Европейски социален фонд
Инвестиции в хората

Проект „Повишаване квалификацията на служителите от администрацията на централно ниво чрез усъвършенстване на знанията и практическите им умения за управление на софтуерни ИТ проекти в съответствие със съвременните методологии“, осъществяван с финансовата подкрепа на Оперативна програма „Административен капацитет” (ОПАК), съфинансирана от Европейския съюз, чрез Европейския социален фонд”, съгласно Договор № К13-22-1/05.03.2014 г.

НАРЪЧНИК

Дейност 5.
ПРОВЕЖДАНЕ НА ОБУЧЕНИЕ ЗА JAVA ПРОГРАМИСТИ ЗА 33 СЛУЖИТЕЛИ НА ЦЕНТРАЛНАТА АДМИНИСТРАЦИЯ И ИЗДАВАНЕ НА СЕРТИФИКАТИ ЗА ПРОВЕДЕНОТО ОБУЧЕНИЕ

Изготвен в изпълнение на Договор № Д-37/11.12.2014 г.
между
МИНИСТЕРСТВО НА ТРАНСПОРТА, ИНФОРМАЦИОННИТЕ ТЕХНОЛОГИИ И СЪОБЩЕНИЯТА
и
„КОНСОРЦИУМ ИТ ОБУЧЕНИЯ 2015“ ДЗЗД

„КОНСОРЦИУМ ИТ ОБУЧЕНИЯ 2015“ ДЗЗД
София 1040, ж.к. Изток, бул. Драган Цанков 36, СТЦ Интерпред, блок А, ет.6; тел: 024210040; имейл: ittraining2015@newhorizons.bg;

Авторски колектив:

Надежда Крачунова-лектор
Тихомир Тотев, Александър Камбуров – асистент-лектори
Ива Колева, Петър Пешев - експерти

Одобрил: Николай Пенев – ръководител на проекта

София, 2015 г.

Съдържание

Речник на термините 10
Java Development Kit (JDK) 12
Java Runtime Environment (JRE) 12
Java Virtual Machine (JVM) 12
Java Integrated Development Environment (IDE) 13
Въведение 13
  Какво е обект? 15
  Какво е клас? 16
  Какво е наследяване? 17
  Какво е интерфейс? 18
  Какво е пакет? 19
Интерфейси и наследяване 20
  Интерфейси 20
    Дефиниране на интерфейс 21
    Тяло на интерфейс 22
    Ползване на интерфейс като тип 22
    Методи по подразбиране 22
  Наследяване 22
    Абстрактни методи и класове 23
    Абстрактни класове и интерфейси 23
Модификатори за достъп 24
  Модификатор за достъп по подразбиране - без ключова дума 24
  Частен модификатор за достъп - private 25
  Публичен модификатор за достъп - public 25
  Защитен модификатор за достъп - protected 26
Hello World приложение 26
  Коментари в изходния код 26
  HelloWorldApp клас дефиниция 27

main метода ... 28
Streams на System класа ... 29
System.in ... 29
System.out ... 29
System.err ... 29
Основи на езика ... 29
Променливи ... 30
Наименоване ... 31
Примитивни типове данни ... 31
Масиви ... 35
Резюме на променливи ... 40
Оператори ... 40
Присвояване, аритметични и унарни оператори ... 41
Равенство, сравняващо и условни оператори ... 45
Побитови оператори ... 47
Резюме на операторите ... 48
Изрази, statements и блокове ... 49
Statements за контрол на потока ... 51
Statements if-then и if-then-else ... 51
Switch statement ... 53
While и do-while statements ... 58
for statement ... 59
Разклоняващи се statements ... 61
Резюме на statements за контрол на потока ... 64
Анотации ... 65
Формат на анотация ... 65
Къде могат да бъдат ползвани анотациите? ... 66
Деклариране на annotation type ... 66
Предефинирани типове анотации ... 68
Типове анотации ползвани от езика Java ... 68

Анотации, прилагани към други анотации ... 69

Типове анотации и pluggable type системи ... 69

Повторяеми анотации ... 70

Стъпка 1: Декларирайте повторяем тип анотация ... 71

Стъпка 2: Декларация на тип анотация-контейнер ... 71

Получаване на анотации ... 71

Съображения относно дизайна ... 72

Generics ... 72

Generic типове ... 72

Прост Box клас ... 73

Generic версия на Box класа ... 73

Конвенция за наименоване на типови параметри ... 73

Извикване и инстанцииране на generic тип ... 74

Диамантът (diamond) ... 74

Множество типови параметри ... 74

Праметъризирани типове ... 75

Collections ... 75

Какво представлява Collections Framework? ... 76

Интерфейси ... 76

Guava Collections ... 78

Изключения ... 78

Какво е изключение? ... 78

Изискване за хващане или специфициране ... 79

Трите вида изключения ... 79

Прихващане и обработка на изключения ... 80

Reflection ... 81

Употреба на Reflection ... 81

Concurrency ... 82

Процеси ... 82

Нишки ... 82

Нишкови обекти ... 82

Дефиниране на начална нишка 82
Паузиране изпълнението чрез sleep 83
Прекъсвания 84
Joins 85
Пример за проста нишка 86
Синхронизация 88
Java API за XML обработка (JAXP) 88
JDBC(TM) достъп до бази данни 89
Java EE 89
История 90
Стандарти и спецификации 90
Основни APIs 90
java.servlet.* 90
javax.websocket.* 91
javax.faces.* 91
javax.faces.component.* 91
javax.el.* 91
javax.enterprise.inject.* 91
javax.enterprise.context.* 91
javax.ejb.* 91
javax.validation.* 91
javax.persistence.* 92
javax.transaction.* 92
javax.security.auth.message.* 92
javax.enterprise.concurrent.* 92
javax.jms.* 92
javax.batch.api.* 92
javax.resource.* 93
Полезни връзки 93
JNDI 93
Архитектура 94
Пакетиране 94
Създаване на Initial Context 94

Looking up на обект 94
Java Servlet 95
Какво е сървлет? 95
Servlet Lifecycle 95
Боравене със servlet lifecycle събития 95
Дефиниране на listener клас 95
Работа със сървлет грешки 96
Създаване и инициализация на сървлет 97
Филтриране на requests и responses 97
Програмни филтри 98
JavaServer Pages 99
Какво е JSP страница? 99
Прост пример за JSP страница 99
Транслация и компилация 101
JavaServer Faces 102
Какво представлява JavaServer Faces приложение? 103
Създаване на просто JavaServer Faces приложение 104
Разработка на managed bean 104
Създаване на уеб страница 105
Mapping на FacesServlet инстанция 105
Lifecycle на hello приложение 105
Bean Validation 107
Enterprise Beans 109
Какво е Enterprise Bean? 109
Ползи от Enterprise Beans 109
Кога да ползваме Enterprise Beans 109
Типове Enterprise Beans 110
Interceptors 110
Persistence 111

Въведение в ORM 111

Какво е ORM? 113

Управление на entities 113

EntityManager интерфейсът 113

Заявки 115

Дескриптори за разгръщане 115

Contexts and Dependency Injection (CDI) 115

Фундаментални услуги 116

Произход 116

Web Services 116

Simple Object Access Protocol (SOAP) 117

WSDL 119

Критика 119

REpresentational State Transfer (REST) 120

WADL 122

Критика 122

Java API for XML Web Services (JAX-WS) 123

Създаване на прост Web Service и клиенти с JAX-WS 123

Изисквания към JAX-WS Endpoint 124

Създаване на service endpoint имплементиращ клас 125

Изграждане, пакетиране и разгръщане на услуга 125

Просто JAX-WS клиентско приложение 125

Писане на клиентско приложение 125

Стартиране на клиентското приложение 126

Прост JAX-WS уеб клиент 126

Писане на сървлет 126

Стартиране на уеб клиента 128

Поддържани от JAX-WS типове 128

Schema-to-Java mapping 129

Java-to-schema mapping 130

RESTful Web Services с JAX-RS 130

Създаване на RESTful root resource class 130

Разработка на RESTful уеб услуги с JAX-RS 131

Преглед на JAX-RS приложение 132

@Path анотацията и URI path templates 133

Отговарящи на HTTP методи и заявки 135

Request method designator анотации 135

Употреба на entity providers за мапинг на HTTP response и request entity bodies 136

Употреба на @Consumes и @Produces за къстъмизиране на заявки и отговори 138

@Produces анотацията 138

@Consumes анотацията 139

Извличане на request параметри 140

Spring Framework 143

Речник на термините

API	Application Programming Interface
CDI	Context and Dependency Injection
CLI	Command-Line Interface
CORBA -	Common Object Request Broker Architecture
COS	Common Object Service
DOM	Document Object Model
DTD	Document Type Definition
EAI	Enterprise Application Integration
EIS	Enterprise Information Systems
EJB	Enterprise JavaBeans
EL	Expression Language
FIFO	First-In First-Out
GC	Garbage Collector
GUI	Graphical User Interface
HTML	HyperText Markup Language
HTTP	Hypertext Transfer Protocol
IDE	Integrated Development Environment
IO	Input/Output
IoC	Inversion Of Control
IPC	Inter Process Communication
J2EE	Java Enterprise Edition
JAR	Java Archive
JASPIC	Java Authentication Service Provider Interface for Containers
Java EE	Java Enterprise Edition
Java SE	Java Standard Edition
JAX-RS	Java API for RESTful Web Services
JAX-WS	Java API for XML Web Services
JAXB	Java Architecture for XML Binding
JAXP	Java API for XML Processing
JCA	Java EE Connector Architecture
JCP	Java Community Process
JDBC	Java Database Connectivity
JDK	Java Development Kit
JMS	Java Message Service
JNDI	Java Naming and Directory Interface
JPA	Java Persistence API
JPQL	Java persistence Query Language
JRE	Java Runtime Environment
JSF	JavaServer Faces
JSON	JavaScript Object Notation
JSP	JavaServer Pages
JSR	Java Specification Request
JTA	Java Transaction API
JVM	Java Virtual Machine
LDAP	Lightweight Directory Access Protocol

LIFO	Last-In First-out
MIME	Multi-Purpose Internet Mail Extensions
OO	Object-Oriented
OOP	Object-Oriented Programming
ORM	Object-Relational Mapping
POJO	Plain Old Java Object
RAML	RESTful API Modeling Language
RDBMS	Relational Database Management System
REST	REpresentational State Transfer
RMI	Remote Method Invocation
RMI-IIOP	RMI over IIOP
RPC	Remote Procedure Calls
SAX	Simple API for XML Parsing
SDK	Software Development Kit
SEI	Service Endpoint Implementation
SMTP	Simple Mail Transfer Protocol
SOAP	Simple Object Access Protocol
SPI	Service Provider Interface
SQL	Structured Query Language
StAX	Streaming API for XML
SVG	Scalable Vector Graphics
TCP	Transmission Control Protocol
UI	User Interface
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
VM	Virtual Machine
W3C	World Wide Web Consortium
WADL	Web Application Description Language
WAR	Web application ARchive
WML	Wireless Markup Language
WS	Web Service
WSDL	Web Service Description/Definition Language
xHTML	Extensible Hypertext Markup Language
XML	EXtensible Markup Language
XSL	EXtensible Stylesheet Language

Java Development Kit (JDK)

Java Development Kit съдържа инструменти, необходими за разработката на Java програми, включително Java Runtime Environment (JRE) за изпълнението им. Инструментите включват компилатор (javac), Java application launcher (java), Appletviewer (за изпълняване на аплети) и др. Компилаторът конвертира Java програмен код в bytecode. Java application launcher отваря JRE, зарежда клас и извиква main метода.

Ако желаете да пишете собствени програми и съответно да ги компилирате, ви е необходимо JDK. За да изпълнявате Java програми JRE би било достатъчно. JRE е предназначено за изпълняване на Java файлове, като включва JVM и runtime libraries.

След компилация на даден Java source файл, изходът не е ".exe". Резултатът от процеса бива ".class" файл, представляващ Java bydecode, разбираем за JVM. Виртуалната машина интерпретира байт кода в машинен такъв, в зависимост от прилежащата операционна система и хардуер под нея.

Java Runtime Environment (JRE)

Java Runtime Environment съдържа Java Virtual Machine (JVM), клас библиотеки и други поддържащи файлове. Не съдържа никакви инструменти за разработка като компилатор, дебъгер и т.н. JVM изпълнява програмата, ползвайки клас библиотеките и други поддържащи файлове, предоставени от JRE. Ако желаете да стартирате Java програма ви е необходимо инсталирано JRE.

Java Virtual Machine (JVM)

Java виртуалната машина предоставя платформено-независим подход за изпълняване на код. По този начин програмистите могат да се съсредоточат върху писането на софтуер, без да се концентрират върху това, как и къде ще бъде изпълняван той. Обърнете внимание обаче, че самата JVM не е платформено независима. Целта й е да помага изпълнението на Java софтуер по платформено независим начин. Когато JVM трябва да интерпретира bytecode към машинен код, тогава трябва да използва native или зависим от операционната система език за да взаимодейства със системата.

JVM се нарича виртуална, защото предоставя машинен интерфейс, който не зависи от операционната система и хардуерната архитектура. Тази независимост е ключова по отношение на write-once run-anywhere (напиши-веднъж изпълни-навсякъде) стойността на Java програмите.

JDK предоставя една или повече имплементации на Java виртуалната машина (VM):

- На платформи, обикновено ползвани за клиентски приложения, JDK идва с VM имплементация наречена **Java HotSpot Client VM** (*client VM*). Клиентската VM е настроена за редуцирано време за старт и memory footprint. Може да бъде извикана чрез употребата на -client command-line опцията при стартиране на приложение.
- На всички платформи JDK идва с имплементация на Java виртуалната машина наречена **Java HotSpot Server VM** (*server VM*). Сървърната виртуална машина е разработена за максимална скорост на изпълнение на програми. Може да бъде извикана, ползвайки -server command-line опцията при стартиране на приложение.

http://docs.oracle.com/javase/8/docs/technotes/guides/vm/

http://www.javabeat.net/what-is-the-difference-between-jrejvm-and-jdk/

Java Integrated Development Environment (IDE)

Представлява софтуерно приложение, подпомагащо потребителите в по-лесното писане и дебъгване на програми. Много IDEs предоставят функционалности като syntax highlighting (подчертаване на синтаксиса) и code completion (завършване на кода), улеснявайки писането на код.

Масово наложени IDEs:

https://www.jetbrains.com/idea/download/

https://eclipse.org/downloads/packages/eclipse-ide-java-ee-developers/lunar

https://netbeans.org/downloads/

http://en.wikibooks.org/wiki/Java_Programming/Java_IDEs

Въведение

КОНЦЕПЦИИ ПРИ ОБЕКТО-ОРИЕНТИРАНОТО ПРОГРАМИРАНЕ

Секцията разглежда концепциите зад обекто-ориентираното програмиране: обекти, съобщения, класове и наследяване. Приключва с демонстриране прилагането на тези концепции и транслирането им в програмен код.

ОСНОВИ НА ЕЗИКА

Секцията описва традиционните характеристики на езика, включая променливи, масиви, типове данни, оператори и контрол на потока.

КЛАСОВЕ И ОБЕКТИ

Текущата секция описва писането на класове, създаването на обекти от тях и използването им.

АНОТАЦИИ

Анотациите са форма на метаданни, предоставяща информация за компилатора. Секцията описва къде и как да използваме ефективно анотации при програмирането.

ИНТЕРФЕЙСИ И НАСЛЕДЯВАНЕ

Секцията описва интерфейси - какво представляват, защо и как бихме желали да пишем и ползваме такива. Описва още начина, по който един клас може да произхожда от друг. Разглежда как подклас (subclass) може да наследява полета (fields) и методи (methods) на суперкласа (superclass). Ще научите, че всички класове произлизат от Object класа. Ще можете да модифицирате методи, които подкласа наследява от суперкласа.

NUMBERS И **STRINGS**

Текущата секция описва употребата на Number и String обектите. Те подобряват т.нар. type safety на кода, правейки потенциалните бъгове по-явни за засичане по време на компилация.

GENERICS

Представляват мощен инструмент в Java, подпомагат type safety.

ПАКЕТИ

Програмният език Java предоставя пакети, подпомагащи организацията и структурирането на класовете и техните взаимовръзки един с друг.

Концепции при обекто-ориентираното програмиране

Ако никога преди не сте ползвали език за обекто-ориентирано програмиране, ще бъде необходимо да научите няколко основни концепции преди да започнете писането на код. Тази секция въвежда в понятията обекти, класове, наследяване, интерфейси и пакети. Всяка точка се фокусира върху приложението на течи концепции, същевременно въвеждайки в синтаксиса (syntax) на програмния език Java.

КАКВО Е ОБЕКТ (**OBJECT**)?

Обект представлява софтуерна единица, обединяваща състояние и поведение. Софтуерните обекти често се ползват за моделиране на обекти от истинския свят и ежедневието. Секцията обяснява как състоянието и поведението биват репрезентирани в рамките на обект, въвежда концепцията за енкапсулация на данните (data encapsulation) и обяснява ползите от подобен тип софтуерен дизайн.

КАКВО Е КЛАС (**CLASS**)?

Клас представлява чертеж или прототип, от който се създават обекти. Секцията описва дефинирането на клас, моделиращ състоянието и поведението на обект от истинския свят. Умишлено се фокусира върху основите, показвайки как дори прост клас може чисто да моделира състояние и поведение.

КАКВО Е НАСЛЕДЯВАНЕ (**INHERITANCE**)?

Наследяването предоставя мощен и натурален механизъм за организиране и структуриране на софтуер. Тази секиция обяснява как класовете наследяват състояние и поведение от техните суперкласове. Обяснява още как даден клас произлиза от друг, ползвайки предоставения за целта синтаксис от езика.

КАКВО Е ИНТЕРФЕЙС (**INTERFACE**)?

Интерфейсът е договор между даден клас и външния свят. Когато клас имплементира интерфейс, се ангажира с предоставяне поведението, публикувано от този интерфейс. Секцията дефинира прост интерфейс и обяснява необходимите промени за даден клас, който го имплементира.

КАКВО Е ПАКЕТ (**PACKAGE**)?

Пакет представлява пространство (namespace) за организиране на класове и интерфейси по логически критерии. Позиционирането на кода в пакети прави поддръжката на големи софтуерни проекти по-лесна. Секцията обяснява защо това е полезно и въвежда в Application Programming Interface (API), предоставено от Java платформата.

Какво е обект?

Обектите са ключ към разбирането на обектно-ориентираната технология. Огледайте се и ще забележите множество примери за обекти от заобикалящия ни свят: вашето куче, бюро, телевизор, велосипед.

Обектите от истинския свят споделят две характеристики: Всички те имат състояние и поведение. Кучетата имат състояние (име, цвят, порода, дали са гладни) и поведение (лаене, хващане, мятане на опашка). Велосипедите също имат състояние (текуща предавка, текущ ритъм на педалите, текуща скорост) и поведение (сменяне на предавка, смяна ритъма на въртене на педалите, натискане на спирачки). Идентифицирайки състоянието и поведението на обектите от истинския свят е чудесен начин да се започне мисленето в парадигмите на обекто-ориентираното програмиране.

Отделете минута да наблюдавате обектите от истинския свят, намиращи се около вас. За всеки обект който видите, си задайте два въпроса: "Какви възможни състояние може да има този обект?" и "Какви възможни действия може да изпълнява този обект?". Запишете вашите наблюдения. Ще забележите, че обектите от истинския свят варират в сложност; вашата настолна лампа може да има само две възможни състояния (включена и изключена) и две възможни поведения (включи, изключи). Вашето настолно радио обаче може да има допълнителни състояния (включено, изключето, височина на звука, текуща станция) и поведение (включи, изключи, увеличи звука, намали звука, търси, сканирай, настрой). Може би ще забележите също, че някои обекти съдържат други обекти. Тези наблюдения от истинския свят се транслират в света на обекто-ориентираното програмиране.

Софтуерните обекти са концептуално сходни с тези на обектите от истинския свят: те се състоят от състояние и свързано поведение. Обектът съхранява неговото състояние в полета (fields; в някои други езици за програмиране се наричат променливи или variables). Обектът разкрива поведението си чрез методи (methods; в някои езици за програмиране се наричат функции или functions). Методите оперират върху вътрешното състояние на обекта и обслужват първичния механичъм за комуникация обект-към-обект. Скриването на вътрешното състояние и изискването всяко взаимодействие да бъде извършено през методи на обекта е познато като енкапсулация на данни (data encapsulation) - фундаментален принцип от обекто-ориентираното програмиране.

Чрез атрибутно състояние (текуща скорост, текущ ритъм на въртене на педалите, текуща предавка) и предоставяйки методи за промяна на състоянието, обектът притежава контрол върху това как на външния свят е позволено да го използва. Например, ако велосипед има 6 скорости, даден метод за промяна на предавката може да отхвърли всяка стойност по-малка от 1 и по-голяма от 6.

Обединяването на код в индивидуални софтуерни обекти предоставя редица преимущества, включващи:

1. Модуларност - Сорс кодът за даден обект може да бъде написан и поддържан независимо от кода на други обекти. Веднъж създаден, даден обект може лесно да бъде предаван в рамките на системата.
2. Скриване на информация - Чрез взаимодействие само с методите на обекта, детайлите за неговата вътрешна имплементация остават скрити за външния свят.
3. Преизползване на кода - Ако даден обект вече съществува (може би написан от друг софтуерен разработчик), можете да го ползвате и във вашата програма. Това позволява на специалистите да имплементират/тестват/дебъгват комплексни, строго специализирани обекти, на които може да се разчита за изпълнения във вашия собствен код.

4. Възможност за подмяна (pluggability) и улесняване дебъгването - Ако определен обект се окаже проблематичен, можете просто да го изтриете от вашето приложение и да го подмените с друг обект. Това е аналог на поправяне на механичен проблем в истинския свят. Ако даден болт се счупи, то само той бива подменен, а не цялата машина.

Какво е клас?

В истинския свят са често срещани индивидуални обекти от един вид. Може да съществуват хиляди велосипеди, всички от един и същи производител и модел. Всеки велосипед е конструиран от същия комплект чертежи, следователно съдържа същите компоненти. В обекто-ориентираната терминология казваме, че този велосипед е инстанция (instance) на клас от обекти, познат като велосипеди. Клас (class) представлява чертеж, от който индивидуалните обекти биват създадени.

Следния Bicycle клас е една възможна имплементация на велосипед:

```
class Bicycle {
   int cadence = 0;
   int speed = 0;
   int gear = 1;
   void changeCadence(int newValue) {
      cadence = newValue;
   }
   void changeGear(int newValue) {
      gear = newValue;
   }
   void speedUp(int increment) {
      speed = speed + increment;
   }
   void applyBrakes(int decrement) {
      speed = speed - decrement;
   }
   void printStates() {
      System.out.println("cadence:" +
         cadence + " speed:" +
         speed + " gear:" + gear);
   }
}
```

Синтаксисът на езика за програмиране Java ще ви изглежда нов, но дизайна на този клас се базира върху предишната дискусия за обекти тип велосипед. Полетата cadance (ритъм на въртене на педалите), speed (скорост) и gear (предавка) репрезентират състоянието на обекта. Методите (changeCadance, changeGear, speedUp и т.н.) дефинират неговото взаимодействие с външния свят.

Може би ще забележите, че клас Bicycle не съдържа main метод. Това се дължи на факта, че горния пример не представлява завършено приложение. Това е само чертеж за велосипеди, който бихме искали да използваме в приложението. Отговорността за създаването и ползването на нови Bicycle обекти принадлежи на някои други класове в приложението.

Ето това е клас BicycleDemo, създаващ два отделни Bicycle обекта и извикващ техните методи:

```
class BicycleDemo {
```

```
    public static void main(String[] args) {

        // Create two different
        // Bicycle objects
        Bicycle bike1 = new Bicycle();
        Bicycle bike2 = new Bicycle();

        // Invoke methods on
        // those objects
        bike1.changeCadence(50);
        bike1.speedUp(10);
        bike1.changeGear(2);
        bike1.printStates();

        bike2.changeCadence(50);
        bike2.speedUp(10);
        bike2.changeGear(2);
        bike2.changeCadence(40);
        bike2.speedUp(10);
        bike2.changeGear(3);
        bike2.printStates();
    }
}
```

Изходът принтиране крайния ритъм на въртене на педалите, скорост и предавка за двата велосипеда:

```
cadence:50 speed:10 gear:2
cadence:40 speed:20 gear:3
```

Какво е наследяване?

Различните видове обекти обикновено имат определено количество сходства един с друг. Планинските велосипеди (mountain bikes), пригодените за път такива (road bikes) и двуместни такива (tandem bikes) например - всички споделят характеристиките на велосипед (текуща скорост, текущ ритъм на педалите, текуща предавка). Същевременно всеки от тях дефинира допълнителни характеристики, различаващи го от останалите - двуместните велосипеди имат две седалки и две кормила; пригодените за път имат ниско кормило; някои планински велосипеди имат допълнително колело за веригата, даващи по-ниска предавка.

Обекто-ориентираното програмиране позволява на класовете да наследят (inherit) общоизползваните състояния и поведения от други класове. Например, Bicycle сега става суперклас (superclass) на MountainBike, RoadBike и TandemBike. В програмния език Java за всеки клас е позволено да има един директен суперклас, като даден суперклас има потенциала за неограничен брой подкласове.

Синтаксисът за създаване на подклас е прост. В началото на вашата клас декларация ползвайте ключовата дума extends, последвана от името на класа който искате да наследите:

```
class MountainBike extends Bicycle {
```

```
    // new fields and methods defining
    // a mountain bike would go here

}
```

Това дава на MountainBike всички полета и методи на Bicycle, същевременно позволявайки на неговия код да се фокусира ексклузивно върху характеристиките, правещи го уникален сам по себе си. Това прави кода на вашите подкласове лесен за четене. Въпреки това трябва да се погрижите за документирането на състояние и поведение, дефинирани от даден суперклас - след като кодът няма да се появи в изходния код на подкласа.

Какво е интерфейс?

Както вече научихте, обектите дефинират собственото си взаимодействие с външния свят чрез методите, които излагат. Методите формират обектната интерфейсна повърхност с външния свят; бутоните на телевизора, например са интерфейс между вас и електрониката от другата страна на кутията. Натискате бутона за пускане за да включите/изключите телевизора.

В най-разпространената си форма, интерфейс представлява група от свързани методи с празни тела. Поведението на велосипед, специфицирано като интерфейс, може да изглежда по следния начин:

```
interface Bicycle {

    //  wheel revolutions per minute
    void changeCadence(int newValue);

    void changeGear(int newValue);

    void speedUp(int increment);

    void applyBrakes(int decrement);
}
```

За да имплементира този интерфейс, името на вашия клас би се променило (на определена марка велосипеди, например ACMEBicycle), като бихте ползвали ключовата дума *implements* в декларацията на класа:

```
class ACMEBicycle implements Bicycle {

    int cadence = 0;
    int speed = 0;
    int gear = 1;

   // The compiler will now require that methods
   // changeCadence, changeGear, speedUp, and applyBrakes
   // all be implemented. Compilation will fail if those
   // methods are missing from this class.

    void changeCadence(int newValue) {
         cadence = newValue;
```

```
    }

    void changeGear(int newValue) {
         gear = newValue;
    }

    void speedUp(int increment) {
         speed = speed + increment;
    }

    void applyBrakes(int decrement) {
         speed = speed - decrement;
    }

    void printStates() {
         System.out.println("cadence:" +
             cadence + " speed:" +
             speed + " gear:" + gear);
    }
}
```

Имплементацията на интерфейс позволява на даден клас да бъде по-формален относно поведението, което обещава да предостави. Интерфейсите формират договор между класовете и външния свят, като този договор се прилага по време на build от компилатора. Ако класът претендира да имплементира интерфейс, то всички методи дефинирани от интерфейса трябва да се появят в сорс кода на класа - за да се компилира той успешно.

Забележка*:* За да успеете да компилирате *ACMEBicycle* класа*,* ще трябва да добавите ключовата дума *public* в началото на имплементацията на интерфейсните методи. Ще научите причините за това в последваща секция.

Какво е пакет**?**

Пакет е пространство (namespace) за организация на набор от свързани класове и интерфейси. Концептуално можете да разглеждате пакетите като подобие на различни папки от файловата система на вашия компютър. Може да съпоставите с HTML страници в дадена папка, картинки в друга, сприптове или приложения в трета. Тъй като софтуерът, написан на езика за програмиране Java може да бъде композиран от стотици или хиляди индивидуални класове, има логика те да бъдат организирани чрез позиционирането на свързани класове и интерфейси в пакети.

Java платформата предоставя огромна клас библиотека (class library, представляващо набор от пакети), подходяща за използване във вашите приложения. Тази библиотека е позната като “Application Programming Interface” или за краткост “API”. Пакетите от нея репрезентират най-често асоциирани с общото програмиране (general-purpose programming) задачи. Например, String обект съдържа състояние и поведение за низове от символи; File обект позволява на програмиста лесно да създава, изтрива, инспектира, сравнява или модифицира файл върху файловата система; Socket обект позволява създаването и употребата на мрежови сокети; различните GUI обекти контролират бутони, чекбоксове и всички останало, свързано с графичния потребителски интерфейс. Съществуват буквално хиляди класове от които може да се избере. Това ви позволява като програмист да се

фокусирате върху дизайна на вашето собствено приложение, вместо върху инфраструктурата, необходима за да го накарате да проработи.

Спецификацията на Java Platform API съдържа пълен списък за всички пакети, интерфейси, класове, полета и методи, предоставени от Java SE платформата. Заредете страницата във вашия браузър и я сложете в bookmarks. Като програмист това ще се превърне в най-важната референтна документация за вас.

https://docs.oracle.com/javase/tutorial/java/concepts/index.html

Интерфейси и наследяване

Интерфейси

Има редица ситуации в софтуерното инженертство в които е важно различни групи програмисти да постигнат съгласие върху “договор” (contract), който определя взаимодействието на техния софтуер. Всяка група трябва да може да пише своя код без знание за начина на реализация кода на другата група. Общо казано интерфейсите представляват подобен договор.

В програмния език Java interface представлява референтен тип, подобен на клас, който съдържа само константи, сигнатури на методи, методи по подразбиране, статични методи и вложени (nested) типове. Телата на методите съществуват само за статични такива или методи по подразбиране. Интерфейсите не могат да бъдат инстанциирани - те могат да бъдат имплементирани от класове или наследявани от други интерфейси.

Дефинирането на интерфейс е подобно на създаването на нов клас:

```
public interface OperateCar {
   // constant declarations, if any

   // method signatures

   // An enum with values RIGHT, LEFT
   int turn(Direction direction,
            double radius,
            double startSpeed,
            double endSpeed);
   int changeLanes(Direction direction,
                   double startSpeed,
                   double endSpeed);
   int signalTurn(Direction direction,
                  boolean signalOn);
   int getRadarFront(double distanceToCar,
                     double speedOfCar);
   int getRadarRear(double distanceToCar,
                    double speedOfCar);
        ......
   // more method signatures
}
```

Обърнете внимание, че сигнатурите на методите нямат къдрави скоби и биват терминирани с “;”.

За реалната употреба на интерфейс се пише клас, който го имплементира. Когато класът имплементира интерфейса, той предоставя тяло на методите, дефинирани в него, например:

```
public class OperateBMW760i implements OperateCar {

   // the OperateCar method signatures, with implementation --
   // for example:
   int signalTurn(Direction direction, boolean signalOn) {
      // code to turn BMW's LEFT turn indicator lights on
      // code to turn BMW's LEFT turn indicator lights off
      // code to turn BMW's RIGHT turn indicator lights on
      // code to turn BMW's RIGHT turn indicator lights off
   }

   // other members, as needed -- for example, helper classes not
   // visible to clients of the interface
}
```

В горния пример автомобилните производители ще имплементират интерфейса. Имплементацията на Chevrolet ще бъде значително различна от тази на Toyota, но и двата производителя ще ползват същия интерфейс. Следователно производителите на системи за навигация ще изградят своите продукти, базирайки ги на GPS данните от местоположението на колата, дигитални карти и данни за трафика. По този начин системите за навигация ще извикват методите на интерфейса: turn, change lanes, brake, accelerate и т.н.

Дефиниране на интерфейс

Декларацията на интерфейс се състои от модификатори, ключовата дума interface, името на интерфейса, списък от родителски интерфейси отделени със запетая (ако има такива), и тялото на интерфейса.

Пример:

```
public interface GroupedInterface extends Interface1, Interface2, Interface3 {

   // constant declarations

   // base of natural logarithms
   double E = 2.718282;

   // method signatures
   void doSomething (int i, double x);
   int doSomethingElse(String s);
}
```

Модификатора за достъп public индикира, че интерфейсът може да бъде достъпван от всеки клас в който и да е пакет. Ако не специфицирате изрично интерфейса като публичен, тогава той би бил достъпен само от класове, дефинирани в същия пакет.

Даден интерфейс може да наследява други интерфейси, респективно класове-подкласове. Докато един клас може да наследява само един друг такъв, интерфейсите могат да наследяват произволна бройка интерфейси.

Тяло на интерфейс

Тялото на интерфейса може да съдържа абстрактни методи, методи по подразбиране и статични методи. Всеки абстрактен метод в рамките на интерфейс завършва с “;”, но няма къдрави скоби (абстрактните методи не включват имплементация). Методите по подразбиране се дефинират с default модификатора, а статичните такива ползват ключовата дума static. Всички абстрактни, методи по подразбиране и статични такива в даден интерфейс имплицитно биват public, така че можете да изпуснете този модификатор.

В допълнение, даден интерфейс може да съдържа декларации на константи. Всички константни стойности, дефинирани в рамките на интерфейс, са имплицитно public, static и final. И тук можете да изпуснете тези модификатори.

Ползване на интерфейс като тип

Когато дефинирате нов интерфейс, вие дефинирате нов референтен (reference) тип данни. Можете да ползвате името на интерфейса където можете да ползвате който и да е друг тип данни. Ако дефинирате референтна променлива, чийто тип е даден интерфейс, всеки обект който присвоявате трябва да бъде инстанция на клас, имплементиращ този интерфейс.

Методи по подразбиране

Методите по подразбиране (default methods) имат за цел да осигурят имплементация по подразбиране. По този начин надграждането на интерфейси посредством добавянето на нови методи става възможно без необходимост от пренаписване на всички имплементиращи ги класове. Старите класове, имплементиращи интерфейс с добавен default метод, ще придобият имплементацията му от интерфейса.

