Spring 4, Cassandra and Tuning

AutoDesk

Class Outline
Day One Spring

- New Features in Spring 4 & 5
  - New features in Spring 4
  - New features in Spring 5
- Migrating from Spring 3 to Spring 4
- Migrating from Spring 4 to Spring 5
- Dependency Injection using JSR-330
- Spring Boot
  - Micro-Services
  - Services in the Cloud
- Spring REST
- Spring Batch

Day Two: Dependency Injection and Cassandra

- Dependency injection
- Cassandra Vs. Hibernate
- Cassandra Configuration
- Cassandra Connections
- Cassandra Repository
- Cassandra Entities
- Data Access with Cassandra
- Embedded Cassandra server for testing

Day Three: Cassandra and Java Tuning Concepts

- Intro to Java Tuning Concepts
- Object Creation
- Queries and Collections
- Thread Tuning
- Concurrency and Locks
- Cassandra Tuning Concepts
- Fetch Type Performance

Introductions

- Bio’s
- Expectations
- Schedule
- Labs
- Questions

New Features in Spring 4 & 5

New features in Spring 4.0

- Any packages. Classes and methods that were deprecated in Spring 3 have been removed from Spring 4
- When upgrading be sure to check and fix all deprecated calls.
- It's easy to use Eclipse to find deprecations:

  Preferences -> Java -> Compiler -> Errors/Warnings -> Deprecated and restricted API section

3rd Party Platforms

- Spring 4 increased requirement levels for some third party platforms, notable requirements include:
- Hibernate Validator 4.3+
- Jackson 2.0+
- Jackson 1.8/1.9 support retained for the time because it was not deprecated until Spring 3.2

Spring 4 Supports Most of Java 8

- Java 8 supported features:
- Lamda functions using Springs callback interface
- Java.time (JSR-310)
- @Repeatable annotation allowing the same annotation to be used multiple times
- The –parameters compiler flag will allow allow parameters types to be retrieved using reflection
- Spring 4 compiled to be compatible with Java 6

-source 1.6 -target 1.6

- If a class uses a Java 8 language feature such as a lambda expression, it has to be compiled with -source 1.8 -target 1.6
- The compilation unit will only work on Java 8+
- If a class in a library uses a new Java 8 interface such as java.util.stream.Stream, the library can still run on a previous Java generation as long as it is being compiled with e.g. -source 1.6 -target 1.6
- The use of that particular Stream-based class is guarded to only kick in when actually running on Java 8+

Compatibility with Older Java Versions

- Spring 4 is compatible with Java versions as far back as Java SE 6 update 18 which was released in January of 2010
- We recommend using Java 7 or 8 for new development projects based on Spring 4

Application Servers and Google's App Engine

- it is possible to deploy a Spring 4 application into a Servlet 2.5 environment
- It is strongly recommended that Servlet 3.0+ is used as a Servlet environment for Spring 4 projects
- Spring 4 test and mock packages for test setups requires the 3.0 specification

Java EE 7 Specification Levels

- Spring 4 supports the following Java 7 EE levels of specifications
  - JMS 2.0
  - JTA 1.2
  - JPA 2.1
  - Bean Validation 1.1
  - JSR-236 Concurrency Utilities

Spring Grails and Domain Specific Languages (DSL)

- Spring Framework 4.0 has Grails like native support for a beans{} “DSL”
- Bean definitions can be embedded in Groovy application scripts with the same format
- This is similar to using XML bean definitions but is more concise
- Using Groovy easily allows bean definitions in bootstrap code

DSL Bean Creation Example

```
def reader = new GroovyBeanDefinitionReader(myAppContext)
reader.beans {
    dataSource(BasicDataSource) {
        driverClassName = "com.datastax.cassandra"
        url = "jdbc:myappdb:mem:grailsDB"
        username = ”accessUser"
        password = ”password"
        settings = [mycassandra:"setting"]
    }
    sessionFactory(SessionFactory) {
        dataSource = dataSource
    }
    myService(MyService) {
        nestedBean = { AnotherBean bean ->
            dataSource = dataSource
        }}}
```

Core Container Improvements

- Autowire pays attention to generic types, Spring now treats generic types as a form of qualifier when injecting Beans
- Using a Spring Data Repository you can now easily inject a specific implementation:
  - @Autowired Repository<Customer> customerRepository
- Spring's meta-annotation now allows custom annotations that can amalgamate multiple Spring annotations

Meta Annotations

- A meta annotation is an annotation that can be applied to another annotation
- Custom annotations can be defined to combine many Spring annotations into a single annotation.

```
@Repository
@Scope("prototype")
@Transactional(propagation =
Propagation.REQUIRES_NEW, timeout = 30,
isolation=Isolation.SERIALIZABLE)
public @interface DefaultDao {}
```

Using a Meta Annotation

- To use the new annotation comprised of multiple Spring annotations simply use the custom annotation:

```
@DefaultDao
public class OrderDaoImpl implements OrderDao
{
  ...
}
```

Defining Sort Order

- @Order tells the highest precedence advice to run first.
- The lower the number, the higher the precedence.
- Given two pieces of 'before' advice, the one with highest precedence will run first

```
@Component
@Order(2)
class VIPCustomer extends Customer{
    public String getName() {
        return ”John Smith”;}}

@Component
@Order(1)
class IPCustomer extends Customer{
    public String getName() {
        return ”John Doe”;}}
```

@Lazy Loading

- As of Spring 4.0 @Lazy can be applied on injection points, such as

  @Autowire or @Inject.

```
@Component(value = ”lazyCustomerType")
@Scope(value = "prototype")
public class CustomerType{..}

@Component(value = "lazyCustomer")
public class Customer{
  @Autowired
  @Lazy
  private CustomerType vipCustomer;

  @Autowired
  @Lazy
  private CustomerTye ipCustomer;
}
```

@Description and @Conditional

- @Description is a new for providing a textual description of the bean which can be useful for monitoring purpose
- @Conditional annotation allows Developers to define user-defined strategies for conditional checking
- Unlike @Profile, @Conditional is not limited to environmental variable and can be used for conditional bean registrations

Web Improvements

- Spring 4.0 is now focused primarily on Servlet 3.0+ environments
- If you are using the Spring MVC Test Framework you will need to ensure that a Servlet 3.0 compatible JAR is in your test classpath

@RestController

- You can now use the new @RestController annotation with Spring MVC applications
- There is no more need to add @ResponseBody to each of your @RequestMapping methods

AsyncRestTemplate

- AsyncRestTemplate accesses the URL and return the output asynchronously
- Output is in the form of ListenableFuture that has get() method to get the result
- Spring 4 provides AsyncRequestCallback class to prepare request object

AsyncRestTemplate Example

```
AsyncRestTemplate asycTemplate = new AsyncRestTemplate();
String url ="http://google.com";
HttpMethod method = HttpMethod.GET;
Class<String> responseType = String.class;
//create request entity using HttpHeaders
HttpHeaders headers = new HttpHeaders();
headers.setContentType(MediaType.TEXT_PLAIN);
HttpEntity<String> requestEntity = new
        HttpEntity<String>("params", headers);
ListenableFuture<ResponseEntity<String>> future =
        asycTemp.exchange(url, method, requestEntity, responseType);
ResponseEntity<String> entity = future.get();

System.out.println(entity.getBody());
```

WebSocket and SockJS,

- A new spring-websocket module provides comprehensive support for WebSocket-based, two-way communication between client and server in web applications
- Spring-websocket is compatible with JSR-356, the Java WebSocket API, and provides SockJS-based fallback options in the form of emulation for use in browsers that don't yet support the WebSocket protocol

STOMP Messaging

- STOMP is Streaming Text Oriented Messaging Protocol
- STOMP clients communicate to a message broker which supports STOMP protocol STOMP uses different commands like connect, send, subscribe, and disconnect to communicate

@EnableWebSocketMessageBroker Annotation

- EnableWebSocketMessageBroker annotation enables a configuration class to support WebSocket
- This annotation supports message broker.

Unit and Integration Testing

- Meta annotations can be used to group most of Springs testing annotations to reduce configuration:
  - @ContextConfiguration
  - @WebAppConfiguration
  - @ContextHierarchy
  - @ActiveProfiles

Active Bean Definition Profiles

- Active bean definition profiles can now be resolved programmatically
- Implement a custom ActiveProfilesResolver and registering it via the resolver attribute of @ActiveProfiles

SocketUtils

- A new SocketUtils class has been introduced in the spring-core module which enables you to scan for free TCP and UDP server ports on localhost.
- This functionality is not specific to testing but can prove very useful when writing integration tests that require the use of sockets, for example tests that start an in-memory SMTP server, FTP server, Servlet container, etc.

Mock Web Package

- As of Spring 4.0 the set of mocks in the org.springframework.mock.web package is now based on the Servlet 3.0 API
- Some of the Servlet API mocks have been improved with easier configurability and enhancements

Let’s take a quick Break

Spring 4.2 Changes

- Core refinements and Web Capabilities
  - Core Container Improvements
  - Data Access Imptovements
  - JMS Improvements
  - Web Improvements

Core Container Improvements

- Annotations such as @Bean get detected and processed on Java 8 default methods
- This allows for composing a configuration class from interfaces with default @Bean methods

@Import

- Configuration classes may declare @Import with regular component classe
- This allows for a mix of imported configuration classes and component classes

Improved @Order

- Configuration classes may now declare an @Order value
- This allows configuration processing in a corresponding order
- Beans can be overridden by name even when they are detected through classpath scanning

@EventListener

- Any public method in a managed bean can be annotated with @EventListener to consume events
- @TransactionalEventListener provides transaction-bound event support

@AliasFor

- The new @AliasFor annotation can be used to declare a pair of aliased attributes within a single annotation or to declare an alias from one attribute in a custom composed annotation to an attribute in a meta-annotation
- Within a single annotation, @AliasFor can be declared on a pair of attributes to signal that they are interchangeable aliases for each other.
- If the annotation() attribute of @AliasFor is set to a different annotation than the one that declares it, the attribute() is interpreted as an alias for an attribute in a meta-annotation (i.e., an explicit meta-annotation attribute override)
- This enables fine-grained control over exactly which attributes are overridden within an annotation hierarchy

Collection Improvements

- Search algorithms have been revamped to easily find meta-annotations
  - Locally composed annotation are now favored over inherited
  - Overridden attributes from meta-annotations are now discoverable on:
    - Interfaces
    - abstract, bridge and interface methods
    - Classes
    - Standard methods, constructors
    - Fields

Conversion Services

- Out of the box converters are part of the DefaultConversionService for:
  - Stream
  - Charset,
  - Currency
  - TimeZone
- Converters can be added individually to any arbitrary ConversionService

Choc Full of Extras

- @NumberFormat can now be used as a meta-annotation.
- JavaMailSenderImpl has a new testConnection() method for checking connectivity to the server.
- ScheduledTaskRegistrar exposes scheduled tasks.

Pooling Aspect Oriented Programming (AOP) and Scripting

- AOP CommonsPool2TargetSources are now able to be pooled using Apache’s commons-pool2
- New class StandardScriptFactory Supports scripted beans, including:
- JavaScript
- Groovy
- Jruby
- Other JSR-223 compliant engines.

Messaging

- The JmsListenerContainerFactory can now control the autoStartup attribute
- Each listener container can now configure the reply Destination.
- @SendTo annotations can now use a SpEL expression
- JmsResponse can now compute the response destination at runtime.
- @JmsListener is now a repeatable annotation to declare several JMS containers on the same method

Web Improvements

- Annotation detection on Java 8 default methods
- Annotation-based application events (@EventListener)
- Full nested path processing for direct field binding
- Standards-based bean scripting via JSR-223 (JRuby, JavaScript)
- JSR-223 based web views (with a focus on JavaScript on Nashorn)
- Rich support for CORS and declarative HTTP caching
- First-class support for HTTP Streaming and Server-Sent Events
- CompletableFuture for handler methods and @Async methods

Let take a break!

Migrating From Spring 3.x to Spring 4.x

The Big Changes

- Spring Framework 4.0 requires Java SE 6
- Specifically, a minimum API level equivalent to JDK 6 update 18
    - 1.6.0_18, as released in January 2010
- Spring 4 requests a minimum of JDK 6 update 21 for known Java bug fixes
- Spring recommends JDK 6 update 25 or higher from a support perspective

Better Yet Java 7 or Java 8

- Java 7 and 8 are recommended for use with Spring Framework 4.0
- Java 8 has been supported in production since Oracle's JDK 8 launch in March 2014

Nitty Gritty

- *There are always issues when migrating*
- *Most errors occur when deprecated methods are still being used*
- *Use Eclipse to find and correct all deprecations*

Deprecations

- A number of deprecations were removed in Spring 4 to clean up clutter
- If you are using the XML Namespace configuration, there are many instances where you will be shielded from deprecation
- Remember a quick search in your workspace should allow you to find all the deprecations

IBM's JDK

- Spring will continue to support JDK 6 and JRE 6 to maintain compatibility with IBM's Webshpere products
- Websphere 7, 8 and 8.5 all shipped with JDK 6

No Big Hangups

- Spring 4.x supports all current versions of its optionally integrated libraries, including:
  - *Hibernate ORM 5.2*
  - *Jackson 2.7/2.8*
  - *OkHttp 3.x*.
- It also embeds the updated *ASM 5.1* and *Objenesis 2.4*.

Specification Requirement List

- Servlet 3.0 *(2.5 supported for deployment)*
- JPA 2.0
- Bean Validation 1.0
- JSF 2.0
- JCache 1.0
- JDO 3.0

Server Requirement List

- Tomcat 6.0.33 / 7.0.20 / 8.0.9
- Jetty 7.5
- JBoss AS 6.1
  - *JBoss EAP 6 recommended, since AS 6/7 community releases have many unresolved bugs*
- GlassFish 3.1
  - *deployment workaround necessary for non-serializable session attributes*
- Oracle WebLogic 10.3.4
  - *JPA 2.0 patch applied*
- IBM WebSphere 7.0.0.9
  - *JPA 2.0 feature pack installed*

Where's The List

- For a list of all requirements and deprecations:
- https://github.com/spring-projects/spring-framework/wiki/Migrating-to-Spring-Framework-4.x

Web Containers

- EE server generations certified for Java EE 6 are recommended
- JPA 2.0 and Servlet 3.0 specifications are highly recommended but it is possible to deploy Spring 4 applications to servers with a Servlet 2.5 container:
  - Google App Engine
  - WebSphere 7
  - WebLogic 10.3
- Tomcat 7 or 8 is recommended but support for Tomcat 6 is still there

Java EE Requirements

- The Spring Framework doesn't require a specific level of Java EE
- Spring requires minimum levels of individual specifications, such as JPA 2.0
- This provides flexibility to run on intermediate server releases
- These intermediate releases contain new specification for individual frameworks like WebSphere Liberty Profile with Servlet 3.1

Lots of Specific Papers for Migrating

- https://github.com/spring-projects/spring-framework/wiki/Migrating-to-Spring-Framework-4.x
- Stackoverflow.com to ask migration questions

Spring 5 Roadmap

Spring 5 Mirrors JDK 9 Release

- First Milestone releases in July 2016
- GA release scheduled for May 2017 alongside JDK 9

Spring 5 Goals

- A key step is to require Java 8+ in order to apply the Java 8 language level to the entire framework codebase
- 5.0 will enable Spring to use Java 8 constructs in its core framework exposing them in core interfaces
- Up until now Spring auto-adapts Java 8 constructs for user code

JDK 9 and More

- Java 8 will be the minimum requirement
- Spring Framework 5.0 will be built on JDK 9 providing comprehensive support for the upcoming generation of the JDK
- Java 9 and Spring 5 will include the new HTTP client API, concurrency enhancements, and more
- Spring 5 also plans JSR-330 revisions and to provide early support for EE 8 level specs such as Servlet 4.0 and JMS 2.1

What are Reactive Web Applications

- The marquee feature for the new release will be first class support for Reactive web applications, derived from work done in Project Reactor which has itself been developed in close collaboration with the RxJava lead.
- Reactive programming is about non-blocking, event-driven applications that scale with a small number of threads with backpressure as a key ingredient that aims to ensure producers do not overwhelm consumers

Reactive Streams

- The Reactive Streams specification of Java 9 enables the ability to communicate demand across layers and libraries from different providers
- An HTTP connection writing to a client can communicate its availability to write all the way upstream to a data repository fetching data from a database so that given a slow HTTP client the repository can slow down too or even pause

Spring 5 and Streams

- Spring Framework 5 uses Reactive Streams and Reactor internally
- Spring 5 provides:
  - Reactive serialization and deserialization to and from JSON (Jackson) and XML (JAXB)
  - Areactive web framework that supports the @Controller programming model
  - A reactive WebClient
- It becomes easy to support input and output streaming scenarios for microservices, scatter/gather, data ingestion, etc.

