigDecimal getSalary() {
    return salary;
  }
```

- **@JsonbProperty** – which is used for specifying a custom field name
- **@JsonbTransient** – when we want to ignore the field during deserialization/serialization
- **@JsonbDateFormat** – when we want to define the display format of the date
- **@JsonbNumberFormat** – for specifying the display format for numeric values
- **@JsonbNillable** – for enabling serialization of null values

Quarkus

- https://quarkus.io
- MicroProfile – optimize J2EE to Microservices
  - JAX-RS, JAXB, CDI
- Supersonic Subatomic Java
- Full-stack Framework
- OpenJDK HotSpot, GraalVM
- Microservices
- Small footprint
- Reduced boot time



Quarkus: Simplified Hibernate ORM with Panache

- Makes mapping simple
- Active record.
- Repository.
- Advanced queries.
- Transactions.
- Lock management.
- Custom IDs
- Mocking

Panache: Active Record example

```java
@Entity
public class Person extends PanacheEntity {
  private String name;
  private LocalDate birth;
  private Status status;

  public enum Status {
    Alive, Deceased
  }
}
```

```java
    // creating a person
    Person person = Person.builder()
      .name("Ada Lovelace").birth(
      LocalDate.of(1815, Month.DECEMBER, 10))
      .status(Status.Deceased).build();

    // persist it
    person.persist();

    // note that once persisted, you don't need
to explicitly save your entity: all
    // modifications are automatically persisted
on transaction commit.

    // check if it's persistent
    if (person.isPersistent()) {
      // delete it
      person.delete();
    }

    // getting a list of all Person entities
    List<Person> allPersons = Person.listAll();
```

Panache: Active Record example

```
// finding a specific person by ID
person = Person.findById(personId);

// finding a specific person by ID
via an Optional
Optional<Person> optional = Person
  .findByIdOptional(personId);
person = optional.orElseThrow(
  NotFoundException::new);

// finding all living persons
List<Person> livingPersons = Person.list(
  "status", Status.Alive);

// counting all persons
long countAll = Person.count();

// counting all living persons
long countAlive = Person.count(
  "status", Status.Alive);
```

```
// delete all living persons
Person.delete(
  "status", Status.Alive);

// delete all persons
Person.deleteAll();

// delete by id
boolean deleted = Person
  .deleteById(personId);

// set the name of all living persons to
'Mortal'
Person.update(
  "name = 'Mortal' where status = ?1"
  , Status.Alive);
```

Panache: Active Record example II

```
//All list methods have equivalent stream
versions.
try (Stream<Person> persons =
  Person.streamAll()) {
    List<String> namesButEmmanuels =
    persons.map(p -> p.getName()
     .toLowerCase())
     .filter(n -> !"emmanuel".equals(n))
     .toList();
}
```

```
@Entity
public class Person extends PanacheEntity {
  private String name;
  private LocalDate birth;
  private Status status;

  public static Person findByName(String name) {
    return find("name", name).firstResult();
  }

  public static List<Person> findAlive() {
    return list("status", Status.Alive);
  }

  public static void deleteStefs() {
    delete("name", "Stef");
  }

  public enum Status {
    Alive, Deceased
  }
```

Panache: Repository pattern

```
@ApplicationScoped
public class PersonRepository implements
  PanacheRepository<Person> {

// put your custom logic here as instance
methods

  public Person findByName(String name) {
    return find("name"
     , name).firstResult();
  }

  public List<Person> findAlive() {
    return list("status", Status.Alive);
  }

  public void deleteStefs() {
    delete("name", "Stef");
  }
}
```

```
@Inject
private PersonRepository
  personRepository;

@GET
public long count() {
  return personRepository.count();
}
```

Quarkus: Getting Started

```
mvn io.quarkus:quarkus-maven-
plugin:1.13.0.Final:create
  -DprojectGroupId=vsb.java2.koz01
  -DprojectArtifactId=
    rest-getting-started
  -DclassName=
    "vsb.java2.rest.GreetingResource"
  -Dpath="/hello"
```