Всички метод декларации в интерфейсите са публични, включително методите по подрзбиране. Следователно можете да пропуснете public модификатора.

В случай, че наследявате интерфейс, който съдържа default метод, можете да направите следното:

- Да пропуснете споменаването на метода по подразбиране, което позволява на вашия разширен интерфейс да наследи default метода
- Предекларация на метода по подразбиране, което го прави abstract
- Предефиниране на default метода, което извършва overriding

Наследяване

В програмния език Java класовете могат да бъдат производни от други класове, следователно да наследяват чужди полета и методи.

Дефиниция: Даден клас, явяващ се производен от друг такъв, се нарича подклас (subclass, derivied class, extended class, child class или дъщерен клас). Класът, от който подкласа е производен, се нарича суперклас (superclass, също base class, parent class или родителски клас).

Освен Object, който няма суперклас, всеки клас има един и само един директен суперклас (единично наследяване или single inheritance). При отсъствието на какъвто и да е друг експлицитен суперклас, всеки клас е имплицитно подклас на Object.

Класовете могат да бъдат производни на класове, производни на други такива и т.н., ултимативно производни от най-горния клас - Object. Подобен клас бива наричан наследник (descended) на всички класове във веригата на наследяване (inheritance chain), простираща се обратно до Object.

Идеята за наследяване е проста и мощна: когато искате да създадете нов клас и вече съществува друг, включващ част от кода от който имате нужда, можете да наследите съществуващия клас. По този начин преизползвате неговите полета и методи, без да е необходимо вие да ги пишете (и дебъгвате!).

Подклас наследява всички членове (полета, методи и вложени класове) от неговия суперклас. Конструкторите не са членове, те не се наследяват от подкласовете, но конструктор на суперкласа може да бъде извикан от подкласа.

Абстрактни методи и класове

Абстрактен клас (abstract class) представлява клас, дефиниран с ключовата дума abstract - който може да включва, а може и да не включва абстрактни методи. Абстрактните класове не могат да бъдат инстанциирани, но могат да бъдат наследявани.

Абстрактен метод представлява метод, деклариран без имплементация (без къдрави скоби, вместо които идва “;”), например:

```
abstract void moveTo(double deltaX, double deltaY);
```

Ако даден абстрактен клас включва абстрактни методи, тогава самият той трябва да бъде деклариран като абстрактен, както следва:

```
public abstract class GraphicObject {
   // declare fields
   // declare nonabstract methods
   abstract void draw();
}
```

Когато бива наследяван абстрактен клас, подкласът обикновено предоставя имплементация на всички абстрактни методи в родителския клас. Ако няма такава, то и наследникът трябва да бъде деклариран абстрактен.

Бележка:

Методите в интерфейс, които не са деклараирани като default или static, са имплицитно abstract, затова abstract модификатора може, но не се ползва в интерфейсни методи.

Абстрактни класове и интерфейси

Абстрактните класове са подобни на интерфейсите. Не могат да бъдат инстанциирани, могат да съдържат микс от методи, декларирани с или без имплементация. В абстрактните класове обаче могат да бъдат декларирани полета, които не са static и final, както и public, protected и private методи. При интерфейсите всички полета автоматично са public, static и финал, а всички методи са public. В допълнение, можете да наследявате само един клас (абстрактен или не), но можете да имплементирате неограничено количество интерфейси.

Кое да ползваме, абстрактни класове или интерфейси?

- Насочете се към ползване на абстрактни класове в следните случаи:
  - Искате да споделите код между няколко логически свързани класа;
  - Очаквате наследниците на абстрактния клас да имат много общи методи или полета, или са нужни модификатори на достъпа, различни от public (protected или прижате);
  - Искате да декларирате нестатични и нефинални полета. Това позволява дефинирането на методи, които могат да достъпят и модифицират състоянието на обекта, към който принадлежат.
- Насочете се към употребата на инферфейси в следните случаи:
  - Очаквате логически несвързани класове да имплементират вашия интерфейс. Например, интерфейсите Comparable и Clonable биват имплементирани от много несвързани класове;
  - Искате да специфицирате поведението на определен тип данни, но не се ангажирате с това кой имплементира поведението;
  - Желаете да се възползвате от множествено наследяване на тип.

https://docs.oracle.com/javase/tutorial/java/IandI/index.html

Модификатори за достъп

Java предоставя т.нар. access modifiers за определянето нива на достъп по отношение на класове, променливи, методи и конструктори. Четирите нива на достъп са както следва:

- видимост в пакета; по подразбиране; без необходими модификатори;
- видимост само в рамките на класа (private);
- видимост за външния свят (public);
- видимост за пакета и подкласовете (protected).

Модификатор за достъп по подразбиране - без ключова дума

Модификатор за достъп по подразбиране означава, че не декларираме изрично модификатор на достъпа за даден клас, поле, метод и т.н.

Променлива или метод, декларирани без посочен модификатор за контрол на достъпа са налични за всеки друг клас в същия пакет. Полетата в даден интерфейс са имплицитно public static final, а методите в него по подразбиране биват public.

Пример:

Променливи и методи могат да бъдат декларирани без никакви модификатори както следва:

```
String version = "1.5.1";

boolean processOrder() {
   return true;
}
```

Частен модификатор за достъп - private

Декларираните като private методи, променливи и конструктори могат да бъдат достъпвани в рамките на техния клас.

Този модификатор за достъп предоставя най-ограниченото ниво на достъп. Класове и интерфейси не могат да бъдат private.

Декларирани като private променливи могат да бъдат достъпвани извън класа, ако в него са налични респективно публични getter методи.

Употребата на private модификатора е основния начин за постигане на енкапсулация и спириване на данни от външния свят.

Пример:

Следният клас използва частен контрол на достъпа

```
public class Logger {
  private String format;
  public String getFormat() {
    return this.format;
  }
  public void setFormat(String format) {
    this.format = format;
  }
}
```

В горния пример променливата format от клас Logger е private, следователно няма начин други класове да получат стойността й или да запазят такава директно в нея.

За да направим променливата достъпна за външния свят дефинираме два публични метода: getFormat(), който връща стойността на format и setFormat(String), който преинициализира нейната стойност.

Публичен модификатор за достъп - public

Декларирани като public, даден клас, метод, конструктор, интерфейс и т.н. могат да бъдат достъпвани от всеки друг клас. Следователно декларирани в рамките на публичен клас такива полета, методи и блокове могат да бъдат достъпвани от всеки наличен клас в клас пътя.

Обърнете внимание, че ако публичният клас който опитваме да достъпим се намира в друг пакет, тогава той трябва да бъде изрично импортнат (imported).

Поради класовото наследяване, всички публични методи и променливи на класа се наследяват от неговите подкласове.

Пример:

Следният метод ползва публичен контрол на достъпа

```
public static void main(String[] arguments) {
  // ...
}
```

main метода на дадено приложение трябва да бъде public. В противен случай не може да бъде извикан от Java интерпретатора за да бъде стартиран класа.

Защитен модификатор за достъп - protected

Декларираните като protected в даден суперклас (клас с наследници) променливи, методи и конструктори могат да бъдат достъпвани само от подкласовете му или който и да е клас в рамките на текущия пакет.

Protected модификатор за достъп не може да бъде употребяван за класове и интерфейси. Методи и полета могат да бъдат декларирани protected, но методи и полета в даден интерфейс не могат.

Защитеният достъп дава на подкласа възможност да използва помощните методи и/или променливи, предотвратявайки употребата им от несвързани класове.

Пример:

Следният родителски клас използва защитения контрол на достъп за да позволи на дъщерните класове да извършат overriding (реимплементация) на openSpeaker() метода:

```
class AudioPlayer {
  protected boolean openSpeaker(Speaker sp) {
     // implementation details
  }
}


class StreamingAudioPlayer {
  boolean openSpeaker(Speaker sp) {
     // implementation details
  }
}
```

В горния случай ако дефинираме openSpeaker() метода като private, то той не би бил достъпен от който и да е клас различен от AudioPlayer. Ако го дефинираме като public, тогава ще бъде достъпен за целия външен свят. За да го изведем за достъп само за подкласовете трябва да ползваме protected модификатора.

http://www.tutorialspoint.com/java/java_access_modifiers.htm

Hello World приложение

```
class HelloWorldApp {
   public static void main(String[] args) {
      System.out.println("Hello World!"); // Display the string.
   }
}
```

Горното “Hello World!” приложение се състои от три основни компонента: коментари в изходния код, дефиниция на HelloWorldApp клас и main метод. Последващите разяснения ще подсигурят базовото разбиране на кода.

Коментари в изходния код

По-долният текст в удебелен шрифт дефинира коментарите в “Hello World!” приложението:

```
/**
 * The HelloWorldApp class implements an application that
```

```
 * simply prints "Hello World!" to standard output.
 */
class HelloWorldApp {
    public static void main(String[] args) {
        System.out.println("Hello World!"); // Display the string.
    }
}
```

Коментарите биват игнорирани от компилатора, но са полезни за програмистите. Програмният език Java поддържа три вида коментари:

```
/* text */
```

Компилаторът игнорира всичко от /* до */.

```
/** documentation */
```

Горното индикира документиращ коментар (document comment или за кратко doc comment). Компилаторът игнорира този тип коментар респективно на коментарите ползващи /* и */. javadoc инструментите ползват документиращите коментари за подготовка автоматизираното генериране на документация.

За повече информация относно javadoc вижте Javadoc™ tool документацията:

https://docs.oracle.com/javase/8/docs/technotes/guides/javadoc/index.html

```
// text
```

Компилаторът игнорира всичко от // до края на реда.

HelloWorldApp клас дефиниция

Следният текст в удебелен шрифт локализира блока код за клас дефиницията на “Hello World!” приложението:

```
/**
 * The HelloWorldApp class implements an application that
 * simply displays "Hello World!" to the standard output.
 */
class HelloWorldApp {
    public static void main(String[] args) {
        System.out.println("Hello World!"); // Display the string.
    }
}
```

Както е показано по-горе, най-базовата форма на клас дефиниция представлява:

```
class name {
    . . .
}
```

Ключовата дума class започва дефиницията на клас на име name, а кодът за всеки клас се позиционира между отварящата и затварящата къдрава скоба, маркирани с удебелен шрифт в предния пример. Засега е достатъчно да знаем, че всяко приложение започва с дефиниция на клас.

main метода

Следният текст в удебелен шрифт обозначава дефиницията на main метода:

```
/**
 * The HelloWorldApp class implements an application that
 * simply displays "Hello World!" to the standard output.
 */
class HelloWorldApp {
    public static void main(String[] args) {
        System.out.println("Hello World!"); //Display the string.
    }
}
```

В програмният език Java всяко приложение трябва да съдържа main метод, притежаващ следната сигнатура (signature):

```
public static void main(String[] args)
```

Модификаторите public и static могат да бъдат изброени в произволен ред (public static или static public), но по конвенция се ползва public static - както е показано по-горе. Можете да наименовате аргументите по ваш избор, но повечето програмисти избират “args” или “argv”.

main методът е подобен на main функцията в C и C++; представлява входна точка за приложението, извикваща последователно всички други методи, необходими за програмата.

main методът приема единичен аргумент: масив от елементи от тип String.

public static void main(**String[] args**)

Масивът е механизма, чрез който runtime системата подава информация към вашето приложение. Например:

java *MyApp arg1 arg2*

Всеки низ в масива се нарича аргумент от командния ред (command-line argument). Подобни аргументи позволяват потребителите да влияят върху операциите, извършвани от приложението - без да е необходима прекомпилация. Например, дадена сортираща програма може да позволи на потребителя да специфицира дали данните да бъдат сортирани в низходящ ред, ползвайки аргументи от командния ред:

-descending

“Hello World!” приложението игнорира аргументите от командния ред, но трябва да имате предвид, че подобни аргументи съществуват.

Финално имаме следния ред:

```
System.out.println("Hello World!”);
```

Горният ред ползва System класа от core библиотеката за да принтира съобщението “Hello World!” на стандартния изход (output). Порции от тази библиотека е позната още като “Application Programming Interface” или “API”.

https://docs.oracle.com/javase/tutorial/getStarted/application/

Streams на System класа

Съществуват 3 streams в System класа, който е част от core библиотеката - System.in, System.out и System.err. Най-употребяваният от тях е System.out за целите на писане на изходно съдържание в конзолата от конзолни приложения.

Тези 3 streams са инициализирани от Java runtime когато JVM стартира, така че няма нужда да си инстанциирате собственоръчни (въпреки, че можете да ги подмените по време на изпълнение).

System.in

Представлява InputStream, който обикновено е свързан към input от клавиатурата по отношение на конзолни приложения. System.in не се ползва толкова често колкото подаването на данни като аргументи на командния ред или конфигурационни файлове. В приложенията с графичен потребителски интерфейс (graphical user interface или GUI) входът към приложението бива реализиран посредством наличния GUI, което се явява отделен входен механизъм от Java IO.

System.out

Представлява PrintStream, който обикновено извежда данни в конзолата. Ползва се често от конзолни програми, като например command-line инструменти. Употребява се често за принтиране на debug statements от дадена програма.

System.err

Представлява PrintStream, който работи подобно на System.out – с изключение на това, че обикновено се употребява за извеждане на съобщения за грешки. Някои програми (например, Eclipse) визуализират изходните данни от System.err в червено.

http://tutorials.jenkov.com/java-io/system-in-out-error.html

Основи на езика

ПРОМЕНЛИВИ

Вече научихте, че обектите пазят тяхното състояние в полета. Езикът за програмиране Java използва също термина променлива (variable). Секцията дискутира тази връзка, както и правилата за наименоване и конвенция, основни типове данни (примитивни, знакови низове и масиви), стойности по подразбиране и литерали.

ОПЕРАТОРИ

Тази секция описва операторите в езика за програмиране Java. Предоставя повечето общоизползвани оператори, след което по-рядко ползваните такива. Включително примерен код, който можете да компилирате и изпълните.

ИЗРАЗИ**, STATEMENTS** И БЛОКОВЕ

Операторите могат да бъдат използвани за изграждането на изрази (expressions), които изчисляват стойности; изразите са основните компоненти на даден statement; statements могат да бъдат групирани в блокове (blocks). Секцията дискутира изрази, statements и блокове, ползвайки примерен код който вече сте виждали.

STATEMENTS ЗА КОНТРОЛ НА ПОТОКА

Секцията описва statements за контрол на потока, поддържани от езика за програмиране Java. Покрива вземането на решения, looping, условни конструкции (branching statements) които позволяват на програмите да изпълняват условно дадени блокове код.

Променливи

Както научихте в предната секция, обектът пази състоянието си в полета.

int **cadence** = 0;

int **speed** = 0;

int **gear** = 1;

Дискусията относно това, какво представлява обект, направи въведение към полетата, но някои въпроси все още не са разгледани, например: Какви са правилата и конвенциите за наименоване на поле? Освен int, какви други типове данни съществуват? Трябва ли полетата да бъдат инициализирани когато се декларират? Присвоена ли е стойност по подразбиране на полетата, при условие, че не са експлицитно инициализирани?

Ще разгледаме отговорите на тези въпроси в текущата секция, но преди това следват няколко технически уточнения. В езика за програмиране Java се ползват и двата термина "поле" и "променлива"; това е чест източник на объркване измежду начинаещи разработчици, след като и двата термина изглеждат като да реферират едно и също нещо.

Програмният език Java дефинира следните типове променливи:

- **Instance** променливи (нестатични полета или non-static fields) - Технически погледнато обектите пазят техните индивидуални състояния в "нестатични полета", представляващи дефинирани такива без ключовата дума *static*. Нестатичните полета са познати още като *instance* променливи*,* тъй като техните стойности са уникални за всяка инстанция на класа (т.е. за всеки обект); currentSpeed на един велосипед е независима от currentSpeed на друг такъв.
- **Class** променливи (статични полета или static fields) - Клас променлива е всяко поле, декларирано със static modifier; това указва на компилатора, че съществува точно едно копие на тази променлива, без значение колко пъти се инициализира класа. Полето, дефиниращо номера на предавки за даден тип велосипед може да бъде маркирано като static, след като концептуално същия брой предавки ще бъдат приложени за всички инстанции. Кодът static int numGears = 6; би създал такова статично поле. В допълнение, ключовата дума *final* може да бъде добавена, за да се индикира че бройката предавки никога няма да се промени.
- **Local** променливи (локални променливи) - Подобно на това как обектите пазят състоянието си в полета, даден метод често ще съхранява собственото си временно състояние в локални променливи. Синтаксисът за деклариране на локална променлива е подобен на този за деклариране на поле (например, int count = 0;). Няма специална ключова дума, определена за локална променлива; това следва изцяло от местоположението на деклариране на променливата - което е между отварящата и затварящата скоба { } на метод. Като такава, дадена локална променлива е видима само в метода, в който е декларирана; не може да бъде достъпвана от останалата част на класа.
- Параметри - Вече сте виждали примери за параметри в Bicycle класа и в main метода на "Hello World!" приложението. Припомнете си, че сигнатурата за main метода е public static void main(String[] args). Тук args променливата е параметърът на този метод. Важно за запомняне е, че параметрите са винаги класифицирани като променливи, а не полета. Това е приложимо също и към други приемащи параметри конструкции (като конструктори и exception handlers), за които ще говором по-долу.

В текущия наръчник при споменаване на "полета в общия случай" (изключвайки локални променливи и параметри), за краткост можем да казваме "полета". Ако дискусията е приложима за всички гореизброени, можем да ги наричаме "променливи". Ако контекстът налага различаване, ще използваме специфичните термини (статични полета, локални променливи и т.н.). Може би ще срещате и термина "член" (member). Полета, методи и вложени типове колективно се наричат членове.

Наименоване

Java като всеки език за програмиране има собствен набор от правила и конвенции за типовете имена, които можете да използвате. Правилата и конвенциите за наименоване на променливи могат да бъдат обобщени както следва:

- Имената на променливите са case-sensitive (прави се разлика между малки и големи букви). Името на променливата може да бъде всеки валиден идентификатор - неограничена по дължина поредица от Unicode букви и числа, започвайки с буква, знак за долар "$", или знак за подчертавка "_". Конвенцията обаче предполага имената на променливите винаги да започват с буква. В допълнение знакът за долар по конвенция изобщо не се ползва. Можете да попаднете в някои ситуации, в които автоматично генерирани имена ще съдържат знак за долар, но при наименоване на вашите променливи трябва да избягвате употребата му. Подобна конвенция съществува и за знака за подчертавка; докато той е технически валиден за начало на имената на вашите променливи, подобна практика не е препоръчителна. Празно пространство не е позволено.
- Последващите символи могат да бъдат букви, числа, знак за долар или подчертавки. Конвенциите са приложими и за това правило. Когато избирате име на вашите променливи, ползвайте пълни думи вместо неясни съкращения. По този начин ще направите вашия код по-лесен за четене и разбиране, като в много случаи той ще бъде и самообяснителен (self-documenting); полетата наименовани cadence, speed и gear например са много по-интуитивни от абревиатурните им версии като s, c и g. Също имайте предвид, че избраните от вас имена не трябва да бъдат ключови или запазени думи (http://docs.oracle.com/javase/tutorial/java/nutsandbolts/_keywords.html).
- Ако избраното от вас име се състои от само една дума, изпишете я с малки букви. Ако се състои от повече от една дума, капитализирайте първата булва на всяка последваща дума. Имената gearRation и currentGear са примери за тази конвенция. Ако вашата променлива съхранява константна стойност, като например static final int NUM_GEARS = 6, конвенцията се изменя като капитализирането се прилага за всяка буква, а думите се разделят с подчертавка. По конвенция, знакът за подчертавка никога не се ползва другаде.

Примитивни типове данни

Програмният език Java е statically-typed, което означава че всички променливи трябва първо да бъдат декларирани преди да могат да бъдат използвани. Това включва заявяване типа на променливата и нейното име, както вече сме виждали:

```
int gear = 1;
```

Горният ред указва на програмата, че полето с име "gear" съществува, съдържа числови данни и има първоначалната стойност "1". Типът данни на променливата определя стойностите, които тя може да съдържа, както и операциит който могат да бъдат извършвани с нея. В допълнение към int, езикът за програмиране Java поддържа седем други примитивни типове данни (primitive data types). Примитивен тип е предефиниран от езика и е наименован с резервирана ключова дума. Примитивните стойности не споделят състояние с други такива. Общо осемте примитивни типове данни, поддържани от Java, са следните:

- byte
- short
- int
- long
- float
- double
- boolean
- char

В допълнение към осемте примитивни типове данни посочени по-горе, езикът за програмиране Java предоставя специална поддръжка за низове от символи чрез java.lang.String класа. Обособяването на вашия низ от символи с двойни кавички автоматично ще създаде нов String обект; например String s = "this is a string"; String обектите са *immutable*, което означава, че веднъж създадени, техните стойности не могат да бъдат променяни. String класа не е технически примитивен тип данни, но имайки предвид специалната поддръжка предоставена от езика, вероятно ще мислите за него като за такъв.

СТОЙНОСТИ ПО ПОДРАЗБИРАНЕ

Не винаги е задължително да се присвояват стойности при декларирането на поле. На декларирани, но неинициализирани полета ще бъдат присвоени резонни стойности по подразбиране от компилатора. В общия случай стойността по подразбиране ще бъде 0 или null, в зависимост от типа данни. Разчитайки на подобни стойности по подразбиране обаче се счита за лош стил на програмиране.

Последващата диаграма обобщава стойностите по подразбиране на горните типове данни:

Тип данни	Стойност по подразбиране (за полета)
byte	0
short	0
int	0
long	0L
float	0.0f
double	0.0d
char	\u0000'
String (или всеки обект)	null
boolean	0

Локалните променливи са малко по-различни; компилаторът никога не присвоява стойност по подразбиране на неинициализирана локална променлива. Ако не можете да инициализирате вашата локална променлива където е декларирана, погрижете се да присвоите стойност преди да се опитате

да я ползвате. Достъп до неинициализирана локална променлива ще доведе до грешка по време на компилация.

ЛИТЕРАЛИ

Може би сте забелязали, че ключовата дума *new* не се ползва при инициализиране на променлива от примитивен тип. Примитивните типове са специални типове данни, построени в самия език; те не са създадени от клас обекти. Литерал (literal) е репрезентацията на фиксирана стойност в сорс кода; литералите са репрезентирали директно в кода без необходимост от изчисление. Както е показано по-долу, възможно е да присвоите литерал на променлива от примитивен ти:

```
boolean result = true;

char capitalC = 'C';

byte b = 100;

short s = 10000;

int i = 100000;
```

ЦЕЛОЧИСЛЕНИ ЛИТЕРАЛИ

Даден целочислен (integer) литерал е от тип long, ако завършва с буквата L или l; в противен случай е int. Препоръчително е ползването на главна буква L, защото малката l е трудноразличима от числото 1.

Стойностите на интегралните типове byte, short, int и long могат да бъдат създадени от int литерали. Стоностите на типа long, които надвишават обема на int, могат да бъдат създадени от long литерали. Integer литералите могат да бъдат представени в следните бройни системи:

- Десетична (decimal) - Base 10, чиито числа се състоят от номерата 0-9; това е бройната система която ползваме в ежедневието си;
- Шестнадесетична (hexadecimal) - Base 16, чиито числа се състоят от номерата 0-9 и буквите A-F;
- Бинарна (binary) - Base 2, чиито номера се състоят от номерата 0 и 1 (можете да създавате бинарни литерали от Java SE 7 и нагоре).

При програмирането в общия случай се ползва десетичната бройна система. Въпреки това, ако имате необходимост да ползвате друга бройна система, следния пример показва правилния синтаксис. Префиксът 0x индикира шестнадесетична и 0b индикира бинарна:

```
// The number 26, in decimal

int decVal = 26;

//  The number 26, in hexadecimal

int hexVal = 0x1a;

// The number 26, in binary

int binVal = 0b11010;
```

ЛИТЕРАЛИ С ПЛАВАЩА ЗАПЕТАЯ

Литералите от тип char и String могат да съдържат всеки Unicode (UTF-16) знак. Ако вашият текстов редактор/среда за разработка и файлова система го позволяват, можете да ползвате такива символи директно във вашия код. Ако не, можете да ползвате т.нар. “Unicode escape”, например, ‘\u0108’ (главно C с диактричен знак ‘~’), или “S\u00ED Se\u00F1or” (Sí Señor на испански). Винаги ползвайте единични кавички за char литералите и двойни кавички за String литералите. Unicode escape поредностите могат да бъдат ползвани и другаде в програмата (например имена на полета), не само при char и String литерали.

Програмният език Java поддържа още някои специални escape поредности за char и String литерали: \b (backspace), \t (tab), \n (нов ред), \f (form feed), \r (carriage return), \" (двойна кавичка), \' (единична кавичка) и \\ (backslash).

Съществува специален null literal, който може да бъде ползван за стойност при всеки референтен тип. null може да бъде присвоено на всяка стойност, освен на примитивни типове. Има малко неща, които могат да бъдат направени с null стойност, освен тестването на нейната наличност. Следователно null бива често ползвана в програми като маркер за индикиране неналичността на обект.

Финално съществува специален вид литерал наречен клас литерал (class literal), формиран от името с добавен ".class"; например String.class. Това реферира към обект (от тип Class), репрезентиращ самия тип.

ПОЛЗВАНЕ НА ЗНАК ЗА ПОДЧЕРТАВКА В ЧИСЛОВИ ЛИТЕРАЛИ

В Java SE 7 и по-късно всякаква бройка подчертавки (__ може да се появи навсякъде между номера в числов литерал. Тази функционалност позволява обособяването на групи от номера в числови литерали, което може да подпомогне четимостта на кода.

Например, ако кодът съдържа номера с много числа, можете да ползвате знака за подчертавка за да отделите числата в групи от по три, сходни на начина по който бихте ползвали пунктуационен маркер като запетайка, празно място или разделител.

Следният пример показва други начини, по които можете да ползвате подчертавката в числови литерали:

```
long creditCardNumber = 1234_5678_9012_3456L;
long socialSecurityNumber = 999_99_9999L;
float pi =  3.14_15F;
long hexBytes = 0xFF_EC_DE_5E;
long hexWords = 0xCAFE_BABE;
long maxLong = 0x7fff_ffff_ffff_ffffL;
byte nybbles = 0b0010_0101;
long bytes = 0b11010010_01101001_10010100_10010010;
```

Можете да поставите подчертавка само между числа; не може да поставяте подчертавки на следните места:

- В началото или края на число;
- Непосредствено до десетична запетая в литерал тип плаваща запетая;
- Преди F или L suffix;
- На позиции където се очаква стрингова репречентация на числа.

Следните примери демонстрират валидни и невалидни (удебелени) поставяния на подчертавки в числови литерали:

```
// Invalid: cannot put underscores
// adjacent to a decimal point
float pi1 = 3_.1415F;
// Invalid: cannot put underscores
// adjacent to a decimal point
float pi2 = 3._1415F;
```

```
// Invalid: cannot put underscores
// prior to an L suffix
long socialSecurityNumber1 = 999_99_9999_L;

// OK (decimal literal)
int x1 = 5_2;
// Invalid: cannot put underscores
// At the end of a literal
int x2 = 52_;
// OK (decimal literal)
int x3 = 5_______2;

// Invalid: cannot put underscores
// in the 0x radix prefix
int x4 = 0_x52;
// Invalid: cannot put underscores
// at the beginning of a number
int x5 = 0x_52;
// OK (hexadecimal literal)
int x6 = 0x5_2;
// Invalid: cannot put underscores
// at the end of a number
int x7 = 0x52_;
```

Масиви

Масив представлява контейнер обект, съдържащ фиксирана бройка стойности от конкретен тип. Дължината на масива е определена при създаването му. След създаване, дължината остава фиксирана. Вече сте виждали пример от масиви в main метода на "Hello World!" приложението. Текущата секция дискутира масивите в по-високо ниво на детайл.

Всяка съдържаща се в масива стойност се нарича елемент (element), като всеки елемент се достъпва по неговия числов индекс (index). Номерирането на елементите започва от 0. Ако масивът има 9 елемента, то тогава последният ще се достъпва с индекс 8.

Следната програма ArrayDemo създава масив от integers, добавя стойности в масива и принтира всяка стойност в стандартния изход.

```
class ArrayDemo {
  public static void main(String[] args) {
    // declares an array of integers
    int[] anArray;

    // allocates memory for 10 integers
    anArray = new int[10];

    // initialize first element
    anArray[0] = 100;
```

```
        // initialize second element
        anArray[1] = 200;
        // and so forth
        anArray[2] = 300;
        anArray[3] = 400;
        anArray[4] = 500;
        anArray[5] = 600;
        anArray[6] = 700;
        anArray[7] = 800;
        anArray[8] = 900;
        anArray[9] = 1000;

        System.out.println("Element at index 0: "
                + anArray[0]);
        System.out.println("Element at index 1: "
                + anArray[1]);
        System.out.println("Element at index 2: "
                + anArray[2]);
        System.out.println("Element at index 3: "
                + anArray[3]);
        System.out.println("Element at index 4: "
                + anArray[4]);
        System.out.println("Element at index 5: "
                + anArray[5]);
        System.out.println("Element at index 6: "
                + anArray[6]);
        System.out.println("Element at index 7: "
                + anArray[7]);
        System.out.println("Element at index 8: "
                + anArray[8]);
        System.out.println("Element at index 9: "
                + anArray[9]);
    }
}
```

Изходът от програмата е следния:

```
Element at index 0: 100
Element at index 1: 200
Element at index 2: 300
Element at index 3: 400
Element at index 4: 500
Element at index 5: 600
Element at index 6: 700
Element at index 7: 800
```

```
Element at index 8: 900
Element at index 9: 1000
```

В ситуация на програмиране в истинския свят вероятно бихте използвали някоя от поддържаните обхождащи конструкции (looping constructs), за да итерирате елементите на масив, вместо да изписвате всеки ред индивидуално като в предходния пример. Въпреки това, примерът ясно илюстрира синтаксиса на масивите.

ДЕКЛАРИРАНЕ НА ПРОМЕНЛИВА ЗА РЕФЕРИРАНЕ КЪМ МАСИВ

Предишната програма декларира масив (наименован anArray) чрез следния код:

```
// declares an array of integers
int[] anArray;
```

Също както при декларациите за променливи от друг тип, декларацията на масив има два компонента - тип на масива и име. Типът на масива се указва като type[], където type е типът данни на съдържащите се елементи; квадратните скоби са специални символи, индикиращи че променливата съдържа масив. Големината на масива не е част от неговия тип (което е и причината скобите да бъдат празни). Името на масива може да бъде всякакво такова, следващо обаче дискутираните по-горе правила за конвенция при наименованието. Както при променливите от друг тип, декларацията всъщност не създава масив; тя указва на компилатора, че тази променлива ще държи масив от определен тип.

Подобно можете да декларирате масиви от други типиве:

```
byte[] anArrayOfBytes;
short[] anArrayOfShorts;
long[] anArrayOfLongs;
float[] anArrayOfFloats;
double[] anArrayOfDoubles;
boolean[] anArrayOfBooleans;
char[] anArrayOfChars;
String[] anArrayOfStrings;
```

Можете също да поставяте скобите след името на масива:

```
// this form is discouraged
float anArrayOfFloats[];
```

Въпреки това конвенцията не препоръчва употребата на тази форма; скобите идентифицират типа на масива и трябва да бъдат поставяни заедно с неговото определяне.

СЪЗДАВАНЕ, ИНИЦИАЛИЗАЦИЯ И ДОСТЪП НА МАСИВИ

Един начин за създаване на масив е чрез оператора *new*. Следващия statement в ArrayDemo програмата заделя (алокира, allocates) масив с достатъчно памет за 10 integer елемента и присвоява масива на anArray променливата.

```
// create an array of integers
anArray = new int[10];
```

Ако този statement липсва, тогава компилацията е неуспешна и следното съобщение за грешка бива изведено:

```
ArrayDemo.java:4: Variable anArray may not have been initialized.
```

Следващите няколко реда присвояват стойности на всеки елемент от масива:

```
anArray[0] = 100; // initialize first element
anArray[1] = 200; // initialize second element
anArray[2] = 300; // and so forth
```

Всеки елелемент на масива се достъпва посредством числовия му индекс:

```
System.out.println("Element 1 at index 0: " + anArray[0]);
System.out.println("Element 2 at index 1: " + anArray[1]);
System.out.println("Element 3 at index 2: " + anArray[2]);
```

Алтернативно можете да ползвате краткия синтаксис за създаване и инициализиране на масив:

```
int[] anArray = {
   100, 200, 300,
   400, 500, 600,
   700, 800, 900, 1000
};
```

Тук дължината на масива се определя от стойностите, предоставени между скобите и разделени със запетаи.

Можете още да декларирате масив от масиви (познат още като многомерен масив или multidimensional array) чрез ползването на две или повече групи от скоби, като например String[][] names. Следователно всеки елемент трябва да бъде достъпен чрез кореспондиращия номер на индекс стойностите.

В програмният език Java многомерен масив е масив, чиито компоненти самите са масиви. Това е различно от C и Fortran, например. Последствие от това е, че редовете могат да варират като дължина, показано в следващата MultiDimArrayDemo програма:

```
class MultiDimArrayDemo {
   public static void main(String[] args) {
      String[][] names = {
         {"Mr. ", "Mrs. ", "Ms. "},
         {"Smith", "Jones"}
      };
      // Mr. Smith
      System.out.println(names[0][0] + names[1][0]);
      // Ms. Jones
      System.out.println(names[0][2] + names[1][1]);
   }
}
```

Изходът от тази програма е:

```
Mr. Smith
Ms. Jones
```

Финално можете да ползвате вграденото *length* property, за да определите големината на всеки масив. Следният код принтира големината на масива в стандартния изход:

```
System.out.println(anArray.length);
```

КОПИРАНЕ НА МАСИВИ

Класът System има метод arraycopy, който можете да ползвате за ефикасно копиране на данни от един масив в друг:

```
public static void arraycopy(Object src, int srcPos,
                             Object dest, int destPos, int length)
```

Двата Object аргумента определят масива от който ще се копира и съответно масива в който ще се копира. Трита int аргумента специфицират началната позиция на входния масив, началната позиция на изходния масив, както и номера на елементите за копиране.