Non-Blocking Data Stream Controller

```
@GetMapping("/Person/{id}/alerts")
public Flux<Alert>
getAccountAlerts(@PathVariable UUID id) {

   return this.repository.getAccount(id)
        .flatMap(account ->
            this.webClient
                .perform(get("/alerts/{key}",
account.getKey()))
                .extract(bodyStream(Alert.class
)));
}
```

MVC vs. Reactive

- There is nothing fundamentally incompatible between the @Controller programming model and the reactive ways
- It all happens underneath the hood to support the reactive model
- On the surface there is no difference except for the full support for reactive types such as *Flux*, *Mono*, *Observable* and *Single* from RxJava both for input and for output

What About JDBC or No SQL Calls

- When we have to call a blocking IO like a JDBC method in a reactive service, the blocking IO call must be placed in a separate thread
- This way reactor threads remain free
- Not perfect but workable

Functional Web Framework

- The HandlerFunction<T> is the starting point for the new framework
- It is a Function<Request, Response<T>>
- Request and Response are newly-defined, immutable interfaces that offer a JDK-8 friendly DSL to the underlying HTTP messages
- The annotation counterpart to HandlerFunction would be a method with @RequestMapping

Create the Repository

```
public interface PersonRepository {
        Mono<Person> getPerson(int id);
        Flux<Person> allPeople();
        Mono<Void> savePerson(Mono<Person> 
person);
    }
```

Expose the Repository

```
RouterFunction<?> route = route(GET("/Person/{id}"),
                          request -> {
                            Mono<Person> person =
Mono.justOrEmpty(request.pathVariable("id"))
                              .map(Integer::valueOf)
                              .then(repository::getPerson);
                            return Response.ok().body(fromPublisher(person,
Person.class));
                          })
                          .and(route(GET("/Person"),
                            request -> {
                              Flux<Person> people = repository.allPeople();
                              return Response.ok().body(fromPublisher(people,
Person.class));
                            }))
                          .and(route(POST("/Person"),
                            request -> {
                              Mono<Person> person = request.body(toMono(Person.class));
                              return Response.ok().build(repository.savePerson(person));
                            }));
```

Run the Handler

```
HttpHandler httpHandler =
RouterFunctions.toHttpHandler(route);
ReactorHttpHandlerAdapter adapter =
  new ReactorHttpHandlerAdapter(httpHandler);
HttpServer server = HttpServer.create("localhost", 8080);
server.startAndAwait(adapter);
```

Spring 5 Worth the Wait

- Spring 5 promises to be choc full of new frameworks
- Reactive programming is the big new paradigm
- Immutable object will be big now that memory is cheap
- Java 9 and new streams will also be highlight
- Java 8 built into the Spring core

Lets take a break!

JSR-330

Dependency Injection

Inversion of Control (IoC)

- Inversion of Control (IoC) means that objects do not create other objects on which they rely to do their work
- Objects get object references that they need from an outside source i.e. an xml configuration or application configuration file

What is Dependency Injection (DI)

- A dependency is any other object that is referenced within a class
- Injection is having those references passed into an object via constructor or property setters
- DI is essentially passing the buck of objection creation and management to an object further up in the dependency graph
- Injection can also occur as parameter injection, most bean definitions accept dependencies as parameter injections

How Does DI Help

- DI separates components and allows separate frameworks to comply with details
- Factories used to create objects can require resources and or require complex computations or IO access to networks or other resources that can result in bottlenecks

```
public DAO(){
 dataAccessObject = DAOFactory.getObject();
}
```

Mocking

- In order to mock factory objects, the factory creation call has to be intercepted and the object mocked based on a test case
- With DI the mock can be passed in at runtime

Injection

- Passing the object into the constructor moves the problem to another object
- Testing and mocking now become a lot easier because we can just pass in a mocked object

```
public DAO(SpecialDAO specialDAO){
 this.dataAccessObject = specialDAO;
}
```

- Avoid Constructor creep and provides separation

@Autowired

- Spring DI is accomplished using the @Autowired annotation
- @Autowired can be applied on a bean's constructor, field, setter method or a config method to autowire the dependency using Spring's dependency injection

```
@Autowired
private PersonRepository repo;
```

@Resource and @Qualifier

- @Resource annotation with ‘name’ attribute
- @Qualifier annotation is often used in conjunction with @Autowired to resolve ambiguity in case more that one bean of injected type exist in application context.

Example

- In the following example we will use the @Qualifier annotation to inject step1 to define a method
- Java usually resolves by type, in this case step one and step 2 are the same type implementing the same interface
- We will inject a qualifier for each of the similar types

@Qualifier

```
@Bean
@Qualifier("s2")
public Step step2(StepBuilderFactory
stepBuilderFactory, @Qualifier("post-
process") Tasklet postProcessTasklet) {
  return
     stepBuilderFactory.get("step2").
     tasklet(postProcessTasklet).
     build();
}
```

@Resource

- Standard @Resource annotation marks a resource that is needed by the application
- It is analogous to @Autowired in that both injects beans by type with no attribute provided
- But with name attribute, @Resource allows you to inject a bean by it's name, which @Autowired does not

@Resource Example

```
@Component("application")
public class Application {
 @Resource(name="applicationUser")
 private ApplicationUser user;
 @Override
 public String toString() {
   return "Application [user=" + user + "]";
 }
}
```

Finding The Bean

- The Application class’s user property is annotated with @Resource(name=”applicationUser”)
- A bean with name ‘applicationUser’ found in applicationContext will be injected here

Dependency Implementation

```
@Component("applicationUser")
public class ApplicationUser {
    private String name = "defaultName”;
 …
    public void setName(String name) {
        this.name = name;
    }
 @Override
public String toString() {
  return "ApplicationUser [name=" + name + "]";
 }
}
```

AppConfig

- @ComponentScan will make Spring auto detect the annotated beans
- Spring scans the specified package and wires them wherever annotations @Resource or @Autowired are used
- Each annotation uses a different scanner to find annotated beans

DI Example

```
@SpringBootApplication
@ComponentScan(basePackages="com.buzr")
@RestController
public class ReadApiApplication {
      @Autowired
      private PersonRepository repo;
      @GetMapping("/Person/{id}")
      public  Person person(@PathVariable UUID id){
            MapId personId = BasicMapId.id("id", id);
            return repo.findOne(personId);
      }                          …
```

Dependency Inversion Pattern (DI*)

- High Level Classes --> Abstraction Layer --> Low Level Classes

- High-level modules should not depend on low-level modules
- All levels should depend on abstractions
- Abstractions should not depend on details
- Details should depend on abstractions

Interfaces

- DI* uses interfaces to provide the layer of abstraction defined in the IoC pattern
- All components become chain testable because higher level components don't depend on lower level components
- Higher level components know only about an Interface definition, no details
- Mocks implements these interfaces to substitute simulations

Lab Overview and Setup

- We will build on our labs to incorporate different technologies we are covering
- We must first setup our Cassandra database
- Next we will set up Eclipse to develop our app
- Feel free to use Gradle, we use Maven

Download and Start Cassandra

- Setting up Cassandra instance
- Download and install Cassandra
  - – apt-get install Cassandra:
  - brew install Cassandra
  - Windows Download and install
- Services start Cassandra
  - $cd apache-cassandra-2.2.8/bin
  - $sudo ./cassandra

Start Cassandra Shell cqlsh

- Cqlsh to start Cassandra shell
  - $cqlsh

```
Connected to Test Cluster at 127.0.0.1:9042.
[cqlsh 5.0.1 | Cassandra 2.2.8 | CQL spec 3.3.1 | Native
protocol v4]
Use HELP for help.
cqlsh>
```

Set Up a Test Database

- First we need to create a Key Space/DB, let’s call it dev, to hold our tables
  - cqlsh> CREATE KEYSPACE dev with replication = { 'class' : 'SimpleStrategy', 'replication_factor' : 3 };

- With replication =

{‘class’:'SimpleStrategy', 'replication_factor':1};

- Initializes DB with itself by itself

Select KeySpace and Create Table

- cqlsh> use dev;

- cqlsh> create table user (userid int primary key, first_name varchar, last_name varchar, secret varchar, email varchar);
- cqlsh> insert into user(userid, first_name, last_name, secret, email) values (1, 'Pungo', 'Seymour', 'heisadog', 'pungo@buzr.com');

Cassandra Configuration

- Cassandra config is an annotated java class
- @configuration is a spring annotation used to configure the context for our DB
- When spring runs it will scan for classes annotated with @configuration for beans to carry out Dependency Injection
- Which means it will look for tables so it can create java classes that will enable us to configure the Cassandra cluster factory
- This will connect the Cassandra db instance to our application

Time For a Lab

- Set up workspace environment and make sure all projects compile

REpresentational State Transfer (REST)

Topics

- Building REST Solutions
- RESTful Web Services
- Key Annotations
- Parameter Annotations
- Running the Service

RESTful Web Services

REST stands for Representational State Transfer

- RESTful web services are designed to work across the Internet.
- REST is an architectural style that is designed to introduce an additional level of abstraction.
- REST specifies constraints, for example a uniform interface.
- REST enhances desirable properties
  - Performance
  - Scalability
  - Modifiability

Do We Need REST?

- Today, the emerging trend of mobile-based applications requires lightweight data transfer in all transactions.
- REST is a services architecture which can accept the request from a browser and respond to the same browser with compatible data types.
- RESTful web services use JSON formats to communicate between browsers and the web services, satisfying the need of lightweight data transfer.

REST Architectural Style

- The REST architectural style defines data and functionality as resources.
- Resources are accessed by using Uniform Resource Identifiers (URIs), i.e., links on the Web.
- Well-defined operations are used to access such resources.

Why REST

- Representational state transfer is an architecture style.
- REST consists of a coordinated set of criteria and constraints that can be used to define an application architecture.
- It is a set of guidelines.
- It is a style of application architecture.

Many Frameworks Offer REST

- There are many frameworks that provide RESTful services out of the box:
  - Spring
  - Jersey

Too Many Choices

Everyone has an opinion on which REST framework is the best. When looking for the right solutions here are some points to look for:

- Complete RESTful API
- Classes for servers and clients
- Java convention consistency
- Complete extension mechanism to interact with other Java web-based technologies
- Well-maintained and mature
- Open source

Designing an API

REST provides a way for developers to design their own API as it relates to a specific application. Certain guidelines should be followed, it should:

- Use web standards where they make sense
- Be friendly to the developer and able to be explored using a browser
- Be simple, intuitive, and consistent
- Provide flexibility to work with the view
- Be efficient
- Maintain balance with the other requirements
- Be open source

RESTful Web Services

- RESTFul web services are based on HTTP methods and the REST architecture.
- RESTFul web services define the base URI for REST services, MIME-types:
  - XML
  - Text
  - JSON
  - User-defined types
- And the supported operation set:
  - POST
  - GET
  - PUT
  - DELETE

HTTP and REST

- The PUT, GET, POST, and DELETE methods are used in REST-based architectures.
  - GET defines read access for a resource without side effects. The resource is never changed via a GET request; the operation is idempotent.
  - PUT creates a new resource.
  - DELETE removes resources.
  - POST updates an existing resource or creates a new resource.

JAX-RS

- JAX RS is a specifications based on JSR 339
- The major release under JAX RS is:
  - JAX RS 2.0

The Java API for RESTful Web Services

- Java defines REST support via the Java Specification Request (JSR) 311.
- This specification is called JAX-RS.
- JAX-RS uses annotations to define the implementation of REST in Java classes.

Let’s Build A Simple Rest API

- The following example will use Spring and REST to construct a way to write data into an Apache Cassandra instance
- We will use this application for the “rest” of the class and build each technology as another piece

The WriteApiApplication Class

```
@RestController
@ComponentScan(basePackages={"com.buzr"})
@SpringBootApplication
public class WriteApiApplication {

	@Autowired
	private PersonRepository repo;
	@PostMapping("/Person")
	public Person person(@RequestBody Person person)
	{
		person.setId(UUID.randomUUID());
		person.setDateCreated(new Date());
		return repo.save(person);
	}
	public static void main(String[] args) {
		SpringApplication.run(WriteApiApplication.class, args);
	}
}
```

Step By Step @RestController

- RestController Annotation Type
- Annotated Interface

```
@Target(value=TYPE)
@Retention(value=RUNTIME)
@Documented
@Controller
@ResponseBody
public @interface RestController
```

@RestController

- @RestController is processed if an appropriate HandlerMapping-HandlerAdapter pair is configured such as the RequestMappingHandlerMapping-RequestMappingHandlerAdapter pair which are the default in the MVC Java config and the MVC namespace

```
Specialized version of the controller annotation where all
of the request mappings are assumed to return ajax style
responses
Request body converts returned object to serialized
version.
RestController assumes all should be serialized
@RestController
```

@ComponentScan Annotation Type

```
@Retention(value=RUNTIME)
 @Target(value=TYPE)
 @Documented
 @Repeatable(value=ComponentScans.class)
public @interface ComponentScan
```

@Configuration

- Configures component scanning directives for use with @Configuration classes
- Indicates that a class declares one or more @Bean methods and may be processed by the Spring container
- Container generates bean definitions and service requests for @Beans at runtime

@ComponentScan

- Provides support parallel with Spring XML's <context:component-scan> element
- Provides a way to look for other components without having to directly link project and packages together
- @ComponentScan uses string a array, to scan multiple sources provided by an array argument:
  - @ComponentScan({"com.buzr.write","com.buzr.read"})

Write API Example

```
@ComponentScan(basePackages={"com.buzr"})
```

- Allows us to pull dependencies from other packages
- Looks for @Bean, @Service, etc..
- @Autowire Spring annotations are scanned
- micro-services are registered in application context, objects, singletons, types and instantiates for dependency injection
- Scan for all injectable components in a base package

@SpringBootApplication

- The @SpringBootApplication annotation is equivalent to using @Configuration, @EnableAutoConfiguration and @ComponentScan with their default attributes

```
//@Configuration @EnableAutoConfiguration @ComponentScan
@SpringBootApplication
public class Application {
    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

@AutoWired

- Autowiring is method of creating an instance of an object and injecting that instance on a specific class that uses it
- Wiring an instance to a class that will use it’s attributes
- In Spring, when the application server initialize the context, it creates a stack/heaps of objects in it’s JVM container
- These objects are available for consumption at any given time as long as the application is running

@Autowired Example

```
public class WriteApiApplication {

Springs way to inject dependency.
Autowiring on fields and constructors
When autowiring fields, spring uses a proxy to construct class and add
dependencies using reflection
Constructor injection allows for fine tuning of fields

    @Autowired
    private PersonRepository repo;
```

@PostMapping Annotation Type

```
@Target(value=METHOD)
@Retention(value=RUNTIME)
@Documented
@RequestMapping(method=POST)
public @interface PostMapping
```

- Annotation for mapping HTTP POST requests onto specific handler methods
- @PostMapping is a composed annotation that acts as a shortcut for

  @RequestMapping(method = RequestMethod.POST)

@PostMapping Example

```
Short cut annotation for request mapping for the http method Post in this
case
@PostMapping("/Person")
```

@RequestBody

- The @RequestBody method parameter annotation indicates that a method parameter should be bound to the value of the HTTP request body

```
public Person person(@RequestBody Person person)
```

- In this example the incoming argument, person, would be converted from JSON and made into a Person object

@RequestBody body of HTML response needs to be de-serialed or unmarshalled into the Person object in this case, Servlet container is instructed on what to do with payload as it is received

```
    public Person person(@RequestBody Person person)
    {
        person.setId(UUID.randomUUID());
        person.setDateCreated(new Date());
        return repo.save(person);


    }
```

Conclusion

This module covered:

- Building REST Solutions
- RESTful Web Services
- REST Annotations
- Example of WriteApi

REST Lab, Create ReadApi

- Please Complete the Rest Lab

- Intentionally left blank

Spring Batch Walkthrough

Batch

- Batch executes jobs in order
- Tasklets
- Demo Rest API read and write API’s using a webservice

Spring Batch Processing

- Spring Batch is an open source framework for batch processing
- It is built as a module within the Spring framework
- Batch processing is the execution of a series of tasks or programs in an ordered way

Spring Batch

- Spring Batch provides mechanisms for processing large amounts of data
  - Transaction management
  - Job processing
  - Resource management
  - Logging
  - Tracing
  - Conversion of data
- The batch framework takes care of the performance and the scalability while processing the records

Iterations

- Spring Batch automates basic batch iterations by providing the capability to process similar transactions as a set, typically in an offline environment and often without any user interaction
- Batch Processing Strategies?
  - Conversion Applications (Facades)
  - Extract Applications
  - Processing and Updating Applications

Current Version of Batch is 3.0.7

- JSR-352
- Spring Integration 4 moves the core messaging APIs to Spring core
- Because of this, Spring Batch 3 will now require Spring 4 or greater
- Spring Batch now supports being run on Java 8, it will still execute on Java 6 or higher as well

Jobs, Steps, ItemReaders, and ItemWriters

- batch processing historically is made up of “Jobs", "Steps" and processing units called ItemReaders and ItemWriters
- ItemReaders and ItemWriters are supplied by the developer

Batch Entities

- ItemReader provides items for processing.
- ItemWriter processes items provided by the ItemReader.
- Spring's PlatformTransactionManager is used to begin and commit transactions during processing
- JobRepository is used to periodically store the StepExecution and ExecutionContext during processing, just before committing
- Commit-interval is the number of items processed before the transaction is committed

ItemReader

- ItemReaders provide data from many different types of input
- The most general examples include:
- Flat File Item Readers read lines of data from a flat file
- XML ItemReaders process XML
- Database resource is accessed to return resultsets which can be mapped to objects for processing

ItemReader Example