```
@Path("/hello")
public class GreetingResource {

  @GET
  @Produces(MediaType.TEXT_PLAIN)
  public String hello() {
    return "Hello RESTEasy";
  }
}
```

https://code.quarkus.io/

Add extension:

- Hibernate ORM with Panache [quarkus-hibernate-orm-panache]
- RESTEasy Classic [quarkus-resteasy]
- RESTEasy Classic JSON-B [quarkus-resteasy-jsonb]
- JDBC Driver - H2 [quarkus-jdbc-h2]
- YAML Configuration [quarkus-config-yaml]

Quarkus: Package and run application

- **./mvnw package**
  - rest-getting-started-1.0.0-SNAPSHOT.jar
  - quarkus-run.jar + quarkus-app/lib/
- **java -jar target/quarkus-app/quarkus-run.jar**
- **./mvnw quarkus:dev**
- **./mvnw compile quarkus:dev**
- **Using main method**

Quarkus: run application + config

```java
@QuarkusMain
public class AppMain {
  public static void main(
    String[] args) {
    Quarkus.run(args);
  }
}
```

```yaml
greeting:
  message: "hello"
quarkus:
  test:
    continuous-testing: disabled
  console:
    enabled: false
  datasource:
    username: app
    password: app
    jdbc:
      url:
jdbc:h2:file:./db/java2
  hibernate-orm:
    database:
      generation: create
```

Quarkus: Native application

- With GrallVM installed
  - ./mvnw package -Pnative.
- Linux executable with docker installed
  - /mvnw package -Pnative -Dquarkus.native.container-build=true
- Creating docker container
  - ./mvnw package -Pnative -Dquarkus.native.container-build=true -Dquarkus.container-image.build=true
-

REST client in Java: Libraries and Frameworks

- Apache CXF
- Jersey
- Spring RestTemplate
- Commons HTTP Client
- Apache HTTP Components (4.2) Fluent adapter
- OkHttp
- Ning Async-http-client
- Feign
- Retrofit
- Volley
- google-http
- Unirest
- Resteasy JakartaEE
- jcabi-http
- restlet
- rest-assured

REST client in Java - OpenAPI - Swagger

- Generate OpenAPI descriptino YAML

```
<dependency>
  <groupId>io.quarkus</groupId>
  <artifactId>Quarkus-smallrye-openapi</artifactId>
</dependency>
```

- http://localhost:8080/q/openapi

REST client in Java - OpenAPI - Swagger

- Generate client – pom.xml
- No modules – delete module-info.java
- Depencencies:

```
<!-- Swagger dependencies BEGIN ********************************-->
<dependency>
<groupId>io.swagger.codegen.v3</groupId>
<artifactId>swagger-codegen-maven-plugin</artifactId>
<version>3.0.52</version>
</dependency>
<dependency>
<groupId>com.google.code.gson</groupId>
<artifactId>gson</artifactId>
<version>2.10.1</version>
</dependency>
<dependency>
<groupId>io.gsonfire</groupId>
<artifactId>gson-fire</artifactId>
<version>1.9.0</version>
</dependency>
<dependency>
<groupId>com.squareup.okhttp3</groupId>
<artifactId>okhttp</artifactId>
<version>4.12.0</version>
</dependency>
<dependency>
<groupId>com.squareup.okhttp3</groupId>
<artifactId>logging-interceptor</artifactId>
<version>4.12.0</version>
</dependency>
<!-- Swagger dependencies END** ********************************-->
```

- Pom.xml – build – blugins

```
<plugin>
  <groupId>org.openapitools</groupId>
  <artifactId>
    Openapi-generator-maven-plugin
  </artifactId>
  <!-- RELEASE_VERSION -->
  <version>7.2.0</version>
  <!-- /RELEASE_VERSION -->
  <executions>
    <execution>
      <goals>
        <goal>generate</goal>
      </goals>
      <configuration>
        <inputSpec>
          http://localhost:8080/q/openapi
        </inputSpec>
        <generatorName>java</generatorName>
      </configuration>
    </execution>
  </executions>
</plugin>
```

Apache CXF

- JAX-RS 2.0 Client API
- Proxy-based API
- CXF WebClient API

```
<dependency>
<groupId>org.apache.cxf</groupId>
<artifactId>cxf-rt-rs-client</artifactId>
<version>3.0.15</version>
</dependency>
```

Apache CFX

```
@Path("/bookstore")
public interface BookStore {
  @GET
  Books getAllBooks();

  @Path("{id}")
  BookResource getBookSubresource(
    @PathParam("id") long id)
    throws NoBookFoundException;
}
```

```
public interface BookResource {
  @GET
  Book getBook();
}
```

```
BookStore store = JAXRSClientFactory.create("http://bookstore.com", BookStore.class);

// (1) remote GET call to http://bookstore.com/bookstore
Books books = store.getAllBooks();

// (2) no remote call
BookResource subresource = store.getBookSubresource(1);