Следната програма ArrayCopyDemo декларира масив от char елементи и образува думата “decaffeinated”. Ползва System.arraycopy метода за копиране на поредността от компонентите на масива във втори такъв:

```
class ArrayCopyDemo {
   public static void main(String[] args) {
      char[] copyFrom = { 'd', 'e', 'c', 'a', 'f', 'f', 'e',
                          'i', 'n', 'a', 't', 'e', 'd' };
      char[] copyTo = new char[7];

      System.arraycopy(copyFrom, 2, copyTo, 0, 7);
      System.out.println(new String(copyTo));
   }
}
```

Изходът от програмата е:

```
caffein
```

МАНИПУЛАЦИИ С МАСИВИ

Масивите са мощни и полезни концепции в програмирането. Java SE предоставя методи за извършване на някои често използвани манипулации, свързани с масиви. Например, ArrayCopyDemo ползва arraycopy метода на System класа, вместо ръчна итерация на елементите от входния масив и позиционирането на всеки един в изходния масив. Това се изпълнява “зад сцената” (behind the scenes), позволявайки на разработчика да ползва един ред код за извикване на метода.

За ваше удобство Java SE предоставя няколко метода за манипулации с масиви (често срещани задачи като копиране, сортиране и търсене в масиви) в java.util.Arrays класа. Например, горният код може да бъде модифициран да ползва copyOfRange метода от java.util.Arrays класа, както ще видите в ArrayCopyOfDemo примера. Разликата е, че ползвайки copyOfRance метода не задължава да бъде предварително създаден изходния масив преди извикването на самия метод, защото изходният масив бива връщан като изход от метода:

```
class ArrayCopyOfDemo {
   public static void main(String[] args) {

      char[] copyFrom = {'d', 'e', 'c', 'a', 'f', 'f', 'e',
         'i', 'n', 'a', 't', 'e', 'd'};

      char[] copyTo = java.util.Arrays.copyOfRange(copyFrom, 2, 9);

      System.out.println(new String(copyTo));
```

```
  }
}
```

Както виждате изходът от тази програма е същата (caffein), въпреки че изисква по-малко редове код. Обърнете внимание, че втория параметър на copyOfRange метода е първиначалния индекс от обема за копиране (включително), докато третият параметър е финалния индекс от обема за копиране (невключително). В този пример обемът за копиране не включва елемента от масива с индекс 9 (който съдържа символ a).

Някои други полезни операции, предоставени от методи на java.util.Arrays класа, са:

- Търсене в масив за определена стойност за получаване индекса, където е позиционирана (binarySearch метода);
- Сравняване на два масива с цел определяне дали са еднакви или не (equals метода);
- Сортиране на масив във възходящ ред. Това може да бъде направено както последователно, ползвайки sort метода, така и паралелно ползвайки parallelSort метода, въведен в Java SE 8. Паралелното сортиране на големи масиви върху мултипроцесорни системи е по-бързо от последователното сортиране на масиви.

Резюме на променливи

Програмният език Java ползва като част от своята терминология както “полета”, така и “променливи”. Instance променливите (нестатични полета) са уникални за всяка инстанция на класа. Клас променливите (статични полета) са полета, декларирани чрез static modifier; съществува точно едно копие на клас променлива, без значение колко пъти бива инстанцииран класа. Локалните променливи съхраняват временното състояние в даден метод. Параметрите са променливи, предоставящо допълнителна информация на метод; както локалните променливи, така и параметрите са винаги класифицирани като “променливи” (не “полета”). При наименоване на полетата от променливи, съществуват правила и конвенции които трябва да бъдат следвани и спазвани.

Осемте примитивни типове данни са - byte, short, int, long, float, double, boolean и char. Класа java.lang.String репрезентира низ от символи. Компилаторът ще присвони смислена стойност по подразбиране за полета от горните типове; за локалните променливи никога не се присвоява стойност по подразбиране.

Литерал е репрезентацията в сорс кода на фиксирана стойност.

Масив представлява контейнер обект, съдържащ фиксирана бройка стойности от даден конкретен тип. Дължината на масива се определя при неговото създаване. След създаването му, дължината остава фиксирана.

Оператори

След като се запознахме с декларацията и инициализацията на променливи, ще разгледаме как можем да правим нещо с тях. Изучаване предоставените от езика за програмиране Java оператори е добро начало. Операторите (operators) са специални символи, изпълняващи специфични операции върху един, два или три операнда (operands), след което бива върнат резултат.

Докато разглеждаме операторите в Java, може да бъде полезно да се запознаете кои от тях имат най-голямо предимство. Операторите в следната таблица са изброени съобразено поредността на тяхното предимство. В горната част на таблицата са операторите с по-високо предимство. Те биват изпълнявани преди операторите с релативно по-ниско предимство. Когато оператори с еднакво предимство се появят в даден израз, правило трябва да определи кой ще бъде изпълнен пръв. Всички

бинарни оператори освен тези за присвояване, се изпълняват отляво надясно; операторите за присвояване се изпълняват от дясно на ляво.

Оператори	Предимство
postfix	expr++ expr—
unary	++expr —expr +expr -expr ~ !
multiplicative	* / %
additive	+ -
shift	<< >> >>>
relational	< > <= >= instanceof
equality	== !=
bitwise AND	&
bitwise exclusive OR	^
bitwise inclusive OR	|
logical AND	&&
ternary	? :
assignment	= += −= ×= ÷= %= &= ^= |= <<= >>= >>>=

При програмирането в общия случай, опеделени оператори се ползват по-често от други; например операторът за присвояване “=“ е далеч по-често използван от ипероатора за unsigned right shift “>>>”. Имайки това предвид, следващата дискусия се фокусира първо върху по-често употребяваните оператори, като в края разглежда по-рядно използваните такива. Всяка дискусия е акомпанирана от примерен код, който можете да компилирате и изпълните. Разглеждането на изхода ще подпомогне научаването.

Присвояване, аритметични и унарни оператори

ПРОСТ ОПЕРАТОР ЗА ПРИСВОЯВАНЕ

Един от най-често използваните оператори, който ще срещате, бива простият оператор “=“. Видяхте оператора в Bicycle класа; присвоява стойността вдясно на операнда вляво:

```
int cadence = 0;
int speed = 0;
int gear = 1;
```

Този оператор може също да бъде ползван върху обекти, за присвояването на обектни референции (object references).

АРИТМЕТИЧНИ ОПЕРАТОРИ

Java предоставя оператори, извършващи добавяне, изваждане, умножение и деление. Вероятно веднага бихте ги разпознали от техните математически еквиваленти. Единственият символ който

може би изглежда нов за вас, е “%”, който разделя един операнд на друг и връща остатъка като резултат.

Оператор	Описание
+	Оператор за добавяне (ползван също при String конкатенацията)
-	Оператор за изваждане
*	Оператор за умножаване
/	Оператор за делене
%	Оператор за остатък при делене

Следващата програма ArithmeticDemo тества аритметичните оператори.

```
class ArithmeticDemo {

  public static void main (String[] args) {

    int result = 1 + 2;
    // result is now 3
    System.out.println("1 + 2 = " + result);
    int original_result = result;

    result = result - 1;
    // result is now 2
    System.out.println(original_result + " - 1 = " + result);
    original_result = result;

    result = result * 2;
    // result is now 4
    System.out.println(original_result + " * 2 = " + result);
    original_result = result;

    result = result / 2;
    // result is now 2
    System.out.println(original_result + " / 2 = " + result);
    original_result = result;

    result = result + 8;
    // result is now 10
    System.out.println(original_result + " + 8 = " + result);
    original_result = result;

    result = result % 7;
    // result is now 3
```

```
        System.out.println(original_result + " % 7 = " + result);
    }
}
```

Програмата ще принтира следното:

```
1 + 2 = 3
3 - 1 = 2
2 * 2 = 4
4 / 2 = 2
2 + 8 = 10
10 % 7 = 3
```

Можете също да комбинирате аритметичните оператори с простия оператор за присвояване, за да създадете т.нар. *compound assignments*. Например, x+=1; и x=x+1; и двете ще инкрементират (increment) стойността на x с 1.

Операторът + може да бъде ползван и при конкатенацията (concatenation, joining, събиране, долепяне или обединяване) на два низа заедно, демонстрирано от следната ConcatDemo програма:

```
class ConcatDemo {
    public static void main(String[] args){
        String firstString = "This is";
        String secondString = " a concatenated string.";
        String thirdString = firstString+secondString;
        System.out.println(thirdString);
    }
}
```

В края на програмата променливата thirdString съдържа “This is a concatenated string.”, което бива принтирано на стандартния изход.

УНАРНИ ОПЕРАТОРИ

Унарните оператори изискват само един операнд; те изпълняват различни операции като инкрементиране/декрементиране (incrementing/decrementing, увеличаване/намаляване) на стойност с едно, отричане (negating) на израз или обръщане (inverting) стойността на boolean.

Оператор	Описание
+	Унарен плюс оператор, индикира позитивна стойност (числата са позитивни и без него)
-	Унарен минус оператор; отрича израз
++	Инкрементиращ оператор; инкрементира стойността с 1
—	Декрементиращ оператор; декрементира стойността с 1
!	Логически комплементарен оператор, обръща стойността на boolean

Следната програма UnaryDemo тества унарните оператори:

```
class UnaryDemo {
```

```
    public static void main(String[] args) {

        int result = +1;
        // result is now 1
        System.out.println(result);

        result--;
        // result is now 0
        System.out.println(result);

        result++;
        // result is now 1
        System.out.println(result);

        result = -result;
        // result is now -1
        System.out.println(result);

        boolean success = false;
        // false
        System.out.println(success);
        // true
        System.out.println(!success);
    }
}
```

Инкрементиращите/декрементиращите оператори могат да бъдат приложени преди (prefix) или след (postfix) операнта. Кодът result++; и ++result; ще резултира и в двата случая в инкрементиран с едно result. Единствената разлика е, че префикс версията (++result) връща инкрементираната стойност предварително, докато постфикс версията (result++) евалюира оригиналната стойност. Ако просто изпълнявате инкрементиране/декрементиране, няма значение коя версия ще изберете. Но ако използвате оператора като част от по-голям израз, кой точно ще изберете може да създаде огромна разлика.

Следната програма PrePostDemo илюстрира разликите между префикс/постфикс унарния инкрементиращ оператор:

```
class PrePostDemo {
    public static void main(String[] args){
        int i = 3;
        i++;
        // prints 4
        System.out.println(i);
        ++i;
        // prints 5
        System.out.println(i);
        // prints 6
```

```
        System.out.println(++i);
        // prints 6
        System.out.println(i++);
        // prints 7
        System.out.println(i);
    }
}
```

Равенство, сравняващо и условни оператори

РАВЕНСТВО И СРАВНЯВАЩИ ОПЕРАТОРИ

Операторите за равенство и сравняване определят дали един операнд е по-голям от, по-малък от, равен на или неравен на друг операнд. Мнозинството от тези оператори вероятно също изглежда познато. Имайте предвид, че трябва да ползвате "==" вместо "=" когато тествате дали две примитивни стойности са еквивалентни.

```
==      equal to
!=      not equal to
>       greater than
>=      greater than or equal to
<       less than
<=      less than or equal to
```

Следната програма ComparisonDemo тества операторите за сравняване:

```
class ComparisonDemo {

    public static void main(String[] args){
        int value1 = 1;
        int value2 = 2;
        if(value1 == value2)
            System.out.println("value1 == value2");
        if(value1 != value2)
            System.out.println("value1 != value2");
        if(value1 > value2)
            System.out.println("value1 > value2");
        if(value1 < value2)
            System.out.println("value1 < value2");
        if(value1 <= value2)
            System.out.println("value1 <= value2");
    }
}
```

Изход:

```
value1 != value2
value1 <  value2
value1 <= value2
```

УСЛОВНИ ОПЕРАТОРИ

Операторите && и || изпълняват условно-и (Conditional-AND) и условно-или (Conditional-OR) операции върху два булеви израза. Тези оператори показват "short-circuiting" поведение, което означава, че вторият операнд бива изпълнен само ако има нужда.

&& Conditional-AND

|| Conditional-OR

Следната програма ConditionalDemo1 тества тези оператори:

```
class ConditionalDemo1 {

    public static void main(String[] args){
        int value1 = 1;
        int value2 = 2;
        if((value1 == 1) && (value2 == 2))
            System.out.println("value1 is 1 AND value2 is 2");
        if((value1 == 1) || (value2 == 1))
            System.out.println("value1 is 1 OR value2 is 1");
    }
}
```

Друг условен оператор е ?:, който може да бъде считан за кратка форма на if-then-else statement. В следния пример операторът трябва да бъде четен като: "Ако someCondition е true, присвои стойността от value1 на result. В противен случай присвои стойността на value2 на result.".

Следната програма ConditionalDemo2 тества ?: оператора:

```
class ConditionalDemo2 {

    public static void main(String[] args){
        int value1 = 1;
        int value2 = 2;
        int result;
        boolean someCondition = true;
        result = someCondition ? value1 : value2;

        System.out.println(result);
    }
}
```

Защото someCondition е true, програмата принтира "1" на екрана. Ползвайте ?: оператора вместо if-then-else statement, ако това би направило кода по-четим; например когато изразите са компактни и без странични ефекти (като присвояване).

ОПЕРАТОРЪТ ЗА СРАВНЕНИЕ НА ТИПОВЕ **INSTANCEOF**

Операторът instanceof сравнява даден обект с посочен тип. Можете да го ползвате за да тествате дали обект е инстанция на даден клас, инстанция на подклас или инстанция на клас, имплементиращ конкретен интерфейс.

Следната програма InstanceofDemo дефинира родителски клас (наименован Parent), прост интерфейс (наименован MyInterface) и дъщерен клас (наименован Child), наследяващ родителския и имплементиращ интерфейса.

```
class InstanceofDemo {
   public static void main(String[] args) {

      Parent obj1 = new Parent();
      Parent obj2 = new Child();

      System.out.println("obj1 instanceof Parent: "
         + (obj1 instanceof Parent));
      System.out.println("obj1 instanceof Child: "
         + (obj1 instanceof Child));
      System.out.println("obj1 instanceof MyInterface: "
         + (obj1 instanceof MyInterface));
      System.out.println("obj2 instanceof Parent: "
         + (obj2 instanceof Parent));
      System.out.println("obj2 instanceof Child: "
         + (obj2 instanceof Child));
      System.out.println("obj2 instanceof MyInterface: "
         + (obj2 instanceof MyInterface));
   }
}

class Parent { }
class Child extends Parent implements MyInterface { }
interface MyInterface { }
```

Изход:

```
obj1 instanceof Parent: true
obj1 instanceof Child: false
obj1 instanceof MyInterface: false
obj2 instanceof Parent: true
obj2 instanceof Child: true
obj2 instanceof MyInterface: true
```

Когато ползвате instanceof оператора, имайте предвид, че null не е инстанция на нищо.

Побитови оператори

Програмният език Java предоставя още оператори, изпълняващо bitwise и bit shift операции върху интегрални (integral) типове. Операторите разгледани в тази секция са по-рядко ползвани. Следователно ще бъдат по-кратко покрити; целта е общо запознаване с това, че съществуват.

Унарният bitwise комплементиращ оператор “~” обръща даден bit pattern; може да бъде приложен върху всеки от интегралните типове, правещ всяка “0” на “1” и всяка “1” на “0”. Например, даден byte

съдържа 8 бита; прилагането на този оператор към стойността чийто bit pattern е “00000000” би променило нейния pattern на “11111111”.

Signed left shift оператора “<<“ измества даден bit pattern наляво, а signed right shift оператора “>>” измества даден bit pattern надясно. Bit pattern се подава от левия операнд, като номерът на позициите за отместване - от десния операнд.

Bitwise & оператора изпълнява побитова AND операция.

Bitwise ^ оператора изпълнява побитова exclusive OR операция.

Bitwise | оператора изпълнява побитова inclusive OR операция.

Следната програма Bitdemo използва побитовия AND оператор за да принтира номер “2” в стандартния изход.

```
class BitDemo {
   public static void main(String[] args) {
      int bitmask = 0x000F;
      int val = 0x2222;
      // prints "2"
      System.out.println(val & bitmask);
   }
}
```

Резюме на операторите

Следната бърза референция обобщава поддържаните от програмния език Java оператори.

Прост оператор за присвояване

= Simple assignment operator

Аритметични оператори

\+ Additive operator (also used
for String concatenation)

\- Subtraction operator

* Multiplication operator

/ Division operator

% Remainder operator

Унарни оператори

\+ Unary plus operator; indicates
positive value (numbers are
positive without this, however)

\- Unary minus operator; negates
an expression

++ Increment operator; increments
a value by 1

-- Decrement operator; decrements
a value by 1

! Logical complement operator;

inverts the value of a boolean

Оператори за равенство и сравняване

== Equal to

!= Not equal to

> Greater than

>= Greater than or equal to

< Less than

<= Less than or equal to

Условни оператори

&& Conditional-AND

|| Conditional-OR

?: Ternary (shorthand for
if-then-else statement)

Оператори за сравняване на типа

instanceof Compares an object to
a specified type

Побитови и bit shift оператори

~ Unary bitwise complement

<< Signed left shift

>> Signed right shift

>>> Unsigned right shift

& Bitwise AND

^ Bitwise exclusive OR

| Bitwise inclusive OR

Изрази, **statements** и блокове

След като вече разбирате променливи и оператори, време е да разгледаме изрази (expressions), *statements* и блокове (blocks). Операторите могат да бъдат ползвани в построяването на изрази, които изчисляват стойности; изразите са основни компоненти на statements; statements могат да бъдат групирани в блокове.

ИЗРАЗИ

Израз представлява конструкция от променливи, оператори и извикване на методи, която бива конструирана съобразно синтаксиса на езика, така че да представя единична стойност. Вече сте виждали примери за изрази, илюстрирани с подчертан шрифт по-долу:

```
int cadence = 0;
anArray[0] = 100;
System.out.println("Element 1 at index 0: " + anArray[0]);

int result = 1 + 2; // result is now 3
if (value1 == value2)
    System.out.println("value1 == value2");
```

Типа данни на връщаната от израза стойност зависи от елементите, ползвани в израза. Изразът cadence = 0 връща int защото операторът за присвояване връща стойност от същия тип данни като неговия ляв операнд; в този случай, cadence е int. Както виждате от другите изрази, даден израз може да връща и други типове данни, като например boolean или String.

Езикът за програмиране Java позволява конструирането на съставни изрази (compound expressions) от няколко други по-малки такива, стига изискания тип данни от първата част на израза да съответства на типа данни от останалата. Ето пример за съставен израз:

```
1 * 2 * 3
```

В този определен пример, поредността в която изразът е изпълнен не е важна, защото резултатът от умножението е независим от реда; изходът винаги е един и също, независимо от поредността на умножението. Това обаче не е валидно за всички изрази. Например, следният израз дава различни резултати, в зависимост от това дали ще изпълним първо събирането или делението:

```
x + y / 100    // ambiguous
```

Можете да укажете точно как да бъде изпълнен даден израз, ползвайки скоби (и). Например, за да направи предишния пример ясен, можем да направим следното:

```
(x + y) / 100  // unambiguous, recommended
```

Ако не укажете изрично индикация за реда на операциите за изпълнение, то той се определя спрямо предимството на операторите, ползвани в израза. Операторите които имат по-високо предимство биват изпълнявани първи. Например, операторът за деление има по-високо предимство от този за добавяне. Следователно следните два statements са еквивалентни:

```
x + y / 100
x + (y / 100) // unambiguous, recommended
```

Когато пишете съставни изрази бъдете експлицитни и индикирайте чрез скоби кои оператори трябва да бъдат изпълнени първо. Тази практика прави кода по-лесен за четене и поддръжка.

STATEMENTS

Statements представляват грубо еквивалент на изречения в натуралните езици. Даден *statement* формира пълна единица на изпълнение. Следните типове изрази могат да бъдат направени в statement чрез терминирането на израза посредством точка и запетая (;).

- Изрази за присвояване
- Всяка употреба на ++ или —
- Извикване на методи
- Изрази за създаване на обекти

Подобни statements се наричат expression statements. Ето и няколко примера за такива:

```
// assignment statement
aValue = 8933.234;
// increment statement
aValue++;
// method invocation statement
System.out.println("Hello World!");
// object creation statement
Bicycle myBike = new Bicycle();
```

В допълнение към изразни (expression) statements съществуват два други вида statements - декларативни (declaration) statements и такива за контрол на потока (control flow statements). Декларативен statement декларира променлива. Вече сте виждали множество примери за декларативни statements:

```
// declaration statement
double aValue = 8933.234;
```

И накрая statements за контрол на потока регулират поредността на изпълнение в програмата.

БЛОКОВЕ

Блок представлява група от нула или повече statements между скоби {} и могат да бъдат ползвани навсякъде където е позволен единичен statement. Следният пример BlockDemo илюстрира употребата на блокове:

```
class BlockDemo {
    public static void main(String[] args) {
        boolean condition = true;
        if (condition) { // begin block 1
            System.out.println("Condition is true.");
        } // end block one
        else { // begin block 2
            System.out.println("Condition is false.");
        } // end block 2
    }
}
```

Statements за контрол на потока

Statements във вашите сорс файлове са основно изпълнявани отгоре надолу, в поредността в която се появяват. Statements за контрол на потока обаче променят потока на изпълнение посредством вземане на определени решения, looping, branching, позволяване на програмата условно да изпълни дадени блокове от код. Текущата секция описва тези decision-making statements (if-then, if-then-else, switch), looping statements (for, while, do-while) и branching statements (break, continue, return) поддържани от езика за програмиране Java.

Statements if-then и if-then-else

IF-THEN STATEMENT

Този statement е най-основния от всички такива за контрол на потока. Указва на програмата да изпълни определена секция от код само, ако даден тест се изпълни към true. Например, Bicycle класа може да позволи на спирачките да намалят скоростта на велосипеда само, ако той вече се намира в движение. Възможна имплементация на applyBrakes метода може да бъде следната:

```
void applyBrakes() {
   // the "if" clause: bicycle must be moving
   if (isMoving){
      // the "then" clause: decrease current speed
      currentSpeed--;
   }
}
```

Ако този тест се изпълни към false (означаващо че велосипедът не е в движение), се прескача към края на наличния if-else statement.

В допълнение, отварящите и затварящите скоби са опционални. Ако then клаузата съдържа един statement то може да изглежда така:

```
void applyBrakes() {
   // same as above, but without braces
   if (isMoving)
      currentSpeed--;
}
```

Решение за това кога да бъдат изпуснати скобите е въпрос на личен вкус. Изпускането им може да направи кода по-чуплив. Ако по-късно се добави втори statement към then клаузата, честа грешка би била да бъде пропуснато добавянето на вече задължителните скоби. Компилаторът не може да хване подобен тип грешка; просто ще получите грешни резултати.

IF-THEN-ELSE STATEMENT

Този statement предоставя вторичен път за изпълнение когато дадена if клауза се изпълни към false. Можете да ползвате if-then-else statement в applyBrakes метода за да предприемете действие, ако спирачките са натиснати когато велосипедът не е в движение. В този случай действието представлява просто принтиране на съобщение за грешка, показвайки че велосипедът е вече спрял.

```
void applyBrakes() {
   if (isMoving) {
      currentSpeed--;
   } else {
      System.err.println("The bicycle has already stopped!");
   }
}
```

Следната програма IfElseDemo присвоява оценка базирана на точковата стойност от тест - A е над 90%, B е за точки над 80% и т.н.

```
class IfElseDemo {
   public static void main(String[] args) {

      int testscore = 76;
      char grade;

      if (testscore >= 90) {
         grade = 'A';
      } else if (testscore >= 80) {
         grade = 'B';
      } else if (testscore >= 70) {
         grade = 'C';
      } else if (testscore >= 60) {
         grade = 'D';
      } else {
         grade = 'F';
```

```
        }
        System.out.println("Grade = " + grade);
    }
}
```

Изходът от програмата е:

```
Grade = C
```

Може би забелязахте, че стойността на testscode може да удовлетвори повече от един израз в съставен statement - 76 >= 70 и 76 >= 60. Обаче веднъж след удовлетворяване на условието, подходящият statement бива изпълнен (grade = ‘C’;) и останалите условия не биват изпълнени.

Switch statement

За разлика от if-then и if-then-else statements, switch statement може да има поредица от позитивни пътища на изпълнение. Switch работи с byte, short, char и int примитивни типове данни. Работи още с изброими типове (enumerated types), String класа и няколко специални класа, обгръщащи (wrapping) определени примитивни типове - Character, Byte, Short и Integer.

Следният примерен код SwitchDemo декларира int наименован month, чиято стойност репрезентира месец. Кодът визуализира името на месеца, базирано върху стойността на month, ползвайки switch statement.

```
public class SwitchDemo {
    public static void main(String[] args) {

        int month = 8;
        String monthString;
        switch (month) {
            case 1:  monthString = "January";
                     break;
            case 2:  monthString = "February";
                     break;
            case 3:  monthString = "March";
                     break;
            case 4:  monthString = "April";
                     break;
            case 5:  monthString = "May";
                     break;
            case 6:  monthString = "June";
                     break;
            case 7:  monthString = "July";
                     break;
            case 8:  monthString = "August";
                     break;
            case 9:  monthString = "September";
                     break;
            case 10: monthString = "October";
                     break;
```

```
            case 11: monthString = "November";
                     break;
            case 12: monthString = "December";
                     break;
            default: monthString = "Invalid month";
                     break;
        }
        System.out.println(monthString);
    }
}
```

В този случай August бива принтирано на стандартния изход.

Тялото на stitch statement е познато като switch блок. Statement в switch block може да бъде обозначен с един или повече case или default етикета. Switch statement изпълнява израза, след което изпълнява всички statements които следват съответния case label.

Можете също да визуализирате името на месеца и чрез if-then-else statements:

```
int month = 8;
if (month == 1) {
    System.out.println("January");
} else if (month == 2) {
    System.out.println("February");
}
...  // and so on
```

Решението за това дали да ползвате if-then-else statements или switch statement се базира върху четимостта и израза, който този statement тества. Даден if-then-else statement може да тества изрази базирани на обем от стойности или условия, докато switch statement тества изрази базирани на единичен integer, изброена стойност или String обект.

Друга точка на интерес представлява break statement. Всеки такъв терминира обграждащия го switch statement. Контролът на потока продължава с първия statement следващ switch блока. Break statements са необходими защото без тях statements в switch блоковете пропадат (fall through) - Всички statements след отговарящия case label се изпълняват в поредност, без значение израза на следващия case label, докато не се срещне break statement. Програмата SwitchDemoFallTrough показва statements в switch блок, които пропадат. Програмата визуализира месеца, кореспондиращ с integer month и последващите месец от годината:

```
public class SwitchDemoFallThrough {

    public static void main(String[] args) {
        java.util.ArrayList<String> futureMonths =
            new java.util.ArrayList<String>();

        int month = 8;

        switch (month) {
            case 1:  futureMonths.add("January");
```

```
            case 2:  futureMonths.add("February");
            case 3:  futureMonths.add("March");
            case 4:  futureMonths.add("April");
            case 5:  futureMonths.add("May");
            case 6:  futureMonths.add("June");
            case 7:  futureMonths.add("July");
            case 8:  futureMonths.add("August");
            case 9:  futureMonths.add("September");
            case 10: futureMonths.add("October");
            case 11: futureMonths.add("November");
            case 12: futureMonths.add("December");
                     break;
            default: break;
        }

        if (futureMonths.isEmpty()) {
            System.out.println("Invalid month number");
        } else {
            for (String monthName : futureMonths) {
               System.out.println(monthName);
            }
        }
    }
}
```

Това е изходът от кода:

```
August
September
October
November
December
```

Технически последният break не е задължителен, защото потокът изпада от горния switch statement. Ползването на break е препоръчително, за по-лесни бъдещи модификации на кода, по-малко податливи на грешки. Секция default отработва всички стойности, непопадащи в никой от case секциите.

Следната програма SwitchDemo2 показва как даден statement може да има множество case labels. Примерният код калкулира броя дни в определен месец:

```
class SwitchDemo2 {
    public static void main(String[] args) {

        int month = 2;
        int year = 2000;
        int numDays = 0;
```

```
        switch (month) {
            case 1: case 3: case 5:
            case 7: case 8: case 10:
            case 12:
                numDays = 31;
                break;
            case 4: case 6:
            case 9: case 11:
                numDays = 30;
                break;
            case 2:
                if (((year % 4 == 0) &&
                     !(year % 100 == 0))
                     || (year % 400 == 0))
                    numDays = 29;
                else
                    numDays = 28;
                break;
            default:
                System.out.println("Invalid month.");
                break;
        }
        System.out.println("Number of Days = "
                           + numDays);
    }
}
```

Това е изходът от горния код:

```
Number of Days = 29
```

УПОТРЕБА НА **STRINGS** В **SWITCH STATEMENTS**

След Java SE 7 включително може да ползвате String обект в израз на switch statement. Следният примерен код StringSwitchDemo визуализира номера на месеца, базирано върху стойността в String на име month:

```
public class StringSwitchDemo {

    public static int getMonthNumber(String month) {

        int monthNumber = 0;

        if (month == null) {
            return monthNumber;
        }

        switch (month.toLowerCase()) {
```

```
        case "january":
            monthNumber = 1;
            break;
        case "february":
            monthNumber = 2;
            break;
        case "march":
            monthNumber = 3;
            break;
        case "april":
            monthNumber = 4;
            break;
        case "may":
            monthNumber = 5;
            break;
        case "june":
            monthNumber = 6;
            break;
        case "july":
            monthNumber = 7;
            break;
        case "august":
            monthNumber = 8;
            break;
        case "september":
            monthNumber = 9;
            break;
        case "october":
            monthNumber = 10;
            break;
        case "november":
            monthNumber = 11;
            break;
        case "december":
            monthNumber = 12;
            break;
        default:
            monthNumber = 0;
            break;
    }

    return monthNumber;
}
```

```
    public static void main(String[] args) {

        String month = "August";

        int returnedMonthNumber =
            StringSwitchDemo.getMonthNumber(month);

        if (returnedMonthNumber == 0) {
            System.out.println("Invalid month");
        } else {
            System.out.println(returnedMonthNumber);
        }
    }
}
```

Изходът от кода е 8.

String в switch израза се сравнява с асоциирания с всеки case label израз, като се ползва String.equals метода. За да може StringSwitchDemo примера да приема всеки месец, независимо от case, month се конвертира към lowercase (малки букви; чрез toLowerCase метода), и всички асоциирани с case labels стрингове са lowercase.

Забележка*:* Този пример проверява дали изразът в горния *switch statement е null.* Уверете се*,* че изразът в даден *switch statement* не е *null,* за да предотвратите хвърлянето на *NullPointerException.*

While и do-while statements

Даден while statement постоянно изпълнява блок от statements докато конкретно условие е true. Неговият синтаксис може да бъде следният:

```
while (expression) {
     statement(s)
}
```

Този while statement изпълнява израз *expression*, който трябва да връща boolean стойност. Ако изразът се изпълнява към true, даденият while statement изпълнява *statement(s)* в while блока. While statement продълважа тестването на израза и изпълнението на блока докато този израз не върне false. Ползвайки while statement за принтиране на стойности от 1 до 10 може да бъде постигнато както следва в WhileDemo програмата:

```
class WhileDemo {
    public static void main(String[] args){
        int count = 1;
        while (count < 11) {
            System.out.println("Count is: " + count);
            count++;
        }
    }
}
```

Можете да имплементирате безкраен цикъл ползвайки while statement както следва:

```
while (true){
   // your code goes here
}
```

Програмният език Java предоставя още do-while statement, който изглежда по следния начин:

```
do {
    statement(s)
} while (expression);
```

Разликата между do-while и while е, че do-while изпълнява своя израз в дъното на цикъла вместо най-отгоре. Следователно, този statement в do блока е винаги изпълнен поне веднъж, което е демонстрирано в следната DoWhileDemo програма:

```
class DoWhileDemo {
   public static void main(String[] args){
      int count = 1;
      do {
         System.out.println("Count is: " + count);
         count++;
      } while (count < 11);
   }
}
```

for statement

for statement е компактен начин за итериране на набор от стойности. Програмистите често го реферират като “for loop”, поради начина по който повторяемо цикли докато дадено условие не бъде удовлетворено. Основната форма на for statement може да бъде описана както следва:

```
for (initialization; termination;
    increment) {
   statement(s)
}
```

Когато се ползва тази версия на for statement, имайте предвид че:

- Горният *initialization* израз инициализира цикъла; изпълнява се веднъж в началото на цикъла.
- Когато *termination* израза се изпълни към false, цикълът бива прекратен.
- Горният *increment* израз бива извикван след всяка итерация на цикъла; абсолютно приемливо е този израз да инкрементира или декрементира дадена стойност.

Следната програма ForDemo ползва основната форма на for statement за да принтира числата от 1 до 10 в стандартния изход:

```
class ForDemo {
   public static void main(String[] args){
        for(int i=1; i<11; i++){
             System.out.println("Count is: " + i);
        }
   }
}
```

Изходът на програмата е:

```
Count is: 1
Count is: 2
Count is: 3
Count is: 4
Count is: 5
Count is: 6
Count is: 7
Count is: 8
Count is: 9
Count is: 10
```

Забележете как кодът декларира променлива в инициализиращия израз. Обхватът на тази променлива се простира от нейната декларация до края на блока на горния for statement, следователно може да бъде ползвана също и в терминиращия и инкрементиращ израз. Ако променливата която контролира гормия for statement не е нужна извън цикъла, то най-добре е да я декларираме в инициализиращия израз. Имената i, j и k за често ползвани за контрол на for цилки; декларирането им в инициализиращия израз ограничава техния живот и редуцира грешки.

Трите израза във for цикъла са опционални; безкраен цикъл може да бъде създаден както следва:

```
// infinite loop
for ( ; ; ) {

   // your code goes here
}
```

for statement има също друга форма, създадена за итерация на Collections и масиви. Тази форма понякога бива реферирана като подобрен (enhanced) for statement и може да бъде ползвана за по-компактни и лесночетими цикли. За целите на демонстрацията разгледайте следния масив, съдържащ числата от 1 до 10:

```
int[] numbers = {1,2,3,4,5,6,7,8,9,10};
```

Следната програма EnhancedForDemo ползва подобрения for за да изцикли масива:

```
class EnhancedForDemo {
   public static void main(String[] args){
       int[] numbers =
          {1,2,3,4,5,6,7,8,9,10};
       for (int item : numbers) {
          System.out.println("Count is: " + item);
       }
   }
}
```

В този пример променливата item държи текущата стойност от numbers масива. Изходът от тази програма е същият като преди:

```
Count is: 1
Count is: 2
Count is: 3
```

```
Count is: 4
Count is: 5
Count is: 6
Count is: 7
Count is: 8
Count is: 9
Count is: 10
```

Препоръчителна е употребата на тази форма на for statement вместо общата форма когато това е възможно.