```
@Service
public class CassandraItemReader implements ItemReader<Person>{

@Autowired
can implement lower lever cassandra operations, JDBC driver directly
instead of provided api
private CassandraOperations cassandraOperations;
private int index=0;
private final Class<Person> clazz = Person.class;
private String cql;
private List<Person> personsCache;

public CassandraItemReader() {
    super();
}
```

Reading the Data

```
@Override
public Person read() throws Exception, UnexpectedInputException, ParseException,
NonTransientResourceException {

    Assert.notNull(cql, "You must set the Select Statement before performing this operation");
    //Assert.notNull(clazz, "You must set the Person type before performing this operation");

        //prevent modification
        final List<Person> persons;
        //ensure we only call the db once.
        if (this.personsCache == null){
                this.personsCache = cassandraOperations.select(this.cql, clazz );
        }
        persons = personsCache;

        if (index < persons.size()) {
                final Person person = persons.get(index);
                index++;
                return person;
        }
                return null;
        }
```

Item Writer

- ItemWriter is the opposite of an ItemReader
- Resources still need to be located, opened and closed but they differ in that an ItemWriter writes out, rather than reading in
- In the case of databases or queues these may be inserts, updates, or sends
- The format of the serialization of the output is specific to each batch job

Item Writer Example

```
@Service
// annotation tells spring to instantiate as a singleton with application scope
public class CassandraItemWriter implements ItemWriter<Person> {

  @Autowired
  private PersonRepository repo;

  @Override
  // implementing interface
  public void write(List<? extends Person> arg0) throws Exception {
      repo.save(arg0);
  }
}
```

Item Processor

- An ItemProcessor is very simple; given one object, transform it and return another

ItemProcessor Example

```
public class UserItemProcessor implements ItemProcessor<String[], Person> {

    private static final int FIRST_NAME_IDX = 0;
    private static final int LAST_NAME_IDX = 1;
    private static final int EMAIL_IDX = 2;
    private static final int SECRET_IDX = 3;

    @Override
    public Person process(String[] arg0) throws Exception {
        Person person = new Person();
        person.setId(UUID.randomUUID());
        person.setDateCreated(new Date());
        person.setFirstName(arg0[FIRST_NAME_IDX]);
        person.setLastName(arg0[LAST_NAME_IDX]);
        person.setEmail(arg0[EMAIL_IDX]);
        person.setSecret(arg0[SECRET_IDX]);
        return person;
    }
}
```

Jobs

- Jobs are abstractions to represent sequences of actions or commands that have to be executed within the batch application
- A Job is an entity that encapsulates an entire batch process
- Jobs are at the top of the Batch hierarchy
- Simple Jobs contain a list of steps and these are executed sequentially or in parallel
- a Job is simply a container for logical steps

Job Example

```
@Bean
  @qualifier is used to inject step 1 first since step 2 is the same type. Java
usually resolves by the
  type in this case we want to resolve by type, "Step" but need to inject correct
value of same type
  public Job importUserJob(JobBuilderFactory jobs,  @Qualifier("s1") Step s1,
                        @Qualifier("s2") Step s2, JobExecutionListener listener) {
     return jobs.get(UUID.randomUUID().toString()).incrementer( new
                    RunIdIncrementer()).listener(listener).flow(s1).next(s2).end().build();
  }
```

Steps

- A Step is a domain object that encapsulates an independent, sequential phase of a batch job
- A Step contains all of the information necessary to define and control the actual batch processing
- Sterps can be relatively simple or very complex
- A StepExecution represents a single attempt to execute a Step

Step Example

```
@Bean
@Qualifier("s1")
public Step step1(StepBuilderFactory stepBuilderFactory, ItemReader<String[]>
                    reader, ItemWriter<Person> writer,
                    ItemProcessor<String[], Person> processor) {
    return stepBuilderFactory.get("step1").<String[],
                    Person>chunk(10).reader(reader).processor(processor)
                    .writer(writer).build();
}

@Bean
@Qualifier("s2")
public Step step2(StepBuilderFactory stepBuilderFactory, @Qualifier("post-
process") Tasklet postProcessTasklet) {
          return stepBuilderFactory.get("step2").
                    tasklet(postProcessTasklet).build();
     }
```

Execution Context

- An ExecutionContext represents a collection of key/value pairs that are persisted and controlled by the framework
- The values allow developers a place to store persistent state that is scoped to a StepExecution or JobExecution
- The execution context is a place to save the state of a batch
- If there is a fatal error or power loss the execution context will have data saved from a periodically defined commit points and can rebuild the job

Harvesting Statistics From Execution Context

- If a flat file contains a list of persons to add to the database that exist across multiple lines, it may be necessary to store how many persons have been processed so that an email can be sent at the end of the Step with the total persons processed and added to the DB
- These stats can be much more detailed that how many lines were read

Job Repository

- JobRepository is the persistence mechanism for batch processes
- It provides CRUD operations for JobLauncher, Job, and Step implementations
- When a Job is first launched, a JobExecution is obtained from the repository, and during the course of execution StepExecution and JobExecution implementations are persisted by passing them to the repository:

Ka-Chunk

- Spring Batch uses a 'Chunk Oriented' processing style within its most common implementation
- Chunk oriented processing refers to reading the data one at a time, and creating 'chunks' that will be written out, within a transaction boundary
- One item is read in from an ItemReader, handed to an ItemProcessor, and aggregated
- Once the number of items read equals the commit interval, the entire chunk is written out via the ItemWriter, and then the transaction is committed

What is a Chunk

- A chunk is defined as the items processed within the scope of a transaction
- Commit intervals define a 'chunk'
- A ChunkListener can be useful to perform logic before a chunk begins processing or after a chunk has completed successfully

```
public interface ChunkListener extends StepListener {
        void beforeChunk();
        void afterChunk();
    }
```

Setting the Chunk

- We set the Chunk to 10 in our example:

```
return stepBuilderFactory.get("step1").<String[],
    Person>chunk(10).reader(reader).processor(processor)
    .writer(writer).build();
```

- We can chain the reader and writer together avoiding the need for an item processor

Tasklets the Anti-Chunk

- Chunk-oriented processing is not the only way to process in a Step
- A tasklet is used when there is no need for a return value to the writer.
- Tasklet is a simple interface with a single method, execute
- Execute is called over and over until the task let returns RepeatStatus.finished or throws an exception

Tasklet Example

```
public class PostProcessUsersTasklet implements Tasklet {

  private ItemReader<Person> reader;
  private Person curr;

  public PostProcessUsersTasklet(ItemReader<Person> reader){
      this.reader = reader;
  }
  @Override
  public RepeatStatus execute(StepContribution arg0, ChunkContext arg1) throws
      Exception {
            while(setNewCurr(reader) != null){
              try {
                    printOutPerson(curr);
              }
              catch(Exception e){
                    e.printStackTrace();
                    return RepeatStatus.CONTINUABLE;
              }}
          return RepeatStatus.FINISHED;
      }
```

Defining the Flow

- The simplest flow scenario is a job where all of the steps execute sequentially
- the Spring Batch namespace allows transition elements to be defined within the step element
- One such transition is the "next" element
- Like the "next" attribute, the "next" element will tell the Job which Step to execute next
- This allows for specific steps to be taken based on the type of exit a method has or a result within a method

Defined Flow Example

```
@Bean
@qualifier is used to inject step 1 first since step 2 is the same type."Step" but need to inject
correct value of same type
public Job importUserJob(JobBuilderFactory jobs,  @Qualifier("s1") Step s1,
                          @Qualifier("s2") Step s2, JobExecutionListener listener) {
      return jobs.get(UUID.randomUUID().toString()).incrementer(
                          new RunIdIncrementer()).listener(listener).flow(s1).next(s2).end().build();
      }
```

Batch Lab

- You have been tasked to fix a bad run of the in class batch script.
- The original batch process reversed the secret and email data and now each person needs to have their data corrected.
- The original csv file must be amended to include the id of the person found in the output of the original batch process and you must make use
- of read/write rest endpoints to reimplement the itemreader and itemwriters in the batch program.

Spring, Cassandra and DI

- Spring configuration supports using Java based @Configuration classes and XML
- CassandraTemplate helper classes perform common Cassandra operations:
  - Integrated object mapping between CQL Tables and POJOs
  - Exception translation into Spring's portable Data Access Exception hierarchy
  - Feature Rich Object Mapping
  - Annotation based mapping metadata
  - Persistence and mapping lifecycle events
  - Java based Query, Criteria, and Update DSLs
  - Automatic implementation of Repository interfaces including support for custom finder methods.

CassandraTemplate Class

- CassandraTemplate contains methods for accessing functionality such as incrementing counters or ad-hoc CRUD operations
- CassandraTemplate provides callback methods so that it is easy for you to get a hold of the low level API artifacts such as Session to communicate directly with Cassandra

DataStax Java Driver

- Spring Data Cassandra uses the DataStax Java Driver version 2.X
- The driver supports:
  - DataStax Enterprise 4/Cassandra 2.0
  - Java SE 6+

Cassandra Dependency

- Maven:

```
<dependency>
   <groupId>org.springframework.data</groupId>
   <artifactId>spring-data-cassandra</artifactId>
   <version>1.0.0.RELEASE</version>
 </dependency>
```

- Gradel

```
dependencies {
        classpath(“org.springframework.boot:spring-boot-gradle-plugin:
                ${springBootVersion}")
        classpath(“com.datastax.cassandra:cassandra-driver-core:
                ${cassandraDriverVersion}")
        }
```

Connecting to Cassandra with Spring

- Create a properties file and or a configuration class
- We will use a configuration class

CassandraConfig extends AbstractCassandraConfiguration

```
@Configuration
@EnableCassandraRepositories(basePackages="com.buzr")
public class CassandraConfig extends AbstractCassandraConfiguration {

    @Override
    protected String getKeyspaceName() {
        return "dev";
    }

    @Bean
    public CassandraClusterFactoryBean cluster() {
        CassandraClusterFactoryBean cluster =
          new CassandraClusterFactoryBean();
        cluster.setContactPoints("127.0.0.1");
        cluster.setPort(9042);

        return cluster;
    }

    @Bean
    public CassandraMappingContext cassandraMapping()
      throws ClassNotFoundException {
        return new BasicCassandraMappingContext();
    }
}
```

@Configuration

- Annotating a class with the @Configuration indicates that the class can be used by the Spring IoC container as a source of bean definitions
- The @Bean annotation tells Spring that a method annotated with @Bean will return an object that should be registered as a bean in the Spring application context

Cassandra Configuration Example

```
@Configuration
@EnableCassandraRepositories(basePackages="com.buzr")
public class CassandraConfig extends AbstractCassandraConfiguration {
```

- @Configuration tells spring to expect a configuration class
- @EnableCassandraRepositories(basePackages lets spring know which base packages to scan for annotated components

AbstractCassandraConfiguration

- @Configuration class intializes Spring beans for Cluster and Session instances
- Spring Data Cassandra provides an AbstractCassandraConfiguration base class to reduce the configuration code needed:

Hooking Into A Cassandra Instance

```
@Bean
public CassandraClusterFactoryBean cluster() {
    CassandraClusterFactoryBean cluster =
      new CassandraClusterFactoryBean();
    cluster.setContactPoints("127.0.0.1");
    cluster.setPort(9042);

    return cluster;
}

@Bean
public CassandraMappingContext cassandraMapping()
  throws ClassNotFoundException {
    return new BasicCassandraMappingContext();
}
```

POJO’s

- CassandraTemplate builds on top of the basic CQL capabilities provided by CqlTemplate
- The template gives you the ability to work with Java objects while Spring Data Cassandra (SDC) takes care of query building and object mapping for you

```
@Table
public class Person {

    public void setDateCreated(Date dateCreated) {
        this.dateCreated = dateCreated;
    }

    @PrimaryKey
    @Column("id")
    private UUID id;

    @Column("first_name")
    private String firstName;

    @Column("last_name")
    private String lastName;

    private String email;

    private String secret;

    @Column(value="create_date")
    private Date dateCreated;
```

@Column(“id)”

- SDC provides annotations to express class member to column mappings and conversions
- SDC also supports working with collection columns and composite keys

Cassandra Repositories

- Creating a repository is accomplished by extending CassandraRepository with a desired entity type:

```
@Repository
Extension of CassandraRepository to enable our own queries extending the
interface
public interface PersonRepository extends CassandraRepository<Person> {

CQL not SQL must be valid against cassandra conventions, i.e. ?0 is a
positional parameter
    @Query("Select * from Person where first_name = ?0")
    Iterable<Person> findByFirstName(String firstName);
}
```

Why Hibernate

- the purpose of ORMs like hibernate or JPA is to hide the complexities of SQL and instead present Object representations of SQL

What About Hibernate

- Hibernate is designed to work with relational databases only, and currently isn't capable of working with NoSQLs like cassandra
- Hibernate OGM is a project intended to fill that gap, but doesn't support Cassandra as of now
- Hibernate OGM provides Java Persistence (JPA) support for NoSQL solutions
- It reuses Hibernate ORM's engine but persists entities into a NoSQL datastore instead of a relational database

Hibernate OGM

- Hibernate OGM supports several ways for searching entities and returning them as Hibernate managed objects:
  - JP-QL queries are converted into native backend queries
  - Datastore specific native queries
  - Full-text queries, using Hibernate Search as indexing engine

Kundera

- Like Hibernate for SQL, Kundera hides the complexities of NoSQL in an intelligent way
- Kundera uses the power of NoSQL but still makes it easy for developers to use the traditional RDBMS paradigm
- Kundera helps in this NoSQL modeling by using optimization techniques such as Embedded
- NoSQL can still create monsters so it si important to understand Cassandra data modeling

Advantages of Kundera

- Handles implicitly Cassandra schema management
- Provides a means for CRUD over Cassandra in JPA way
- Cassandra is a non relational distributed database, developers using Cassandra are still looking for JPA like use cases, polyglot persistence

Cassandra Lab

Create a parent to a collection of children entities mapping

Create a repository that interfaces with new entity

Cassandra Connections

- The Cassandra driver communicates with Cassandra over TCP, using the Cassandra binary protocol
- This protocol is asynchronous, which allows each TCP connection to handle multiple simultaneous requests

Connection Steps

- when a query is executed, a stream id gets assigned to the current connection
- the driver writes a request containing the stream id and the query on the connection, and then proceeds without waiting for the response
- Once the request has been written to the connection, it is in flight
- The response contains the stream id, which allows the driver to trigger a callback that will complete the corresponding query

Pooling

- The Cassandra driver uses a pool of connections to each host
- For each Session object, there is one connection pool per connected host.
- The number of connections per pool is configurable

Configuring the Pool

- Connections pools are configured with a PoolingOptions object

```
PoolingOptions poolingOptions = new PoolingOptions();
//customize options...