// {3} remote GET call to http://bookstore.com/bookstore/1
Book b = subresource.getBook();
```

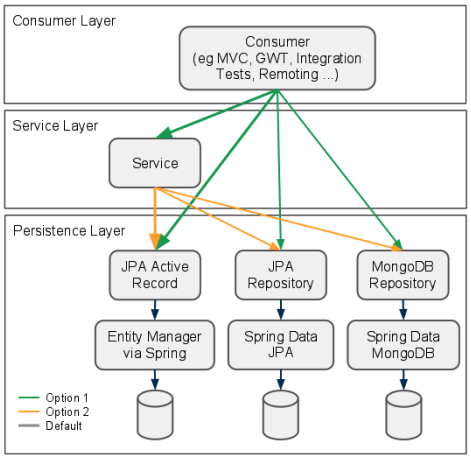
Spring.io

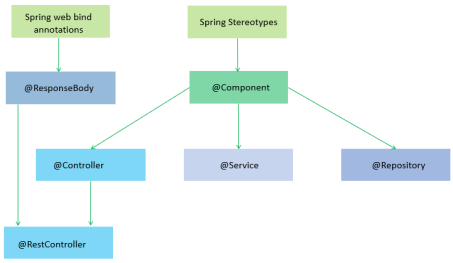
- The Spring Framework provides a comprehensive programming and configuration model for modern Java-based enterprise applications - on any kind of deployment platform.
- Container for CDI
- Framework for WEB application – not only
- Many extensions

Spring.io - starting

- https://start.spring.io/ - create maven project for you
  - Spring WEB
  - Spring Data JPA
  - H2 Database

Spring - @Component





Spring.io – running app

```
@SpringBootApplication
public class Java2Lect07Application {

  public static void main(String[] args) {
    SpringApplication.run(Java2Lect07Application.class, args);
  }

}
```

Spring.io - Rest Controller

```
@RestController
@RequestMapping(path = "api")
public class RestApiController {
  @GetMapping("/")
  public String hello() {
    return "Hi!";
  }
  @GetMapping({ "/persons", "/persons/" })
  public List<Person> getAllPersosn() {
    return Arrays.asList(new Person("David", 10), new Person("Jan", -98));
  }
  @PostMapping("/persons")
  public Person savePersosn(@RequestBody Person person) {
    System.out.println(person);
    return person;
  }
}
```

Spring.io - JPA

```
public interface PersonRepository extends JpaRepository<Person,
Long>, CustomPersonRepository{
}


public class RestApiController {

  @Autowired
  PersonRepository personRepository;

  @GetMapping({ "/persons", "/persons/" })
  public List<Person> getAllPersosn() {
    return personRepository.findAll();
```

CDI

- Framework/container need bean
- Found class with proper type (same, descendant, implementation)
- Create instance – constructor with no parameters
- JVM run class constructor
- Framework/container inspect bean and inject autowired/injected properties
- Framework/container run method anotated with @PostConstruct
- Framework/container use bean

CDI

- Framework/container need bean
- Found class with proper type (same, descendant, implementation)
- Create instance – constructor with parameters - autowiring
- JVM run constructor
- ~~Framework/container inspect bean and inject autowired/injected properties~~
- Framework/container run method anotated with @PostConstruct
- Framework/container use bean

CDI bean - example

```java
@Service
public class PersonBL {

  @Autowired
  protected PersonRepository
    personRepository;

  public Person save(
    Person entity) {
    return personRepository
     .save2(entity);
  }
```

```java
@Service
public class PersonBL2 {

  protected PersonRepository
    personRepository;

  public PersonBL2(
    PersonRepository
      personRepository) {
    this.personRepository =
      personRepository;
  }