Разклоняващи се statements

BREAK STATEMENT

Този statement има две форми - labeled и unlabeled. Вече видяхте unlabeled формата в предишната дискусия на switch statement. Можете да ползвате също и unlabeled break за да терминирате for, while или do-while цикъл, както е показано в следната BreakDemo програма:

```
class BreakDemo {
   public static void main(String[] args) {

      int[] arrayOfInts =
         { 32, 87, 3, 589,
           12, 1076, 2000,
           8, 622, 127 };
      int searchfor = 12;

      int i;
      boolean foundIt = false;

      for (i = 0; i < arrayOfInts.length; i++) {
         if (arrayOfInts[i] == searchfor) {
            foundIt = true;
            break;
         }
      }

      if (foundIt) {
         System.out.println("Found " + searchfor + " at index " + i);
      } else {
         System.out.println(searchfor + " not in the array");
      }
   }
}
```

Тази програма търси за числото 12 и масив. Горният break statement в подчертан шрифт прекъсва външния for loop когато стойността бива открита. Контрола на потока се измества към следващия statement след цикъла. Изходът от програмата е:

```
Found 12 at index 4
```

Unlabeled break statement терминира най-вътрешния switch, for, while или do-while statement, но даден labeled break терминира външния statement. Следната програма BreakWithLabelDemo е подобна на предишната, но използва вложени for цикли за търсене на дадена стойност в двумерен (two-dimensional) масив. Когато стойността бъде открита, даден labeled break терминира външния for loop (labeled "search"):

```
class BreakWithLabelDemo {
    public static void main(String[] args) {

        int[][] arrayOfInts = {
            { 32, 87, 3, 589 },
            { 12, 1076, 2000, 8 },
            { 622, 127, 77, 955 }
        };
        int searchfor = 12;

        int i;
        int j = 0;
        boolean foundIt = false;

    search:
        for (i = 0; i < arrayOfInts.length; i++) {
            for (j = 0; j < arrayOfInts[i].length;
                 j++) {
                if (arrayOfInts[i][j] == searchfor) {
                    foundIt = true;
                    break search;
                }
            }
        }

        if (foundIt) {
            System.out.println("Found " + searchfor + " at " + i + ", " + j);
        } else {
            System.out.println(searchfor + " not in the array");
        }
    }
}
```

Това е изходът от програмата:

```
Found 12 at 1, 0
```

Горният break statement терминира съответния labeled statement; не пренасочва потока на контрол към този label. Контролният поток е трансфериран към непосредствения statement, следващ labeled (terminated) statement.

CONTINUE STATEMENT

Този statement пропуска текущата итерация на даден for, while или do-while цикъл. Тази unlabeled форма пропуска към края на тялото на най-вътрешния цикъл и изпълнява булевият израз, контролиращ цикъла. Следната програма ContinueDemo обхожда даден String, преброявайки появите на буква “p”. Ако текущият знак не е “p”, тогава посочения continue statement пропуска останалата част от цикъла и продължава към следващия знак. Ако пък знакът е “p”, програмата инкрементира броя на буквите.

```
class ContinueDemo {
   public static void main(String[] args) {

      String searchMe = "peter piper picked a " + "peck of pickled peppers";
      int max = searchMe.length();
      int numPs = 0;

      for (int i = 0; i < max; i++) {
         // interested only in p's
         if (searchMe.charAt(i) != 'p')
            continue;

         // process p's
         numPs++;
      }
      System.out.println("Found " + numPs + " p's in the string.");
   }
}
```

Ето това е изходът на програмата:

```
Found 9 p's in the string.
```

За да видите ефекта по-ясно, опитайте да изтриете горния continue statement и прекомпилирайте. Когато пуснете програмата отново, броят ще бъде грешен, казвайки че са открити 35 “p” вместо 9.

Даден labeled continue statement пропуска текущата итерация на външен цикъл, маркирайки го със съответния label. Следната примерна програма ContinueWithLabelDemo ползва вложени цикли за търсене на подниз (substring) в друг такъв. Два вложени цикъла са задължителни - единият за итерация на подниза, другият за итерация на низа в който се търси. Следната програма ContinueWithLabeldemo ползва labeled форма на continue, за да пропусне итерацията на външния цикъл.

```
class ContinueWithLabelDemo {
   public static void main(String[] args) {
      String searchMe = "Look for a substring in me";
      String substring = "sub";
      boolean foundIt = false;
```

```
        int max = searchMe.length() -
                substring.length();

    test:
        for (int i = 0; i <= max; i++) {
            int n = substring.length();
            int j = i;
            int k = 0;
            while (n-- != 0) {
                if (searchMe.charAt(j++) != substring.charAt(k++)) {
                    continue test;
                }
            }
            foundIt = true;
                break test;
        }
        System.out.println(foundIt ? "Found it" : "Didn't find it");
    }
}
```

Ето и изходът от програмата:

```
Found it
```

RETURN STATEMENT

Последният от разклоняващите се statements е т.нар. return staement. Той излиза от текущия метод, а контрола на потока се връща към мястото, от където е бил извикан. return statement има две форми - една която връща стойност, и такава която не връща нищо. За да върнете стойност (или съответно израз, калкулиращ стойност) просто я подайте непосредствено след ключовата дума *return*.

```
return ++count;
```

Типът данни на връщаната стойност трябва да съвпата с посочената за връщане такава в декларацията на метода. Когато даден метод бъде деклариран като void, ползвайте формата на return, която не връща стойност.

```
return;
```

Резюме на statements за контрол на потока

Разгледаният if-them statement е най-базовият от всички statements за контрол на потока. Указва на програмата да изпълни определена секция код само, ако даден тест се изпълни към true. if-then-else statement предоставя вторичен път на изпълнение когато дадена “if” клауза върне false. За разлика от if-then и if-then-else, switch statement позволява за всякаква бройка от възможни пътища на изпълнение. Разгледаните по-горе while и do-while statements непрекъснато изпълняват блок от statements докато дадено условие връща true. Разликата между do-while и while е, че първият изпълнява изразите си в do блока винаги поне веднъж. Горегазгледаният for statement предоставя компактен начин за итерация на набори от стойности. Има две форми, едната от които е пригодена за циклене на колекции и масиви.

https://docs.oracle.com/javase/tutorial/java/nutsandbolts/index.html

Анотации

Анотациите представляват метаданни, предоставящи информация за програма, като те самите не са част от нея. Анотациите нямат директен ефект върху операциите, резултат от анотирания код.

Анотациите имат редица приложения, сред които:

- Информация за компилатора - Анотациите могат да бъдат ползвани от компилатора с цел засичане на грешки (errors) или подтискане на предупреждения (warnings).
- **Compile-time** и **deployment-time** обработка - Софтуерните инструменти могат да обработват информацията от анотациите с цел генериране на програмен код, XML файлове и т.н.
- **Runtime** обработка - Някои анотации са налични за употреба в runtime.

Формат на анотация

В най-простата си форма една анотация изглежда по подобен начин:

```
@Entity
```

Знакът @ индикира за компилатора че следващото го е именно анотация. В следващия пример името на анотацията е Override:

```
@Override
void mySuperMethod() { ... }
```

Анотацията може да включва елементи, които могат да бъдат конкретно наименовани или не, като те имат стойности:

```
@Author(
  name = "Benjamin Franklin",
  date = "3/27/2003"
)
class MyClass() { ... }
```

или

```
@SuppressWarnings(value = "unchecked")
void myMethod() { ... }
```

Ако има само един наличен елемент *value*, тогава името може да бъде изпуснато, като:

```
@SuppressWarnings("unchecked")
void myMethod() { ... }
```

Ако анотацията няма елементи, тогава скобите могат да бъдат изпуснати, както при примера с @Override.

Възможно е още употребата на множество анотации към една и съща декларация:

```
@Author(name = "Jane Doe")
@EBook
class MyClass { ... }
```

Ако анотациите са от един и същи тип, то повторната им употребата се нарича повтарящо се анотиране:

```
@Author(name = "Jane Doe")
@Author(name = "John Smith")
```

```
class MyClass { ... }
```

Повтарящите се анотации се поддържат от Java SE 8.

Анотацията може да бъде един от типовете, дефинирани в java.lang или java.lang.annotation пакетите на Java SE API. В предишния пример, Override и SuppressWarnings са предефинирани анотации. Възможно е също дефинирането на собствени анотации. Author и Ebook в предишния пример са допълнително дефинирани анотации.

Къде могат да бъдат ползвани анотациите?

Анотациите могат да бъдат приложени върху декларации: на класове, полета, методи и други програмни елементи. Когато се използват върху декларация, всяка анотация по конвенция се поставя на нов ред.

От Java SE 8 анотациите могат да бъдат приложени при употребата на типове. Ето и няколко примера:

- Израз за инстанцииране на клас:

```
new @Interned MyObject();
```

- Type cast:

```
myString = (@NonNull String) str;
```

- implements клауза:

```
class UnmodifiableList<T> implements
    @Readonly List<@Readonly T> { ... }
```

- Декларация при хвърляне на изключение :

```
void monitorTemperature() throws
    @Critical TemperatureException { ... }
```

Горната форма на анотиция се нарича *type annotation*.

Деклариране на **annotation type**

Много анотации заместват коментарите в кода.

Да предположим, че е наложена практиката при започване тялото на класа да бъде предоставяна важна информация под формата на коментар, например:

```
public class Generation3List extends Generation2List {
  // Author: John Doe
  // Date: 3/17/2002
  // Current revision: 6
  // Last modified: 4/12/2004
  // By: Jane Doe
  // Reviewers: Alice, Bill, Cindy

  // class code goes here
}
```

За да добавим същите метаданни с анотация е необходимо първо да дефинираме тип анотация. Синтаксисът е следния:

```
@interface ClassPreamble {
  String author();
```

```
  String date();
  int currentRevision() default 1;
  String lastModified() default "N/A";
  String lastModifiedBy() default "N/A";
  // Note use of array
  String[] reviewers();
}
```

Дефиницията на тип анотация изглежда подобно на тази за интерфейс, като пред ключовата дума *interface* бива поставен знак "@" (@ = AT, като при annotation type). Типовете анотации са форма на интерфейс.

Тялото на горе дефинираната анотация съдържа декларация на елементи, което изглежда подобно на тази за методи. Могат да бъдат дефинирани и опционални стойности по подразбиране.

След като типа анотация е дефиниран, той може да бъде използват с попълнени прилежащите му стойности, например:

```
@ClassPreamble (
  author = "John Doe",
  date = "3/17/2002",
  currentRevision = 6,
  lastModified = "4/12/2004",
  lastModifiedBy = "Jane Doe",
  // Note array notation
  reviewers = {"Alice", "Bob", "Cindy"}
)
public class Generation3List extends Generation2List {

// class code goes here

}
```

Бележка: За да може информация за @ClassPreamble да бъде отразена в генерираната Javadoc документация, то трябва да анотирате самата дефиниция на @ClassPreamble - с @Documented анотацията:

```
// import this to use @Documented
import java.lang.annotation.*;

@Documented
@interface ClassPreamble {

  // Annotation element definitions

}
```

Предефинирани типове анотации

Набор от типове анотации са предефинирани в самото Java SE API. Някои типове анотации се ползват от Java компилатора, а някои се ползват при дефиницията на други анотации.

Типове анотации ползвани от езика Java

Предефинираните типове анотации от java.lang пакета са @Deprecated, @Override и @SupporessWarnings.

@Deprecated анотацията индикира, че маркирания елемент е deprecated (остарял вариант на реализация) и вече не трябва да бъде използван. Компилаторът генерира предупреждение когато дадена програма ползва метод, клас или поле, анотирани с @Deprecated анотацията. Когато даден елемент е deprecated, той трябва да бъде документиран ползвайки Javadoc @deprecated тага, подобно на демонстрираното в следващия пример. Употребата на знак “@“ в Javadoc коментарите и в самите анотации не е съвпадение: те са свързани концептуално. Обърнете внимание, че Javadoc тага започва с малко “d”, а самата анотация с голямо “D”.

```
/ Javadoc comment follows
   /**
    * @deprecated
    * explanation of why it was deprecated
    */
   @Deprecated
   static void deprecatedMethod() { }
}
```

@Override анотацията информира компилатора, че елементът е предназначен за overriding на елемента, деклариран в суперкласа.

```
 // mark method as a superclass method
   // that has been overridden
   @Override
   int overriddenMethod() { }
```

Употребата на горната анотация при overriding на методи не е задължителна, но спомага за предотвратяването на грешки. Ако даден метод е маркиран с @Override и не извърши коректен overriding на метода от суперкласа, заради анотацията ще възникне предупреждение.

```
   // use a deprecated method and tell
   // compiler not to generate a warning
   @SuppressWarnings("deprecation")
    void useDeprecatedMethod() {
        // deprecation warning
        // - suppressed
        objectOne.deprecatedMethod();
    }
```

Всяко предупреждение от компилатора може да бъде категоризирано. В спецификацията на езика Java фигурират две категории: *deprecation* и *unchecked*. Unchecked предупреждение може да възникне при употребата на стар код, написан преди появата на generics. За да подтиснете множество категории предупреждения, използвайте следния синтаксис:

@SuppressWarnings({"unchecked", "deprecation"})

При прилагането към метод или конструктор, @SafeVarargs анотацията предполага, че кодът не извършва потенциално несигурни операции върху varargs параметъра. При употребата й unchecked предупрежденията биват подтиснати.

@FunctionalInterface анотацията е въведена в Java SE 8, като индикира, че дадена декларацията на тип е предназначена за функционален интерфейс, дефинирано от спецификацията на езика Java.

Анотации, прилагани към други анотации

Анотациите с приложение за други анотации се наричат мета-анотации. Съществуват няколко такива, дефинирани в java.lang.annotation пакета.

@Retention анотацията специфицира как маркираната анотация бива запазена:

- RetentionPolicy.SOURCE - Маркираната анотация е задържана само на ниво сорс код и бива игнорирана от компилатора.
- RetentionPolicy.CLASS - Маркираната анотация се държи от компилатора по време на компилация, но бива игнорирана от Java виртуалната машина (JVM).
- RetentionPolicy.RUNTIME - Маркираната анотация е задържана от JVM, съответно може да бъде ползвана от runtime средата.

@Documented анотацията индикира, че елементите върху които се ползва трябва да бъдат документирани ползвайки Javadoc инструмент. (По подразбиране анотациите не се включват в Javadoc.)

@Target анотацията маркира друга анотация, за да ограничи върху кои елементи от езика може да бъде приложена тя. Target анотацията специфицира един от следните типове елементи като стойност:

- ElementType.ANNOTATION_TYPE може да бъде приложена към тип анотация.
- ElementType.CONSTRUCTOR може да бъде приложена към конструктор.
- ElementType.FIELD може да бъде приложена към поле (field) или характеристика (property).
- ElementType.LOCAL_VARIABLE може да бъде приложена към локална променлива.
- ElementType.METHOD може да бъде приложена на ниво метод.
- ElementType.PACKAGE може да бъде приложена към декларацията на пакет.
- ElementType.PARAMETER може да бъде приложена към параметри на метод.
- ElementType.TYPE може да бъде приложена към всеки елемент на класа.

@Inherited анотацията индикира, че типа анотация може да бъде наследен от супер класа. (Това не е така по подразбиране.) Когато потребителят направи проверка за типа анотация и класът няма анотация за този тип, неговият суперклас бива запитан за типа анотация. Тази анотация важи само за клас декларации.

@Repeatable анотацията е въведена в Java SE 8, като индикира, че маркираната анотация може да бъде приложена повече от веднъж към употребата на същата декларация или тип.

Типове анотации и **pluggable type** системи

Преди Java SE 8 анотациите можеха да бъдат приложени само към декларации. От Java SE 8 насам вече могат да се прилагат към всяка употреба на тип. Това означава, че анотациите могат да бъдат

ползвани навсякъде където ползвате тип. Пример за това къде се ползват типове са изрази за създаване инстанции на класове (new), casts, implements клаузи и throws клаузи. Тази форма на анотация се нарича *type annotation.*

Типовите анотации са създадени с цел подобряване анализа на начина за подсигуряване по-строга проверка на типа в Java програмите. Java SE 8 не предоставя type checking framework, но позволява да бъдат написали (или свалени) type checking frameworks, имплементиращи един или повече pluggable модули, които биват ползвани в тандем с компилатора.

Например, можете да подсигурите това, че на дадена променлива в програмата никога няма да бъде присвоена null стойност, чрез което да избегнете хвърлянето на *NullPointerException*. Можете да напишете проверка за това. Съответно бихте могли да модифицирате кода, така че да анотира тази конкретна променлива, индикирайки че никога няма да й бъде присвоена стойност null. Декларацията на тази променлива може да изглежда така:

```
@NonNull String str;
```

Когат компилирате кода, включвайки NonNull модула през командния ред, компилатора принтира предупреждение, ако засече потенциален проблем, позволявайки да модифицирате кода, за да избегнете грешка. След като поправите кода, за да предотвратите възникването на предупреждения, тази конкретна грешка няма да възникне когато програмата работи.

Можете да ползвате множество type-checking модули, като всеки от тях да проверява за различен вид грешки. По този начин може да надградите една Java type система, добавяйки специфични проверки когато и където желаете.

С разумната употреба на type annotations и наличието на pluggable type checkers може да бъде писан по-силен и неподатлив на грешки код.

В много случаи няма да бъде наложително писането на собствени type checking модули. Има външни вече разработени такива. Например, при желание от ваша страна можете да се възползвате от т.нар. Checker Framework, създаден от University of Washington. Този framework включва NonNull модул, както и модул за стандартни изрази и mutex lock модул. За повече информация вижте Checker Framework.

Повторяеми анотации

Съществуват някои ситуации, където бихте желали да приложите същата анотация към декларация или употреба на тип. От Java SE 8 насам са въведени т.нар. *repeating annotations*, които позволяват горепосоченото.

Например, пишете код за ползване на timer услуга, позволяваща стартирането на метод по определено време и график, подобно на UNIX cron service. В такъв случай бихте желали да настроите timer за изпълняването на метод doPeriodicCleanup, в последния ден на месеца и всеки петък в 23:00ч. За да настроите подобен timer, създайте @Schedule анотация и я приложете два поредни пъти към doPeriodicCleanup метода. Първата употреба специфицира последния ден на месеца, а втората такава - петък в 23:00ч, демонстрирано чрез следния примерен код:

```
@Schedule(dayOfMonth="last")
@Schedule(dayOfWeek="Fri", hour="23")
public void doPeriodicCleanup() { ... }
```

Предишният пример прилага анотация към метод. Можете да повтаряте анотация навсякъде където бихте ползвали стандартна такава. Например, имате клас за обработка на изключения при

неоторизиран достъп. Анотирате класа с една @Alert анотация за мениджъри и друга за администратори:

```
@Alert(role="Manager")
@Alert(role="Administrator")
public class UnauthorizedAccessException extends SecurityException { ... }
```

За целите на съвместимостта, повторяемите анотации се пазят в контейнер анотация (container annotation), която бива автоматично генерирана от Java компилатора. За да бъде извършено това са нужни поне две декларации във вашия код.

Стъпка 1: Декларирайте повторяем тип анотация

Типа анотация трябва да бъде маркиран с @Repeatable мета-анотацията. Следният пример дефинира повторяем тъп анотация @Schedule:

```
import java.lang.annotation.Repeatable;


@Repeatable(Schedules.class)
public @interface Schedule {
  String dayOfMonth() default "first";
  String dayOfWeek() default "Mon";
  int hour() default 12;
}
```

Стойността на @Repeatable мета-анотацията в скобите представлява контейнер анотацията, която Java компилатора генерира, за да запази повторяемите анотации. В този пример типа анотация-контейнер е Schedules, така че потворяемите анотации @Schedule се пазят в @Schedules анотацията.

Прилагането на една и съща анотация към декларация без първо да се декларира като повторяема резултира в грешка при компилация.

Стъпка 2: Декларация на тип анотация-контейнер

Типа анотация-контейнер трябва да има елемент *value* от тип масив. Компонентният тип на масива трябва да бъде повторяем тип анотация. Декларацията за Schedules, съдържаща тип анотация е както следва:

```
public @interface Schedules {
   Schedule[] value();
}
```

Получаване на анотации

В Reflection API има няколко метода, които могат да бъдат използвани за извличането на анотации. Поведението на методите които връщат единична анотация, като AnnotatedElement.getAnnotationByType(Class<T>), ще върнат единична анотация, ако само една такава от извикания елемент е налична. Ако повече от една анотация от извикания елемент е налична, можете да ги получите чрез вземане първо на тяхната анотация-контейнер. По този начин стар код (legacy code) продължава да работи. Други методи са въведени в Java SE 8, сканиращи чрез контейнер-анотацията за връщане на множество анотации наведнъж, като например AnnotatedElement.getAnnotations(Class<T>).

Съображения относно дизайна

Когато се прави дизайн на тип анотация трябва да се вземе предвид кардиналността на анотациите от този тип. Вече е възможно тази анотация да бъде използвана нула пъти, един или много - ако е маркирана като @Repeatable. Възможно е също да се ограничи приложението на тип анотация чрез употребата на @Target мета-анотацията. Например, можете да създадете повторяем тип анотация, който може да бъде прилаган само върху методи и полета. Важно е дизайна на типа анотация да бъде внимателно направен, подсигурявайки гъвкава и мощна употреба от страна на програмиста, който я ползва.

https://docs.oracle.com/javase/tutorial/java/annotations/index.html

Generics

Generics прибавят стабилност към кода, правейки възможно повече бъгове да бъдат засечени по време на компилация.

Generics позволяват типове (класове и интерфейси) да бъдат параметри при дефинирането на класове, интерфейси и методи. Подобно на по-познатите формални параметри, използвани при декларацията на даден метод, типовите праметри предоставят начин за преизползване на същия код с различен вход. Разликата е, че входът при формалните параметри са стойности, докато входът при типовите параметри са типове.

Ползващият generics код има много преимущества:

- По-силни type checks по време на компилация.

Java компилаторът прилага силна проверка за типа относно generic кода и генерира грешки, ако засече нарушения. Поправянето на грешки по време на компилация е по-лесно от поправянето на грешки по време на изпълнение, които могат да бъдат трудни за откриване.

- Елиминиране на casts.

Следният примерен код без generics изисква casting:

```
List list = new ArrayList();
list.add("hello");
String s = (String) list.get(0);
```

След пренаписването на горния код ползвайки generics, casting вече не се изисква:

```
List<String> list = new ArrayList<String>();
list.add("hello");
String s = list.get(0);   // no cast
```

- Позволяване имплементацията на алгоритми с широко приложение.

Чрез употребата на generics програмистите могат да имплементират алгоритми с широко приложение, които работят върху колекции от различни типове, могат да бъдат персонализирани (customized) и са type-safe и по-лесни за четене.

Generic типове

Generic тип представлява generic клас или интерфейс, който е параметъризиран вместо типизиран. Следният Box клас ще бъде модифициран, за да демонстрира концепцията.

Прост Box клас

Нека разгледаме non-generic *Box* клас, който оперира върху обекти от всеки тип. Необходимо е да предостави два метода: set, който добавя обект към box, и get, който го връща:

```
public class Box {
    private Object object;

    public void set(Object object) { this.object = object; }
    public Object get() { return object; }
}
```

След като методите приемат или връщат тип *Object*, можете да подадете каквото пожелаете, с изключение на примитивни типове данни. Не съществува начин за верифициране употребата на класа по време на компилация. Дадена част от кода може да поставя Integer в box и да очаква Integer при получаването, докато друга може по погрешка да подава String, което ще резултира в грешка по време на изпълнение.

Generic версия на Box класа

Даден generic клас се дефинира в следния формат:

```
class name<T1, T2, ..., Tn> { /* ... */ }
```

Секцията с параметри за типа, обособена чрез счупени скоби (<>), следва името на класа. Специфицира параметри за типа (наречени още типови променливи) T1, T2, …, и Tn.

За да пригодите Box класа към употреба на generics трябва да създадете generic типова декларация чрез променяне "*public class Box*" на "*public class Box<T>*" в кода. Това въвежда типова променлива T, която може да бъде ползвана навсякъде вътре в класа.

С тази промяна клас Box изглежда по следния начин:

```
/**
 * Generic version of the Box class.
 * @param <T> the type of the value being boxed
 */
public class Box<T> {
    // T stands for "Type"
    private T t;

    public void set(T t) { this.t = t; }
    public T get() { return t; }
}
```

Забележете, че всички появи на Object биват заменени с T. Типова променлива може да бъде всеки непримитивен тип: всеки клас, интерфейс, масив или друг тип променлива.

Същата техника може да бъде приложена при създаването на generic интерфейси.

Конвенция за наименоване на типови параметри

По конвенция имената на типовите параметри са единични главни букви. Това целенасочено създава ярък контраст спрямо конвенциите за наименоване на променливи: без тази конвенция би било

трудно да се направи разликата между типови променливи и имена на обикновени класове и интерфейси.

Най-често употребяваните имена за типови параметри са:

- E - Element (ползва се широко в Java Collections Framework)
- K - Key
- N - Number
- T - Type
- V - Value
- S, U, V и т.н. - 2-ри, 3-ти, 4-ти тип

Можете да забележите тези имена в Java SE API и текущия документ.

Извикване и инстанцииране на generic тип

За да реферирате generic Box класа от вашия код трябва първо да извършите извикване на generic типа, което замества T с някои конкретни стойности, например Integer:

```
Box<Integer> integerBox;
```

За извикването на generic тип може да се мисли по подобен начин на обикновеното извикване на метод, като вместо подаване на аргумент към него подавате типов аргумент - в случая Integer - на самия Box клас.

Бележка: Терминология относно типов параметър и типов аргумент - Много разработчици ползват темрмините “типов параметър” и “типов аргумент” взаимозаменяемо, но тези термини не означават едно и също нещо. По време на писането за създаване на параметъризиран тип се подават типови аргументи. Следователно T във Foo<T> представлява типов параметър, докато String във Foo<String> f представлява типов аргумент.

Както при декларирането на променливи, горният код всъщност не създава нов Box обект. Той декларира, че ingegerBox ще държи референция към “Box от Integer”, което е начина на прочитане на Box<Integer>.

Извикването на generic типове е широко познато като параметъризиран тип.

За да инстанциирате този клас ползвайте ключовата дума *new*, като поставите <Integer> между името на класа и кръглите скоби:

```
Box<Integer> integerBox = new Box<Integer>();
```

Диамантът (diamond)

След Java SE 7 можете да заместите типовите аргументи, необходими за извикването на конструктор на generic клас с празни скоби за типови аргументи (<>), като компилаторът определя типа в зависимост от контекста. Тази двойка от счупени скоби <> бива неофициално наричана диамант. Наприме, можете да създадете инстанция на Box<Integer> както следва:

```
Box<Integer> integerBox = new Box<>();
```

Множество типови параметри

Даден generic клас може да има множестви типови параметри. Например, generic OrderedPair клас, който имплементира generic Pair интерфейс:

```
public interface Pair<K, V> {
   public K getKey();
   public V getValue();
}

public class OrderedPair<K, V> implements Pair<K, V> {

   private K key;
   private V value;

   public OrderedPair(K key, V value) {
            this.key = key;
            this.value = value;
   }

   public K getKey()   { return key; }
   public V getValue() { return value; }
}
```

По-долу се създават две инстанции на OrderedPair класа:

```
Pair<String, Integer> p1 = new OrderedPair<String, Integer>("Even", 8);
Pair<String, String>  p2 = new OrderedPair<String, String>("hello", “world");
```

Кодът new OrderedPair<String, Integer> инстанциира K като String и V като Integer. Следователно типовите параметри на OrderedPair конструктора са String и Integer. Благодарение на наличния в Java autoboxing е валидно също и подаването на String и int.

Както бе споменато по-горе, тъй като Java компилаторът може да подразбере типовете на K и V от декларацията OrderedPair<String, Integer>, то при инициализацията може да бъде ползвана diamond нотацията:

```
OrderedPair<String, Integer> p1 = new OrderedPair<>("Even", 8);
OrderedPair<String, String>  p2 = new OrderedPair<>("hello", “world");
```

При създаването на generic интерфейс се следват същите конвенции като тези за създаването на generic клас.

Праметъризирани типове

Можете още да заместите типови параметри (например K или V) с параметъризиран тип (например List<String>). Ето и пример, ползващ OrderedPair<K, V>:

```
OrderedPair<String, Box<Integer>> p = new OrderedPair<>("primes", new Box<Integer>(…));
```

https://docs.oracle.com/javase/tutorial/java/generics/index.html

Collections

Колекция - наричана още контейнер - представлява обект, който групира множество елементи в един общ елемент. Колекциите се ползват за запазване, извличане, манипулация и агрегиране на данни. Обикновено репрезентират данни, формиращи естествена група, например ръка при карти за игра

(колекция от карти), mail folder (колекция от писма), телефонна директория (свързване на имена към телефонни номера).

Какво представлява **Collections Framework?**

Collections framework е унифицирана архитектура за репрезентиране и манипулиране на колекции. Всички collections frameworks съдържат следното:

- Интерфейси: Представляват абстрактни типове данни, които репрезентират колекции. Интерфейсите позволяват колекциите да бъдат манипулирани независимо от детайлите на тяхната репрезентация. В обекто-ориентираните езици интерфейсите обикновено формират йерархия.
- Имплементации: Това са конкретните имплементации на collection интерфейсите. В същността си представляват преизползваеми структури от данни.
- Алгоритми: Представляват методи за извършване на полезни изчисления, като търсене и сортиране, върху обекти имплементиращи collection интерфейси. Алгоритмите са полиморфични (polymorphic): т.е. същият метод може да бъде използван върху много различни имплементации на подходящия collection интерфейс. В същността си алгоритмите са преизползваема функционалност.

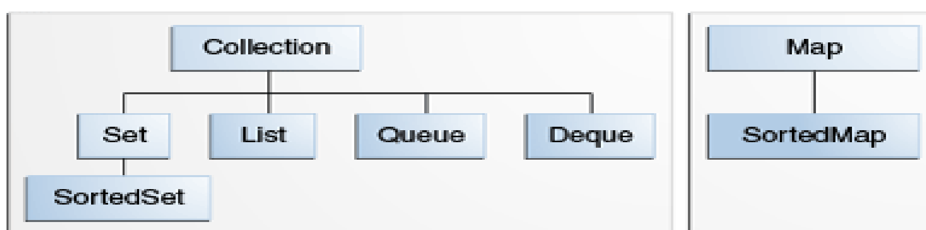
Интерфейси

Основните collection интерфейси енкапсулират различни типове колекции, показани на изображението по-долу. Тези интерфейси позволяват колекциите да бъдат манипулирани независимо от детайлите на тяхната репрезентация. Главните collection интерфейси са основата на Java Collections Framework. Както може да видите в долупосоченото изображение, основните collection интерфейси формират йерархия.

Set е специален вид Collection, SortedSet е специален вид Set и т.н. Обърнете внимание, че йерархията се състои от две отделни дървета - Map не е истинска Collection.

Забележете, че всички основни collection интерфейси са generic. Например, това е декларацията на Collection интерфейса:

```
public interface Collection<E>…
```



Синтаксисът <E> издава, че интерфейсът е generic. Когато декларирате Collection инстанция, може и трябва да специфицирате типовете на съдържащите се в нея елементи. Специфицирането на типа позволява на компилатора да верифицира (по време на компилация) че типът обект, който слагате в колекцията, е коректен - следователно редуцирате грешките по време на изпълнение.

Когато разберете как да ползвате тези интерфейси ще знаете повечето от необходимото относно Java Collections Framework. Текущата секция дискутира общи правила за ефективна употреба на интерфейсите, включая кога кой интерфейс да бъде използван.

За да запази количеството на основните collection интерфейси в рамките на разумното, Java платформата не предоставя отделни интерфейси за всеки вариант на всеки тип колекция. (Такива

варианти може да включват immutable, fixed-size и appent-only.) Вместо това, операциите по модификация във всеки интерфейс са определени като опционални - като дадена имплементация може да избере да не поддържа всички операции. Ако неподдържана операция бъде извикана, колекцията хвърля UnsupportedOperationException. Имплементациите са отговорни за документиране на това кои точно операции те самите поддържат. Всички имплементации от Java платформата с общо предназначение поддържат всички опционални операции.

Следният списък описва основните collection интерфейси:

- **Collection** - първичният (root) елемент на collection йерархията. Колекция репрезентира група от обекти, познати като нейни елементи. Collection интерфейса е най-малкия общ деноминатор, който всички колекции имплементират и се ползва за предаване на колекции и за манипулацията им когато се търси максимално генерализиране. Някои типове колекции позволяват дупликиращи елементи, други не. Някои са подредени, други са неподредени. Java платформата не предоставя директни имплементации на този интерфейс, но предоставя имплементации на по-специфични подинтерфейси, например на Set и List.
- **Set** - колекция която не може да съдържа дупликиращи се елементи. Този интерфейс моделира математическата set абстракция и се ползва за репрезентацията на sets, като карти за игра образуващи ръка, курсовете в графика на студент, или процесите вървящи върху дадена машина.
- **List** - подредена колекция (понякога наричана *sequence*). Lists могат да съдържат дупликирани елементи. Употребяващият List основно притежава прецизен контрол върху това къде в списъка влиза всеки елемент, като може да достъпва елементите по техния числов индекс (index, position). Ако сте използвали Vector, значи сте запознати с основната функционалност на List.
- **Queue** – колекция, ползвана за държане на елементи преди тяхната обработка. Освен основните операции на Collections, Queue предоставя допълнителни операции по въвеждане, екстракция и инспектиране. Обикновено, но незадължително, подрежда елементите по FIFO (first-in, first-out) маниер. Измежду изключенията попадат приоритетните опашки, които подреждат елементите спрямо предоставения comparator или естествената подредба на елементите. Каквато и подредба да бъде използвана, главата на опашката е елементът който ще бъде изтрит чрез извикване на remove или полл. В дадена FIFO опашка всички нови елементи се въвеждат откъм края на опашката. Други видове опашки могат да ползват различни правила за позициониране. Всяка Queue имплементация трябва да специфицира своите характеристики на подредба.
- **Deque** - колекция използвана за държане на множество елементи непосредствено преди тяхната обработка. Освен основните Collection операции, Deque предоставя допълнителни операции по въвеждане, екстракция и инспектиране. Deques могат да бъдат ползвани както като FIFO (first-in, first-out), така и като LIFO (last-in, first-out). В deque всички нови елементи могат да бъдат въвеждани, извличани и премахвани и в двата края.
- **Map** - обект, който свързва ключове със стойности. Map не може да съдържа дублиращи се ключове; всеки ключ може да сочи максимум към една стойност. Ако сте ползвали Hashtable, значи вече сте запознати с основите на Map.