Cluster cluster = Cluster.builder()
 .withContactPoints("127.0.0.1")
 .withPoolingOptions(poolingOptions)
 .build();
```

Pool Size

- Connection pools have a variable size, which gets adjusted automatically depending on the current load
- There will always be at least a core number of connections, and at most a max number

```
poolingOptions
.setCoreConnectionsPerHost(HostDistance.LOCAL,  4)
.setMaxConnectionsPerHost( HostDistance.LOCAL, 10)
.setCoreConnectionsPerHost(HostDistance.REMOTE, 2)
.setMaxConnectionsPerHost( HostDistance.REMOTE, 4);
```

Aquiring Pool Policies

- The easiest way to monitor pool usage is with Session.getState

```
final LoadBalancingPolicy loadBalancingPolicy =
        cluster.getConfiguration().getPolicies().getLoadBalancingPolicy();
final PoolingOptions poolingOptions =
        cluster.getConfiguration().getPoolingOptions();
```

Monitoring The Load session.getState()

```
Session.State state = session.getState();
for (Host host : state.getConnectedHosts()) {
    HostDistance distance = loadBalancingPolicy.distance(host);
    int connections = state.getOpenConnections(host);
    int inFlightQueries = state.getInFlightQueries(host);
    System.out.printf("%s connections=%d, current load=%d, max
            load=%d%n", host, connections, inFlightQueries, connections *
            poolingOptions.getMaxRequestsPerConnection(distance));
```

- Intentionally Left Blank

Advanced Concurrent Programming

Presentation Topics

In this presentation we will cover:

- Introduction to Concurrent Libraries
- Working with Synchronizers
- Using the Execution Framework

Objectives

When we are done, you should be able to:

- Identify two motivations for the concurrent libraries
- List key components of the execution framework
- Describe one synchronizer

Introduction to Concurrent Libraries

Who in the world is Doug Lea?

Introduction to Concurrent Libraries

- Java provides built-in basic structures for concurrent programming
- Beyond “basic” concurrent solutions, built-in facilities are limited; foundational but not complete
- As a result, community created own concurrency oriented libraries to address complex situations

Java Thread Model Limitations

Based on block-structured locking

- Locks associated with entire objects
  - Can’t notify specific thread based on condition
  - Have to notify unknown waiting thread
- No way to:
  - “take back” or timeout attempt to acquire lock
  - Modify lock semantics
- No “built-in”:
  - Pooling mechanism
  - Auto-blocking lists
  - Limited atomic operation support

Overview of Concurrent Libraries

- Introduced as part of Java SE 5.0
- Driven by JSR 166
  - Adaptation of Doug Lea's `util.concurrent` package
  - Defined in three packages:
    - `java.util.concurrent`
    - `java.util.concurrent.atomic`
    - `java.util.concurrent.locks`

Motivations for Concurrent Libraries

- Address limitations of Java's thread model
- Standardize and simplify common concurrency mechanisms
  - Lower complexity in development concurrent programs
  - Increase maintainability of concurrent code
  - Lessen common "concurrency" issues
- Provide robust, efficient, and high-performance utilities

java.util.concurrent Package

- Main concurrency package
- Contains classes to aid in concurrency development
- Three main facilities:
  - Concurrent collections
  - Execution framework
  - Synchronizers

Concurrent Collections

Concurrent Collections

- Extend Collections framework into concurrency world
  - More scalable than standard `Collections` classes
  - Compliant with `Collection` framework
- Provide thread safety
  - More "lightweight" than synchronized
  - Typically synchronize on manipulation
  - Typically retrieval is not synchronized
- Contains:
  - List
  - Map
  - Set
  - Queue

Concurrent Collections : Iterators

- Standard `java.util.Iterator`
  - Fail-fast implementation
  - If underlying collection changes, `Iterator` throws `ConcurrentModificationException`
- Concurrency Collections
  - Weakly-consistent implementation
  - Support concurrent modifications
  - May reflect underlying changes while iterating

Concurrent Collections : Lists

- Add concurrency support to lists
  - Alternative to `Collections.synchronizedList`
  - Uses Concurrency APIs for thread-safety
- `CopyOnWriteArrayList`
  - Modifications on the list cause are performed on a copy of an array
  - Efficient because no locking on traversal
  - Not-efficient because of memory copy

Concurrent Collections : Maps

- Two interfaces:
  - `ConcurrentMap`
  - `ConcurrentNavigableMap`
- Provide atomic operations for Map
  - `putIfAbsent`
  - `remove`
  - `replace`
- Implementations include:
  - `ConcurrentHashMap`
  - `ConcurrentSkipListMap`

Concurrent Collections : Set

- Implementations include:
  - `CopyOnWriteArraySet`
  - `ConcurrentSkipListSet`
- More fine grained access control

Collection Framework : Queues

New `Queue` interface added to `java.util`

- Represents some form of waiting list
- Implementations have different ordering algorithms
  - First-in-first-out (FIFO)
  - Last-in-first-out (LIFO)
  - Natural ordering
  - Priority
- Defined in terms of
  - Head (start of queue)
  - Tail (end of queue)

Collection Framework: Qs [cont.]

- Support normal-collection behaviors
  - `java.util.Collection`
  - `java.util.Iterable`

- Support new behaviors
  - Insertion:
    - `offer` - inserts element into queue; if space available
  - Removal:
    - `remove` - removes head of queue or throws `NoSuchElementException`
    - `poll` - removes head of queue or `null`
  - Viewing
    - `element` - retrieves head or throws `NoSuchElementException`
    - `peek` - retrieves head or `null`

Concurrent Queues

- Concurrency libraries provide concurrent implementations of `Queue` interface
  - `BlockingQueue`
    - Adds waiting functionality to queue
    - `put` - adds to queue or waits for space
    - `take` - removes from queue or waits for availability
  - `BlockingDeque`

Concurrent Queues [cont.]

Sample implementations:

- Bounded Implementations
  - `ArrayBlockingQueue`
  - `LinkedBlockingQueue`
  - `LinkedBlockingDeque`

- Unbounded Implementations
  - `PriorityBlockingQueue`
  - `DelayQueue`

- Synchronous Implementation
  - `SynchronousQueue`
  - Take waits for put / put waits for take
  - No “internal” capacity

Using LinkedBlockingQueue

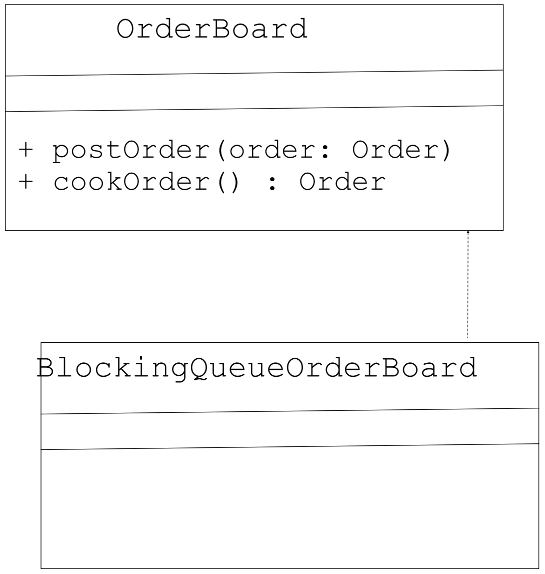
- Can be used to simplify producer - consumer problem

- Current solution uses `OrderBoard`
  - `OrderBoard` manages synchronization
    - Obtains list object lock before modifying list
    - Release list object lock after modifying list
  - `OrderBoard` manages availability
    - Determines whether insert operation is valid
    - Determines whether remove operation is valid
    - Synchronizes threads appropriately

- `LinkedBlockingQueue` alternative
  - Manages synchronization of access
  - Manages availability of access

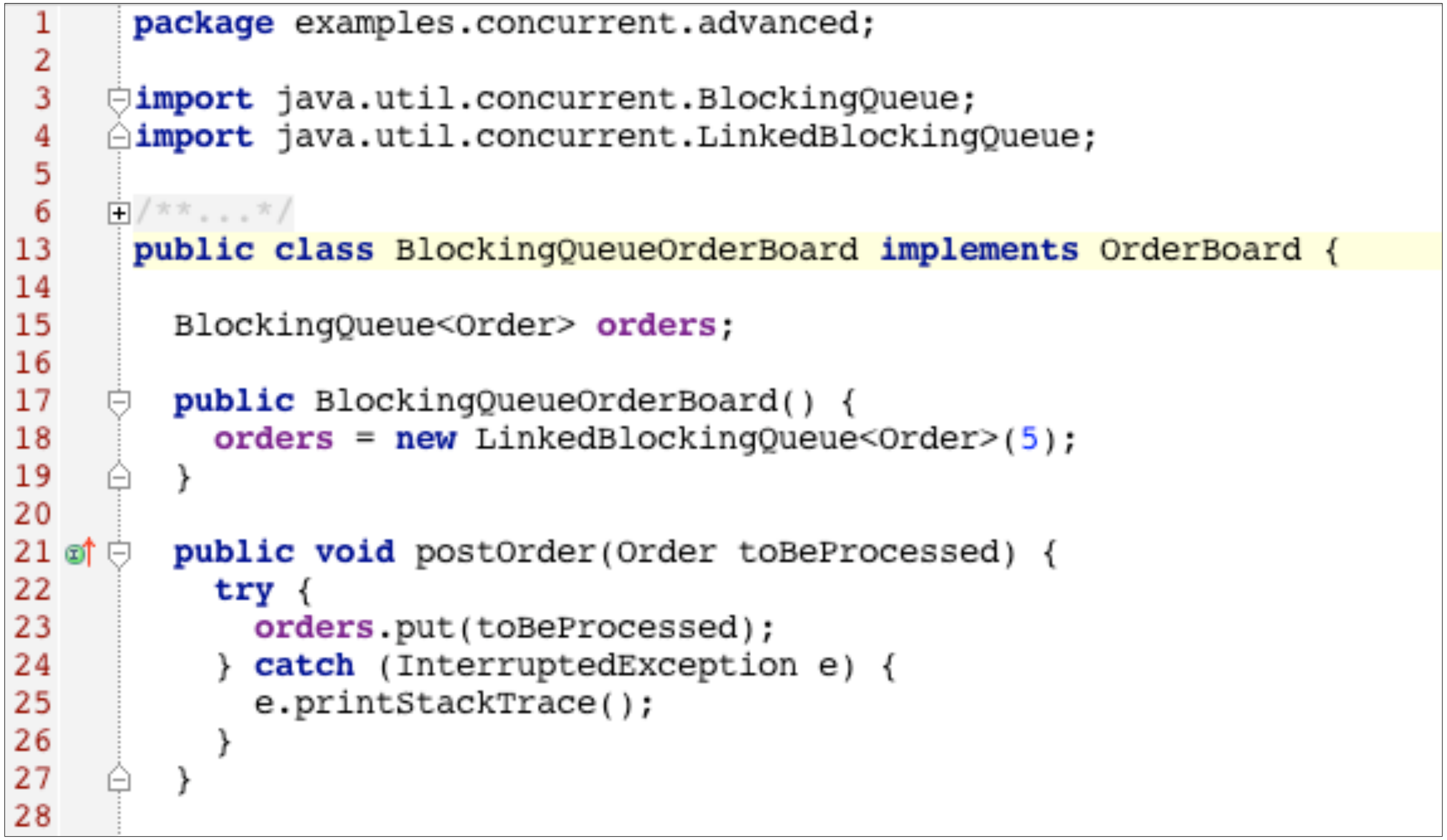
OrderBoard

- `OrderBoard` has been redesigned
  - Extracted interface
  - Enables us to create different implementations
  - Don’t need to modify cook or waiter



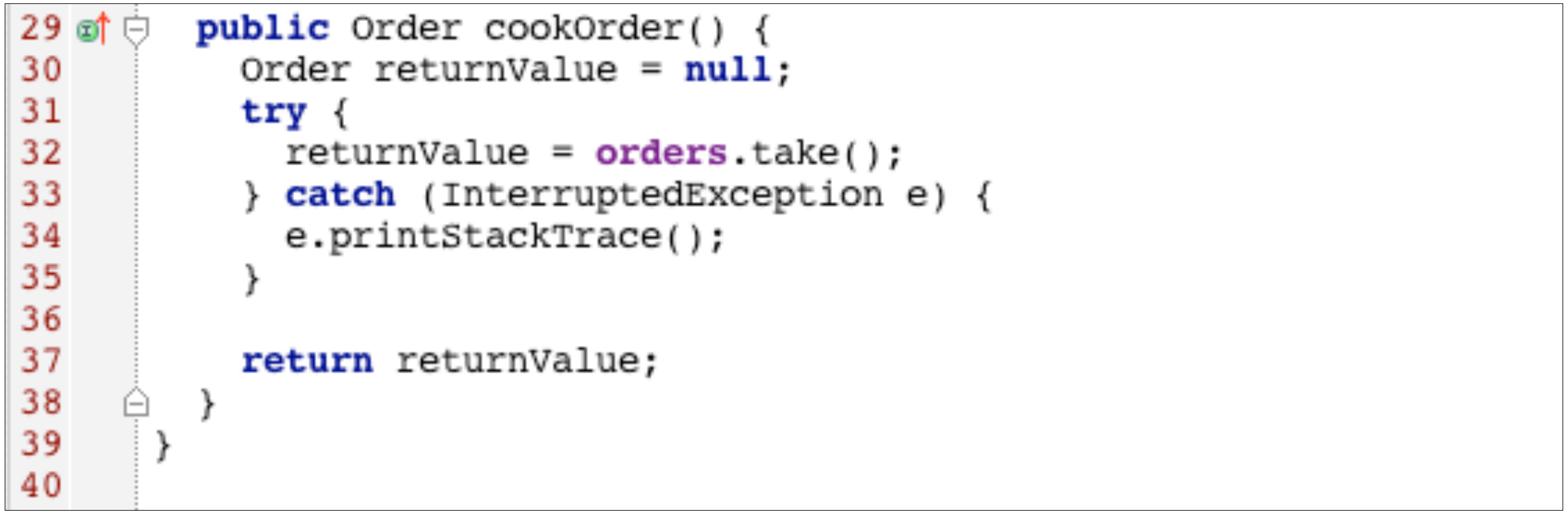
- `BlockingQueueOrderBoard`
  - Implementation of `OrderBoard`
  - " Uses a bounded `BlockingQueue`
  - " No synchronization
  - " No queue empty / full management
  - " Simplifies original `OrderBoard`

BlockingQueue Example



```
package examples.concurrent.advanced;

import java.util.concurrent.BlockingQueue;
import java.util.concurrent.LinkedBlockingQueue;

/**...*/
public class BlockingQueueOrderBoard implements OrderBoard {

  BlockingQueue<Order> orders;

  public BlockingQueueOrderBoard() {
    orders = new LinkedBlockingQueue<Order>(5);
  }

  public void postOrder(Order toBeProcessed) {
    try {
      orders.put(toBeProcessed);
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
  }

```

BlockingQueue Example [cont.]