  public Person save(
      Person entity) {
    return personRepository
      .save2(entity);
  }
```

CDI magic

- Framework/container automatically create transactions, take care of bean life-cycle, take care about security
- How?
- CDI bean are injected into framework.
- You can inject CDI been to your classes (CDI bean).
- You probably do not get the object of class you want (inject).
  - You get object of descendant class which is generated automatically.
  - All method are overridden and do “The magic“ you want (annotated) – transactions, security, lifecycle

6th Lecture

- Internationalization
- BigInteger, BigDecimal
- Concurrency
  - Lock objects
  - Executors
  - Concurrent collections
  - Atomic variables
  - ThreadLocalRandom
  - CompletableFuture

Internationalization

- Support for different character sets and for universal (UTF-8, UTF-16)
- Support for specific settings: format (number, date, ….), currencies, texts and other resources (multimedia, data)
- Locale is a class that identifies a combination of language and region:
  - Locale(String language)
  - Locale(String language, String country)
  - Locale czechLocale = new Locale(“cs”,”CZ”)

String localization

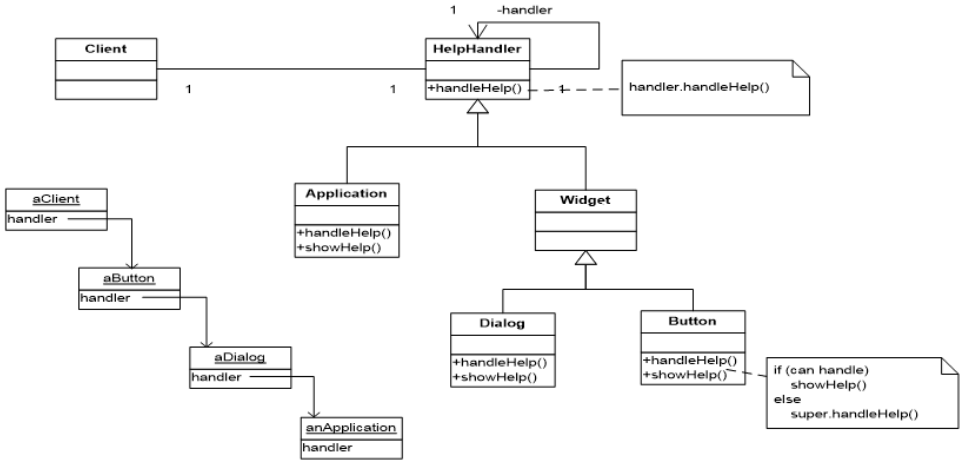
```
ResourceBundle bundle = ResourceBundle.getBundle("MessageBundle", locale);
System.out.println("" + bundle.getString("greetings"));
```

- Files in resources directory:
  - MessageBundle.properties
    greetings = Hello
  - MessageBundle_de.properties
    greetings = Hallo
  - MessageBundle_fr.properties
    greetings = Bonjour
  - MessageBundle_it.properties
    greetings = Ciao
- For German, French and Italian is used given text and default for other (Hello).

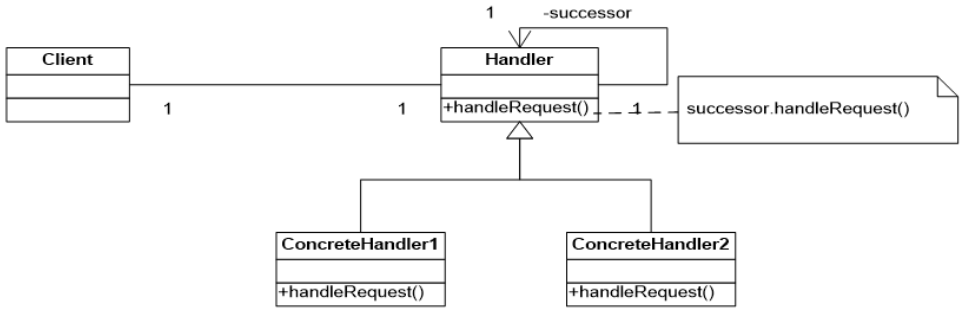
Chain of Responsibility Design Pattern

- It avoids coupling senders of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.

Chain of Responsibility Example



Chain of Responsibility Design Pattern



Chain of Responsibility – resource bundles

- ExampleResource
- ExampleResource_en
- ExampleResource_en_US
- ExampleResource_en_US_UNIX

```
Locale locale = Locale.of("en", "en_US");
ResourceBundle exampleBundle = ResourceBundle.getBundle(
  "package.ExampleResource", locale);
```

Formatting (with predefined format)

- Numbers

```
NumberFormat nf = NumberFormat.getNumberInstance(Locale.FRANCE);
String valueStr = nf.format(Math.PI);
System.out.println(valueStr);
//------------------------
//Output is: 3,142
```

- Currency

```
Locale locale_enGB = locale.UK;
Currency currency = Currency.getInstance(locale_enGB);
NumberFormat currencyFormat = NumberFormat.getCurrencyInstance(locale_enGB);
System.out.println(currency.getDisplayName() + ": " + currencyFormat.format(100.0));
//-----------------------------------------------
//Output is: British Pound: £100.00
```

- Datetime