Последните два основни collection интерфейса са просто сортирани версии на Set и Map:

- **SortedSet** - представлява Set, който поддържа своите елементи във възходящ ред. Предоставени са няколко допълнителни операции, възползващи се от подредбата. Сортираните sets се ползват за натурално подредени sets, например списъци от думи.

- **SortedMap** - Map който държи своите елементи във възходящ ред. Представлява Map аналога на SortedSet. Сортираните maps се ползват за натурално подредените колекции от двойки ключ/стойност, например речници и телефонни указатели.

https://docs.oracle.com/javase/tutorial/collections/index.html

Guava Collections

Проектът Guava съдържа няколко core библиотеки на Google, една от които разширява collections. Представлява допълнение към collections екосистемата на JDK, явявайки се една от най-узрелите и популярни части на Guava.

- Immutable collections, за дефанзивно програмиране, константни колекции и подобрена ефективност.
- Нови типове колекции, за употреба в случаи, които JDK колекциите не адресират достатъчно добре: multisets, multimaps, tables, bidirectional maps и т.н.
- Можен инструментариум по отношение на колекции, за често извършвани операции, непредоставени от java.util.Collections.
- Допълнителни инструменти, за писане на Collection декоратор, имплементиране на Iterator и т.н.

https://code.google.com/p/guava-libraries/wiki/GuavaExplained

Изключения

Програмният език Java използва *exceptions* за справяне с грешки и други изключителни събития.

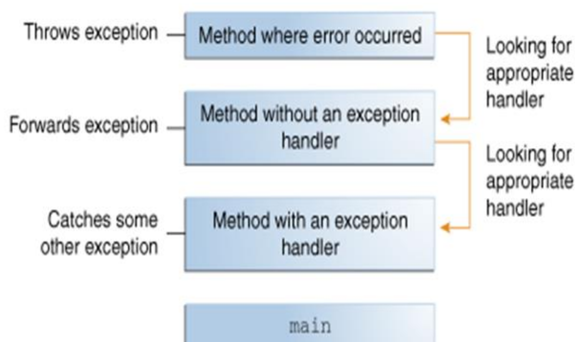
Какво е изключение?

Терминът *exception* е кратко наименование на фразата “exceptional event”.

Дефиниция: Изключение представлява събитие, възникнало по време на изпълнението на програма с нарушен поток на инструкциите.

Когато в рамките на даден метод възникне грешка, той създава обект и го подава на runtime системата. Този обект се нарича exception object и съдържа информация за грешката, включая нейния тип и състоянието на програмата към момента на възникване на грешката. Създаването на обект за изключение и подаването му на runtime системата се нарича хвърляне на изключение (throwing an exception).

След като даден метод хвърли изключение, runtime системата прави опити да намери нещо, което да го обработи. Наборът от възможни “неща” за обработка на изключение е подреденият списък от методи, извикани да отработят метода, където е възникнала грешката. Този списък от методи е познат като стек на извикване (call stack).



Runtime системата претърсва стека на извикване за метод, съдържащ блок от код, който може да отработи изключението. Този блок от код се нарича *exception handler*. Търсенето започва с метода, в който е възникнала грешката, и продължава през методите от стека на извикване в обратен на извикването им ред. Когато бъде намерен подходящ handler, runtime системата подава изключението на него. Даден exception handler се счита за подходящ ако типа на хвърления exception обект съвпада с типа, който може да бъде отработен от този handler.

Избраният exception handler прихваща изключението (catch the exception). Ако runtime системата изтощително претърсва всички методи в стека на извикване без да намери подходящ exception handler, както е показано в голното изображение, runtime системата (и съответно програмата) терминира.

Употребата на изключения за управление на грешки има някои преимущества пред традиционните техники за error-management.

Изискване за хващане или специфициране

Валидният Java програмен код трябва да удовлетвори изискването за хващане или специфициране (catch or specify requirement). Това означава, че кодът който може да хвърля изключение трябва да бъде ограден от някое от следните:

- try statement, който прихваща изключението. Try трябва да предостави handler за изключението;
- метод, специфициращ възможността за хвърлянето на изключение. Методът трябва да предостави throws клауза, която изброява възможните изключения.

Код, който не удовлетворява това изискване, няма да може да бъде компилиран.

Не всички изключения са предмет на изискването за хващане или специфициране. За да разберем защо, трябва да разгледаме трите основни категории изключения, само една от които е обект на изискването.

Трите вида изключения

Първият тип изключение се нарича checked exception. Съществуват изключителни условия, които едно добре написано приложение трябва да очаква и при които трябва да има възможност да се възстанови. Например, да предположим, че дадено приложение изиска от потребителя да въведе име на входен файл, след което го отваря чрез подаване името му на конструктора на java.io.FileReader. В нормални обстоятелства потребителят въвежда име на съществуващ достъпен файл, така че конструирането на FileReader обекта бива успешно, и изпълнението на програмата продължава както нормално. Понякога обаче потребителят предоставя име на несъществуващ файл, следователно конструкторът хвърля изключение java.io.FileNotFoundException. Една добре написана програма ще хване това изключение и ще нотифицира потребителя за грешката, с възможност за въвеждане на поправеното име.

Checked изключения са обект на изискването за хващане или специфициране. Всички изключения са checked exceptions, освен тези индикирани от Error, RuntimeException или техните подкласове.

Вторият тип изключение се нарича грешка (error). Съществуват особени условия, външни за приложението, които обикновено то не очаква и от които не може да се възстанови. Например, да предположим, че дадено приложение успешно отваря файл за вход, но не е способно да прочете файла поради хардуерен проблем или системен проблем. Неуспешното четене ще хвърли java.io.IOError. Приложението може да избере да хване това изключение, за да уведоми потребителя - но може да предпочете да принтира stack trace и да прекъсне изпълнение (exit).

Грешките не са обект на изискването за хващане или специфициране. Грешките са тези изключения, индикирани от Error и неговите подкласове.

Третият вид изключение е изключение по време на изпълнение (runtime exception). Съществуват изключителни условия, вътрешни за приложението, които обикновено то не очаква и от които не може да се възстанови. Обикновено те индикират програмни бъгове, като например логически грешки или неправилна употреба на дадено API. Нека допуснем например, че приложението описано по-горе подава името на файла на конструктора на FileReader. Ако поради логическа грешка стойност null бива подадена на конструктора, то той би хвърлил NullPointerException. Приложението може да хване това изключение, но вероятно е по-смислено бъгът, отговорен за възникването на това изключение, да бъде елиминиран.

Изключенията по време на изпълнение не са предмет на изискването за хващане или специфициране. Runtime изключенията са тези индикирани от RuntimeException и неговите подкласове.

Грешките и изключенията по време на изпълнение са колективно познати като unckecked exceptions.

Прихващане и обработка на изключения

Секцията описва употребата на трите компонента на exception handler - try, catch и finally блоковете - при написването на exception handler. В Java SE 7 е въведен try-with-resources, предназначен за ситуации ползващи Closable ресурси, като например streams.

Следният пример дефинира и имплементира клас ListOfNumbers. При конструирането си той създава ArrayList, съдържащ 10 Integer елемента с поредни стойности 0-9. ListOfNumbers класа дефинира също и метод writeList, който записва списъка от номера в текстов файл, наименован OutFile.txt. Примерът използва output класове дефинирани в java.io пакета.

```
// Note: This class will not compile yet.
import java.io.*;
import java.util.List;
import java.util.ArrayList;

public class ListOfNumbers {

    private List<Integer> list;
    private static final int SIZE = 10;

    public ListOfNumbers () {
        list = new ArrayList<Integer>(SIZE);
        for (int i = 0; i < SIZE; i++) {
            list.add(new Integer(i));
        }
    }

    public void writeList() {
            // The FileWriter constructor throws IOException, which must be caught.
        PrintWriter out = new PrintWriter(new FileWriter("OutFile.txt"));
        for (int i = 0; i < SIZE; i++) {
            // The get(int) method throws IndexOutOfBoundsException, which must be caught.
```

```
            out.println("Value at: " + i + " = " + list.get(i));
        }
        out.close();
    }
}
```

Първият ред с удебелен шрифт вика конструктор. Конструкторът инициализира output stream във файл. Ако този файл не може да бъде отворен, конструкторът хвърля IOException. Вторият ред с удебелен шрифт представлява извикване на get метода от клас ArrayList, който хвърля IndexOutOfBoundsException, ако стойността на аргумента е прекалено малка (по-малка от 0) или прекалено голяма (повече от броя текущо налични елементи, съдържащи се в ArrayList).

Ако се опитате да компилирате ListOfNumbers класа, компилаторът ще изведе съобщение за грешка относно хвърленото изключение от FileWriter конструктора. Обаче не извежда съобщение за грешка относно изключението, хвърлено от get. Причината е, че изключението хвърлено от конструктора, IOException, представлява checked exception, а това хвърляно от get метода, IndexOutOfBoundsException, представлява unckecked exception.

https://docs.oracle.com/javase/tutorial/essential/exceptions/index.htm

Reflection

Употреба на **Reflection**

Reflection се използва обикновено от програми, изискващи възможността да изследват или модифицират в движение поведението на приложения в рамките на Java виртуалната машина. Това е относително напредничава функция и трябва да бъде използвана само от разработчици, притежаващи стабилни познания относно фундаментите на езика. Reflection е мощна техника и може да позволи на приложенията да изпълняват операции, които привидно не са възможни.

Reflection дава възможност за инспектиране на класове, интерфейси, полета и методи по време на изпълнение - без да знаем конкретните им имена по време на компилацията. Възможно е още да бъдат инициирани нови обекти, да бъдат извиквани методи и да се четат/пишат стойности на полета посредством reflection.

Пример:

Следният код представлява демонстрация употребата на reflection

```
Method[] methods = MyObject.class.getMethods();


for(Method method : methods){
    System.out.println("method = " + method.getName());
}
```

В примерът по-горе се взема Class обектът от класа на име MyObject. Ползвайки клас обектът, примерът взема списък от методите на този клас, итерира ги и принтира техните имена.

http://docs.oracle.com/javase/tutorial/reflect/index.html

http://tutorials.jenkov.com/java-reflection/index.html

Concurrency

Още в основите на Java платформата е заложено многонишковото програмиране, включая езика и Java клас библиоетеките. От версия 5.0 насам Java платформата включва concurrency APIs на високо ниво.

Процеси

Даден процес има самостоятелна среда на изпълнение. Един процес обикновено има пълен частен набор от базови ресурси по време на изпълнение; в частност всеки процес има собствена памет.

Процесите често биват разглеждани като синоними на програми или приложения. Обаче това, което потребителят вижда като единично приложение може всъщност да бъде набор от коопериращи процеси. За улеснение комуникацията между процесите, повечето операционни системи поддържат Inter Process Communication (IPC) ресурси, като например pipes и sockets. IPC се ползва не само за комуникация между процеси в рамките на една система, но и между процеси върху различни такива.

Повечето имплементации на Java виртуална машина вървят като единичен процес. Дадено Java приложение може да създаде допълнителни процеси ползвайки ProcessBuilder обект.

Нишки

Нишките са понякога наричани олекотени процеси (lightweight processes). Както процесите, така и нишките предоставят среда на изпълнение, но създаването на нова нишка изисква по-малко ресурси от създаването на нов процес.

Нишките съществуват в рамките на един процес - всеки процес има поне една. Нишките споделят процесните ресурси, включая памет и отворени файлове. Това ги прави по-ефективни, но потенциално проблематични при комуникация.

Изпълнението на множество нишки е основополагаща функционалност на Java платформата. Всяко приложение има поне една нишка - или няколко, ако броим “системните” нишки, които се занимават с управление на паметта и сигналите. Но от гледна точка на разработващия приложението, стартирате само една нишка, която се нарича основна (main thread). Тази нишка има способността да създава допълнителни такива.

Нишкови обекти

Всяка нишка бива асоциирана с инстанция на класа Thread. Има две основни стратегии за употреба на Thread обекти, за целите на създаване на concurrent приложение:

- За директен контрол върху създаването и управлението на нишки, инстанциирайте Thread всеки път, когато приложениет се нуждае от иницииране на асинхронна задача;

- За абстрактно управление на нишки от останалата част на приложенеието, подайте задачата на приложението на executor.

Дефиниране на начална нишка

Приложение, създаващо инстанция на Thread, трябва да предоставя кода който ще бъде изпълнен в тази нишка. Има два начина това да бъде направено:

- Предоставяне на Runnable обект. Runnable интерфейса дефинира единичен метод run, предназначен да съдържа кода, изпълняван от нишката. Runnable обекта бива подаден на Thread конструктора, както в следния HelloRunnable пример:

```
public class HelloRunnable implements Runnable {

    public void run() {
        System.out.println("Hello from a thread!");
    }

    public static void main(String args[]) {
        (new Thread(new HelloRunnable())).start();
    }

}
```

- Подклас на Thread. Thread класа сам по себе си имплементира Runnable, въпреки че неговия run метод не прави нищо. Дадено приложение може да имплементира подклас на Thread, предоставяйки своя собствена имплементация на run, както е в следния HelloThread пример:

```
public class HelloThread extends Thread {

    public void run() {
        System.out.println("Hello from a thread!");
    }

    public static void main(String args[]) {
        (new HelloThread()).start();
    }

}
```

Обърнете внимание, че и двата примера извикват Thread.start за да поставят начало изпълнението на новата нишка.

Кой от горните подходи трябва да ползвате? Първият, употребяващ Runnable обект, е по-общ защото Runnable обекта може да бъде подклас на друг освен Thread. Вторият подход е по-лесен за употреба в прости приложения, но е ограничен от факта, че вашият клас трябва да бъде наследник на Thread.

Класът Thread дефинира множество методи, полезни за управлението на нишки. Те включват статични методи, предоставящи информация за нишката на метода, или афектиращи статуса й. Другите методи се извикват от други нишки, въвлечени в управлението на текущата такава и Thread обекта.

Паузиране изпълнението чрез sleep

Thread.sleep кара текущата нишка да преустанови изпълнението си за определен период от време. Това е ефективен подход за освобождаване на процесорна мощ за другите нишки на приложението, или други приложения, които могат да вървят върху системата. sleep метода може да бъде ползван още за въвеждане на темпо/пулсация (pacing), демонстрирано в примера по-долу - изчаквайки за друга нишка натоварена със задачи с изисквания откъм времето.

Предоставени са две разширени (overloaded) версии на sleep метода: една специфицираща времето за пауза до милисекунди, и друга специфицираща го в наносекунди. Няма гаранция обаче, че тези

времена ще бъдат прецизни, защото те са ограничени от инструментите, предоставени от наличната операционна система. Също периодът на пауза може да бъде прекъснат чрез interrupts. Във всеки случай, не трябва да разчитате, че извикан sleep ще паузира нишката за прецизен времеви период.

Следният SleepMessages пример ползва sleep за да принтира съобщения на 4-секундни интервали:

```
public class SleepMessages {
   public static void main(String args[])
      throws InterruptedException {
      String importantInfo[] = {
         "Mares eat oats",
         "Does eat oats",
         "Little lambs eat ivy",
         "A kid will eat ivy too"
      };

      for (int i = 0;
          i < importantInfo.length;
          i++) {
         //Pause for 4 seconds
         Thread.sleep(4000);
         //Print a message
         System.out.println(importantInfo[i]);
      }
   }
}
```

Обърнете внимание, че main декларира хвърляне на InterruptedException. Изключението бива хвърляно от sleep метода когато друга нишка прекъсне текущата по време на активен sleep. След като приложението не е дефинирало друга нишка за прекъсването, в примера няма нужда да се хваща InterruptedException изключението.

Прекъсвания

Прекъсване (interrupt) е индикация, че нишката трябва да прекрати това, което прави - и да започне да прави нещо друго. Програмистът може да реши точно как дадена нишка реагира на прекъсване, но честа практика е нишката да терминира.

Нишката изпраща прекъсване чрез извикване на interrupt метода на Thread обекта за нишката, която искаме да прекъснем. За да може механизма на прекъсване да сработи коректно, прекъснатата нишка трябва да поддържа собствено прекъсване.

ПОДДРЪЖКА НА ПРЕКЪСВАНЕ

Как дадена нишка поддържа собственото си прекъсване? В зависимост от това какво точно прави в текущия момент. Ако нишката често извиква други методи, които хвърлят InterruptedException, то просто излиза от run метода след като хване изключението. Например нека предположим, че централния цикъл на съобщения в SleepMessages примера е в run метода на нишковия Runnable обект. Тогава той може да бъде модифициран за да поддържа прекъсвания по този начин:

```
for (int i = 0; i < importantInfo.length; i++) {
```

```
    // Pause for 4 seconds
    try {
        Thread.sleep(4000);
    } catch (InterruptedException e) {
        // We've been interrupted: no more messages.
        return;
    }
    // Print a message
    System.out.println(importantInfo[i]);
}
```

Много методи, които хвърлят InterruptedException, като например sleep, са разработени да прекъснат текущата си операция и излязат мигновенно когато получат прекъсване.

Какво се случва ако нишката върви дълго време без да извика метод, хвърлящ InterruptedException? Тогава трябва периодично да извиква Thread.interrupted, което връща true ако е получено прекъсване.

Пример:

```
for (int i = 0; i < inputs.length; i++) {
    heavyCrunch(inputs[i]);
    if (Thread.interrupted()) {
        // We've been interrupted: no more crunching.
        return;
    }
}
```

В този пример, кодът просто тества за прекъсване и нишката излиза, ако такова е било получено. В по-комплексни приложения вероятно би било по-логично да се хвърля InterruptedException:

```
if (Thread.interrupted()) {
    throw new InterruptedException();
}
```

Това позволява на кодът, отговорен за прекъсването, да бъде централизиран в catch клауза.

СТАТУС ФЛАГ НА ПРЕКЪСВАНЕ

Механизмът за прекъсване е имплементиран посредством вътрешен флаг, познат като статус на прекъсване (interrupt status). Извикването на Thread.interrupt настройва този флаг. Когато дадена нишка провери за прекъсване чрез извикване на статичния метод Thread.interrupted, статусът на прекъсване бива изчистен. Нестатичният isInterrupted метод, който се ползва от една нишка за да запита за статуса на прекъсване друга такава, не променя флага за статуса на прекъсване.

По конвенция, всеки моетод който излиза чрез хвърляне на InterruptedException изчиства статуса на прекъсване при хвърлянето. Въпреки това, възможно е статусът на прекъсване да бъде настроен непосредствено след това, от друга нишка извикала interrupt.

Joins

Метода join позволява дадена нишка да изчака завършването на друга такава. Ако t е Thread обект, чиято нишка в момента бива изпълнявана,

```
t.join();
```

кара текущата нишка да паузира изпълнението си докато нишката на t терминира. Разширяването на join метода позволява на програмиста да специфицира период на изчакване. Както при sleep метода, join е зависим от операционната система по отношение на времето, затова не трябва да разчитате, че join ще чака точно толкова колкото сте указали.

Както sleep, join реагира на прекъсване чрез излизане с InterruptedException.

Пример за проста нишка

Следният пример демонстрира някои засегнати по-горе концепции. SimpleThreads се състои от две нишки. Първата е основната нишка (main), която всяко Java приложение притежава. main нишката създава нова нишка от Runnable обекта, MessageLoop, и изчаква цикълът да приключи. Ако изпълнението на MessageLoop нишката отнеме прекалено дълго време, основната такава я прекъсва.

Нишката MessageLoop принтира серия от съобщения. Ако бъде прекъсната преди да принтира всички свои съобщения, MessageLoop нишката принтира съобщение за това и излиза.

```
public class SimpleThreads {

    // Display a message, preceded by
    // the name of the current thread
    static void threadMessage(String message) {
        String threadName =
            Thread.currentThread().getName();
        System.out.format("%s: %s%n",
                          threadName,
                          message);
    }

    private static class MessageLoop
        implements Runnable {
        public void run() {
            String importantInfo[] = {
                "Mares eat oats",
                "Does eat oats",
                "Little lambs eat ivy",
                "A kid will eat ivy too"
            };
            try {
                for (int i = 0;
                     i < importantInfo.length;
                     i++) {
                    // Pause for 4 seconds
                    Thread.sleep(4000);
                    // Print a message
                    threadMessage(importantInfo[i]);
                }
            } catch (InterruptedException e) {
```

```
            threadMessage("I wasn't done!");
        }
    }
}

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

    // Delay, in milliseconds before
    // we interrupt MessageLoop
    // thread (default one hour).
    long patience = 1000 * 60 * 60;

    // If command line argument
    // present, gives patience
    // in seconds.
    if (args.length > 0) {
        try {
            patience = Long.parseLong(args[0]) * 1000;
        } catch (NumberFormatException e) {
            System.err.println("Argument must be an integer.");
            System.exit(1);
        }
    }

    threadMessage("Starting MessageLoop thread");
    long startTime = System.currentTimeMillis();
    Thread t = new Thread(new MessageLoop());
    t.start();

    threadMessage("Waiting for MessageLoop thread to finish");
    // loop until MessageLoop
    // thread exits
    while (t.isAlive()) {
        threadMessage("Still waiting...");
        // Wait maximum of 1 second
        // for MessageLoop thread
        // to finish.
        t.join(1000);
        if (((System.currentTimeMillis() - startTime) > patience)
              && t.isAlive()) {
            threadMessage("Tired of waiting!");
            t.interrupt();
            // Shouldn't be long now
```

```
            // -- wait indefinitely
            t.join();
          }
        }
        threadMessage("Finally!");
    }
}
```

Синхронизация

Нишките комуникират основно чрез споделяне достъп до полета и обекти. Тази форма на комуникация е изключително ефективна, но прави два вида грешки възможни: смущения на нишки и грешки при консистентността на паметта. Инструментът, необходим за предотвратяването на тези грешки, се нарича синхронизация.

Синхронизацията от своя страна може да доведе до съревнование между нишки, което възниква когато две или повече нишки опитат да достъпят същия ресурс едновременно и карат Java runtime да изпълни една или повече нишки забавено, или дори да преустанови тяхното изпълнение. Starvation и livelock представляват форми на съперничество между нишки.

ThreadLocal класа позволява създаването на променливи, които могат да бъдат четени и писани само от текущата нишка. По този начин, дори две нишки да изпълняват същия код, и този код реферира ThreadLocal променлива, тогава двете нишки не могат да виждат ThreadLocal променливите на другата такава.

```
private ThreadLocal myThreadLocal = new ThreadLocal();
```

В горният пример бива инстанцииран нов ThreadLocal обект. Необходимо е това да бъде направено само веднъж, като няма значение от коя нишка се извършва. Всички нишки ще виждат същата ThreadLocal инстанция, но стойностите на ThreadLocal, присвоени ползвайки set метода, ще бъдат видими само за нишката, която е записала стойността. Дори две различни нишки да запишат две различни стойности на един и същи ThreadLocal обект, те не могат да видят стойността на другата нишка.

```
myThreadLocal.set("A thread local value”);
```

Четенето на стойността, запазена в ThreadLocal, се прави по следния начин:

```
String threadLocalValue = (String) myThreadLocal.get();
```

Пакетът java.util.concurrent съдържа помощни класове, ползвани често в concurrent програмирането.

http://docs.oracle.com/javase/tutorial/essential/concurrency/index.html

http://tutorials.jenkov.com/java-concurrency/threadlocal.html

http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/package-summary.html

Java API за **XML** обработка **(JAXP)**

Java API за XML обработка (JAXP) е предназначено за обработване на XML данни, ползвайки приложения написани на програмния език Java. JAXP използва parser стандартите Simple API for XML Parsing (SAX) и Document Object Model (DOM), така че можете да изберете да прочетете вашите данни като stream от събития или да построите обектна репрезентация. JAXP също поддържа Excensible Stylesheet Language Transformations (XSLT) стандарта, давайки контрол върху презентацията на данните и позволяването на конвертиране данните в други XML документи или

различни формати като HTML. JAXP предоставя още поддръжка на namespaces, позволявайки работата с DTDs с иначе потенциални naming конфликти. Финално, от версия 1.4 насам, JAXP имплементира Streaming API for XML (StAX) стандарта.

С насочен към гъвкавостта дизайн, JAXP позволява употребата на XML-съвместим parser вътре във вашето приложение. Това се случва благодарение на т.нар. pluggability layer, който позволява plug in на имплементация на SAX или DOM API. Този pluggability layer позволява още включването на XSL процесор, давайки контрол върху визуализацията на XML данните.

http://docs.oracle.com/javase/tutorial/jaxp/index.html

JDBC(TM) достъп до бази данни

JDBC™ API притежава дизайн, съобразен с простата му употреба. Това означава, че JDBC улеснява максимално работата с бази данни. Може да достъпва всеки вид tabular данни, особено данни пазени в релационни бази данни.

JDBC помага писането на Java приложения, извършващи следните дейности:

- Свързане с data source, като база данни
- Пращане на заявки и update statements към базата
- Получаване и обработка на резултатите, получени от базата в отговор на заявка

http://docs.oracle.com/javase/tutorial/jdbc/index.html

Java EE

Java Enterprise Edition представлява стандартен community-driven enterprise софтуер. Разработена е посредством Java Community Process (JCP), като принос имат експерти от индустрията, комерсиални и open-source организации, Java потребителски групи и множество индивиди. Всяка версия интегрира нови функционалности, отговарящи на индустриалните нужди, като подобрява преносимостта и увеличава производителността на разработчиците.

Платформата предоставя API и runtime environment за разработка и употреба на enterprise софтуер, включая мрежови приложения, web services и други големи, многослойни, скалируеми, надеждни и сигурни мрежови приложения. Java EE разширява Java Standart Edition (Java SE) платформата, предоставяйки API за object-relational mapping (ORM), дистрибутирани и многослойни архитектури и web services. Платформата инкорпорира дизайн, базиран върху модулни компоненти, вървящи върху приложен сървър (application server). Софтуерът за Java EE е основно разработен на програмния език Java. Платформата залага на конвенция пред конфигурация и анотиране за конфигурация. Опционално XML може да бъде ползван за припокриване на анотации, или за отколение от стойностите по подразбиране за платформата.

История

Платформата бе позната като Java 2 Platform, Enterprise Edition или J2EE преди името да бъде сменено на Java Platform, Enterprise Edition или Java EE във версия 5. Текущата версия се нарича Java EE 7.

- J2EE 1.2 (12 декември 1999)
- J2EE 1.3 (24 септември 2001)
- J2EE 1.4 (11 ноември 2003)
- Java EE 5 (11 май 2006)
- Java EE 6 (10 декември 2009)
- Java EE 7 (28 май 2013, но 5 април 2013 според спецификацията. 12 юни 2013 бе планираната начална дата)
- Java EE 8 (очаква се 2016)

Стандарти и спецификации

Java EE е дефинирана от своята спецификация:

http://www.oracle.com/technetwork/java/javaee/tech/index.html

Както при други Java Community Process спецификации, доставчиците на софтуер трябва да посрещнат някои изисквания, за да могат да обявят продуктите си като Java EE съвместими такива.

Java EE включва няколко API спецификации, като RMI, e-mail, JMS, web services, XML и др., и дефинира тяхната координация. Java EE също предоставя някои компоненти спецификации, включая Enterprise JavaBeans, connectors, servlets, JavaServer Pages и няколко web service технологии. Това позволява на разработчиците да създават enterprise приложения, които са преносими и скалируеми, и се интегрират с по-старите технологии. Даден Java EE приложен сървър (application server) може да поеме транзакции, сигурност, скалируемост, concurrency и управление на разгърнатите компоненти, помагайки на разработчиците да се концентрират повече върху бизнес логиката на компонентите, вместо върху инфраструктурата и задачите по интеграцията.

Основни APIs

Java EE APIs включват няколко технологии, разширяващи функционалността на базовите Java SE APIs.

- Java EE 7 Platform Packages
- Java EE 6 Platform Packages
- Java EE 5 Platform Packages

java.servlet.*

Servlet спецификацията дефинира набор от APIs за обслужване основно на HTTP заявки. Включва и JavaServer Pages (JSP) спецификацията.

JSR 369 - Java Servlet 4.0, https://jcp.org/en/jsr/detail?id=369

javax.websocket.*

Java API за WebSocket спецификацията дефинира набор от APIs за обслужване на WebSocket connections.

JSR 356 - Java API for WebSocket, https://jcp.org/en/jsr/detail?id=356

javax.faces.*

Пакетът дефинира корена на JavaServer Faces (JSF) API. JSF представлява технология за конструиране на потребителски интерфейси чрез компоненти.

JSR 372 - JavaServer Faces (JSF 2.3) Specification, https://jcp.org/en/jsr/detail?id=372

javax.faces.component.*

Пакетът дефинира компонентната част на JavaServer Faces API. Тъй като JSF е основно ориентирана към компоненти, този е един от основните пакети.

javax.el.*

Пакетът дефинира класове и интерфейси за Java EE Expression Language. Този Expression Language (EL) представлява прост език, оригинално разработен за да удовлетворява специфични нужди на разработчици на уеб приложения. Използва се основно в JSF за binding (връзка) между компонентите и (backing) beans, както и в CDI за да наименова beans, но може да бъде ползван в цялата платформа.

JSR 341 - Expression Language 3.0, https://jcp.org/en/jsr/detail?id=341

javax.enterprise.inject.*

Пакетите дефинират injection анотациите за Contexts and Dependency Injection (CDI) APIs.

JSR 346 - Contexts and Dependency Injection for Java EE 1.1, https://jcp.org/en/jsr/detail?id=346

javax.enterprise.context.*

Тези пакети дефинират контекст анотациите и интерфейсите за CDI API.

javax.ejb.*

Enterprise JavaBean (EJB) спецификацията дефинира набор от олекотени APIs, които даден обектен контейнер (EJB container) ще поддържа за да предостави транзакции (ползвайки JTA), remote procedure calls (посредством RMI или RMI-IIOP), concurrency control, dependency injection и access control за бизнес обекти. Този пакет съдържа Enterprise JavaBeans класове и интерфейси, дефиниращи договори между enterprise bean и неговите клиенти, и между enterprise bean и ejb container.

JSR 220 - Enterprise JavaBeans 3.0, https://jcp.org/en/jsr/detail?id=220

javax.validation.*

Този пакет съдържа анотации и интерфейси за поддръжка на декларативна валидация, предоставена от Bean Validation API. Bean Validation предоставя унифициран начин за constraints на beans (например, JPA model classes), които могат да бъдат наложени cross-layer. В Java EE, JPA се съобразява с bean validation constraints в наличния persistence layer, докато JSF се съобразява на ниво view layer.

JSR 349 - Bean Validation 1.1, https://jcp.org/en/jsr/detail?id=349

javax.persistence.*

Пакетът съдържа договори между persistence provider и managed класове и клиентите на Java Persistence API (JPA).

JSR 338 - Java Persistence 2.1, https://jcp.org/en/jsr/detail?id=338

javax.transaction.*

Пакетът предоставя Java Transaction API (JTA), съдържащо интерфейси и анотации за интеракция с transaction поддръжката, предложена от Java EE. Въпреки че това API се абстрахира от същинските детайли на ниско ниво, интерфейсите се считат за такива на ниско ниво, следователно разработчиците могат да разчитат на прозрачна употреба на транзакции от по-високо ниво на EJB абстракции, или ползвайки анотации предоставени от това API в комбинация със CDI managed beans.

JSR 907 - Java Transaction API (JTA), https://jcp.org/en/jsr/detail?id=907

javax.security.auth.message.*

Този пакет предоставя основите на Java Authentication SPI (JASPIC), съдържащ интерфейси и класове за изграждането на автентикационни модули за сигурни Java EE приложения. Автентикационните модули са отговорни за интеракцията с потребителя (например, пренасочването към дадена форма или към OpenID provider), верифицирайки потребителския вход (например чрез LDAP lookup, заявка към базата или запитване на OpenID provider с token), получавайки набор от групи/роли, в които автентикираният потребител се намира/притежава.

JSR 196 - Java Authentication Service Provider Interface for Containers, https://jcp.org/en/jsr/detail?id=196

javax.enterprise.concurrent.*

Този пакет предоставя интерфейси за интеракция директно с наличния по подразбиране managed thread pool в Java EE платформата. Опционално може да бъде ползван executor service, работещ на по-високо ниво върху същия thread pool. Същите интерфейси могат да бъдат ползвани за дефинирани от потребителя managed thread pools, но това се базира върху специфична за вендора (vendor-specific) конфигурация и не се покрива от Java EE спецификацията.

JSR 236 - Concurrency Utilities for Java EE, https://jcp.org/en/jsr/detail?id=236

javax.jms.*

Този пакет дефинира Java Message Service (JMS) API. Това API предоставя сходен начин за създаване, изпращане, получаване и четене на съобщения от enterprise messaging системи.

JSR 914 - Java Message Service (JMS) API, https://jcp.org/en/jsr/detail?id=914

javax.batch.api.*

Пакетът дефинира входния AP за Java EE Batch Applications. Batch Applications API предоставя възможности за изпълняване на дългосрочно изпълняващи се задачи в background, които потенциално включват големи количества дани и които може да имат нужда да бъдат периодично изпълнявани.

JSR 352 - Batch Applications for the Java Platform, https://jcp.org/en/jsr/detail?id=352

javax.resource.*

Този пакет дефинира Java EE Connector Architecture (JCA) API. То представлява Java-базирано решение за връзка с приложни сървъри и enterprise информационни системи (EIS) като част от enterprise application integration (EAI) решения. Представлява API на ниско ниво, насочено към vendors, с които даден application developer обикновено няма досег.

JSR 322 - Java EE Connector Architecture 1.6, https://jcp.org/en/jsr/detail?id=322

Полезни връзки

Днес Java EE предлага богата enterprise софтуерна платформа с над 20 съвместими Java EE 6 имплементации за избор, предоставящи нисък риск и множество възможности.