```
    public Order cookOrder() {
        Order returnValue = null;
        try {
            returnValue = orders.take();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        return returnValue;
    }
}
```

Lab: Order Board

- **GOAL:** Refactor the order board to use a blocking queue from the concurrency library. The resulting code should be less complex, less lines of code. Yet the functionality will be exactly the same.

- **NOTE:** Keep an old copy of the order board around. It will be used in other upcoming labs.

- **DURATION:** 45 minutes
  - 30 minutes - development
  - 15 minutes - group code review.

Execution Framework

Delegate, delegate, delegate

Task Execution

- Two task execution frameworks built into Java
- `Thread` as an execution framework
  - `Runnable` becomes task
  - Thread governs when `run` is executed
  - No support for canceling, scheduled execution, etc.
- `java.util.Timer` as execution framework
  - Introduced in 1.3
  - Task represented as `TimerTask`
  - Supports canceling, fixed rate scheduling, date-based scheduling
  - No real-time timing guarantees - relies on wait mechanism

Task Execution [cont.]

- Both task execution frameworks are “functional”, but somewhat incomplete

- Generally you need a more robust execution framework that provides:
  1. Thread reuse and pooling
  2. Task scheduling
  3. Task canceling
  4. Decoupling of task registration from execution

Concurrency Execution Framework

- Part of `java.util.concurrent` package
  - Decouples task execution from `Thread` dependency
  - Supports more robust task handling
  - Implemented using Factory and command-pattern
- Built around three key concepts:
  - Tasks
  - Executors
  - Execution services

Execution Framework Tasks

- Two “tasks” in concurrency execution framework
- `java.lang.Runnable`
  - Standard `Runnable`
  - Implement `run` method
  - Don’t worry about Threading semantics

- `java.util.concurrent.Callable`
  - Similar to a `Runnable`, in concept
  - Single method to implement
    - `public V call()`
    - Can return value
    - Can throw checked exceptions

Executors

- Entities that execute tasks
- Represented by `java.util.concurrent.Executor`
  - Decouples task submission from execution
  - Does not define how `Runnable` will be executed
  - `Executor` implementation could be:
    - Dedicated single-thread based
    - Thread-pool based
    - Current-thread based
  - Does not define when `Runnable` will be executed
  - Single task submission method

    ```
    public void execute(Runnable cmd)
    ```

Execution Services

" Entities responsible for execution and management of tasks

" Two types:

1.`java.util.concurrent.ExecutorService`

"Interface extensions of `Executor`

"Adds management capabilities to `Executor`

–Supports blocking - `awaitTermination`

–Shutdown - `shutdown`

–Service monitoring - `isShutdown` / `isTerminated`

"Enhances task handling

–Submission – supports `Callable and Runnable`

1.Management – returns `Future`

Types of Execution Services [cont.]

- Two types (cont):
  1. `java.util.concurrent.ScheduledExecutorService`
     - Interface extensions of `ExecutorService`
     - Adds scheduling capabilities to `ExecutorService`
       - Supports Callable and Runnable
       - Single-schedule execution
       - Fixed-rate scheduled execution
       - Fixed-delay scheduled execution
       - No “date-based” scheduled execution

Task Management

- Every scheduled task has an associated “handle”
- Handle used for task cancellation and monitoring
- Handles are decoupled from:
  - Service - no way to get execution service reference from handle
  - Task - no way to get task reference from handle
- Two types of “handles”
  - `java.util.concurrent.Future`
  - `java.util.concurrent.ScheduledFuture`
  -

    NOTE: Handle type dependent on scheduling mechanism

Executor Implementations

- Create your own `Executor` implementation
  - Easiest way is to define a `ThreadFactory`
  - Associate it with `ThreadPoolExecutor`

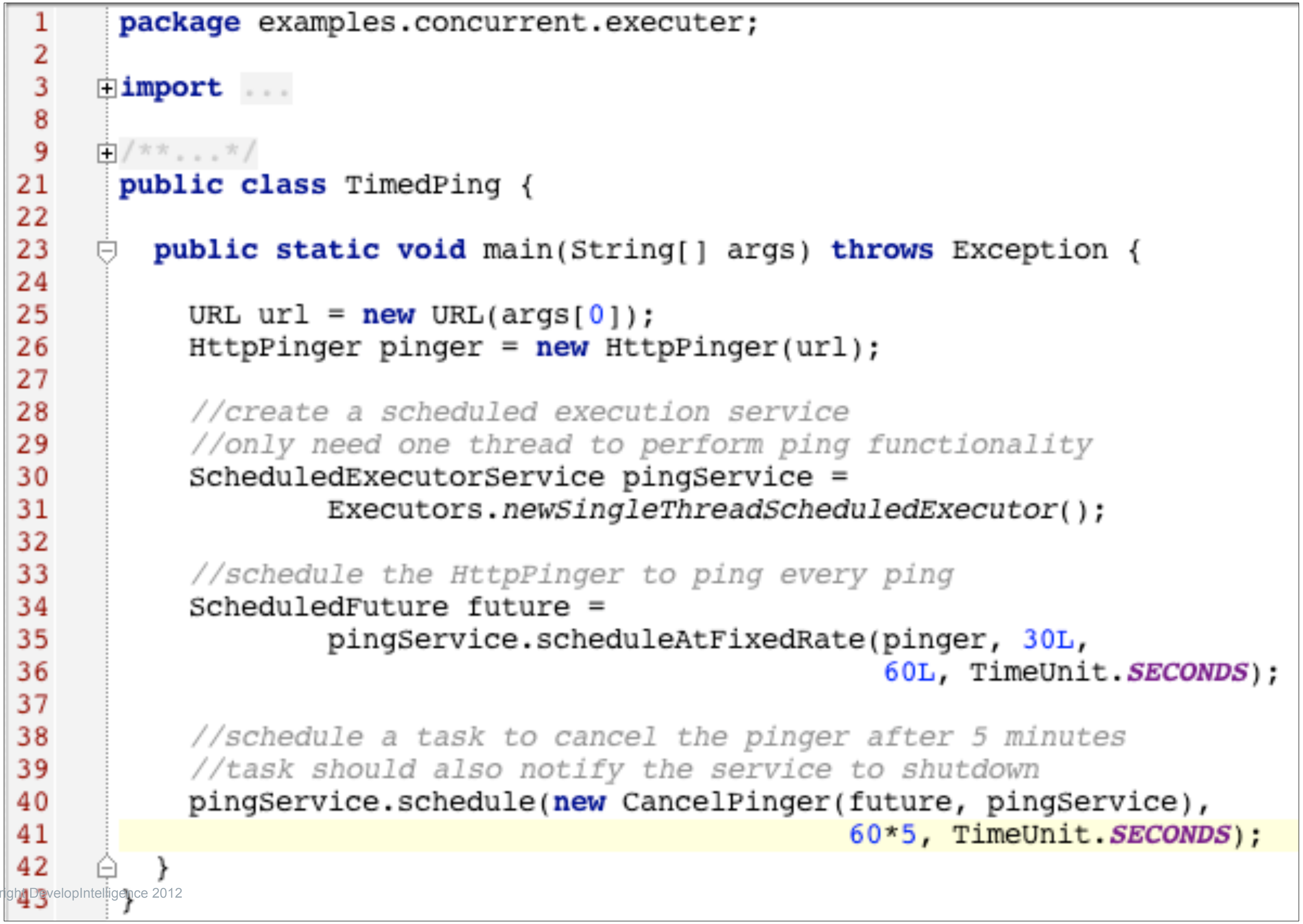
- Or utilize the built-in implementations
  - `Executors` class is a factory
    - Can be used to create `ExecutorServices`
      - Single thread
      - Cached thread pool
      - Fixed thread pool
      - Scheduled thread pool
    - Can be used to create `Callable` objects out of `Runnable` objects

Execution Framework Example

A Simplistic example

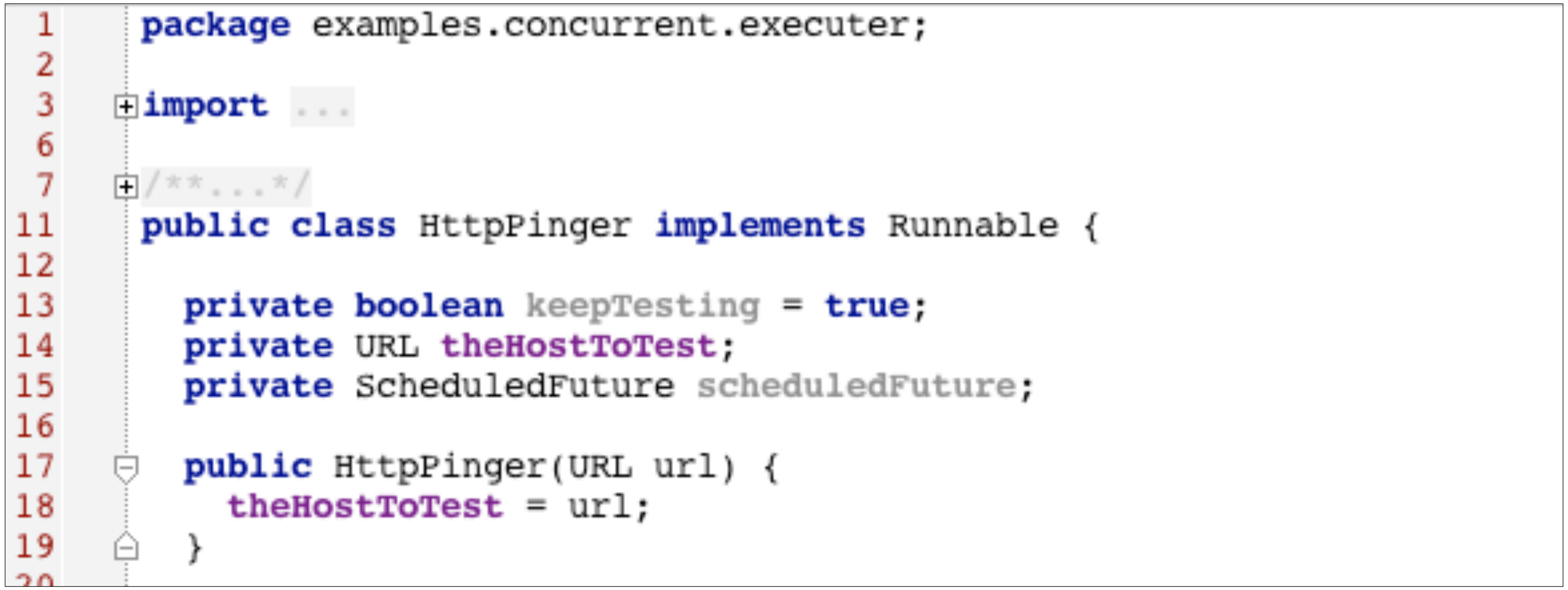
- Intended to illustrate use of an execution service
- Built around scheduled execution of a task
- Task performs HTTP ping-like functionality to determine availability of web server

Execution Framework Example



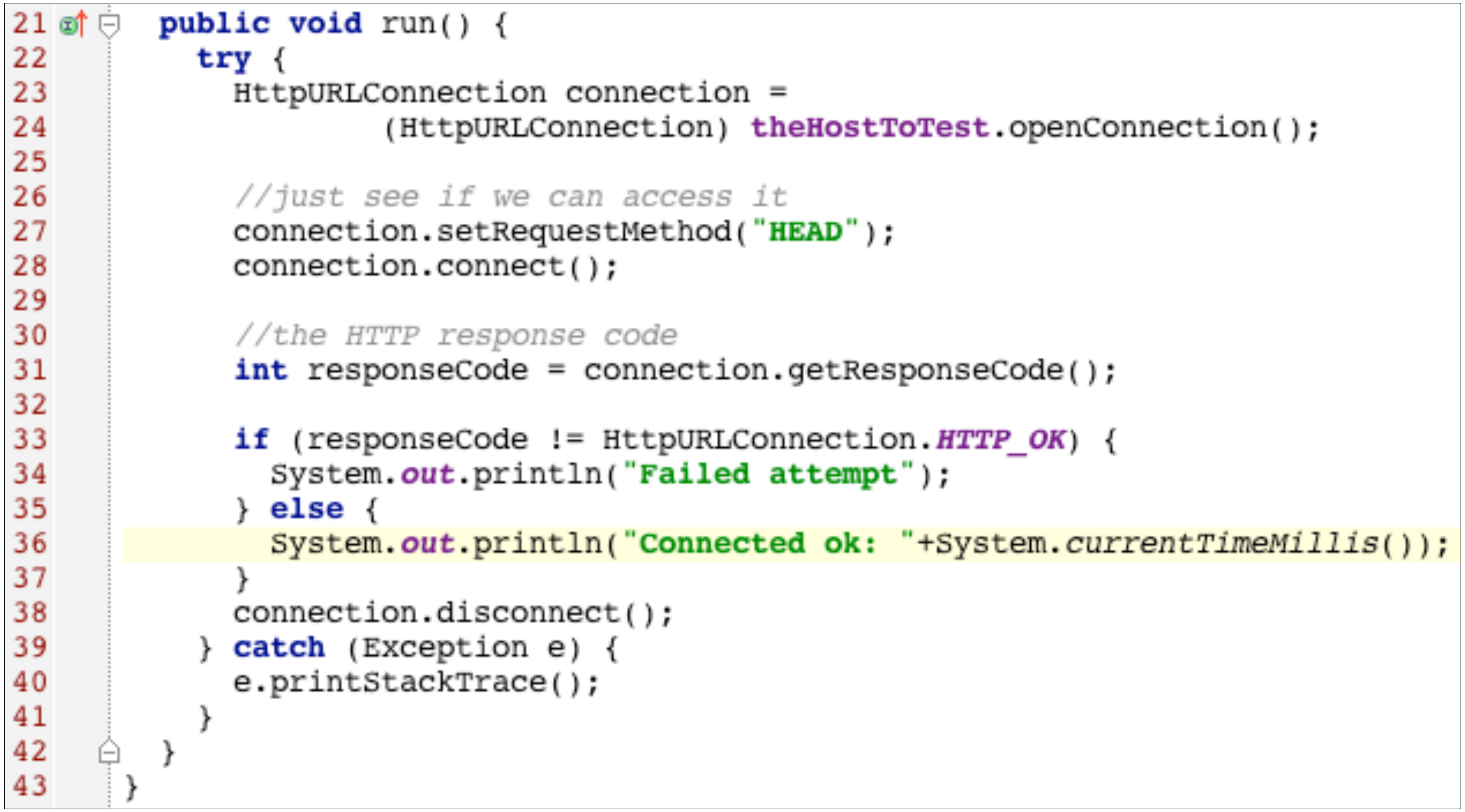
```
package examples.concurrent.executer;

import ...

/**...*/
public class TimedPing {

  public static void main(String[] args) throws Exception {

    URL url = new URL(args[0]);
    HttpPinger pinger = new HttpPinger(url);

    //create a scheduled execution service
    //only need one thread to perform ping functionality
    ScheduledExecutorService pingService =
            Executors.newSingleThreadScheduledExecutor();

    //schedule the HttpPinger to ping every ping
    ScheduledFuture future =
            pingService.scheduleAtFixedRate(pinger, 30L,
                                            60L, TimeUnit.SECONDS);

    //schedule a task to cancel the pinger after 5 minutes
    //task should also notify the service to shutdown
    pingService.schedule(new CancelPinger(future, pingService),
                                        60*5, TimeUnit.SECONDS);
  }
}
```

Exec. Framework Example [cont.]



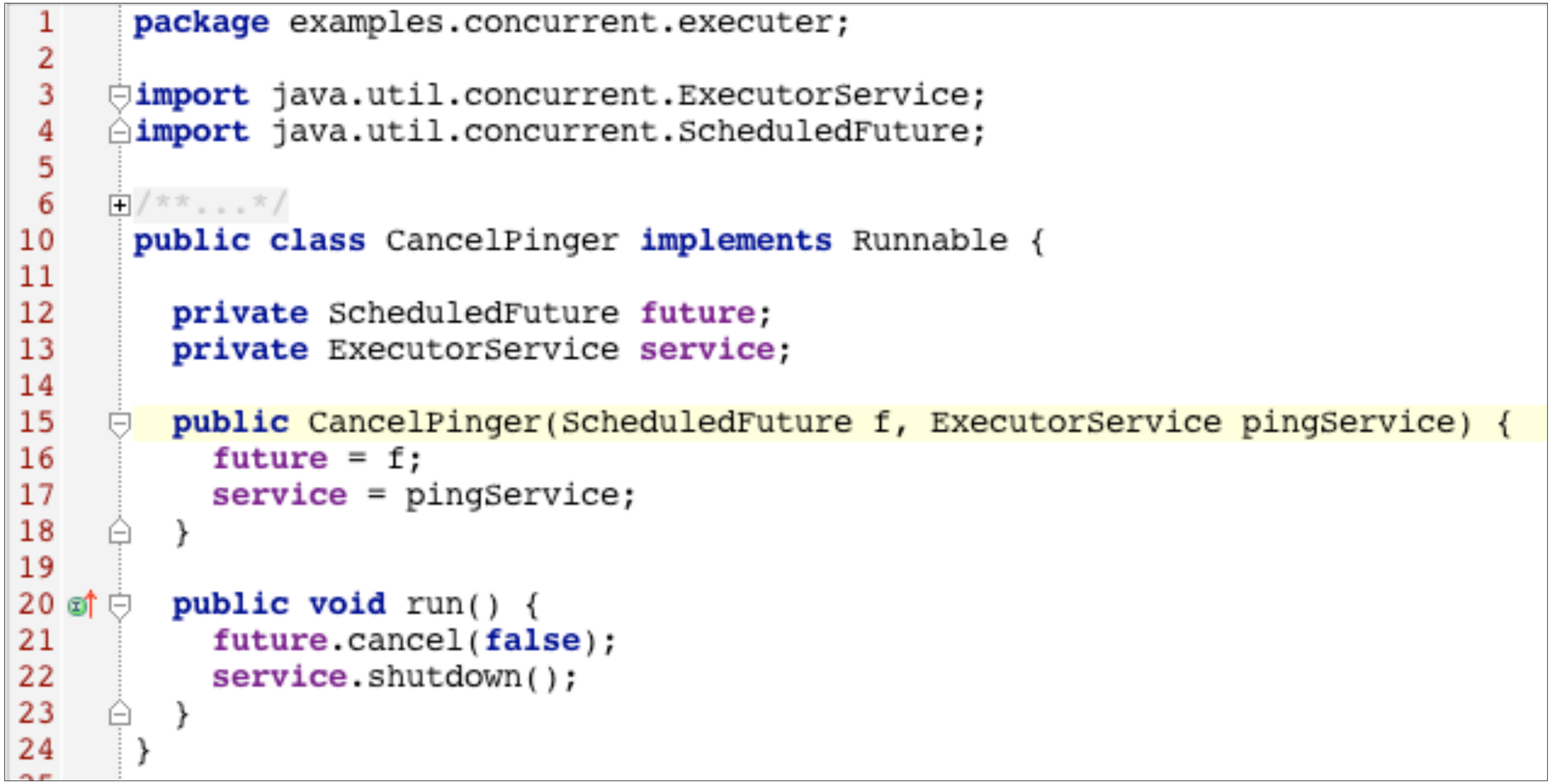
```
package examples.concurrent.executer;

import ...

/**...*/
public class HttpPinger implements Runnable {

  private boolean keepTesting = true;
  private URL theHostToTest;
  private ScheduledFuture scheduledFuture;

  public HttpPinger(URL url) {
    theHostToTest = url;
  }
```

Exec. Framework Example [cont.]



```
  public void run() {
    try {
      HttpURLConnection connection =
              (HttpURLConnection) theHostToTest.openConnection();

      //just see if we can access it
      connection.setRequestMethod("HEAD");
      connection.connect();

      //the HTTP response code
      int responseCode = connection.getResponseCode();

      if (responseCode != HttpURLConnection.HTTP_OK) {
        System.out.println("Failed attempt");
      } else {
        System.out.println("Connected ok: "+System.currentTimeMillis());
      }
      connection.disconnect();
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Exec. Framework Example [cont.]



```java
package examples.concurrent.executer;

import java.util.concurrent.ExecutorService;
import java.util.concurrent.ScheduledFuture;

/**...*/
public class CancelPinger implements Runnable {

  private ScheduledFuture future;
  private ExecutorService service;

  public CancelPinger(ScheduledFuture f, ExecutorService pingService) {
    future = f;
    service = pingService;
  }

  public void run() {
    future.cancel(false);
    service.shutdown();
  }
}
```

Lab: Task Execution

- **GOAL:** Implement the example. Allow more than one “pinger” to exist and “ping” the health of a website. After a website has been determined to be up (10 successful pings), change the frequency (the delay) from ever 30 seconds to every minute.