```
DateTimeFormatter dateTimeFormat =
DateTimeFormatter.ofLocalizedDateTime(FormatStyle.FULL).withLocale(Locale.of("cs", "CZ"));
System.out.println(dateTimeFormat.format(ZonedDateTime.now()));
//-------------------------------------------------------------------------------------------------------------
//Output is:čtvrtek 21. listopadu 2019 15:33:34 Středoevropský standardní čas
```

Big values

- **Double** does not have unlimited precision

```
double val = 0.1;
for (int i = 0; i < 10; i++) {
  val += 0.1;
}
System.out.printf("val = " + val);
//--- output ---------------------
//val = 1.0999999999999999
```

BigDecimal and BigInteger

- https://www.baeldung.com/java-bigdecimal-biginteger

BigDecimal

- BigDecimal represents an immutable arbitrary-precision signed decimal number.
  - Unscaled value
  - Scale (32 bit)
- High-precision arithmetic
- Variety constructors(String, character array, int, long, and BigInteger) and factory method valueOf (double, long)

BigDecimal operations

- Arithmetic operation - **add**, **subtract**, **multiply**, **divide**, ...
- Relational operation – compareTo, equals (compares also scale)
- Functions – **abs**, **pow**, **sqrt**,..
- Various attributes – **precision**, **scale**, **sign**
- Rounding – 8 modes

BigInteger

- immutable arbitrary-precision integers
- used when integers involved are larger than the limit of long type
- Constructor (String, byte array) and valueOf (long)

BigInteger operations

- Similar to int and long but cannot overflow
- Arithmetic, bitwise, - as methods
- Bit manipulation methods
- GCD, modular arithmetic, prime generation, primality testing,

Money in Java

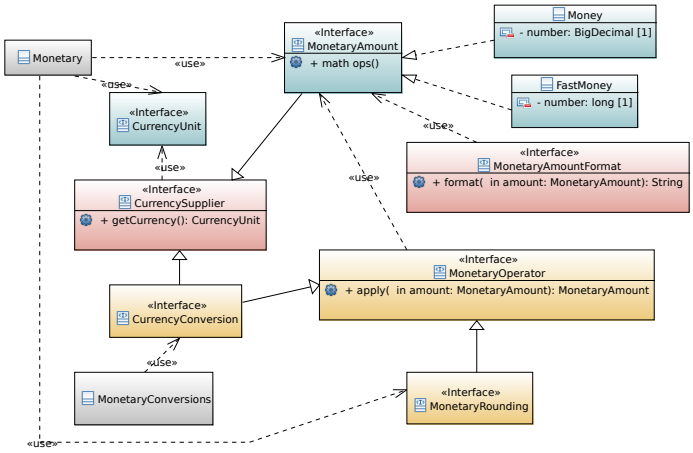
“A large proportion of the computers in this world manipulate money, so it's always puzzled me that money isn't actually a first class data type in any mainstream programming language. “

Martin Fowler

Money in Java - standards

- Joda money
- JSR 354
  - To provide an API for handling and calculating monetary amounts
  - To define classes representing currencies and monetary amounts, as well as monetary rounding
  - To deal with currency exchange rates
  - To deal with formatting and parsing of currencies and monetary amounts

JSR 354 - model



JSR 354 - CurrencyUnit

```
@Test
void givenCurrencyCode_whenString_thanExist() {
  CurrencyUnit usd = Monetary.getCurrency("USD");

  assertNotNull(usd);
  assertEquals(usd.getCurrencyCode(), "USD");
  assertEquals(usd.getNumericCode(), 840);
  assertEquals(usd.getDefaultFractionDigits(), 2);
}

@Test
void givenCurrencyCode_whenNoExist_thanThrowsError() {
  UnknownCurrencyException thrown = Assertions.assertThrows(
    UnknownCurrencyException.class, () -> {
      Monetary.getCurrency("AAA");
    });
  assertEquals("Unknown currency code: AAA", thrown.getMessage());
}
```

JSR 354 - MonetaryAmount

```
@Test
void givenAmounts_whenStringified_thanEquals() {
  CurrencyUnit usd = Monetary.getCurrency("USD");
  MonetaryAmount fstAmtUSD = Monetary
    .getDefaultAmountFactory().setCurrency(usd)
    .setNumber(200).create();
  Money moneyof = Money.of(12, usd);
  FastMoney fastmoneyof = FastMoney.of(2, usd);

  assertEquals("USD", usd.toString());
  assertEquals("USD 200", fstAmtUSD.toString());
  assertEquals("USD 12", moneyof.toString());
  assertEquals("USD 2", fastmoneyof.toString());
}
```

JSR 354 – Monetary Arithmetic