- JSR 366 - Java EE 8
- JSR 367 - The Java API for JSON Binding
- JSR 368 - Java Message Service 2.1
- JSR 369 - Java Servlet 4.0
- JSR 370 - Java API for RESTful Web Services 2.1
- JSR 371 - Model-View-Controller 1.0
- JSR 372 - Java Server Faces 2.3
- JSR 373 - Java EE Management API 1.0
- JSR 374 - Java API for JSON Processing 1.1
- JSR 375 - Java EE Security API 1.0

http://www.oracle.com/technetwork/java/javaee/overview/index.html

http://en.wikipedia.org/wiki/Java_Platform,_Enterprise_Edition

JavaServer Pages (JSP)

Java Web Applications

Enterprise JavaBeans (EJB)

JavaBeans Validation (Bean Validation)

Java Persistence API (JPA)

Context Dependency Injection (CDI)

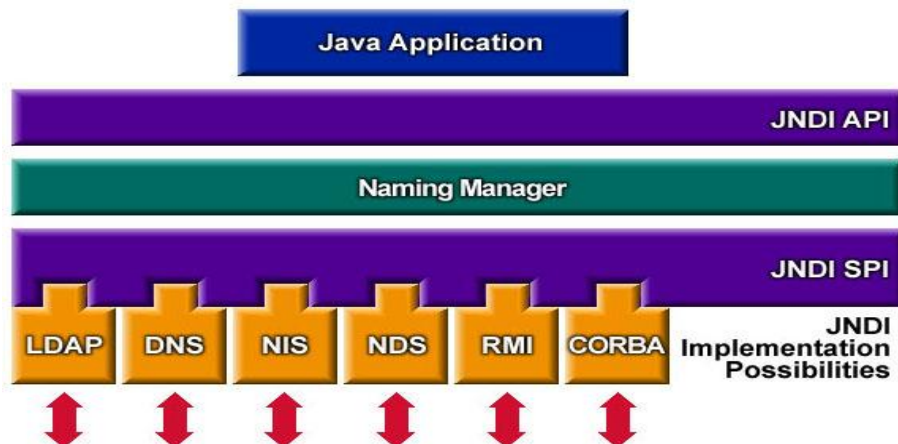
Web Services (SOAP & REST)

JNDI

Java Naming and Directory Interface (JNDI) представлява API, предоставящо функционалности по отношение на наименоване и директорийност на приложения, ползващи езика Java. Създадено е да бъде независимо от конкретна имплементация на директорийна услуга. По този начин множество директории - нови, възникващи и вече разгърнати - могат да бъдат достъпвани по сходен начин.

Архитектура

JNDI архитектурата се състои от API и service provider interface (SPI). Java приложенията ползват JNDI API за достъп до разнообразни naming и directory services. SPI позволява на тези услуги да бъдат включени по прозрачен начин, позволявайки на приложението да ползва JNDI API за достъп до услуги.



Пакетиране

JNDI е част от Java SE. За да ползвате JNDI трябва да имате JNDI класове и един или повече service providers. Java 2 SDK v1.3 включва три service providers за следните naming/directory services:

- Lightweight Directory Access Protocol (LDAP)
- Common Object Request Broker Architecture (CORBA) Common Object Services (COS) name service
- Java Remote Method Invocation (RMI) Registry

Могат да бъдат употребявани и други външни service providers.

Създаване на Initial Context

Пример:

```
Hashtable env = new Hashtable();
env.put(Context.INITIAL_CONTEXT_FACTORY,
    "com.sun.jndi.fscontext.RefFSContextFactory");
Context ctx = new InitialContext(env);
```

В main метода по-горе се създава initial context. Индикира се ползването на файловата система като service provider чрез настройката на environment properties параметъра (репрезентиран от Hashtable класа) към InitialContext конструктора.

Looking up на обект

Следният код демонстрира извършването на looking up за обект с определено име:

```
Object obj = ctx.lookup(name);
```

http://docs.oracle.com/javase/jndi/tutorial/trailmap.html

Java Servlet

Какво е сървлет?

Сървлет представлява клас, използван за надграждане възможностите на сървъри, сертиращи приложения, достъпвани посредством request-response модела. Въпреки че сървлетите могат да отговарят на всеки тип request, обикновено се ползват за разширяване на приложения върху уеб сървъри. За подобни приложения, Java Servlet технологията дефинира HTTP-специфични сървлет класове.

javax.servlet и javax.servlet.http пакетите предоставят интерфейси и класове за писане на сървлети. Всички сървлети трябва да имплементират Servlet интерфейса, който дефинира lifecycle методите. Когато се имплементира service с широко приложение, можете да ползвате или разширите GenericServlet класа, предоставен от наличното Java Servlet API. HttpServlet класа предоставя методи като например doGet и doPost, за обслужване на специфични услуги за HTTP протокола.

Servlet Lifecycle

Жизненият цикъл на сървлета се контролира от контейнера, в който сървлетът е разгърнат. Когато даден request пристигне към сървлета, контейнерът изпълнява следните стъпки:

1. Ако не съществува инстанция на сървлета, уеб контейнерът:
    1. Зарежда сървлет класа.
    2. Създава инстанция на сървлет клас.
    3. Инициализира сървлет инстанцията чрез извикване на init метода.
2. Извиква service метода, подавайки request и response обектите.

Ако е необходимо изтриване на сървлета, контейнерът финализира сървлета чрез извикване на неговия destroy метод.

Боравене със servlet lifecycle събития

Чрез дефиниране на listener обекти, чиито методи да бъдат извиквани при възникване на servlet lifecycle събитие, можете да извършвате мониторинг и да реагирате на събития от жизнения цикъл на сървлета. За да използвате тези listener обекти, трябва да дефинирате и специфицирате listener клас.

Дефиниране на listener клас

Listener клас се дефинира като имплементация на listener интерфейс. Долупосочената таблица представя събитията, които могат да бъдат обект на мониторинг и кореспондиращия интерфейс, който трябва да бъде имплементиран за целта. Когато бива извикан метод на listener, на него се подава събитие, съдържащо информация за него. Например, на методите в HttpSessionListener интерфейса се подава HttpSessionEvent, който съдържа HttpSession.

Обект	Събитие	Listener интерфейс и event клас
Web context	Инициализация и унищожаване	javax.servlet.ServletContextListener и ServletContextEvent
Web context	Добавяне, итзриване или заместване на атрибут	javax.servlet.ServletContextAttributeListener и ServletContextAttributeEvent

Обект	Събитие	Listener интерфейс и event клас
Session	Създаване, инвалидация, активация, пасивация и timeout	javax.servlet.http.HttpSessionListener, javax.servlet.http.HttpSessionActivationListener и HttpSessionEvent
Session	Добавяне, изтриване или заместване на атрибут	javax.servlet.http.HttpSessionAttributeListener и HttpSessionBindingEvent
Request	Започнала е обработката на даден сървлет request от web компоненти	javax.servlet.ServletRequestListener и ServletRequestEvent
Request	Добавяне, изтриване или заместване на атрибут	javax.servlet.ServletRequestAttributeListener и ServletRequestAttributeEvent

Използвайте @WebListener анотацията, за да дефинирате listener за прихващане на събития за различни операции върху определен web application context. Класовете анотирани с @WebListener трябва да имплементират един от следните интерфейси:

- javax.servlet.ServletContextListener
- javax.servlet.ServletContextAttributeListener
- javax.servlet.ServletRequestListener
- javax.servlet.ServletRequestAttributeListener
- javax.servlet..http.HttpSessionListener
- javax.servlet..http.HttpSessionAttributeListener

Например, следният примерен код дефинира listener, който имплементира два от горните интерфейса:

```
import javax.servlet.ServletContextAttributeListener;
import javax.servlet.ServletContextListener;
import javax.servlet.annotation.WebListener;

@WebListener()
public class SimpleServletListener implements ServletContextListener,
        ServletContextAttributeListener {
   ...
```

Работа със сървлет грешки

По време на работата на даден сървлет могат да възникнат произволно количество изключения. При възникването на такова, уеб контейнерът генерира страница по подразбиране, съдържаща следното съобщение:

A Servlet Exception Has Occurred

Можете да специфицирате, че контейнерът трябва да връща специфична страница за грешки, при възникването на дадено изключение.

Създаване и инициализация на сървлет

Използвайте @WebServlet анотацията за да дефинирате сървлет компонент в дадено уеб приложение. Тази анотация е специфицирана върху клас и съдържа метаданни относно декларацията на сървлета. Анотацията трябва да специфицира поне един URL pattern. Това се прави чрез употребата на urlPatterns или value атрубута на анотацията. Всички други атрибути са опционални, с настройки по подразбиране. Ползвайте value атрибута когато единственият атрибут на анотацията се явява посочения URL pattern; в противен случай ползвайте urlPatterns атрибута, когато ползвате и други атрибути.

Класовете анотирани с @WebServlet трябва да наследяват javax.servlet.http.HttpServlet класа. Например, следният примерен код дефинира сървлет с URL pattern /report:

```
import javax.servlet.annotation.WebServlet;

import javax.servlet.http.HttpServlet;


@WebServlet("/report")

public class MoodServlet extends HttpServlet {

   ...
```

Уеб контейнерът инициализира сървлет след зареждане и инстанцииране на сървлет класа и преди предаване на requests от клиента. За къстъмизация на процеса с цел позволяване на сървлета да чете persistent configuration данни, инициализира ресурси и извършва всякакви други еднократни дейности, можете или да припокриете init метода на Servlet интерфейса, или да специфицирате initParams атрибутите на @WebServlet анотацията. Тези атрибути съдържат @WebInitParam анотацията. Ако не може да завърши инициализиращия процес, даден сървлет хвърля UnavailableException.

Ползвайте инициализиращ параметър за да предоставите данни, необходими на определен сървлет. В контаст, контекстов параметър предоставя данни, достъпни за всички компоненти на уеб приложение.

Филтриране на requests и responses

Филтър представлява обект, който може да трансформира хедър и съдържание (или и двете) на даден request или resopnse. Филтрите се различават от уеб компонентите по това, че те обикновено сами по себе си не създават response. Вместо това, даден филтър предоставя функционалност, която бива “прикачвана” към даден вид уеб ресурс. В следствие на това, филтрите не трябва да имат зависимости от даден уеб ресурс, за който се явяват филтър; по този начин могат да бъдат композирани с повече от един тип уеб ресурс.

Основните задачи, извършвани от даден филтър, са следните:

- Заявка за request с цел последваща работа с него;
- Блокиране на request-и-response двойка от продължаване работата им нататък;
- Модифициране на request хедъри и данни. Това се прави чрез предоставяне къстъмизирана версия на наличния request;
- Модифициране на response хедъри и данни. Това се прави чрез предоставяне къстъмизирана версия на наличния response;

- Интерация с външни ресурси.

Приложението на филтрите включва автентикация, logging, конверсия на изображения, компресиране на данни, криптиране, XML трансформации и т.н.

Можете да конфигурирате даден уеб ресурс да бъде филтриран от верига (chain) от нула, един или повече филтри в специфична поредност. Тази верига е специфицирана когато уеб приложението съдържащо компонента бива разгърнато, и съответно е инстанциирана когато уеб контейнерът зареди компонента.

Програмни филтри

Наличното API за филтриране е дефинирано от Filter, FilterChain и FilterConfig интерфейсите в javax.servlet пакета. Можете да дефинирате филтър чрез имплементиране на Filter интерфейса.

Ползвайте @WebFilter анотацията, за да дефинирате филтър в уеб приложение. Тази анотация е специфицирана върху клас и съдържа метаданни относно декларацията на филтъра. Анотираният филтър трябва да специфицира поне един URL pattern. Това се прави чрез употребата на urlPatterns или value атрибута на анотацията. Всички други атрибути са опционални, с настройки по подразбиране. Ползвайте value атрибута когато единственият такъв в анотацията е URL pattern; ползвайте urlPatterns атрибута при наличието на допълнителни атрибути.

Класовете анотирани с @WebFilter анотацията трябва да имплементират javax.servlet.Filter интерфейса.

За да добавите конфигурация към филтъра, специфицирайте initParams атрибута на @WebFilter анотацията. initParams атрибъта съдържа @WebInitParam анотацията. Следният примерен код дефинира филтър, специфицирайки инициализиращия параметър:

```
import javax.servlet.Filter;
import javax.servlet.annotation.WebFilter;
import javax.servlet.annotation.WebInitParam;

@WebFilter(filterName = "TimeOfDayFilter",
urlPatterns = {"/*"},
initParams = {
   @WebInitParam(name = "mood", value = "awake")})
public class TimeOfDayFilter implements Filter {
   ....
```

Най-важният метод на Filter интерфейса е doFilter, на който биват подавани request, response и filter chain обектите. Този метод може да извършва следните действия:

- Изследване на request хедърите;
- Късътмизация на request обекта ако филтърът желае да модифицира request хедърите или данните;
- Късътмизация на response обекта ако филтърът желае да модифицира request хедърите или данните;
- Извикване на следващото entity във филтър веригата. Ако текущият филтър е последният филтър във веригата, приключваща с target уеб компонента или статичен ресурс, то следващото entity е ресурса в края на веригата; в противен случай се явява следващият филтър, конфигуриран в наличния WAR. Филтърът извиква следващото entity чрез извикване на doFilter метода върху

обекта от веригата, подавайки му наличните request и response с които е извикан, или с техните wrapped версии които потенциално е създал. Алтернативно, филтрите могат да изберат да блокират request чрез неизвикване на следващото entity. В по-късен етап, филтърът е отговорен за попълване на response;

- Изследване на response хедърите след извикване на следващия филтър във веригата;
- Хвърляне на изключение за индикиране на грешка по време на работа.

В допълнение на doFilter, трябва да имплементирате и init и destroy методите. init методът се извиква от контейнера когато филтърът бива инстанцииран. Ако желаете да подадете инициализационни параметри на филтъра, можете да ги получите от FilterConfig обекта, подаван на init.

http://docs.oracle.com/javaee/6/tutorial/doc/bnagb.html

JavaServer Pages

JavaServer Pages (JSP) технологията позволява лесното създаване на уеб съдържание, с прилежащи както статични, така и динамични компоненти. JSP технологията предоставя всички динамични възможности на Java Servlet технологията, но и един по-натурален подход към създаване на статичното съдържание.

Основните функции на JSP са както следва:

- Език за разработка на JSP страници, които са текстово-базирани документи които описват как да бъде обработен request и да се конструира response;
- Expression language за достъпване на server-side обекти;
- Механизми за дефиниране на разширения на JSP езика.

JSP технологията съдържа още API, ползвано от разработчици на уеб контейнери.

Какво е JSP страница?

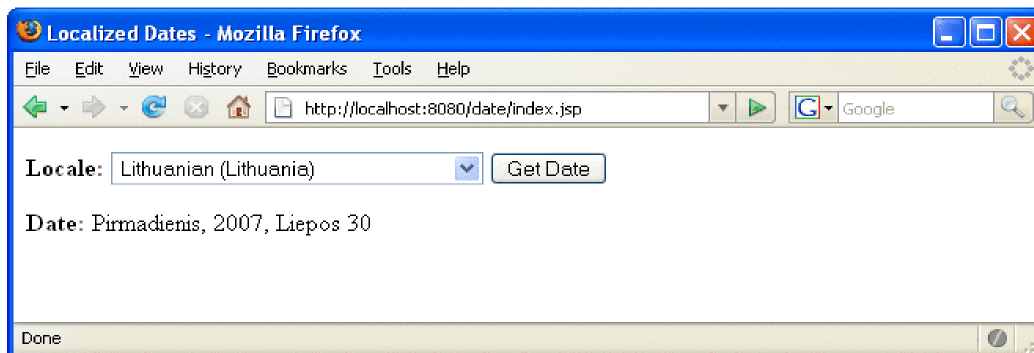
Дадена JSP страница представлява текстов документ, съдържащ два вида текст: статични данни, които могат да бъдат изразени чрез всеки текстово-базиран формат (като например, HTML, SVG, WML и XML), и JSP елементи, които конструират динамично съдържание.

Препоръчително е файловото разширение на сорс файла на JSP страници да бъде .jsp. Страницата може да бъде композирана от top файл, който включва други файлове, съдържащи или пълна JSP страница, или фрагмент от такава. Препоръчителното разширение за сорс файл на фрагмент от JSP страница е .jspf.

JSP елементите в дадена JSP страница могат да бъдат изразени в два синтаксиса, стандартен и XML, като даден файл може да ползва само един синтаксис. JSP страница в XML синтаксис представлява XML документ и може да бъде манипулирана от иснтрументи и APIs за XML документи.

Прост пример за JSP страница

Долупосочената уеб страница представлява форма, позволяваща избора на местоположение и визуализираща датата посредством маниер, подходящ за него.



JSP страницата index.jsp представлява типична смесица на статичен HTML markup и JSP елементи.

Текстът в удебелен шрифт в примерния код съдържа следните типове JSP конструкции:

- Директива за страницата (<%@page ... %>) настройва типа съдържание, връщано от нея;
- Директива за tag library (<%@taglib ... %>) импортва определени tag библиотеки;
- jsp:useBean представлява стандартен елемент, който създава обект, съдържащ колекция от локации и инициализира идентификатор, който сочи към този обект;
- JSP expression language експрешъните (${ }) получават стойността на обектните характеристики. Стойностите се ползват за настройка на custom tag атрибутни стойности и създаване на динамично съдържание;
- Custon tags настройва променлива (c:set), итерира колекция от имена на локации (c:forEach), и условно вкарва HTML текст в response (c:if, c:choose, c:when, c:otherwise);
- jsp:setProperty е друг стандартен елемент, който настройва стойността на обектна характеристика;
- Функцията (f:equals) тества равенството на атрибут и текущия елемент на колекция. (Вграденият == оператор обикновено се ползва за тестване на равенство.).

Сорс кодът на горната JSP страница:

```
<%@ page contentType="text/html; charset=UTF-8" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core
"
        prefix="c" %>
<%@ taglib uri="/functions" prefix="f" %>
<html>
<head><title>Localized Dates</title></head>
<body bgcolor="white">
<jsp:useBean id="locales" scope="application"
    class="mypkg.MyLocales"/>

<form name="localeForm" action="index.jsp" method="post">
```

```
<c:set var="selectedLocaleString" value="${param.locale}" />
<c:set var="selectedFlag"
    value="${!empty selectedLocaleString}" />
<b>Locale:</b>
<select name=locale>
<c:forEach var="localeString" items="${locales.localeNames}" >
<c:choose>
   <c:when test="${selectedFlag}">
      <c:choose>
         <c:when
             test="${f:equals(selectedLocaleString, localeString)}" >
            <option selected>${localeString}</option>
         </c:when>
         <c:otherwise>
            <option>${localeString}</option>
         </c:otherwise>
      </c:choose>
   </c:when>
   <c:otherwise>
      <option>${localeString}</option>
   </c:otherwise>
</c:choose>
</c:forEach>
</select>
<input type="submit" name="Submit" value="Get Date">
</form>

<c:if test="${selectedFlag}" >
   <jsp:setProperty name="locales"
      property="selectedLocaleString"
      value="${selectedLocaleString}" />
   <jsp:useBean id="date" class="mypkg.MyDate"/>
   <jsp:setProperty name="date" property="locale"
      value="${locales.selectedLocale}"/>
   <b>Date: </b>${date.date}</c:if>
</body>
</html>
```

Транслация и компилация

По време на фазата по транслация, всеки тип данни от дадена JSP страница бива третиран по различен начин. Статичните данни се трансформират в код, който ще пусне данните в наличния response stream. JSP елементите се третират както следва:

- Директивите се ползват за контрол върху това как уеб контейнера транслира и изпълнява дадена JSP страница;

- Скриптинг елементите (оформящи scriptlets) се вкарват в сървлет класа на JSP страницат;
- Expression language експрешъните се подават като параметри при извикване на JSP expression evaluator;
- jsp:[set|get]Property елементите се конвертират в метод извиквания на JavaBeans компоненти;
- jsp:[include|forward] елементите се конвертират в извиквания на Java Servlet API;
- jsp:plugin елементът се конфертира в извикване на tag handler, имплементиращ custom tag.

В приложния сървър, сорс кодът за даден сървлет, създаден от JSP страница на име pageName се намира в следния файл:

domain-dir/generated/jsp/j2ee-modules/WAR-NAME/pageName_jsp.java

Например, сорс кодът на индекс страницата (наименована index.jsp) в горния пример ще бъде наименован по следния начин:

domain-dir/generated/jsp/j2ee-modules/date/index_jsp.java

И двете фази по транслация и компилация могат да предизвикат грешки, наблюдавани само когато страницата бива извикана за пръв път. Ако дадена грешка се появи по време на някоя от тези фази, сървърът ще върне JasperException и съобщение, включващо името на JSP страницата и реда на възникване на грешката.

След като страницата е била транслирана и компилирана, нейният сървлет (в по-голямата му част) следва сървлет жизнения цикъл (servlet lifecycle):

1. Ако не съществува инстанция на сървлет за JSP страницата, тогава контейнерът:
    1. Зарежда сървлета за JSP страницата.
    2. Инстанциира сървлет класа.
    3. Инициализира сървлет инстанцията чрез извикване на jspInit метода.
2. Контейнерът извиква _jspService метода, подавайки наличните request и response обекти.

Ако контейнерът трябва да изтрие сървлет на JSP страница, извиква jspDestroy метода.

http://docs.oracle.com/javaee/5/tutorial/doc/bnagx.html

JavaServer Faces

JavaServer Faces технологията представлява server-side компонентен framework за изграждане на Java уеб приложения.

JavaServer Faces технологията се състои от следното:

- API за репрезентиране на компоненти и управление на тяхното състояние (state); боравене със събития, server-side валидация и конвертиране на данни; дефиниране на навигация между страниците; поддръжка на интернационализация и достъпност; и предоставяне на възможност за разширяване на всички тези функции;
- Tag библиотеки за добавяне на компоненти към уеб страници и за свързането им със server-side обекти.

JavaServer Faces предоставя добре дефиниран програмен модел и редица tag библиотеки. Tag библиотеките съдържат tag handlers, имплементиращи компонентни тагове. Тези функции значително улесняват бремето по изграждане и поддръжка на уеб приложения със server-side user interfaces (UIs). С минимални усилия, можете да извършите следните задачи:

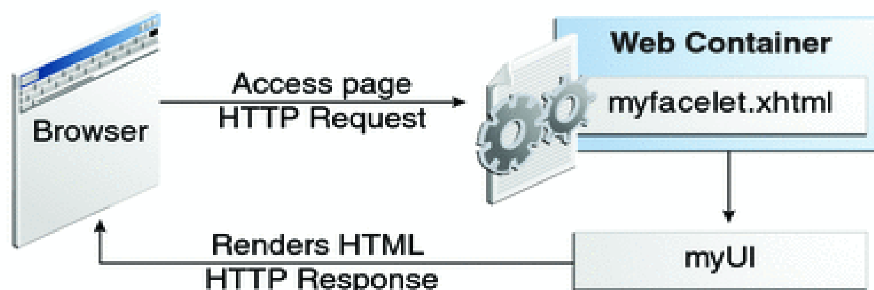
- Създаване на уеб страница;
- Пускане на компоненти в уеб страница чрез добавяне на компонентни тагове;
- Свързане на компоненти от страницата към server-side данни;
- Свързане на генерирани от компонентите събития към server-side application код;
- Запазване и възстановяване състоянието на приложението отвъд живота на server requests;
- Преизползването и наследяването на компоненти чрез късттъмизация.

Какво представлява JavaServer Faces приложение?

Функционалността предоставена от JavaServer Faces приложение е подобна на всяко друго Java уеб такова. Типичното JavaServer Faces приложение включва следните части:

- Набор от уеб страници, в които биват поставяни компонентите;
- Набор от тагове за добавяне на компоненти към уеб страница;
- Набор от managed beans, представляващи олекотени container-managed обекти (POJOs) с минималистични изисквания. Те поддържат малък набор от базови services, като например ресурсно инжектиране (resource injection), lifecycle callbacks и интерсептори (interceptors);
- Web deployment descriptor (web.xml file);
- Опционално, един или повече application configuration resource files, като например faces-config.xml файла, който може да бъде ползван за дефиниране на навигация между страниците и конфигуриране на beans и други custom обекти като компоненти;
- Опционално, набор от custom обекти, които могат да включват custom компоненти, валидатори (validators), конвъртъри (converters) или листенерс, създадени от разработчика;
- Опционално, набор от custom tags за репрезентацията на custom обекти в страницата.

Изображението по-долу демонстрира интеракцията между клиент и сървър в типично JavaServer Faces приложение. В отговор на клиентския request, дадена уеб страница бива rendered от уеб контейнера.



Уеб страницата, myfacelet.xhtml е изградена ползвайки JavaServer Faces компонентнит тагове. Те се ползват за добавяне на компоненти към наличното view (репрезентирано чрез myUI в диаграмата), представляващо server-side репрезентация на страницата. В допълнение към компонентите, уеб страницата може също да реферира обекти, като например:

- Всякакви event listeners, validators и converters, регистрирани в компонентите;
- JavaBeans компоненти, прихващащи данните и обработващи специфичната за приложението функционалност на компонентите.

За всеки клиентски рекуест, наличното view бива rendered като response. Рендърването представлява процес където на базата на наличното server-side view, уеб контейнерът генерира изходни данни като HTML или xHTML, които могат да бъдат прочетени от клиента, като например браузър.

Създаване на просто JavaServer Faces приложение

JavaServer Faces технологията предоставя лесен и user-friendly процес за създаване на уеб приложения. Разработването на просто JavaServer Faces приложение обикновено изисква изпълняване на следните задачи:

- Разработка на managed beans;
- Създаване на уеб страници ползвайки компонентни тагове;
- Mapping на javax.faces.webapp.FacesServlet инстанцията.

Примерът по-долу представлява Hello приложение, включващо managed bean и уеб страница. Когато бива достъпена от клиента, уеб страницата извежда съобщение “Hello World!”.

Разработка на managed bean

Managed bean представлява олекотен container-managed обект. Компонентите в дадена страница биват асоциирани с managed beans, предоставящи логиката в приложението. Примерният managed bean Hello.java съдържа следния код:

```
package hello;


import javax.faces.bean.ManagedBean;


@ManagedBean
public class Hello {


    final String world = "Hello World!";


    public String getworld() {
        return world;
    }
}
```

Горният примерен managed bean инициализира променлива world с низовата стойност “Hello World!”. @ManagedBean анотацията регистрира managed bean като ресурс с JavaServer Faces имплементация.

Създаване на уеб страница

В типично Facelets приложение, уеб страниците се създават във вид на xHTML. Примерната уеб страница beanhello.xhtml представлява проста xHTML страница със следното съдържание:

```
<html lang="en"
    xmlns="http://www.w3.org/1999/xhtml"
    xmlns:h="http://java.sun.com/jsf/html">
  <h:head>
     <title>Facelets Hello World</title>
  </h:head>
  <h:body>
     #{hello.world}
  </h:body>
</html>
```

Дадена Facelets xHTML страница може също да съдържа няколко други елемента.

Уеб страницата се свързва с managed bean посредством наличния Expression Language (EL) value expression #{hello.world}, който извлича стойността на world характеристиката от managed bean Hello. Обърнете внимание, че hello реферира managed bean Hello. Ако няма специфицирано име в @ManagedBean анотацията, то той винаги се достъпва с малка първа буква на класа.

Mapping на FacesServlet инстанция

Финалната задача изисква mapping на FacesServlet, което се случва посредством web deployment descriptior (web.xml). Един типичен mapping на FacesServlet изглежда така:

```
<servlet>
  <servlet-name>Faces Servlet</servlet-name>
  <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
  <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
  <servlet-name>Faces Servlet</servlet-name>
  <url-pattern>/faces/*</url-pattern>
</servlet-mapping>
```

Горният файлов сегмент репрезентира част от JavaServer Faces web deployment descriptor. Дескрипторът за уеб разгръщане може още да съдържа друго съдържание, релевантно за конфигурацията на JavaServer Faces приложение.

Lifecycle на hello приложение

Всяко уеб приложение има жизнен цикъл. Широко разпространени задачи, като например боравене с прииждащи requests, декодиране на параметри, модифициране и запазване на състояние и rendering на уеб страници към браузъра, всички се формират по време на web application lifecycle. Някои frameworks за уеб приложения скриват детайлите от жизнения цикъл от ползващите ги, докато други изискват изричното им ръчно управление.

По подразбиране JavaServer Faces се справя автоматично с повечето lifecycle actions вместо вас. Същевременно обаче извежда различни stages от request lifecycle, така че да можете да модифицирате или извършвате различни действия с тях, ако вашето приложение го налага.

В началото не е необходимо потребителят да разбира жизнения цикъл на JavaServer Faces приложение, но тази информация може да бъде полезна при създаване на по-комплексни приложения.

Жизненият цикъл на JavaServer Faces приложенията започва и завършва със следните дейности: Клиентът прави request за дадена уеб страница, а сървърът отговаря със страицата. Жизненият цикъл се състои от две основни фази: execute и render.

По време на execute фазата, следните действия биват извършени:

- Приложното view бива изградено или възстановено;
- Прилагат се стойностит на request параметрит;
- Изпълняват се конвертиране и валидации за компонентните стойности;
- Managed beans са обновени с компонентните стойности;
- Извиква се логиката на приложението.

При първия (initial) request, само самото view бива изградено. За последващите (postback) requests, някои или всички други действия са изпълнени.

В render фазата, наличното requested view е rendered като response към клиента. Рендърването обикновено представлява процес по генериране на изходни данни, като например HTML или xHTML, които могат да бъда тпрочетени от клиента, обикновено браузър.

Следното кратко описание на примерното JavaServer Faces приложение, минавайки през жизнения си цикъл, обобщава случващите се действия. След разгръщане върху GlassFish server, hello примерното приложение минава през следните сцени (stages):

1. Когато приложението бива изграждано и разгръщано върху сървъра, то е в неинициирано състояние (uninitiated state).
2. Когато клиентът направи първоначалия request за beanhello.xhtml уеб страницата, hello Facelets приложението бива компилирано.
3. Компилираното Facelets приложение е изпълнено, и ново компонентно дърво е конструирано за hello приложението, и позиционирано в javax.faces.context.FacesContext.
4. Компонентното дърво е популирано (populated) с компонента и managed bean характеристиката, асоциирана с него, репрезентирана чрез EL expression hello.world.
5. Ново view бива построено, базирано на компонентното дърво.
6. Това view е rendered към клиента като response.
7. Компонентното дърво се унищожава автоматично.
8. При последващи requests компонентното дърво бива изграждано наново, като запазеното състояние бива приложено.

Bean Validation

Валидиране входните данни, получени от потребителя, с цел запачване итегритета на данните, представлява важна част от логиката на дадено приложение. Валидацията на данни може да бъде реализирана в различни слоеве дори при най-простите приложения.

JavaBeans Validation (Bean Validation) представлява модел за валидация, предоставен от Java EE платформата. Поддържа ограничения (constraints) във вид на анотации, поставени на полета, методи или класове от даден JavaBeans компонент.

Ограниченията могат да бъдат вградени или дефинирани от потребителя. Вторите такива се наричат custom constraints. Няколко вградени ограничения са налични в javax.validation.constraints пакета.

Списък на наличните вградени ограничения:

Ограничение	Описание	Пример
@AssertFalse	Стойността на поле или характеристика трябва да бъде false	@AssertFalse boolean isUnsupported;
@AssertTrue	Стойността на поле или характеристика трябва да бъде true	@AssertTrue boolean isActive;
@DecimalMax	Стойността на поле или характеристика трябва да бъде десетична такава, по-малка или равна на номера посочен във value елемента	@DecimalMax("30.00") BigDecimal discount;
@DecimalMin	Стойността на поле или характеристика трябва да бъде десетична такава, по-голяма или равна на номера посочен във value елемента	@DecimalMin("5.00") BigDecimal discount;
@Digits	Стойността на поле или характеристика трябва да бъде номер между специфицирания диапазон. integer елементът специфицира максимума интегрални числа за номера, а fraction елементът специфицира максимума фракционални числа за номера	@Digits(integer=6, fraction=2) BigDecimal price;
@Future	Стойността на поле или характеристика трябва да бъде дата в бъдещето	@Future Date eventDate;
@Max	Стойността на поле или характеристика трябва да бъде integer стойност, по-малка или равна на посочения номер във value елемента	@Max(10) int quantity;

Ограничение	Описание	Пример
@Min	Стойността на поле или характеристика трябва да бъде integer стойност, по-голяма или равна на посочения номер във value елемента	@Min(5) int quantity;
@NotNull	Стойността на поле или характеристика не трябва да бъде null	@NotNull String username;
@Null	Стойността на поле или характеристика трябва да бъде null	@Null String unusedString;
@Past	Стойността на поле или характеристика трябва да бъде дата в миналото	@Past Date birthday;
@Pattern	Стойността на поле или характеристика трябва да отговаря на regular expression, дефиниран в regexp елемента	@Pattern(regexp="\\(\\d{3}\\)\\d{3}-\\d{4}") String phoneNumber;
@Size	Големината на поле или характеристика е оценена и трябва да отговаря на специфицираните граници. Ако полето или характеристиката са String, големината на полето е преценена. Ако полето или характеристиката са колекция, големината на колекцията е преценена. Ако полето или характеристиката е Map, то големината на този Map е оценена. Ако полето или характеристиката са масив, големината на масивът е преценена. Използвайте някой от опционалните min или max елементи за да специфицирате границите.	@Size(min=2, max=240) String briefMessage;

В следният пример, constraint е поставен на поле ползвайки вграденото @NotNull ограничение:

```
public class Name {
    @NotNull
    private String firstname;

    @NotNull
    private String lastname;
}
```

Можете също да позиционирате повече от едно ограничение върху един JavaBeans компонентен обект. Например, можете да добавите допълнително ограничение по отношение големината на полетата за firstname и lastname:

```
public class Name {
    @NotNull
    @Size(min=1, max=16)
    private String firstname;

    @NotNull
    @Size(min=1, max=16)
    private String lastname;
}
```

http://docs.oracle.com/javaee/6/tutorial/doc/bnaph.html

Enterprise Beans

Enterprise beans представляват Java EE компоненти, имплементиращи Enterprise JavaBeans (EJB) технологията. Enterprise beans вървят в EJB контейнер, runtime environment в рамките на GlassFish server. Въпреки прозрачността към разработчика на приложението, EJB контейнера предоставя services на системно ниво, като например транзакции и сигурност по отношение на неговите enterprise beans. Тези services позволяват бързо изграждане и разгръщане на enterprise beans, което формира ядрото на транзакционните Java EE приложения.

Какво е Enterprise Bean?