- **HINT:** You will need more than a single threaded executor.

- **DURATION:** 45 minutes
  - 30 minutes - development
  - 15 minutes - group code review.

Working with Synchronizers

The Great Barrier Reef

Synchronizers

" Utility classes
1. Used to help coordinate control flow of Threads
2. Have their own state to determine “go” or “wait”
3. Potential replacements for synchronization blocks

" Three broad types:
- Latches
- Barriers
- Semaphores

Latches

- Synchronizer that delays progress of threads
  - Threads are delayed until a terminal state is reached
  - Once reached, all threads can proceed

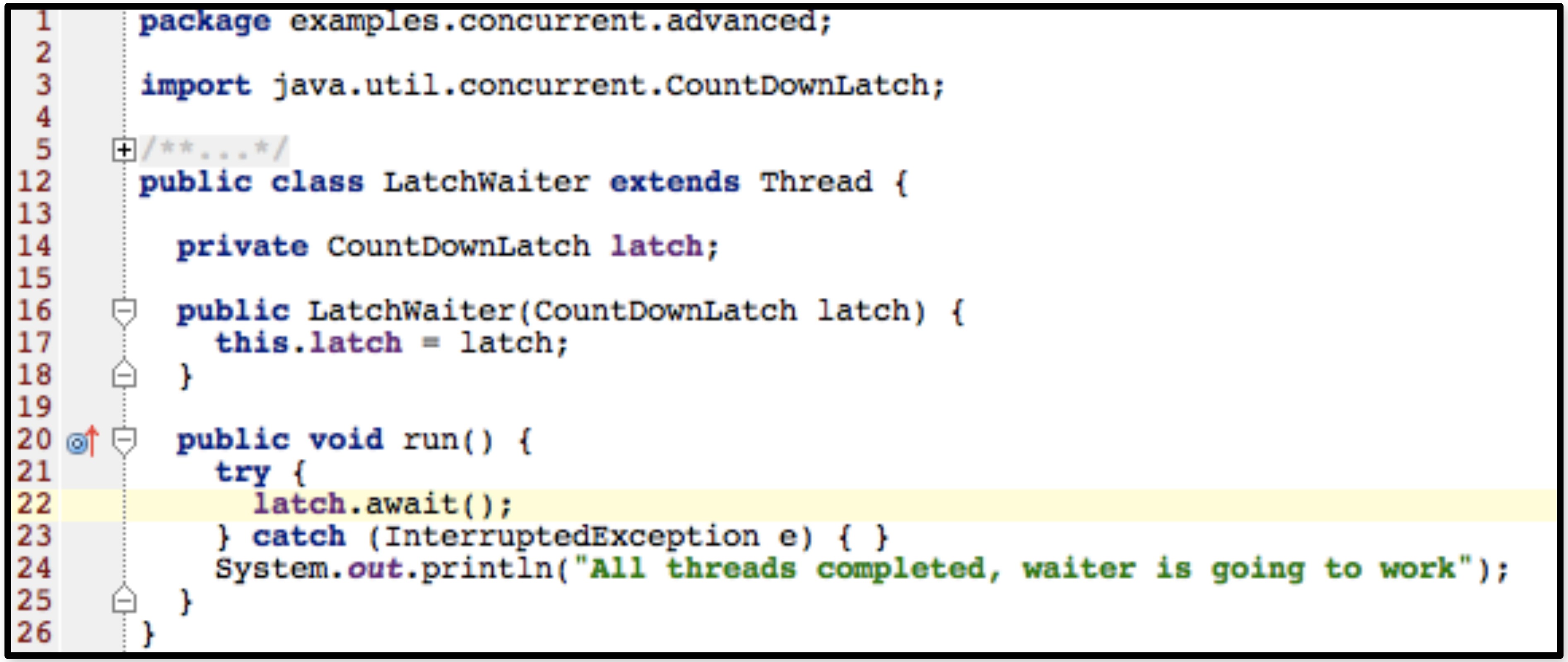
- Function like a gate:
  - When gate is closed, can't go through
  - When gate is open, can go through
  - Once gate is open, stays open

- Useful when trying to synchronize:
  - Resource initialization
  - Service startup
  - Application shutdown

CountDownLatch

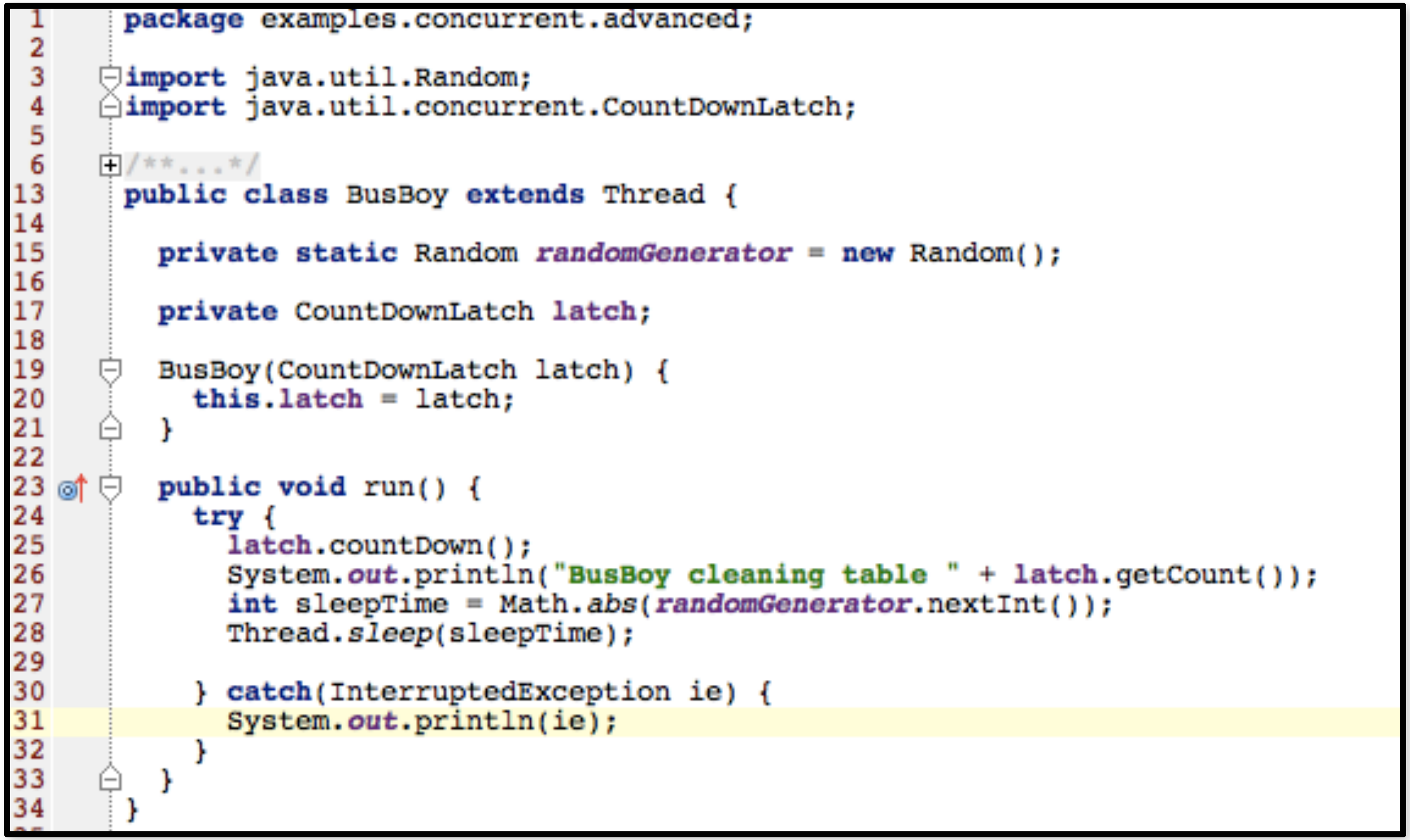
- Implementation of a latch
  - Forces threads to wait until a predefined number of “events” occur
  - Utilizes a counter
  - As events occur, counter decrements
  - When `count` becomes 0, latch is released
- Can not be reused

Latch Example : LatchWaiter



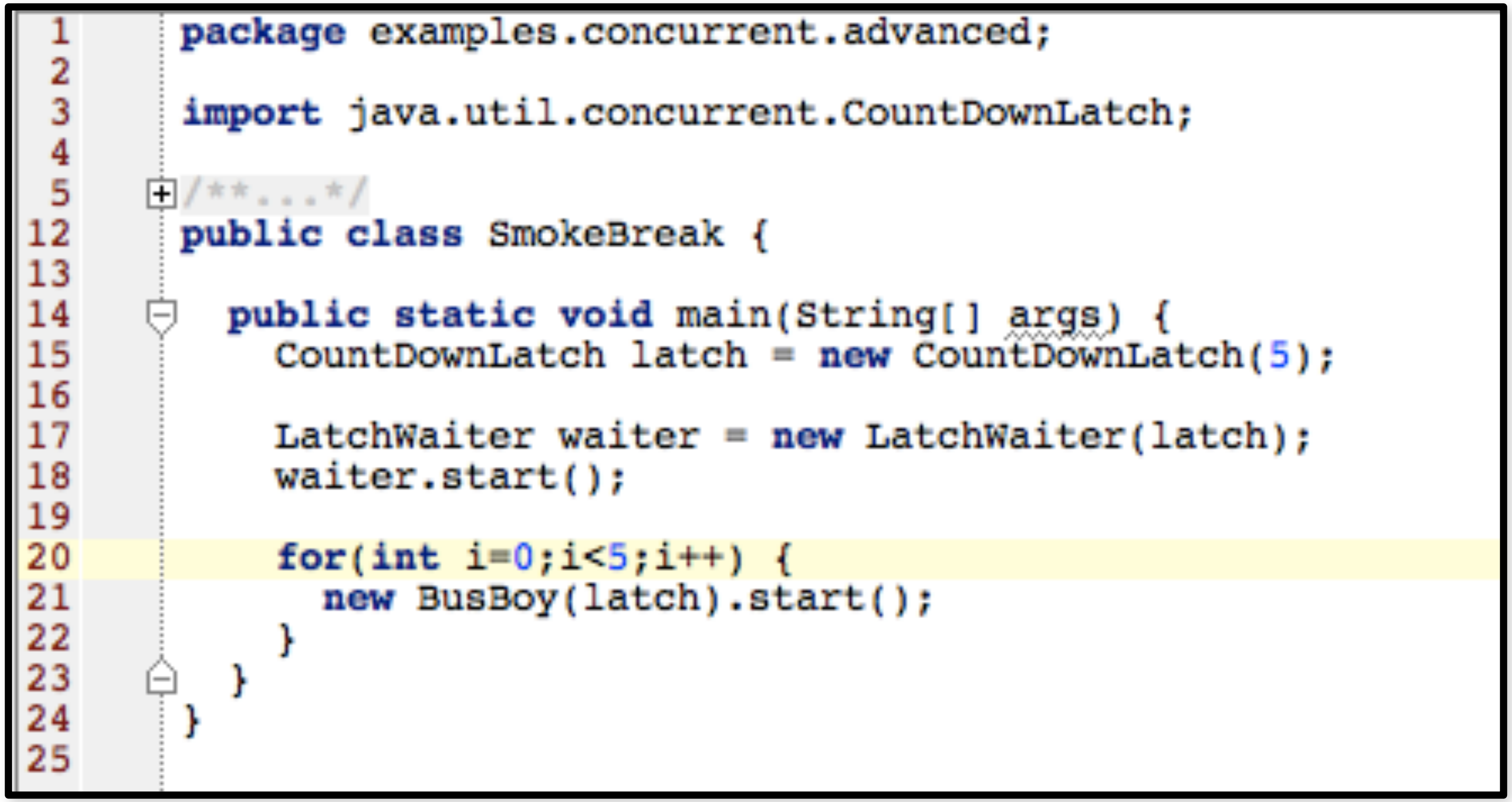
```
package examples.concurrent.advanced;

import java.util.concurrent.CountDownLatch;

/**...*/
public class LatchWaiter extends Thread {

  private CountDownLatch latch;

  public LatchWaiter(CountDownLatch latch) {
    this.latch = latch;
  }

  public void run() {
    try {
      latch.await();
    } catch (InterruptedException e) { }
    System.out.println("All threads completed, waiter is going to work");
  }
}
```

Latch Example : BusBoy(s)



```java
package examples.concurrent.advanced;

import java.util.Random;
import java.util.concurrent.CountDownLatch;

/**...*/
public class BusBoy extends Thread {

  private static Random randomGenerator = new Random();

  private CountDownLatch latch;

  BusBoy(CountDownLatch latch) {
    this.latch = latch;
  }

  public void run() {
    try {
      latch.countDown();
      System.out.println("BusBoy cleaning table " + latch.getCount());
      int sleepTime = Math.abs(randomGenerator.nextInt());
      Thread.sleep(sleepTime);

    } catch(InterruptedException ie) {
      System.out.println(ie);
    }
  }
}
```

Latch Example : SmokeBreak



```
package examples.concurrent.advanced;

import java.util.concurrent.CountDownLatch;

/**...*/
public class SmokeBreak {

  public static void main(String[] args) {
    CountDownLatch latch = new CountDownLatch(5);

    LatchWaiter waiter = new LatchWaiter(latch);
    waiter.start();

    for(int i=0;i<5;i++) {
      new BusBoy(latch).start();
    }
  }
}
```

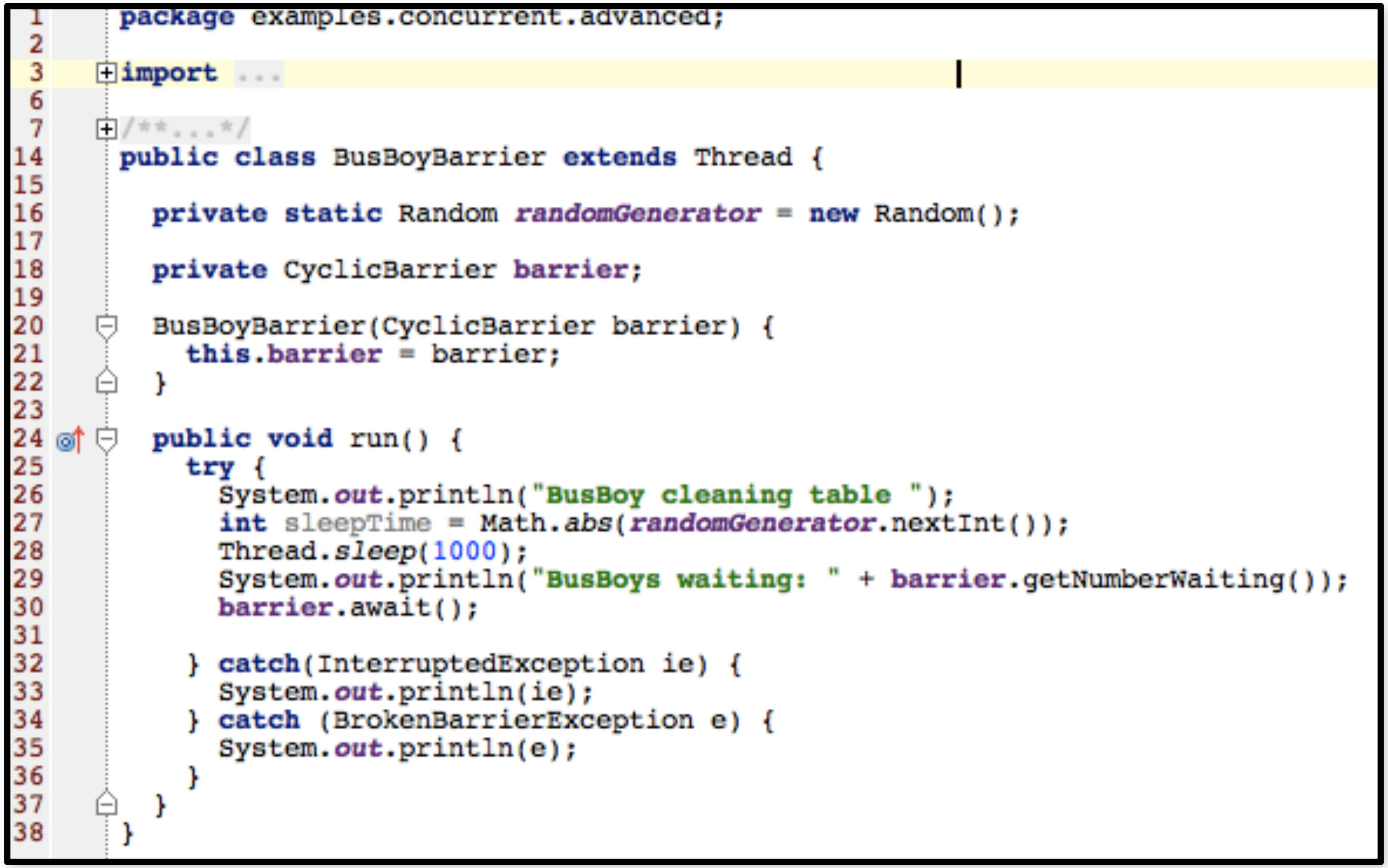
Barriers

- Block threads until some “event” occurs
  - Used to “join” groups of threads
  - All threads must reach rendezvous point at same time
  - Once all threads reach barrier, then proceed
- Threads don’t die when they reach barrier
  - Different than `Thread.join()`
  - They can continue processing
- Could be implemented using `wait` / `notify` mechanics
  - But might be messy
  - And potentially error prone