```
@Test
void givenCurrencies_whenCompared_thanNotequal() {
  MonetaryAmount oneDolar = Monetary.getDefaultAmountFactory()
    .setCurrency("USD").setNumber(1).create();
  Money oneEuro = Money.of(1, "EUR");
  assertFalse(oneEuro.equals(FastMoney.of(1, "EUR")));
  assertTrue(oneDolar.equals(Money.of(1, "USD")));
}
@Test
void givenAmounts_whenSummed_thanCorrect() {
  MonetaryAmount[] monetaryAmounts = new MonetaryAmount[] {
    Money.of(100, "CHF"), Money.of(10.20, "CHF")
    , Money.of(1.15, "CHF") };
  Money sumAmtCHF = Money.of(0, "CHF");
  for (MonetaryAmount monetaryAmount : monetaryAmounts) {
    sumAmtCHF = sumAmtCHF.add(monetaryAmount);
  }
  assertEquals("CHF 111.35", sumAmtCHF.toString());
}
```

JSR 354 – Monetary Rounding

```
@Test
void givenAmount_whenRounded_thanEquals() {
  MonetaryAmount fstAmtEUR = Monetary
    .getDefaultAmountFactory().setCurrency("EUR")
    .setNumber(1.30473908).create();
  MonetaryAmount roundEUR = fstAmtEUR
    .with(Monetary.getDefaultRounding());

  assertEquals("EUR 1.30473908", fstAmtEUR.toString());
  assertEquals("EUR 1.3", roundEUR.toString());
}
```

JSR 354 – Currency Conversion

```
@Test
void givenAmount_whenConversion_thenNotNull() {
  MonetaryAmount oneDollar = Monetary
    .getDefaultAmountFactory().setCurrency("USD")
    .setNumber(1).create();

  CurrencyConversion conversionEUR = MonetaryConversions
    .getConversion("EUR");
  MonetaryAmount convertedAmountUSDtoEUR = oneDollar
    .with(conversionEUR);

  assertEquals("USD 1", oneDollar.toString());
  assertNotNull(convertedAmountUSDtoEUR);
}
```

JSR 354 - Formatting

```
@Test
void givenLocale_whenFormatted_thanEquals() {
  MonetaryAmount oneDollar = Monetary
    .getDefaultAmountFactory().setCurrency("USD")
    .setNumber(1).create();

  MonetaryAmountFormat formatUSD = MonetaryFormats
    .getAmountFormat(Locale.US);
  String usFormatted = formatUSD.format(oneDollar);

  assertEquals("USD 1", oneDollar.toString());
  assertNotNull(formatUSD);
  assertEquals("USD1.00", usFormatted);
}
```

JSR 354 – Formatting II

```
@Test
void givenAmount_whenCustomFormat_thanEquals() {
  MonetaryAmount oneDollar = Monetary
    .getDefaultAmountFactory().setCurrency("USD")
    .setNumber(1).create();

  MonetaryAmountFormat customFormat = MonetaryFormats
    .getAmountFormat(AmountFormatQueryBuilder
      .of(Locale.US).set(CurrencyStyle.NAME)
      .set("pattern", "00000.00 ¤").build());
  String customFormatted = customFormat.format(oneDollar);

  assertNotNull(customFormat);
  assertEquals("USD 1", oneDollar.toString());
  assertEquals("00001.00 US Dollar", customFormatted);
}
```

Concurrency - references

- https://docs.oracle.com/javase/tutorial/essential/concurrency/forkjoin.html

Lock object

- Similar to mechanism of synchronized sections
- Can back out of and attempt to acquire lock
  - *tryLock()*

Executors

- Executor interfaces
- Thread Pool
- Fork/Join

Executors – Executor Interfaces

- **Executor** - a simple interface that supports launching new tasks.
- **ExecutorService** - a subinterface of Executor, which adds features that help manage the lifecycle, both of the individual tasks and of the executor itself.
-
- **ScheduledExecutorService** - a subinterface of ExecutorService, supports future and/or periodic execution of tasks.
-

Executor

- Behavior depends on implementation.