Написан на програмния език Java, даден enterprise bean представлява server-side компонент, който енкапсулира бизнес логиката на дадено приложение. Бизнес логиката представлява кодът, който изпълнява предназначението на приложението. В дадено приложение за контрол на инвентаризация например, enterprise beans може да имплементират бизнес логиката в методи наречени checkInventoryLevel и orderProduct. Чрез извикване на тези методи, клиентите могат да достъпват услугите по инвентаризация, предоставени от приложението.

Ползи от Enterprise Beans

По различни причини enterprise beans опростяват разработката на големи, дистрибутирани приложения. Първо защото EJB контейнерът предоставя services на систено ниво на enterprise beans, следователно bean разработчикът може да се концентрира върху разрешаването на бизнес проблеми. EJB контейнерът, за разлика от bean разработчика, е отговорен за тези system-level servives, като например управление на транзакциите и оторизация.

Второ, защото по-скоро beans отколкото клиента съдържа бизнес логиката на приложението, следователно разработчика на клиента може да се фокусира върху неговата репрезентация. Клиентският разработчик не се занимава с имплементиране на бизнес правила или достъп да базата данни. Като резултат на това, клиентите са по-малки, което е изключително важно за клиенти, вървящи върху по-малки устройства.

Трето, защото enterprise beans са преносими компоненти, следователно едно ново приложение може да бъде построено от вече съществуващи беанс. Ползвайки стандартните APIs, тези приложения могат да вървят върху всеки съвместим с Java EE сървър.

Кога да ползваме Enterprise Beans

Трябва да се насочите към употребата на enterprise beans, ако вашето приложение притежава някой от следните изисквания:

- Приложението трябва да бъде скалируемо. За да се посрещне нарастващият брой потребители, може да имате нужда да дистрибутирате компонентите на приложението на множество машини. Enterprise beans могат не само да вървят върху различни машини, но също така тяхната локация остава прозрачна за клиентите;

- Транзакционалността трябва да подсигури интегритета на данните. Enterprise beans поддържат транзакции, механизмите за управление на едновременен достъп до споделени обекти;

- Приложението ще има различни видове клиенти. Само с няколко реда код, отдалечените клиенти могат лесно да локализират наличните enterprise beans. Тези клиенти могат да бъдат малки, различни и многобройни.

Типове Enterprise Beans

Тип	Предназначение
Session	Изпълнява задача за даден клиент; опционално, може да имплементира web service
Message-driven	Държи се като listener за определен тип messaging, като например Java Message Service (JMS) API

Interceptors

Интерсепторите се ползват във връзка с Java EE managed класове, за да позволят на разработчиците да извикат interceptor методи върху асоцииран target class, заедно с method invocations или събития от жизнения цикъл. Честата употреба на interceptors е logging, одитиране и profiling.

Спецификацията за Interceptors 1.1 е част от финалния release на JSR 318, Enterprise JavaBeans 3.1, https://jcp.org/en/jsr/detail?id=318

Даден интерсептор може да бъде дефиниран в рамките на target class като interceptor method, или в даден асоцииран клас наречен interceptor class. Интерсептор класовете съдържат методи, извиквани заедно с методите на lifecycle events на target класа.

Интерсептор класовет и методите биват дефинирани посредством употребата на метаданни във вид на анотации, или на deployment descriptor в приложението, съдържащ описаните interceptors и target classes.

Бележка:

Приложения, ползващи deployment descriptor за дефиниране на интерсептори не са преносими между Java EE сървъри.

Интерсептор методите в рамките на target класа или в даден interceptor class са анотирани с една от следните анотации:

Interceptor Metadata Annotation	Описание
javax.interceptor.AroundInvoke	Обозначава метода като interceptor метод
javax.interceptor.AroundTimeout	Обозначава метода като timeout interceptor, за вмъкване на timeout методи за enterprise bean timers

Interceptor Metadata Annotation	Описание
javax.annotation.PostConstruct	Обозначава метода като interceptor такъв за post-construct lifecycle събития
javax.annotation.PreDestroy	Обозначава метода като interceptor метод за pre-destroy lifecycle събития

http://docs.oracle.com/javaee/6/tutorial/doc/gijsz.html

http://docs.oracle.com/javaee/6/tutorial/doc/gkigq.html

Persistence

Java Persistence API предоставя на Java разработчиците възможност за реализация на object/relational mapping за управление на релационни данни в Java приложения. Java Persistence се състои от четири области:

- Java Persistence API;
- Query language;
- Java Persistence Criteria API;
- Object/relational mapping метаданни.

Въведение в ORM

Когато работим с обекто-ориентирани системи, съществува несъответствие между обектния модел и релационната база данни. RDBMS системите репрезентират данните в плосък формат, докато Java репрезентира данните в граф (graph) от взамосвързани обекти.

Нека вземем за пример следния клас:

```
public class Employee {
  private int id;
  private String first_name;
  private String last_name;
  private int salary;

  public Employee() {}
  public Employee(String fname, String lname, int salary) {
    this.first_name = fname;
    this.last_name = lname;
    this.salary = salary;
  }
  public int getId() {
    return id;
  }
  public String getFirstName() {
    return first_name;
```

```
  }
  public String getLastName() {
    return last_name;
  }
  public int getSalary() {
    return salary;
  }
}
```

Обекти от горния клас трябва да бъдат съхранявани и извлизани в/от следната RDBMS таблица:

```
create table EMPLOYEE (
  id INT NOT NULL auto_increment,
  first_name VARCHAR(20) default NULL,
  last_name  VARCHAR(20) default NULL,
  salary     INT  default NULL,
  PRIMARY KEY (id)
);
```

Първият проблем възниква, ако се появи нуждата да модифицираме дизайна на нашата база данни след като сме разработили няколко страници от нашето приложение.

Второ, зареждането и съхранението на обекти в релационна база води до следните пет проблема:

Разминаване	Описание
Гранулярност	Понякога е наличен обектен модел, съдържащ повече класове от наличните в база данни кореспондиращи таблици
Наследяване	RDBMS не дефинира нищо наподобяващо концепцията за наследяване, явяваща се натурална парадигма в обекто-ориентираните програмни езици
Идентичност	RDBMS дефинира точно едно понятие за “еднаквост”: primary key. Java обаче дефинира обектна идентичност (a == b) и обектно равенство (a.equals(b)).
Асоцииране	Обекто-ориентираните езици репрезентират асоциациите ползвайки обектни референции, а RDBMS репрезентира дадена асоциация като foreign key колона
Навигация	Начинът за достъп на обекти в Java и RDBMS са фундаментално различни

Object-Relational Mapping (ORM) представлява решение за справяне с гореизброените проблеми.

Какво е ORM?

ORM е програмна техника за конвентиране на данни между релационни бази и обекто-ориентирани програмни езици като Java. Дадена ORM система притежава следните преимущества в сравнение с чисто JDBC:

- Позволява на бизнес кода да достъпва обекти вместо таблици в базата;
- Скрива детайли по SQL заявките от OO логиката;
- Базира се върху JDBC;
- Няма нужда да се съобразяваме с имплементацията на базата;
- Базирани върху бизнес концепции Entities вместо структури в базата;
- Бърза разработка на приложения.

Управление на entities

Наличните entities се управляват от т. нар. entity manager, репрезентирн от javax.persistence.EntityManager инстанция. Всяка EntityManager инстанция е асоциирана с persistence context: набор от managed entity инстанции, съществуващи в определен data store. Даден persistence context дефинира обхвата на създаване, перзистване и изтриване на entity instances. Интерфейсът дефинира методите, използвани за интеракция с persistence контекста.

EntityManager интерфейсът

EntityManager API създава и изтрива persistent entity инстанции, намира entities по техния primary key, и позволява изпълняването на заявки върху тях.

УПРАВЛЕНИЕ НА НИВО КОНТЕЙНЕР

Ползвайки container-managed entity manager инстанцията на EntityManager persistence контекстът бива автоматично подавана от контейнера към компонентите на приложението, ползващи EntityManager инстанцията в рамките на единична JTA транзакция.

JTA транзакции обикновено включват извиквания на множество приложни компоненти. За завършването на JTA транзакция, тези компоненти обикновено се нуждаят от единичен persistence context. Това се поражда от инжектирането на EntityManager в компонентите на приложението посредством javax.persistence.PersistenceContext анотацията. Наличният persistence context автоматично подава текущата JTA транзакция, а EntityManager референции от същия persistence unit предоставят достъп до persistence контекста в рамките на транзакцията. Чрез автоматичното подаване на persistence context, компонентите на приложението нямат нужда да подават референции на EntityManager инстанциите един на друг, за да направят промени в рамките на единична транзакция. Java EE контейнерът управлява жизнения цикъл на container-managed entity managers.

За да се сдобиете с EntityManager инстанция, инжектирайте го в компонент на приложението както следва:

```
@PersistenceContext
EntityManager em;
```

УПРАВЛЕНИЕ НА НИВО ПРИЛОЖЕНИЕ

При application-managed entity manager, persistence контекстът не бива подаван на компонентите на приложението, и жизненият цикъл на EntityManager инстанциите е управляван от самото приложение.

Управляваните от приложението entity managers се ползват когато приложенията имат нужда от достъп до persistence context, който не бива подаден в JTA транзакция през EntityManager инстанции в определен persistence context. EntityManager и неговите асоциирани persistence контексти се създават и унищожават експлицитно от приложението. Ползват се още когато директно инжектираните инстанции на EntityManager не могат да бъдат реализирани, тъй като EntityManager инстанциите не са thread-safe. EntityManagerFactory инстанциите са thread-safe.

В този случай приложенията създават инстанции на EntityManager посредством createEntityManager метода на javax.persistence.EntityManagerFactory.

За да се сдобиете с инстанция на EntityManager, първо ви е необходима инстанция на EntityManagerFactory класа чрез инжектирането му в компонента на приложението ползвайки javax.persistence.PersistenceUnit анотацията:

```
@PersistenceUnit
EntityManagerFactory emf;
```

След това можете да получите EntityManager по сления начин:

```
EntityManager em = emf.createEntityManager();
```

Управляваните на ниво приложение entity managers не въвеждат автоматично JTA transaction context-а. Подобни приложения трябва ръчно да се сдобият с достъп до JTA transaction manager и да добавят информация за разграничаване на транзакциите при изпълнение на entity операции. javax.transaction.UserTransaction интерфейсът дефинира методи за начало, commit и roll back на транзакции. Инжектирайте инстанция на UserTransaction чрез създаване на instance променлива, анотирана с @Resource:

```
@Resource
UserTransaction utx;
```

За започване на транзакция извикайте UserTransaction.begin метода. Когато всички операции по даденото entity са приключили, извикайте UserTransaction.commit метода за да комитнете транзакцията. UserTransaction.rollback метода се ползва за връщане на текущата транзакция.

Примерът по-долу демонстрира управление на транзакции:

```
@PersistenceContext
EntityManagerFactory emf;
EntityManager em;
@Resource
UserTransaction utx;
...
em = emf.createEntityManager();
try {
  utx.begin();
  em.persist(SomeEntity);
  em.merge(AnotherEntity);
  em.remove(ThirdEntity);
  utx.commit();
```

```
} catch (Exception e) {
  utx.rollback();
}
```

Заявки

Java Persistence API предоставя следните методи за заявки към entities:

- Java Persistence query language (JPQL) представлява прост, низово-ориентиран език подобен на SQL, използван за заявки върху entities и техните relationships. http://docs.oracle.com/javaee/6/tutorial/doc/bnbtg.html
- Criteria API се ползва за създаването на typesafe заявки върху entities и техните relationships. http://docs.oracle.com/javaee/6/tutorial/doc/gjitv.html

Повече за заявките можете да прочетете тук:

http://docs.oracle.com/javaee/6/tutorial/doc/gjise.html

Дескриптори за разгръщане

EntityManager API е чудесно, но как сървърът знае в коя база данни се предполага, че трябва да ползва? Как конфигурираме наличния object relational mapping engine и cache за по-добра производителност и разследване на проблеми? Файлът persistence.xml предоставя необходимата гъвкавост за конфигуриране на EntityManager.

persistence.xml файлът представлява стандартен конфигурационен файл в JPA. Трябва да бъде включен в META-INF директорията в даден JAR файл, съдържащ entity beans. Файлът трябва да дефинира persistence-unit с уникално име в рамките на текущия classloader. Provider атрибута специфицира конкретната имплементация на JPA EntityManager. jta-data-source сочи към JNDI името на базата за текущия persistence unit.

Пример:

```
<persistence>
  <persistence-unit name="myapp">
    <provider>org.hibernate.ejb.HibernatePersistence</provider>
    <jta-data-source>java:/DefaultDS</jta-data-source>
    <properties>
      property name="hibernate.dialect"
            value="org.hibernate.dialect.HSQLDialect"/>
      <property name="hibernate.hbm2ddl.auto" value="create-drop"/>
    </properties>
  </persistence-unit>
</persistence>
```

http://docs.oracle.com/javaee/6/tutorial/doc/bnbpz.html

http://www.tutorialspoint.com/hibernate/orm_overview.htm

Contexts and Dependency Injection (CDI)

CDI представлява една от няколкото Java EE функционалности, подпомагаща свързването на web и transactional tiers на Java EE платформата. CDI е набор от услуги, които ползвани заедно, позволяват

на разработчиците лесната употреба на enterprise beans заедно с JavaServer Faces технологията в уеб приложения.

JSR 299, Contexts and Dependency Injection for Java EE, https://jcp.org/en/jsr/detail?id=299 (познато преди като Web Beans). Свързани спецификации ползвани от CDI включват:

- JSR 330 - Dependency Injection for Java, https://jcp.org/en/jsr/detail?id=330
- JSR 316 - Java Platform, Enterprise Edition 6 (Java EE 6) Specification, https://jcp.org/en/jsr/detail?id=316

Фундаментални услуги

- Contexts: Способността за binding на жизнения цикъл и интеракциите на stateful компоненти към добре дефинираните и разширяеми lifecycle contexts
- Dependency injection: Сопсобността за инжектиране на компоненти в приложението по typesafe начин, включая способността за избор по време на разгръщане на това коя имплементация на определен интерфейс да бъде инжектирана.

В допълнение, CDI предоставя следните услуги:

- Интеграция с Expression Language (EL), която позволява на компонентът да бъде ползван директно в рамките на JavaServer Faces страница или JavaServer Pages такава;
- Способността за декорация на инжектираните компоненти;
- Способността за асоцииране на интерсептори в компонентите, ползвайки typesafe interceptor bindings;
- Event-notification model;
- Web conversation scope в допълнение към трите стандартни такива (request, session и application), дефинирани от Java Servlet спецификацията;
- Пълен Service Provider Interface (SPI), позволяващ на външни frameworks да се интегрират чисто в Java EE среда.

Произход

Идеята за CDI произлиза от Seam 3, представляваща външен за Java платформата framework. Повече можете да видите тук:

https://docs.jboss.org/seam/3/latest/reference/en-US/html/

http://docs.oracle.com/javaee/6/tutorial/doc/gjbnr.html

Web Services

Терминът web API като цяло обхваща и двете страни на комуникацията между компютърни системи по мрежата - API services предоставяни от сървъра, както и API предоставяно от клиента като например, web browser. Сървърната част от web API представлява програмен интерфейс на дефинирана система за request-response съобщения, като обикновено е реферирана като Web Service. Съществуват няколко дизайн модела за уеб услуги, но двата най-разпространени са SOAP и REST.

Simple Object Access Protocol (SOAP)

SOAP разчита основно на XML, като заедно със схемите (schemas) дефинира строго типизирано framework за съобщения. Всяка операция предоставяна от услугата е експлицитно дефинирана, с прилежаща XML структура на заявката и отговора за тази операция. Всеки входен параметър е дефиниран по подобен начин и заключен към тип - например integer, string или друг комплексен обект.

Всичко това е посочено в Web Service Description (или в по-късни версии Definition) Language (WSDL). WSDL често бива обяснен като договор между доставчика и консумиращия дадена услуга. В програмната терминология за WSDL можем да мислим като метод сигнатура за web service.

Пример:

Проста обмяна на съобщения изглежда по следния начин

Заявка от клиент:

```
POST http://www.stgregorioschurchdc.org/cgi/websvccal.cgi HTTP/1.1
Accept-Encoding: gzip,deflate
Content-Type: text/xml;charset=UTF-8
SOAPAction: "http://www.stgregorioschurchdc.org/Calendar#easter_date"
Content-Length: 479
Host: www.stgregorioschurchdc.org
Connection: Keep-Alive
User-Agent: Apache-HttpClient/4.1.1 (java 1.5)
<?xml version="1.0"?>
<soapenv:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:cal="http://www.stgregorioschurchdc.org/Calendar">
<soapenv:Header/>
<soapenv:Body>
  <cal:easter_date soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
  <year xsi:type="xsd:short">2014</year>
</cal:easter_date>
</soapenv:Body>
</soapenv:Envelope>
```

Отговор от услуга:

```
HTTP/1.1 200 OK
Date: Fri, 22 Nov 2013 21:09:44 GMT
Server: Apache/2.0.52 (Red Hat)
SOAPServer: SOAP::Lite/Perl/0.52
Content-Length: 566
Connection: close
Content-Type: text/xml; charset=utf-8
<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```
xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
<SOAP-ENV:Body>
  <namesp1:easter_dateResponse
xmlns:namesp1="http://www.stgregorioschurchdc.org/Calendar">
<s-gensym3 xsi:type="xsd:string">2014/04/20</s-gensym3>
</namesp1:easter_dateResponse>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

От горния пример можете да видите изпратеното по HTTP съобщение. SOAP е агностичен по отношение на транспортния протокол, следователно може да бъде изпращан посредством почти всеки протокол като HTTP, SMTP, TCP или JMS. SOAP съобщенията сами по себе си трябва да бъдат XML-форматирани. Както при всеки един XML документ, и тук трябва да бъде наличен един root елемент - в случая явяващ се Envelope (плик). Той съдържа два задължителни елемента - Header и Body. Останалата част от елементите в това съобщение са описани в съответния WSDL.

Акомпаниращият WSDL, дефиниращ горната услуга, изглежда по следния начин (детайлите не са важни, но целият документ бива представен с цел завършеност):

```
<?xml version="1.0"?>
<definitions xmlns:tns="http://www.stgregorioschurchdc.org/Calendar"
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns="http://schemas.xmlsoap.org/wsdl/"
name="Calendar" targetNamespace="http://www.stgregorioschurchdc.org/Calendar">
<message name="EasterDate">
  <part name="year" type="xsd:short"/>
</message>
<message name="EasterDateResponse">
  <part name="date" type="xsd:string"/>
</message>
<portType name="EasterDateSoapPort">
  <operation name="easter_date" parameterOrder="year">
    <input message="tns:EasterDate"/>
    <output message="tns:EasterDateResponse"/>
  </operation>
</portType>
<binding name="EasterDateSoapBinding" type="tns:EasterDateSoapPort">
  <soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
  <operation name="easter_date">
    <soap:operation soapAction="http://www.stgregorioschurchdc.org/Calendar#easter_date"/>
    <input>
     <soap:body use="encoded" namespace="http://www.stgregorioschurchdc.org/Calendar" encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
    </input>
```

```
    <output>
      <soap:body use="encoded" namespace="http://www.stgregorioschurchdc.org/Calendar" encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
    </output>
  </operation>
</binding>
<service name="Calendar">
  <port name="EasterDateSoapPort" binding="tns:EasterDateSoapBinding">
    <soap:address location="http://www.stgregorioschurchdc.org/cgi/websvccal.cgi"/>
  </port>
</service>
</definitions>
```

Обърнете внимание, че всички части от тялото на съобщението са описани в горния документ. Също, въпреки че основното предназначение на документа е да бъде четен от компютър, той е сравнително лесен за четене от човек с програмни познания.

WSDL

WSDL дефинира всеки аспект от дадено SOAP съобщение. Възможно е дори дефинирането на това дали всеки елемент или атрибут е позволен да бъде повторен множество пъти, дали е задължителен или опционален, и дали е наложителна специфична поредност на елементите.

Честа заблуда е това, че WSDL е изискване за SOAP услуга. SOAP е разработен преди WSDL, следователно WSDL е опционален. Въпреки това е значително по-трудно да се комуникира с уеб услуга, която няма WSDL.

От друга страна, ако даден разработчик е запитан за интерфейс на текущ SOAP web service, необходимо е само да предостави WSDL. Съществуват инструменти за откриване на услугата, генериращи методи с подходящи параметри в почти всеки програмен език от точи WSDL. Много тестови инструменти на пазара работят по същия начин - тестър предоставя URL към WSDL и инструментите генерират всички извиквания с примерни параметри за всички налични методи.

Критика

WSDL може да изглежда отлично на пръв поглед - самодокументиращ и съдържащ почти пълната картина на всичко необходимо за интеграцията с услуга - но може да се превърне и в бреме. WSDL представлява договор между вас (предоставящият услугата) и всеки друг от вашите клиенти (консумиращи услугата). Ако желаете да промените вашето API, дори нещо изключително малко като добавяне на опционален параметър, то и самият WSDL трябва да бъде променен. А подобни промени означават промени и при клиента - всички ваши клиенти трябва да прекомпилират техните приложения с новия WSDL. Тази малка промяна значително увеличава бремето върху екипите по разработка (и от двете страни на комуникацията), както и тестващите такива. Поради тази причина на WSDL се гледа като заключване към версия, и повечето доставчици са противници на промяната на техните APIs.

Освен това, докато SOAP предоставя гъвкавост, като например способността да бъде пренесен посредством всеки транспортен протокол, никой всъщност не се възползва особено от това. Благодарение посоката на развитие на Интернет, всичко значимо върви по HTTP. Има нови преимущества, повечето от които биват затруднени от инфраструктурните рутери, отказващи да рутират нестандартен HTTP трафик. Само помислете - колко дълго време в световен мащаб се налага миграцията към IPv6?

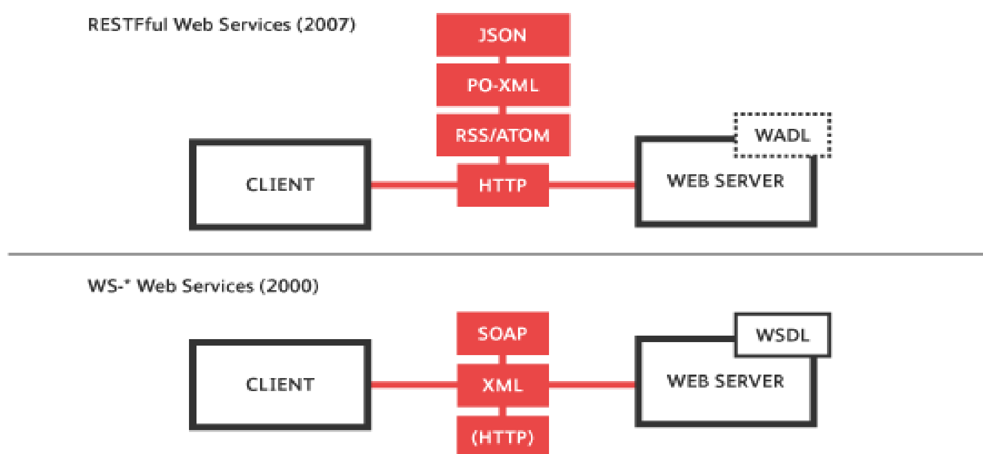
При всяка ситуация при която големината на изпратеното съобщение няма значение, или при която контролирате всичко от край до край, SOAP почти винаги е най-добрият избор. Това е приложимо основно към директната сървър-към-сървър комуникация, основно използвана за вътрешна комуникация само в рамките на дадена организация. Масовото разпространение на устройства с малко памет и процесорна мощ, свързани към множество услуги едновременно, определено налага по-лек и гъвкав модел.

REpresentational State Transfer (REST)

REST бързо става предпочитания дизайн модел за публични APIs. Налични са някои интересни статистики по отношение нарастване употребата на REST, например:

http://www.programmableweb.com/apis

REST е архитектурен стил, за разлика от SOAP който е стандартизиран протокол. REST употребява съществуващи и широко разпространени технологии, специфично HTTP, като не създава никакви допълнителни стандарти. Може да структурира данни в XML, YAML или който и да е друг машинночетим формат. Обикновено предпочитаният такъв бива JavaScript Object Notation (JSON). Както може да се очаква от JavaScript, обектите не са строго типизирани. REST следва обекто-ориентираната парадигма на съществително-глагол. REST е изключително data-driven в сравнение със SOAP, който е силно function-driven. В REST парадигмата метаданните са структурирани йерархично и репрезентирани в самия URI; в това се изразява съществителното. HTTP стандартът предлага няколко глагола, репрезентиращи операции или действия, които можете да изпълнявате върху данните. Най-често ползваните са GET, POST, PUT и DELETE.



Например, нека си представим, че имаме необходимост от три операции - потребителски вход, изход и получаване баланса от сметката на текущия потребител. Докато даден SOAP service би имплементирал всяка от тези като отделни операции (с потребителско име и парола подавани като аргументи на операцията за вход), REST би дефинирал URI като http://sample.test/api/session. Дадена POST операция (с потребителско име и парола подадени в тялото) към този URI би постигнало вход на потребителя и отваряне на сесия. Съответно дадена DELETE операция към същия URI би постигнала терминиране на сесията - ефективен изход. GET заявка към *http://sample.test/api/session/balance* би получила баланса на текущия потребител.

Пример:

Примерна обмяна на съобщение би съдържала следното

Request:

```
GET http://www.catechizeme.com/catechisms/catechism_for_young_children/daily_question.js HTTP/1.1
Accept-Encoding: gzip,deflate
Host: www.catechizeme.com
Connection: Keep-Alive
User-Agent: Apache-HttpClient/4.1.1 (java 1.5)
```

Response:

```
HTTP/1.1 200 OK
Date: Fri, 22 Nov 2013 22:32:22 GMT
Server: Apache
X-Powered-By: Phusion Passenger (mod_rails/mod_rack) 3.0.17
ETag: "b8a7ef8b4b282a70d1b64ea5e79072df"
X-Runtime: 13
Cache-Control: private, max-age=0, must-revalidate
Content-Length: 209
Status: 200
Keep-Alive: timeout=2, max=100
Connection: Keep-Alive
Content-Type: js; charset=utf-8
{
  "link": "catechisms\/catechism_for_young_children\/questions\/36",
  "catechism": "Catechism for Young Children",
  "a": "Original sin.",
  "position": 36,
  "q": " What is that sinful nature which we inherit from Adam called?"
}
```

Горното съобщение бива пратено по HTTP, ползвайки GET глагола. Забележете също, че URI който задължително присъства и при SOAP където request не носи значение, тук има такова. Тялото на съобщението е значително по-малко, като в горния пример дори отсъства.

Дадена REST услуга също има схема в Web Application Description Language (WADL). WADL схемата за горното извикване би изглеждала по следния начин:

```
<?xml version="1.0"?>
<application xmlns="http://wadl.dev.java.net/2009/02">
<doc xml:lang="en" title="http://www.catechizeme.com"/>
<resources base="http://www.catechizeme.com">
<resource path="catechisms/{CATECHISM_NAME}/daily_question.js" id="Daily_question.js">
<doc xml:lang="en" title="Daily_question.js"/>
<param xmlns:xs="http://www.w3.org/2001/XMLSchema" name="CATECHISM_NAME" style="template" type="string"/>
<method name="GET" id="Daily_question.js">
  <doc xml:lang="en" title="Daily_question.js"/>
  <request/>
  <response status="200">
    <representation mediaType="json" element="data"/>
```

```
    <representation mediaType="js; charset=utf-8" element="data"/>
  </response>
</method>
</resource>
</resources>
</application>
```

WADL ползва XML синтаксис за описание на метаданните и наличните действия. Може също да бъде написан точно толкова стриктно колкото WSDL - дефинирайки типове, опционални параметри и т.н.

WADL

WADL не притежава механизъм за репрезентиране на самите данни, които са обект на изпращане към URI. Това означава, че WADL е способен да документира само половината информация, нужна за комуникацията с услугата. Нека вземем за пример CATECHEISM_NAME параметъра в горния пример. WADL указва единствено къде в самия URI принадлежи параметърът, и че трябва да бъде низова стойност. Обаче, ако трябва да извлечете валидните стойности сами, вероятно би ви отнело доста време. Обърнете внимание, че е възможно добавянето на схема към вашия WADL, така че да можете да дефинирате дори комплексни типове променливи, като например изброими такива. Това обаче се практикува по-рядно от предоставянето на WADL.

WADL е напълно опционален, като повечето пъти дори отсъства изцяло. Поради характера на услугата, за да можете да я ползвате смислено почти сигурно е, че ще ви е нужда допълнителна документация.

Критика

Малкият обем данни и широкото разпространение на HTTP стандарта прави REST доста привлекателна опция за публични APIs. Окомплектовани с JSON, който прави промени като добавяне на опционален параметър много прости, става изключително гъвкав и позволява чести обновления без засягане на клиентите.

Най-големият недостатък се явява WADL - опционален и непритежаващ определена нужна информация. Съществуват няколко frameworks адресиращи този проблем, подпомагайки документирането и продуцирането на RESTful APIs, като например:

- Swagger - http://swagger.io
- RAML - http://raml.org
- JSON-home - https://github.com/otto-de/jsonhome

Въпреки това, тъй като няма чист "стандарт", изникват множество различни frameworks. Това би затруднило работата с това API, ако предпочитате чисто дефинирани стандарти. Съществуват дискусии относно най-добри практики в областта, например, "Richardson Maturity Model" на Martin Fowler:

http://martinfowler.com/articles/richardsonMaturityModel.html

http://www.soapui.org/testing-dojo/world-of-api-testing/soap-vs--rest-challenges.html

Java API for XML Web Services (JAX-WS)

JAX-WS представлява технология за изграждане на уеб услуги и клиенти, комуникиращи посредством XML. JAX-WS позволява на разработчиците да пишат message-oriented web services, както и такива базирани на Remote Procedure Calls (RPC-oriented).

В JAX-WS всяка едно извикване на операция бива репрезентирано чрез XML-базиран протокол като SOAP. Спецификацията на SOAP дефинира обгръщащата структура (envelope), правилата за енкодинг и конвенциите за репрезентиране извикванията и отговорите на уеб услугата. Тези извиквания и отговори биват пренасяни посредством SOAP съобщения (XML файлове) по HTTP.

Въпреки че SOAP съобщенията са комплексни, JAX-WS API скрива тази сложност от разработчика на приложението. От страната на сървъра, разработчикът специфицира операциите чрез дефиниране на методи в интерфейс, написан на програмния език Java. Разработчикът също пише един или повече класа, имплементиращи тези методи. Клиентските програми също са лесни за създаване. Даден клиент създава прокси (proxy - локален обект, репрезентиращ услугата), след което просто извиква методите на проксито. Ползвайки JAX-WS разработчикът не генерира или извършва parsing на SOAP съобщенията. Конвертирането на API извиквания и отговори към SOAP съобшения бива отговорност на JAX-WS runtime системата.

Посредством JAX-WS, клиентите и уеб услугите имат голямо преимущество - платформената независимост на програмния език Java. В допълнение, JAX-WS не е рестриктивен - JAX-WS клиент може да достъпи web service, който не върви върху Java платформата, и обратното. Гъвкавостта е възможна защото JAX-WS използва технологии, дефинирани от W3C - HTTP, SOAP и WSDL. WSDL специфицира XML формат, описващ услугата като набор от крайни точки (endpoints), опериращи върху съобщения.

Създаване на прост Web Service и клиенти с JAX-WS

Изображението по-долу илюстрира как JAX-WS технологията управлява комуникацията между дадена уеб услуга и клиент.

Началната точка за разработка на JAX-WS уеб услуга бива Java клас, анотиран с javax.jws.WebService анотацията. @WebService анотацията дефинира този клас като web service endpoint.

Даден service endpoint interface или service endpoint implementation (SEI) представлява Java интерфейс или респективно клас, деклариращ методите които клиентът може да извиква от услугата. Интерфейсът не е задължителен при построяването на JAX-WS endpoint. Имплементиращият клас на уеб услугата имлицитно дефинира SEI.

Можете да специфицирате експлицитно интерфейс чрез добавяне на endpointInterface елемент към @WebService анотацията в имплементиращия клас. Тогава трябва да предоставите интерфейс, който дефинира публичните методи, налични в endpoint имплементиращия клас.

Основните стъпки за създаване на уеб услуга и клиент са както следва:

1. Написване на имплементиращия клас.

2. Компилиране на имплементиращия клас.
3. Пакетиране на файловете в WAR файл.
4. Разгръщане на WAR файла. Артефактите на уеб услугата, ползвани за комуникация с клиенти, се генерират от приложния сървър по време на разгръщане.
5. Нписване на клиентския клас.
6. Употреба на wsimport Ant task за генериране и компилиране на web service artefacts, необходими за свързане с услугата.
7. Компилиране на клиентския клас.
8. Пускане на клиента.

Бележка:

Повечето IDEs, включая NetBeans, извършват задачите по wsimport автоматично.

Изисквания към JAX-WS Endpoint

JAX-WS крайните точки трябва да отговарят на следните изисквания:

- Имплементиращият клас трябва да бъде анотиран с javax.jws.WebService или javax.jws.WebServiceProvider анотацията;
- Имплементиращият клас може експлицитно да реферира даден SEI чрез endpointInterface елемента на @WebService анотацията, но това не е задължително. Ако няма специфициран endpointInterface елемент на @WebService, то SEI бива имплицитно дефиниран за имплементиращия клас;
- Бизнес методите на имплементиращия клас трябва да бъдат public и не могат да бъдат декларирани static или final;
- Изложените към клиентите на уеб услуги бизнес методи трябва да бъдат анотирани с javax.jws.WebMethod;
- Изложените към клиентите на уеб услуги бизнес методи трябва да имат съвместими с JAXB параметри и return types. За повече информация вижте тук: http://docs.oracle.com/javaee/6/tutorial/doc/bnazc.html;
- Имплементиращият клас не трябва да бъде като деклариран final, нито abstract;
- Имплементиращият клас трябва да има публичен конструктор по подразбиране (default public constructor);
- Имплементиращият клас не трябва да дефинира finalize метод;
- Имплементиращият клас може да ползва javax.annotation.PostConstruct или javax.annotation.PreDestroy анотациите върху методите си за lifecycle event callbacks.

@PostConstruct методът бива извикан от контейнера преди имплементиращият клас да изпрати отговор към клиентите на уеб услугата.

@PreDestroy методът бива извикан от контейнера преди текущият endpoint да бъде премахнат от операцията.

Създаване на service endpoint имплементиращ клас

В долния пример имплементиращият клас Hello е анотиран като крайна точка на уеб услуга посредством @WebService анотацията. Hello декларира един метод на име sayHello, анотиран с @WebMethod анотацията, който излага анотирания метод към клиентите на уеб услугата. Методът sayHello връща поздрав към клиента, ползвайки името подадено му за композиране на поздрава. Имплементиращият клас също трябва да дефинира default public конструктор без аргументи.