Barriers : `CyclicBarrier`

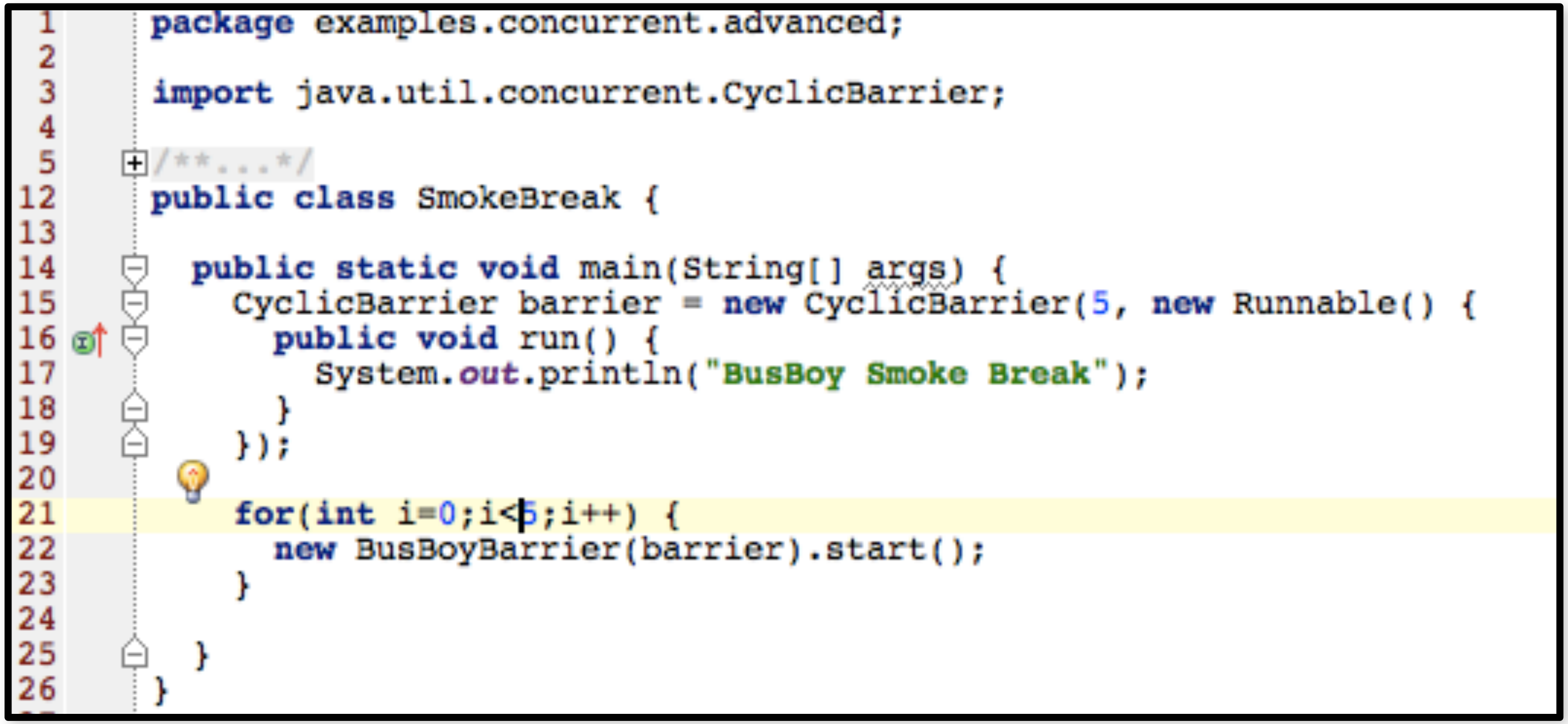
- Provides "blocking" point for threads
  - Constructed with number of threads in party
  - Each thread calls `await` when it gets to point
  - `CyclicBarrier` blocks `await` thread until all members of party arrive
  - Once all arrive, releases threads
- Can be reused - `reset` the barrier

CyclicBarrier Example : BusBoyBarrier



```
package examples.concurrent.advanced;

import ...

/**...*/
public class BusBoyBarrier extends Thread {

  private static Random randomGenerator = new Random();

  private CyclicBarrier barrier;

  BusBoyBarrier(CyclicBarrier barrier) {
    this.barrier = barrier;
  }

  public void run() {
    try {
      System.out.println("BusBoy cleaning table ");
      int sleepTime = Math.abs(randomGenerator.nextInt());
      Thread.sleep(1000);
      System.out.println("BusBoys waiting: " + barrier.getNumberWaiting());
      barrier.await();

    } catch(InterruptedException ie) {
      System.out.println(ie);
    } catch (BrokenBarrierException e) {
      System.out.println(e);
    }
  }
}
```

CyclicBarrier Example : SmokeBreak



```
package examples.concurrent.advanced;

import java.util.concurrent.CyclicBarrier;

/**...*/
public class SmokeBreak {

  public static void main(String[] args) {
    CyclicBarrier barrier = new CyclicBarrier(5, new Runnable() {
      public void run() {
        System.out.println("BusBoy Smoke Break");
      }
    });

    for(int i=0;i<5;i++) {
      new BusBoyBarrier(barrier).start();
    }

  }
}
```

Exchanger

" Barrier with data passing semantics

" Used with two threads
1. Meet at rendezvous point
2. Once there, `exchange` data

    ? Continue on processing

Semaphore

Formalization of counting semaphore

1. Counting associated with set number of permits

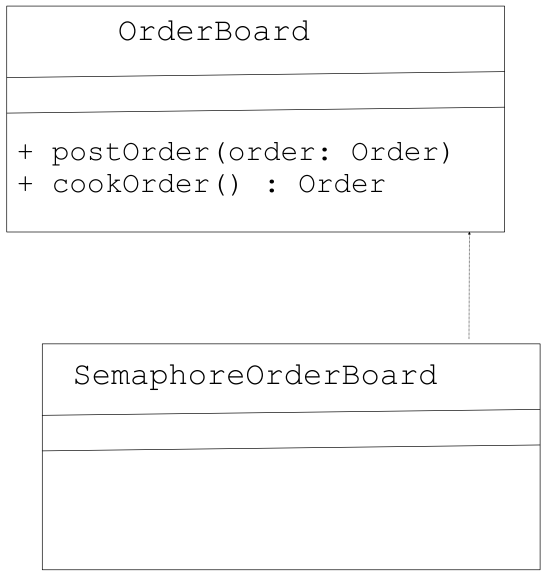
" `Semaphore` with one permit is considered a mutex

" Removes counting semantics found in many synchronization techniques

Semaphore [cont.]

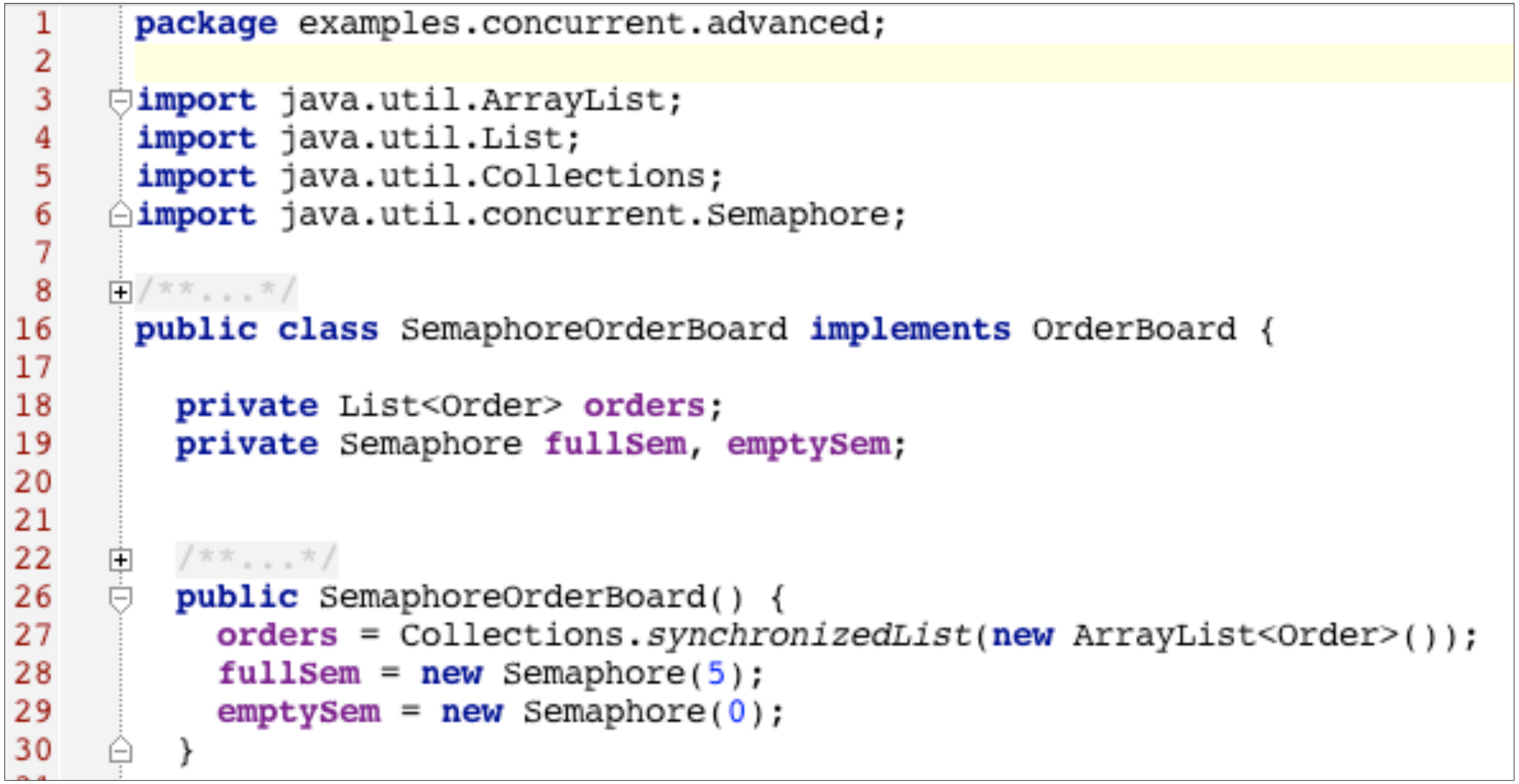
- Permits provide access control
  - Initialized to the number of resources it controls
  - Two key methods:
    - `acquire`
      - Decreases the number of available permits
      - Will wait if no permits available
    - `release` - increases number of available permits
  - Thread can hold more than one permit
  - Permit can be released by non-holding thread

OrderBoard Redesign



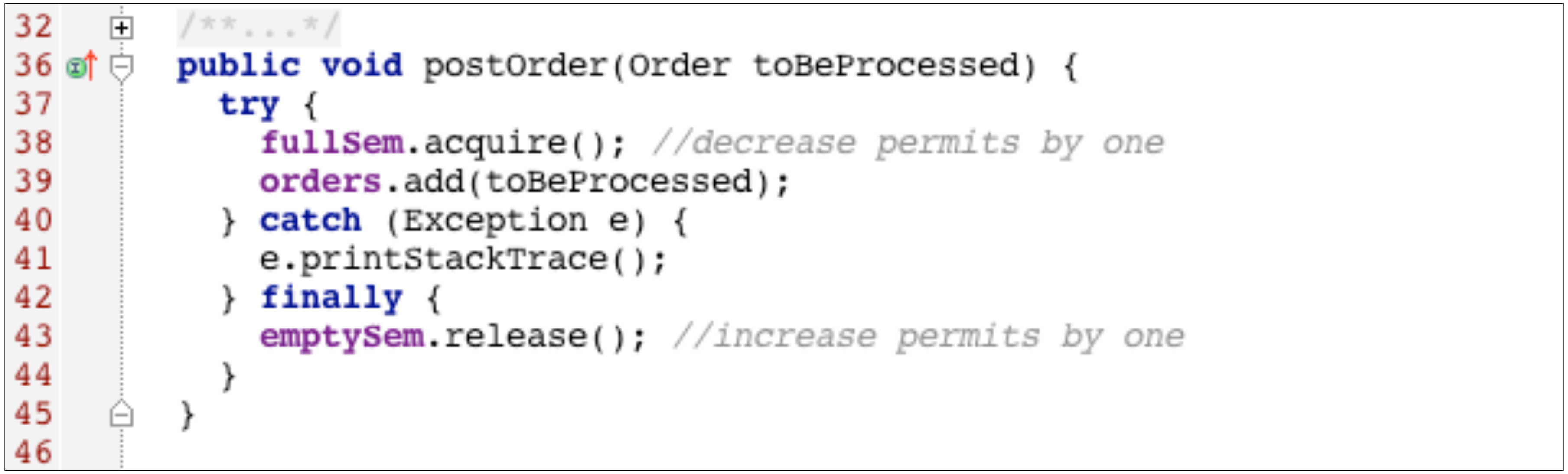
- `SemaphoreOrderBoard`
  1. Implementation of `OrderBoard`
  - Uses two Semaphores
    - `fullSem` - initialized with 5 permits
    - `emptySem` - initialized with 0 permits
  - No synchronization - uses synchronized list
  - No queue empty / full management
  - Simplifies original `OrderBoard`

Semaphore Example



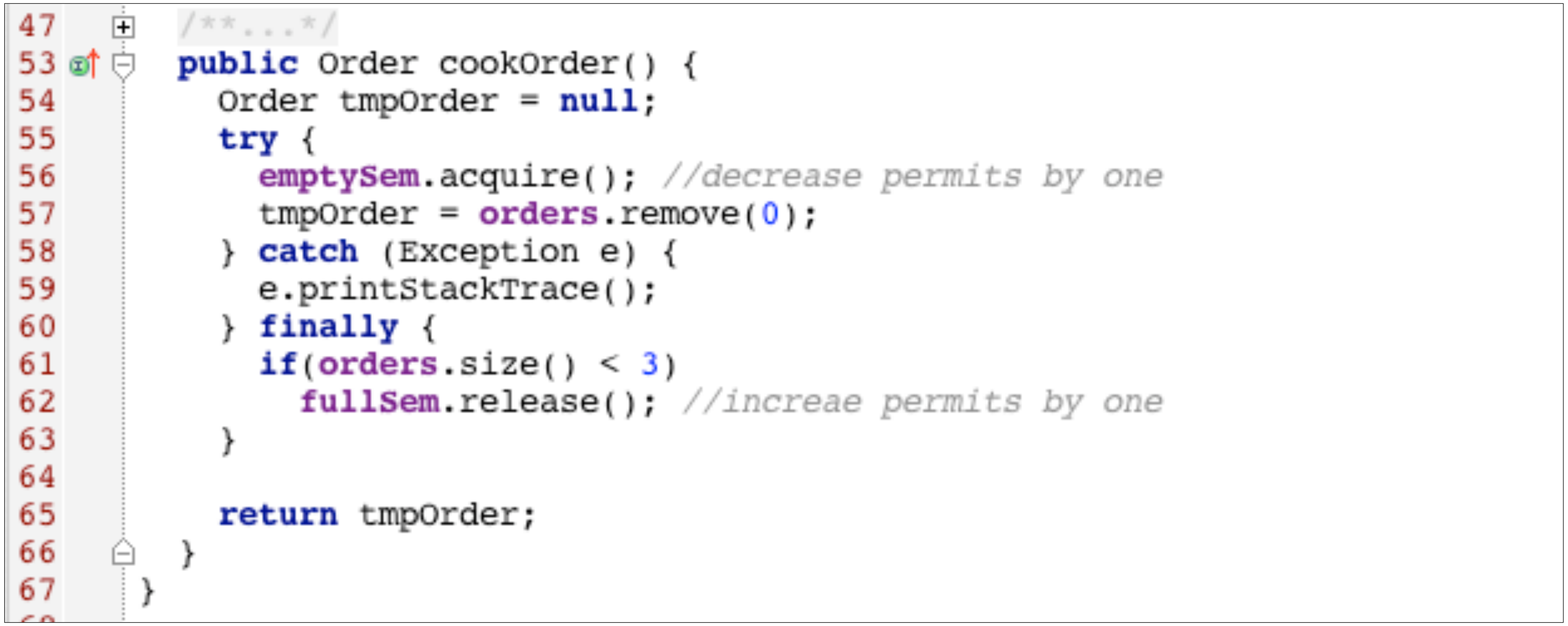
```
package examples.concurrent.advanced;

import java.util.ArrayList;
import java.util.List;
import java.util.Collections;
import java.util.concurrent.Semaphore;

/**...*/
public class SemaphoreOrderBoard implements OrderBoard {

   private List<Order> orders;
   private Semaphore fullSem, emptySem;


   /**...*/
   public SemaphoreOrderBoard() {
      orders = Collections.synchronizedList(new ArrayList<Order>());
      fullSem = new Semaphore(5);
      emptySem = new Semaphore(0);
   }
```

Semaphore Example [cont.]



```
/**...*/
public void postOrder(Order toBeProcessed) {
   try {
      fullSem.acquire(); //decrease permits by one
      orders.add(toBeProcessed);
   } catch (Exception e) {
      e.printStackTrace();
   } finally {
      emptySem.release(); //increase permits by one
   }
}
```

Semaphore Example [cont.]



```
/**...*/
public Order cookOrder() {
    Order tmpOrder = null;
    try {
        emptySem.acquire(); //decrease permits by one
        tmpOrder = orders.remove(0);
    } catch (Exception e) {
        e.printStackTrace();
    } finally {
        if(orders.size() < 3)
            fullSem.release(); //increae permits by one
    }

    return tmpOrder;
}
}
```

Lab: Rewrite Order Board

- **GOAL:** Refactor the order board to use a Semaphore concurrency library. The resulting code should be less complex, less lines of code. Yet the functionality will be exactly the same.