```
(new Thread(r)).start();

// --------------------------

e.execute(r);
```

ExecutorService

- extends Executor:
  - shutdown()
  - shutdownNow()
  - isShutdown()
  - isTerminated()
  - awaitTermination(long, TimeUnit)
  - submit(Callable<T>)
  - submit(Runnable, T)
  - submit(Runnable)
  - invokeAll(Collection<? extends Callable<T>>)
  - invokeAll(Collection<? extends Callable<T>>, long, TimeUnit)
  - invokeAny(Collection<? extends Callable<T>>)
  - invokeAny(Collection<? extends Callable<T>>, long, TimeUnit)

ScheduledExecutorService

- extension of ExecutorService
  - schedule
  - scheduleAtFixedRate
  - scheduleWithFixedDelay

Thread Pools

- Consist of worker threads – can be used to execute multiple tasks
- Minimizes the overhead due thread creation
- Factory methods in java.util.concurrent.Executors:
  - newFixedThreadPool
  - newCachedThreadPool
  - newSingleThreadExecutor
  - ScheduledExecutorService – different versions
  - newVirtualThreadPerTaskExecutor
- java.util.concurrent.ThreadPoolExecutor and java.util.concurrent.ScheduledThreadPoolExecutor

Fork/Join

- Implementation of **ExecutorService** – designed for work that could be broken into smaller pieces
- The goal is to use all the available processing power
- Distinct from a thread pool implementing work-stealing algorithm
- The main class is **ForkJoinPool**

Fork/join basic scheme usage

- if (my portion of the work is small enough)
  - do the work directly
- else
  - split my work into two pieces
- invoke the two pieces and wait for the results

Fork/join – example

```
public class ForkBlur extends
RecursiveAction {
  private int[] mSource;
  private int mStart;
  private int mLength;
  private int[] mDestination;
  // Processing window size;
  // should be odd.
  private int mBlurWidth = 15;

  public ForkBlur(int[] src,
    int start, int length
    , int[] dst) {
      mSource = src;
      mStart = start;
      mLength = length;
      mDestination = dst;
  }
  protected void computeDirectly() {
//…
  }
  protected static int
    sThreshold = 100000;

  protected void compute() {
    if (mLength < sThreshold) {
      computeDirectly();
      return;
    }

    int split = mLength / 2;

    invokeAll(new ForkBlur(
      mSource, mStart, split
      , mDestination),
    new ForkBlur(mSource, mStart
      + split, mLength – split
      , mDestination));
  }
}
```

Fork/join – run example

1) Create a task that represents all of the work to be done.

```
// source image pixels are in src
// destination image pixels are in dst
ForkBlur fb = new ForkBlur(src, 0, src.length, dst);
```

2) Create the ForkJoinPool that will run the task

```
ForkJoinPool pool = new ForkJoinPool();
```

3) Run the task.

```
pool.invoke(fb);
```

Concurrent Collections

- **BlockingQueue** – blocks or times out when full/empty during addition/retrieving
- **ConcurentMap** – interface that defines useful atomic operation
- **ConcurentNavigableMap** – extends ConcurrentMap

Atomic Variables

- Provides thread-safe operation for variable holding and modification
- AtomicInteger, AtomicLong, …
- **volatile** – key word for Memory Visibility and Order beetween thread
  - To ensure that updates to variables propagate predictably to other threads, we should apply the volatile modifier to those variables.

ThreadLocalRandom

- For concurrent access its using provides better performance

```
int r = ThreadLocalRandom.current().nextInt(4, 77);
```

CompletableFuture

- https://www.callicoder.com/java-8-completablefuture-tutorial/

CompletableFuture - description

- extends **Future** – methods **done**, **isDone** – with:
  - can be manually completed,
  - manually perform further action on a CompletableFuture's result without blocking,
  - multiple CompletableFutures can be chained together,
  - multiple CompletableFutures can be combined together,
  - exception handling.

Creating CompletableFuture with constructor

```
CompletableFuture<String> completableFuture =
  new CompletableFuture<String>();


String result = completableFuture.get();


completableFuture.complete("Future's Result");
```

Factory methods in CompletableFuture - runAsync

```
//Run a task specified by a Runnable Object asynchronously.
CompletableFuture<Void> future = CompletableFuture.runAsync(
  new Runnable() {
  @Override
  public void run() {
    // Simulate a long-running Job
    try {
      TimeUnit.SECONDS.sleep(1);
    } catch (InterruptedException e) {
      throw new IllegalStateException(e);
    }
    System.out.println(
      "I'll run in a separate thread than the main thread.");
  }
});

//Block and wait for the future to complete
future.get();
```

Factory methods in CompletableFuture - supplyAsync

```
// Run a task specified by a Supplier object asynchronously
CompletableFuture<String> future = CompletableFuture.supplyAsync(
  new Supplier<String>() {
  @Override
  public String get() {
    try {
      TimeUnit.SECONDS.sleep(1);
    } catch (InterruptedException e) {
      throw new IllegalStateException(e);
    }
    return "Result of the asynchronous computation";
  }
});

// Block and get the result of the Future
String result = future.get();
System.out.println(result);
```

CompletableFuture – use Executor

```
//Variations of runAsync() and supplyAsync() methods
public static CompletableFuture<Void>
runAsync(Runnable runnable)
public static CompletableFuture<Void>
runAsync(Runnable runnable, Executor executor)
public static <U> CompletableFuture<U>
supplyAsync(Supplier<U> supplier)
public static <U> CompletableFuture<U>
supplyAsync(Supplier<U> supplier, Executor executor)
```

Transforming and acting on a CompletableFuture

- thenApply, thenAccept, thenRun

```
//Create a CompletableFuture
CompletableFuture<String> whatsYourNameFuture = CompletableFuture
  .supplyAsync(() -> {
  try {
    TimeUnit.SECONDS.sleep(1);
  } catch (InterruptedException e) {
    throw new IllegalStateException(e);
  }
  return "David";
});
//Attach a callback to the Future using thenApply()
CompletableFuture<String> greetingFuture =
  whatsYourNameFuture.thenApply(name -> {
  return "Hello " + name;
});

//Block and get the result of the future.
System.out.println(greetingFuture.get()); // Hello David
```

Combine two dependent futures using thenCompose()

```
public CompletableFuture<User> getUserDetail(
  String userId) {
  return CompletableFuture.supplyAsync(() -> {
    return UserService.getUserDetails(userId);
  });
}
public CompletableFuture<Double> getCreditRating(
  User user) {
  return CompletableFuture.supplyAsync(() -> {
    return CreditRatingService.getCreditRating(user);
  });
}

CompletableFuture<Double> result =
  getUserDetail(userId).thenCompose(
   user -> getCreditRating(user));
```

Combine two independent futures using thenCombine()

```
System.out.println("Retrieving weight.");
CompletableFuture<Double> weightInKgFuture =
CompletableFuture.supplyAsync(() -> {
  try {
    TimeUnit.SECONDS.sleep(1);
  } catch (InterruptedException e) {
    throw new IllegalStateException(e);
  }
  return 65.0;
});

System.out.println("Retrieving height.");
CompletableFuture<Double> heightInCmFuture =
CompletableFuture.supplyAsync(() -> {
  try {
    TimeUnit.SECONDS.sleep(1);
  } catch (InterruptedException e) {
    throw new IllegalStateException(e);
  }
  return 177.8;
});
```

```
System.out.println("Calculating BMI.");
CompletableFuture<Double> combinedFuture =
weightInKgFuture.thenCombine(heightInCmFuture,
    (weightInKg, heightInCm) -> {
      Double heightInMeter = heightInCm / 100;
      return weightInKg / (heightInMeter *
heightInMeter);
    });

System.out.println("Your BMI is - " +
combinedFuture.get());
```

Combining multiple CompletableFutures together

```
public static CompletableFuture<Void>
allOf(CompletableFuture<?>... cfs)
```

```
public static CompletableFuture<Object>
anyOf(CompletableFuture<?>... cfs)
```

Handle exceptions using exceptionally() callback

```
Integer age = -1;

CompletableFuture<String> maturityFuture =
  CompletableFuture.supplyAsync(() -> {
  if (age < 0) {
    throw new IllegalArgumentException("Age can not be negative");
  }
  if (age > 18) {
    return "Adult";
  } else {
    return "Child";
  }
}).exceptionally(ex -> {
  System.out.println("Oops! We have an exception - " + ex.getMessage());
  return "Unknown!";
});

System.out.println("Maturity : " + maturityFuture.get());
```

Handle exceptions using the generic handle() method

```
Integer age = -1;
CompletableFuture<String> maturityFuture =
  CompletableFuture.supplyAsync(() -> {
  if (age < 0) {
    throw new IllegalArgumentException("Age can not be negative");
  }
  if (age > 18) {
    return "Adult";
  } else {
    return "Child";
  }
}).handle((res, ex) -> {
  if (ex != null) {
    System.out.println("Oops! We have an exception - " + ex.getMessage());
    return "Unknown!";
  }
  return res;
});
System.out.println("Maturity : " + maturityFuture.get());
```

7th lecture

- Java NIO
- Serialization