- **NOTE:** Keep an old copy of the order board around. It will be used in other upcoming labs.

- **DURATION:** 45 minutes
  - 30 minutes - development
  - 15 minutes - group code review.

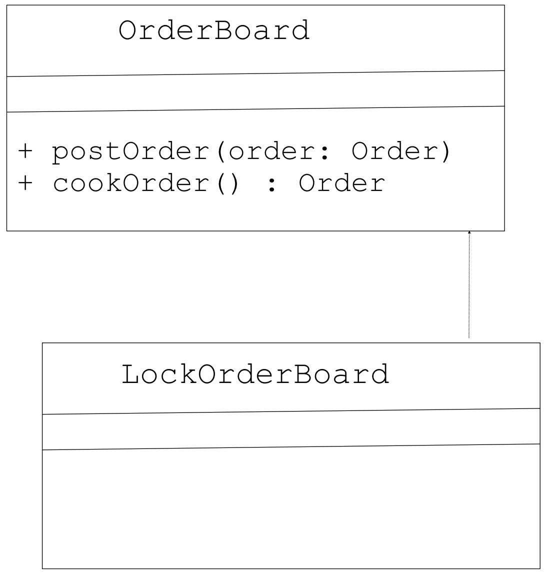
Other Concurrency Packages

java.util.concurrent.locks

- Locking and waiting condition framework
  - Alternative to monitor lock mechanism
  - Provides greater flexibility
    - Locks do not require synchronized blocks
    - Locks support re-entrance and fairness policies
    - Locks have multiple conditions
    - Waiting based on condition not “object lock”

- Key components:
  - `java.util.concurrent.locks.Lock`
  - `java.util.concurrent.locks.Condition`
  - `java.util.concurrent.locks.ReentrantLock`

OrderBoard Redesign



" `LockOrderBoard`

1. Implementation of `OrderBoard`

" Uses one Lock

"`ReentrantLock`

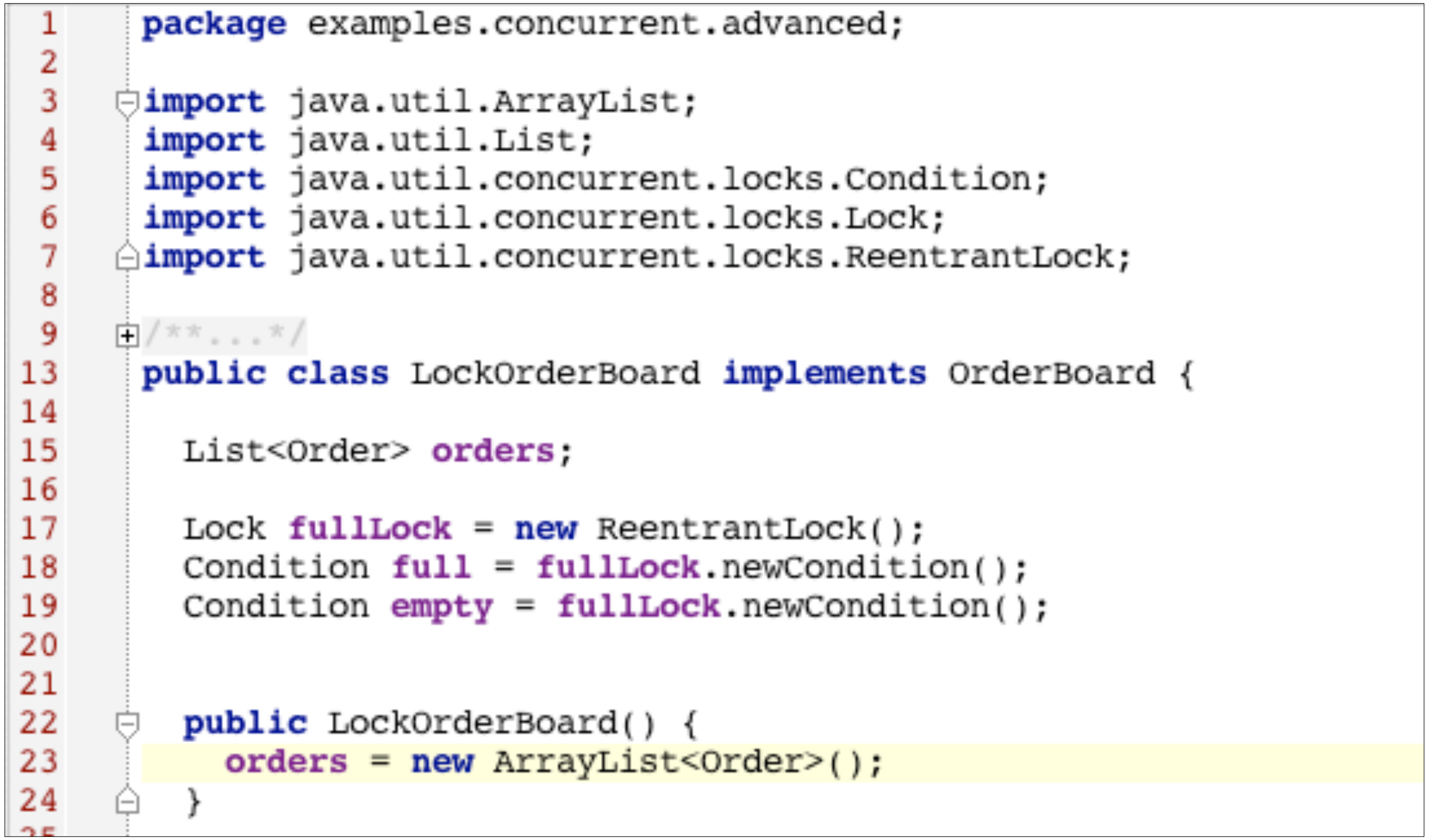
"Used to synchronize access to orders list

" Access controlled by two conditions

1. `full`
2. `empty`

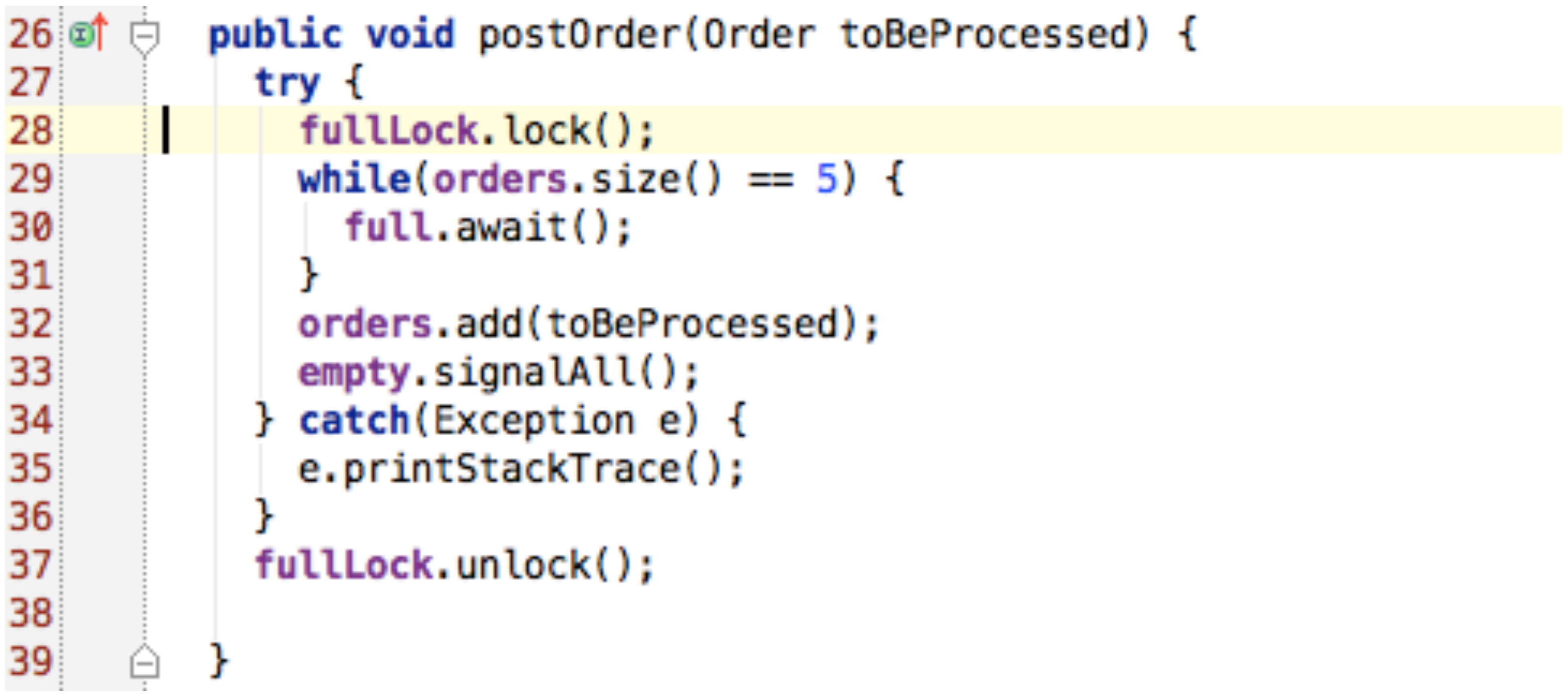
" Queue empty / full management

Lock Example



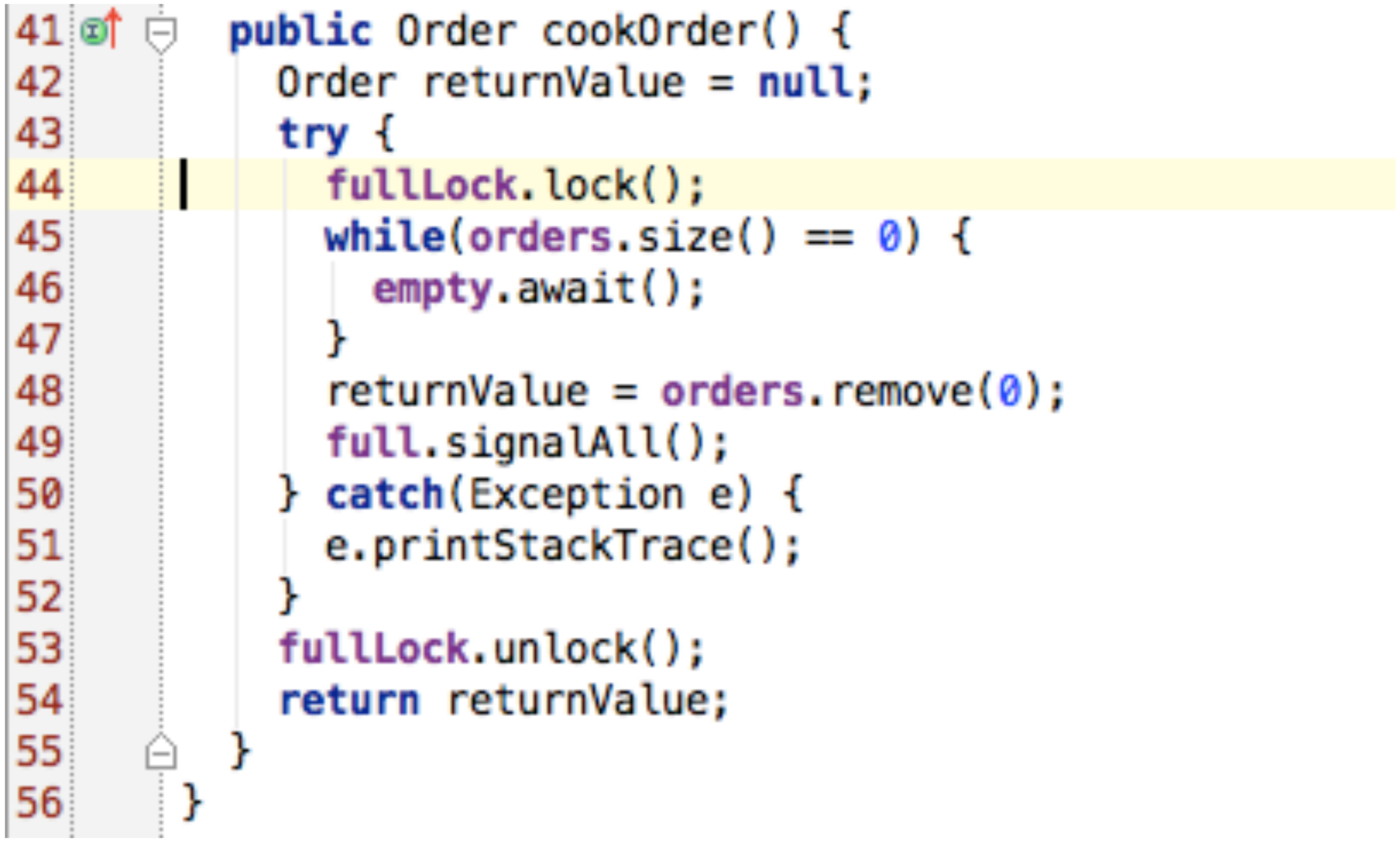
```
package examples.concurrent.advanced;

import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

/**...*/
public class LockOrderBoard implements OrderBoard {

  List<Order> orders;

  Lock fullLock = new ReentrantLock();
  Condition full = fullLock.newCondition();
  Condition empty = fullLock.newCondition();


  public LockOrderBoard() {
    orders = new ArrayList<Order>();
  }
```

Lock Example [cont.]



```
public void postOrder(Order toBeProcessed) {
    try {
        fullLock.lock();
        while(orders.size() == 5) {
            full.await();
        }
        orders.add(toBeProcessed);
        empty.signalAll();
    } catch(Exception e) {
        e.printStackTrace();
    }
    fullLock.unlock();

}
```

Lock Example [cont.]



```
public Order cookOrder() {
  Order returnValue = null;
  try {
    fullLock.lock();
    while(orders.size() == 0) {
      empty.await();
    }
    returnValue = orders.remove(0);
    full.signalAll();
  } catch(Exception e) {
    e.printStackTrace();
  }
  fullLock.unlock();
  return returnValue;
}
}
```

java.util.concurrent.atomic

- Toolkit of classes
  1. Provide atomic manipulation of variables
  2. Uses lock-free thread-safe implementation

- Extends `volatile` using `compareAndSet` functionality
  1. Relies on enhancements made to JVM
  2. Take advantage of compare-and-swap or load-linked/store-condition hardware based operations
  3. Foundational for entire concurrency package

- May or may not be used at application development level

Summary

" Java SE 5.0 exponentially expands concurrent programming in Java
" Concurrency libraries provide standardization to common concurrent problems
" Concurrency libraries provide:
    1. Concurrent collections
    2. Synchronizers
    3. Locks
    4. Atomic wrappers

Advanced Concurrency Lab

" **GOAL**: Create a basic site map creation utility.

The site map creation utility should generate a text file containing all of the domain-specific URLs found on a given website. Associated with each domain, should be a indicator denoting whether the URL was accessible. The site map utility does not need to track images, only <a href> tags.

Use multiple threads to make the utility efficient. There should be one thread that finds <a href>s, and one thread that processes <a href>s.

When all of the <a href>s are processed, the application should generate a text file and then shut-down gracefully.

" **HINTS:** A map could be used to represent the “indexed” site. A queue could be used as the shared resource representing the unindexed / untested elements of the site.

Advanced Concurrency Lab

- **DURATION:** 120 minutes
  - 100 minutes – development (consider pair programming)
  - 20 minutes – group code review