```
package helloservice.endpoint;
import javax.jws.WebService;
import javax.jws.WebMethod;

@WebService
public class Hello {
   private String message = new String("Hello, ");

   public void Hello() {
   }

   @WebMethod
   public String sayHello(String name) {
      return message + name + ".";
   }
}
```

Изграждане, пакетиране и разгръщане на услуга

Можете да ползвате IDE или Ant за building, packaging и deploy на helloservice приложението. След разгръщане можете да видите WSDL файла чрез зареждане на следния URL в уеб браузъра:

http://localhost:8080/helloservice/HelloService?wsdl

Просто JAX-WS клиентско приложение

HelloAppClient класът представлява самостоятелно (stand-alone) клиентско приложение, достъпващо sayHello метода на HelloService. Извикването бива извърпено чрез порт, локален обект, държащ се като прокси за отдалечената услуга. Портът бива създаден по време на разработка чрез wsimport, което генерира JAX-WS portable артефакти, базирани върху WSDL файл.

Писане на клиентско приложение

След извикване на отдалечените методи от порта, клиентът извършва следните стъпки:

1. Използва генерирания helloservice.endpoint.HelloService класа, репрезентиращ услугата на URI на разгърнатия WSDL файл на услугата:

```
import helloservice.endpoint.HelloService;
import javax.xml.ws.WebServiceRef;

public class HelloAppClient {
   @WebServiceRef(wsdlLocation =
    "META-INF/wsdl/localhost_8080/helloservice/HelloService.wsdl")
```

```
   private static HelloService service;
```

2. Получава прокси към услугата, познато още като порт, чрез извикване на getHelloPort върху услугата:

```
helloservice.endpoint.Hello port = service.getHelloPort();
```

Портът имплементира SEI, дефиниран от услугата.

3. Извиква sayHello метода на порта, подавайки към услугата низ от символи:

```
return port.sayHello(arg0);
```

Пълният изходен код на HelloAppClient:

```
package appclient;


import helloservice.endpoint.HelloService;
import javax.xml.ws.WebServiceRef;


public class HelloAppClient {
   @WebServiceRef(wsdlLocation =
     "META-INF/wsdl/localhost_8080/helloservice/HelloService.wsdl")
   private static HelloService service;


   /**
    * @param args the command line arguments
    */
   public static void main(String[] args) {
     System.out.println(sayHello("world"));
   }


   private static String sayHello(java.lang.String arg0) {
      helloservice.endpoint.Hello port = service.getHelloPort();
      return port.sayHello(arg0);
   }
}
```

Стартиране на клиентското приложение

Можете да ползвате IDE или Ant за building, packaging, deployment и стартиране на appclient приложението. За да направите build на клиента, трябва първо да имате разгърнат helloservice.

Прост JAX-WS уеб клиент

HelloServlet представлява сървлет, който подобно на Java клиента извиква sayHello метода на уеб услугата. Подобно на горното клиентско приложение, текущото прави извикванията чрез порт.

Писане на сървлет

За да извика метода от порта, клиентът извършва следните стъпки:

1. Import на HelloService endpoint и WebServiceRef анотацията:

```
import helloservice.endpoint.HelloService;
```

```
...
import javax.xml.ws.WebServiceRef;
```

2. Дефиниране на референция за уеб услугата, специфицирайки WSDL локация:

```
@WebServiceRef(wsdlLocation =
  "WEB-INF/wsdl/localhost_8080/helloservice/HelloService.wsdl")
```

3. Деклариране на уеб услуга, след което дефиниране на private метод, който извиква sayHello метода от порта:

```
private HelloService service;
...
private String sayHello(java.lang.String arg0) {
    helloservice.endpoint.Hello port = service.getHelloPort();
    return port.sayHello(arg0);
}
```

4. Извикване на private метода в сървлета:

```
out.println("<p>" + sayHello("world") + "</p>");
```

Следва същинската част на HelloServlet кода:

```
package webclient;

import helloservice.endpoint.HelloService;
import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.xml.ws.WebServiceRef;

@WebServlet(name="HelloServlet", urlPatterns={"/HelloServlet"})
public class HelloServlet extends HttpServlet {
    @WebServiceRef(wsdlLocation =
      "WEB-INF/wsdl/localhost_8080/helloservice/HelloService.wsdl")
    private HelloService service;

    /**
     * Processes requests for both HTTP <code>GET</code>
     *   and <code>POST</code> methods.
     * @param request servlet request
     * @param response servlet response
     * @throws ServletException if a servlet-specific error occurs
     * @throws IOException if an I/O error occurs
     */
```

```
    protected void processRequest(HttpServletRequest request,
            HttpServletResponse response)
    throws ServletException, IOException {
        response.setContentType("text/html;charset=UTF-8");
        PrintWriter out = response.getWriter();
        try {

            out.println("<html lang=\"en\">");
            out.println("<head>");
            out.println("<title>Servlet HelloServlet</title>");
            out.println("</head>");
            out.println("<body>");
            out.println("<h1>Servlet HelloServlet at " +
                request.getContextPath () + "</h1>");
            out.println("<p>" + sayHello("world") + "</p>");
            out.println("</body>");
            out.println("</html>");

        } finally {
            out.close();
        }
    }

    // doGet and doPost methods, which call processRequest, and
    //   getServletInfo method

    private String sayHello(java.lang.String arg0) {
        helloservice.endpoint.Hello port = service.getHelloPort();
        return port.sayHello(arg0);
    }
}
```

Стартиране на уеб клиента

Можете да ползвате IDE или Ant за building, пакетиране, разгръщане и стартиране на webclient приложението. За да извърщите build на клиента, трябва първо да разгърнете helloservice.

Поддържани от JAX-WS типове

JAX-WS делегира мапинга (mapping) на типовете от програмния език Java към XML дефиниции на JAXB. Приложният разработчик няма нужда да знае детайли по отношение на тези мапинги, но трябва да знае, че не всеки клас в Java езика може да бъде ползван като параметър на метод или return type в JAX-WS.

Следващата секция обяснява свързването на schema-to-Java и Java-to-schema типове данни по подразбиране.

Schema-to-Java mapping

Езикът Java предоставя по-богат набор от типове данни, отколкото XML схема.

XML schema type	Java data type
xsd:string	java.lang.String
xsd:integer	java.math.BigInteger
xsd:int	int
xsd.long	long
xsd:short	short
xsd:decimal	java.math.BigDecimal
xsd:float	float
xsd:double	double
xsd:boolean	boolean
xsd:byte	byte
xsd:QName	javax.xml.namespace.QName
xsd:dateTime	javax.xml.datatype.XMLGregorianCalendar
xsd:base64Binary	byte[]
xsd:hexBinary	byte[]
xsd:unsignedInt	long
xsd:unsignedShort	int
xsd:unsignedByte	short
xsd:time	javax.xml.datatype.XMLGregorianCalendar
xsd:date	javax.xml.datatype.XMLGregorianCalendar
xsd:g	javax.xml.datatype.XMLGregorianCalendar
xsd:anySimpleType	java.lang.Object
xsd:anySimpleType	java.lang.String
xsd:duration	javax.xml.datatype.Duration

XML schema type	Java data type
xsd:NOTATION	javax.xml.namespace.QName

Java-to-schema mapping

Java class	XML data type
java.lang.String	xs:string
java.math.BigInteger	xs:integer
java.math.BigDecimal	xs:decimal
java.util.Calendar	xs:dateTime
java.util.Date	xs:dateTime
javax.xml.namespace.QName	xs:QName
java.net.URI	xs:string
javax.xml.datatype.XMLGregorianCalendar	xs:anySimpleType
javax.xml.datatype.Duration	xs:duration
java.lang.Object	xs:anyType
java.awt.Image	xs:base64Binary
javax.activation.DataHandler	xs:base64Binary
javax.xml.transform.Source	xs:base64Binary
java.util.UUID	xs:string

http://docs.oracle.com/javaee/6/tutorial/doc/bnayl.html

RESTful Web Services с JAX-RS

Jersey, референтната имплементация на JAX-RS, имплементира поддръжка за анотации, дефинирани в JSR 311, позволявайки на разработчиците лесно да изграждат RESTful уеб услуги ползвайки програмния език Java.

Създаване на RESTful root resource class

Root resource класовете са POJOs, които са или анотирани с @Path, или имат поне един метод, анотиран с @Path или request method designator, като например @GET, @PUT, @POST или @DELETE. Resource методите представляват методи на ресурсния клас, анотирани с request method designator.

Разработка на RESTful уеб услуги с JAX-RS

JAX-RS е API в програмния език Java, направено за улеснение разработката на приложения, ползващи REST архитектура.

JAX-RS API ползва анотации за да опрости разработката на RESTful уеб услуги. Разработчиците декорират Java класовете с JAX-RS анотации, за да дефинират ресурси и действия, които могат да бъдат извършвани с тях. JAX-RS анотациите биват по време на изпълнение, следователно runtime reflection ще генерира спомагателните (helper) класове и артефакти за ресурсите. Даден Java EE архив, съдържащ JAX-RS ресурсни класове, ще има конфигурирани ресурси, генерирани спомагателни класове и артефакти, и изложени ресурси към клиентите - посредством разгръщане на архива върху Java EE сървър.

Долната таблица изброява някои дефинирани от JAX-RS анотации, заедно с кратко обяснение начина им на употреба. За повече информация можете да видите тук: http://docs.oracle.com/javaee/7/api/

Анотация	Описание
@Path	Стойността на @Path анотацията е релативен URI път, индикиращ къде ще бъде разгърнат Java класа - например /helloworld. Можете да влагате и променливи в самите URIs, за да направите URI path template. Например, можете да попитате за името на потребител, след което да го подадете на приложението като променлива в URI: /helloworld/{username}
@GET	@GET анотацията представлява request method designator и кореспондира със съответния HTTP метод. Анотираният с този request method designator Java метод ще обработва HTTP GET заявки. Поведението на даден ресурс е определяно от HTTP метода, на който отговаря даден ресурс.
@POST	@POST анотацията представлява request method designator и кореспондира със съответния HTTP метод. Анотираният с този request method designator Java метод ще обработва HTTP POST заявки. Поведението на даден ресурс е определяно от HTTP метода, на който отговаря даден ресурс.
@PUT	@PUT анотацията представлява request method designator и кореспондира със съответния HTTP метод. Анотираният с този request method designator Java метод ще обработва HTTP PUT заявки. Поведението на даден ресурс е определяно от HTTP метода, на който отговаря даден ресурс.
@DELETE	@DELETE анотацията представлява request method designator и кореспондира със съответния HTTP метод. Анотираният с този request method designator Java метод ще обработва HTTP DELETE заявки. Поведението на даден ресурс е определяно от HTTP метода, на който отговаря даден ресурс.
@HEAD	@HEAD анотацията представлява request method designator и кореспондира със съответния HTTP метод. Анотираният с този request method designator Java метод ще обработва HTTP HEAD заявки. Поведението на даден ресурс е определяно от HTTP метода, на който отговаря даден ресурс.

Анотация	Описание
@PathParam	@PathParam анотацията представлява тип на параметър, който можете да извлечете за употреба във вашия ресурсен клас. URI path параметрите биват извлечени от заявения URI, а имената им кореспондират с имената на URI path template променливите, специфицирани в @Path анотацията на ниво клас.
@QueryParam	@QueryParam анотацията представлява тип на параметър, който можете да извлечете за употреба във вашия ресурсен клас. Query параметрите са извлечени от заявените URI query параметри.
@Consumes	@Consumes анотацията се използва за специфициране на MIME media types изпратени от клиента, които ресурсът може да консумира.
@Produces	@Produces анотацията се използва за специфициране на MIME media types, които ресурсът може да продуцира и изпраща обратно към клиента - например “text/plain”.
@Provider	@Provider анотацията се ползва за всичко, което е от интерес за JAX-RS runtime, например MessageBodyReader и MessageBodyWriter. За HTTP заявките MessageBodyReader се ползва за мапинг между HTTP request entity body и параметри на метод. От страна на отговора, връщаната стойност бива мапната към HTTP response entity body чрез употребата на MessageBodyWriter. Ако приложението има нужда да предостави допълнителни метаданни, като например HTTP headers или различен status code, даден метод може да върне Response, който обгръща (wraps) наличното entity и който може да бъде построен ползвайки Response.ResponseBuilder.

Преглед на JAX-RS приложение

Следният примерен код представлява прост пример за root resource клас, ползващ JAX-RS анотации:

```
package com.sun.jersey.samples.helloworld.resources;
import javax.ws.rs.GET;
import javax.ws.rs.Produces;
import javax.ws.rs.Path;
// The Java class will be hosted at the URI path "/helloworld"
@Path("/helloworld")
public class HelloWorldResource {

    // The Java method will process HTTP GET requests
    @GET
    // The Java method will produce content identified by the MIME Media
    // type "text/plain"
    @Produces("text/plain")
    public String getClichedMessage() {
        // Return some cliched textual content
        return "Hello World";
    }
}
```

- Стойността на @Path анотацията е релативен URI път. В горния пример Java класът може да бъде разгърнат върху URI път /helloworld. Това е много прост пример за употреба на @Path анотацията със статичен URI път. В URIs могат да бъдат влагани променливи. URI path templates представляват URIs с вложени променливи в URI синтаксиса.
- @GET анотацията представлява request method designator, заедно с @POST, @PUT, @DELETE и @HEAD, дефинирани от JAX-RS и кореспондиращи с подобно наименованите HTTP методи. В примера анотираният Java метод ще обработва HTTP GET заявки. Поведението на ресурса е определено от HTTP метода, на който отговаря той.
- @Produces анотацията се ползва за специфициране на MIME media types, които ресурсът може да продуцира и изпрати обратно към клиента. В този пример Java методът ще продуцира репрезентации, идентифицирани от MIME media type "text/plain".
- @Consumes анотацията се ползва за специфициране на MIME media types, които даден ресурс може да консумира като подадени от клиента. Примерът може да бъде модифициран, така че да добавя връщано от getClichedMessage метода съобщение по следния начин:

```
@POST
@Consumes("text/plain")
public void postClichedMessage(String message) {
        // Store the message
}
```

@Path анотацията и URI path templates

@Path анотацията идентифицира URI path template, към който ресурсът отговаря и бива специфицирана на ниво метод на ресурса. Стойността на @Path анотацията е частичен URI path template, релативен за базовия URI на сървъра върху който е разгърнат ресурсът, context root на приложението и URL pattern към който JAX-RS runtime отговаря.

URI path templates са URIs с вложени в URI синтаксиса променливи. Тези променливи биват заместени по време на изпълнение, с цел ресурсът да отговори на request, базиран на заместения URI. Променливите са обозначени с къдрави скоби. Пример:

```
@Path("/users/{username}")
```

В горния пример от потребителя е поискано за въвеждане име, след което отговаря JAX-RS уеб услугата, конфигурирана за този URI path template. Например, ако потребителят въведе име "Galileo", уеб услугата отговаря на следния URL:

http://example.com/users/Galileo

За да получи стойността за потребителско име, @PathParam анотацията може да бъде ползвана върху параметър на request метод, например:

```
@Path("/users/{username}")
public class UserResource {

   @GET
   @Produces("text/xml")
   public String getUser(@PathParam("username") String userName) {
      ...
```

```
    }
}
```

По подразбиране URI променливата трябва да отговаря на regexp “[^/]+?”. Тази променлива може да бъде къстъмизирана чрез специфициране на различен regular expression след името й. Например, ако потребителското име се състои само от малки, големи букви и числа, припокрийте regexp-а по подразбиране в дефиницията на променливата:

```
@Path("users/{username: [a-zA-Z][a-zA-Z_0-9]*}")
```

В горния пример променливата за потребителско име ще съответства (matching) само на потребителски имена, започващи с малка или голяма буква и нула или повече числа и подчертавка. Ако дадено потребителско име не съответства на този template, към клиента ше бъде пратен отговор 404 (Not Found).

Не е задължително стойността на @Path да има предходни или последващи наклонени черти (/). JAX-RS runtime подава URI path templates по същия начин, независимо дали биват предшествани или завършвани от пространства.

Даден URI path template има една или множество променливи, с името на всяка от които оградено с къдрави скоби - { за започване на името и } за края й. В предходния пример username е името на променливата. По време на изпълнение ресурсът конфигуриранда отговаря на предходния URI path template ще се опита да обработи URI данните, кореспондираши с локацията на {username} в посочения URI като данни за променливата username.

Например, ако искате да разгърнете ресурс, кореспондиращ на URI path template

http://example.com/myContextRoot/resources/{name1}/{name2}/

трябва да разгърнете приложението на Java EE сървър, отговарящ на заявки на URI http://example.com/myContextRoot, след което да декорирате вашия ресурс със следната @Path анотация:

```
@Path("/{name1}/{name2}/")
public class SomeResource {
    ...
}
```

В този пример URL pattern за JAX-RS помощния сървлет, специфициран в web.xml, бива по подразбиране:

```
<servlet-mapping>
    <servlet-name>My JAX-RS Resource</servlet-name>
    <url-pattern>/resources/*</url-pattern>
</servlet-mapping>
```

Име на променлива може да бъде ползвано повече от веднъж в рамките на един и същ URI path template.

Ако даден знак в стойността на променливата може да създаде конфликт с резервиран такъв от URI, конфликтният знак ще бъде заместен с процентен енкодинг. Например, празното пространство в стойността на променливата ще бъде заместено с %20.

При дефиниране на URI path templates, внимавайте дали резултиращият от заместването URI ще бъде валиден.

Примерите по-долу демонстрират URI path template променливи и как URIs са оформени след заместване. Следните имена на променливи и стойности биват ползвани в примерите:

- name1: james
- name2: gatz
- name:
- location: Main%20Street
- question: why

Бележка:

Стойността на променлива name2 е празен низ.

URI path template	**URI след заместване**
http://example.com/{name1}/{name2}/	http://example.com/james/gatz/
http://example.com/{question}/{question}/{question}/	http://example.com/why/why/why/
http://example.com/maps/{location}	http://example.com/maps/Main%20Street
http://example.com/{name3}/home/	http://example.com//home/

Отговарящи на HTTP методи и заявки

Поведението на даден ресурс е определено от HTTP методите (обикновено GET, POST, PUT, DELETE, към които ресурсът отговаря.

Request method designator анотации

Request method designator анотациите представляват анотации по време на изпълнение, дефинирани от JAX-RS и кореспондиращи на наименованите по подобен начин HTTP методи. В рамките на ресурсен клас файл, HTTP методите биват мапнати към програмния език Java чрез употребата на request method designator анотации. Поведението на ресурса бива оптеделяно от това на кой HTTP метод отговаря той. JAX-RS дефинира набор от request method designators за обичайните HTTP методи @GET, @POST, @PUT, @DELETE и @HEAD. Можете също да създавате ваши собствени request method designators.

Следният пример (изваден от storage service sample) демонстрира употребата на PUT метода за създаване и обновяване на storage контейнер:

```
@PUT
public Response putContainer() {
    System.out.println("PUT CONTAINER " + container);

    URI uri =  uriInfo.getAbsolutePath();
    Container c = new Container(container, uri.toString());
```

```
    Response r;
    if (!MemoryStore.MS.hasContainer(c)) {
        r = Response.created(uri).build();
    } else {
        r = Response.noContent().build();
    }

    MemoryStore.MS.createContainer(c);
    return r;
}
```

По подразбиране JAX-RS runtime автоматично ще поддържа методите HEAD и OPTIONS, ако те не бъдат експлицитно имплементирани. За HEAD, наличният runtime ще извика имплементирания GET метод ако има такъв, и ще игнорира отговора, ако бъде върнат такъв. В допълнение, JAX-RS runtime ще върне WADL документ описващ ресурса.

Декорирани с request method designators методи трябва да връщат void, тип на програмния език Java или javax.ws.rs.core.Response обект. Множество параметри могат да бъдат извлечени от URI чрез употребата на @PathParam или @QueryParam anotaciite. Конвертиране между Java tipowe i ентитъ бодъ бива отговорност на entity provider, например MessageBodyReader или MessageBodyWriter. Методи, който трябва да предоставят допълнителни метаданни в отговор, трябва да връщат инстанция на Response класа. ResponseBuilder класът предоставя удобен начин за създаване инстанция на Response, ползвайки builder pattern. PUT и POST HTTP методите очакват налично HTTP request body, затова можете да ползвате MessageBodyReader за методи, отговарящи на PUT и POST заявки.

И двете @PUT и @POST могат да се ползват за създаването или обновяването на ресурс. POST може да означава всичко, затова при употребата му самото приложение определя семантиката. PUT има добре дефинирана семантика. При употребата на PUT за създаване, клиентът декларира URI за новосъздадения ресурс.

PUT има доста чиста семантика за създаване и обновяване на ресурс. Заявката от клиента трябва да бъде същата репрезентация, получена при употребата на GET, ако media type бива същият. PUT не позволява даден ресурс да бъде частично обновен, честа грешка при опит за употреба на PUT метод. Разпространен application pattern бва употребата на POST за създаване на ресурс и връщането на отговор 201 с location header, чиято стойност бива URI на новосъздадения ресурс. При този pattern уеб услугата декларира URI за новосъздадения ресурс.

Употреба на entity providers за мапинг на HTTP response и request entity bodies

Entity providers предоставят мапинг услуги между репрезентациите и техните асоциирани Java типове. Твата типа entity providers са MessageBodyReader и MessageBodyWriter. За HTTP заявки, MessageBodyReader се ползва за мапинг на HTTP request entity body към параметри на метод. От страна на отговора, return стойността е мапната към HTTP response entity body чрез употребата на MessageBodyWriter. Ако приложението има нужда да предоставя допълнителни метаданни като например, HTTP headers или различен status code, методът може да върне Response, който обгръща (wraps) наличното и entity и който може да бъде построен ползвайки Response.ResponseBuilder.

Долната таблица демонстрира стандартните типове, поддържани автоматично за HTTP request и response entity bodies. Необходимо е да пишете entity provider само ако не изберете измежду стандартните типове.

Java тип	Поддържан Media тип
byte[]	Всички media типове (*/*)
java.lang.String	Всички текстови media типове (text/*)
java.io.InputStream	Всички media типове (*/*)
java.io.Reader	Всички media типове (*/*)
java.io.File	Всички media типове (*/*)
javax.activation.DataSource	Всички media типове (*/*)
javax.xml.transform.Source	XML media типове (text/xml, application/xml и application*/+xml)
javax.xml.bind.JAXBElement и предоставените от приложението JAXB класове	XML media типове (text/xml, application/xml и application*/+xml)
MultivaluedMap<String, String>	Съдържание на форма (application/x-www-form-urlencoded)
StreamingOutput	Всички media типове (*/*), само за MessageBodyWriter

Следният пример демонстрира употребата на MessageBodyReader с @Consumes и @Provider анотациите:

```
@Consumes("application/x-www-form-urlencoded")

@Provider

public class FormReader implements MessageBodyReader<NameValuePair> {
```

Долният пример демонстрира употребата на MessageBodyWriter с @Produces и @Provider анотациите:

```
@Produces("text/html")

@Provider

public class FormWriter implements

    MessageBodyWriter<Hashtable<String, String>> {
```

Следният пример демонстрира употребата на ResponseBuilder:

```
@GET

public Response getItem() {

  System.out.println("GET ITEM " + container + " " + item);


  Item i = MemoryStore.MS.getItem(container, item);

  if (i == null)

    throw new NotFoundException("Item not found");

  Date lastModified = i.getLastModified().getTime();

  EntityTag et = new EntityTag(i.getDigest());

  ResponseBuilder rb = request.evaluatePreconditions(lastModified, et);

  if (rb != null)
```

```
        return rb.build();

    byte[] b = MemoryStore.MS.getItemData(container, item);
    return Response.ok(b, i.getMimeType()).
            lastModified(lastModified).tag(et).build();
}
```

Употреба на @Consumes и @Produces за къстъмизиране на заявки и отговори

Изпратената към ресурс и в последствие върнатат към клиента информация бива специфицирана като MIME media type в headers на HTTP заявка или отговор. Ползвайки следните анотации можете да укажете с кои MIME media types на репрезентации даден ресурс може да отговори или да продуцира:

- javax.ws.rs.Consumes
- javax.ws.rs.Procudes

По подразбиране ресурсният клас може да отговори или да продуцира всички MIME media types на репрезентации, специфицирани в HTTP request и response headers.

@Produces анотацията

@Produces анотацията се ползва за специфициране на MIME media types или репречентации, които даден ресурс може да продуцира или изпрати обратно на клиента. Ако @Produces бива приложена на ниво клас, то всички методи в ресурса трябва да продуцират специфицираните MIME types по подразбиране. Ако бъде приложена на ниво метод, анотацията презаписва всички @Produces анотации, приложени на ниво клас.

Ако в даден ресурс няма методи, способни да продуцират MILE type в клиентска заявка, JAX-RS runtime изпраща обратно HTTP “406 Not Acceptable” грешка.

Стойността на @Produces представлява масив от String, а именно MIME types. Пример:

```
@Produces({"image/jpeg,image/png"})
```

Следният пример демонстрира приложението на @Produces както на ниво клас, така и на ниво метод:

```
@Path("/myResource")
@Produces("text/plain")
public class SomeResource {
    @GET
    public String doGetAsPlainText() {
        ...
    }

    @GET
    @Produces("text/html")
    public String doGetAsHtml() {
        ...
    }
}
```

По подразбиране doGetAsPlainText методът има MIME media type от @Produces анотацията на ниво клас. Анотацията @Produces на doGetAsHtml метода припокрива тази на ниво клас, специфицирайки че методът продуцира HTML.

Ако ресурсният клас е способен да продуцира повече от един MIME media type, избраният ресурсен метод ще кореспондира с най-приложимия медиа тъпе, деклариран от клиента. По-конкретно, Accept хедъра на HTTP request декларира кое е най-приемливото. Например, ако Accept хедърът бива "Accept: text/plain", то ще бъде извикан doGetAsPlainText метода. Алтернативно, ако Accept хедърът бива "Accept: text/plain;q=0.9, text/html", което декларира че клиентът може да приема media types от вид text/plain и text/html, но предпочита последното, ще бъде извикан doGetAsHtml метода.

В декларацията на @Produces могат да бъдат декларирани повече от един media type. Следният примерен код демонстрира това:

```
@Produces({"application/xml", "application/json"})
public String doGetAsXmlOrJson() {
   ...
}
```

Методът doGetAsXmlOrJson ще бъде извикан ако единият от media types application/xml и application/json е приемлив. Ако и двата са еднакво приемливи, първият ще бъде избран. Горният пример реферира експлицитно MIME media types за яснота. Възможно е да рефериране константни стойности, които могат да редуцират правописните грешки. За повече информация вижте тук: http://jsr311.java.net/nonav/releases/1.0/javax/ws/rs/core/MediaType.html

@Consumes анотацията

@Consumes анотацията се ползва за специфициране на това кои MIME media types на репрезентациите даден ресурс може да приема или консумира от клиента. Ако @Consumes е приложена на ниво клас, всички отговарящи методи приемат специфицираните MIME типове по подразбиране. Ако бъде приложена на ниво метод, @Consumes припокрива анотацията на ниво клас.

Ако ресурсът не може да консумира MIME типът от клиентската заявка, JAX-RS runtime изпраща обранто HTTP 15 ("Unsupported Media Type") грешка.

Стойността на @Consumes бива масив от String стойности за MIME types. Пример:

```
@Consumes({"text/plain,text/html"})
```

Следният пример показва приложението на @Consumes, както на ниво клас, така и на ниво метод:

```
@Path("/myResource")
@Consumes("multipart/related")
public class SomeResource {
   @POST
   public String doPost(MimeMultipart mimeMultipartData) {
      ...
   }
   @POST
   @Consumes("application/x-www-form-urlencoded")
   public String doPost2(FormURLEncodedProperties formData) {
      ...
   }
```

```
}
```

MIME media типът по подразбиране за doPost метода е този от @Consumes анотацията на ниво клас. doPost2 методът припокрива анотацията на ниво клас, специфицирайки URL-encoded form data.

Ако няма ресурсни методи, които могат да отговорят на заявения MIME type, то HTTP 415 (“Unsupported Media Type”) грешка ще бъде върната към клиента.

HelloWorld примерът може да бъде модифициран да изпраща съобщение чрез употребата на @Consumes, демонстрирано чрез следния примерен код:

```
@POST
@Consumes("text/plain")
public void postClichedMessage(String message) {
   // Store the message
}
```

В горния пример Java методът ще консумира репрезентациите, идентифицирани от MIME media type text/plain. Обърнете внимание, че ресурсният метод връща void. Това означава, че няма връщана репрезентация, и че ще бъде върнат отговор със status code HTTP 204 (“No Content”).

Извличане на request параметри

Параметрите на ресурсен метод могат да бъдат анотирани с параметърно-базирани анотации, с цел извличането на информация от заявката. В предишен пример бе демонстрирана употребата на @PathParam параметръра от path компонента на заявения URL, отговарящ на декларирания път в @Path.

Във вашия ресурсен клас можете да извлечете следните типове параметри за употреба:

- Query
- URI path
- Form
- Cookie
- Header
- Matrix

Query параметрите биват извличани от заявените URI query параметри, и биват специфицирани посредством javax.ws.rs.QueryParam анотацията в аргументите на метода. Следният пример от sparklines примерното приложение демонстрира употребата на @QueryParam, за целите на извличане на query параметри от Query компонента на заявения URL:

```
@Path("smooth")
@GET
public Response smooth(
        @DefaultValue("2") @QueryParam("step") int step,
        @DefaultValue("true") @QueryParam("min-m") boolean hasMin,
        @DefaultValue("true") @QueryParam("max-m") boolean hasMax,
        @DefaultValue("true") @QueryParam("last-m") boolean hasLast,
        @DefaultValue("blue") @QueryParam("min-color") ColorParam minColor,
```

```
    @DefaultValue("green") @QueryParam("max-color") ColorParam maxColor,
    @DefaultValue("red") @QueryParam("last-color") ColorParam lastColor
    ) { ... }
```

Ако query параметъра step съществува в query компонента на заявения URI, стойността на step ще бъде извлечена и парсната (parsed) като 32-bit signed integer, и присвоена на step метод параметъра. Ако step не съществува, стойността по подразбиране 2, декларирана в @DefaultValue анотацията, ще бъде присвоена на step метод параметъра. Ако step стойността не може да бъде парсната като 32-bit signed integer, то бива върната HTTP 400 (“Client Error”) грешка.

Дефинираните от потребителя типове от програмния език Java могат да бъдат използвани като query параметри. Следният примерен код демонстрира ColorParam класа, ползван в предния пример за query параметър:

```
public class ColorParam extends Color {
  public ColorParam(String s) {
    super(getRGB(s));
  }

  private static int getRGB(String s) {
    if (s.charAt(0) == '#') {
      try {
        Color c = Color.decode("0x" + s.substring(1));
        return c.getRGB();
      } catch (NumberFormatException e) {
        throw new WebApplicationException(400);
      }
    } else {
      try {
        Field f = Color.class.getField(s);
        return ((Color)f.get(null)).getRGB();
      } catch (Exception e) {
        throw new WebApplicationException(400);
      }
    }
  }
}
```

Конструкторът за ColorParam приема един низов параметър.

Както @QueryParam, така и @PathParam могат да бъдат ползвани само върху следните Java типове:

- Всички примитивни типове освен char
- Всички wrapper класове на примитивните типове освен Character
- Всеки клас с конструктор, приемащ един низов аргумент
- Всеки клас със статичен метод на име valueOf(String), който приема един низов аргумент

- List<T>, Set<T> или SortedSet<T>, където T съответства на вече изброен критерий. Понякога параметрьте могат да съдържат повече от една стойности със същото име. В такъв случай тези типове могат да бъдат ползвани за получаването на всички стойности.

Ако @DefaultValue не се ползва заедно с @QueryParam и query параметърът не е наличен в заявката, стойността ще бъде празна колекция за List, Set или SortedSet, null за други обектни типове и примитивните стойности по подразбиране.

URI path параметрите биват извлизани от заявения URI и имената на параметрите кореспондират с имената в URI path template, специфицирани в @Path анотацията на ниво клас. URI параметрите са специфицирани посредством javax.ws.rs.PathParam анотацията в аргументите на метода. Следният примерен код демонстрира употребата на @Path променливи и @PathParam анотацията в метод:

```
@Path("/{username}")
public class MyResourceBean {
   ...
   @GET
   public String printUsername(@PathParam("username") String userId) {
      ...
   }
}
```

В горния пример URI path template променливата username е специфицирана като параметър на printUsername метода. Анотацията @PathParam е приложена върху променливата с име username. По време на изпълнение, преди извикването на printUsername, стойността на username бива извлечена от наличния URI и casted към String. Резултиращия низ бива достъпен за метода посредством userId променливата.

Ако дадената URI path template променлива не може да бъде кастната към специфицния тип, тогава JAX-RS runtime връща към клиента грешка HTTP 400 (“Bad Request”). Ако @PathParam анотацията не може да бъде кастната към специфицния тип, JAX-RS runtime връща към клиента грешка HTTP 404 (“Not Found”).

@PathParam параметъра и другите параметър-базирани анотации (@MatrixPara, @HeaderParam, @CookieParam и @FormParam) се подчиняват на същите правила като @QueryParam.

Cookie параметри, индикирани чрез декорация на параметър с javax.ws.rs.CookieParam, извличат информация от cookies, декларирани в cookie-related HTTP headers. Header параметри, индикирани посредством декориран параметър с javax.ws.rs.HeaderParam, извличат информация от HTTP headers. Matrix параметри, индикирани чрез декориране на параметър с javax.ws.rs.MatrixParam извличат информация от сегменти на URL пътя.

Form параметри, индикирани чрез декорация на параметър с javax.ws.rs.FormParam, извличат информация от request репречентация с MIME media type application/x-www-form-urlencoded и съответстват на енкодинга, специфициран от HTML форма, по начин описан тук:

http://www.w3.org/TR/html401/interact/forms.html#h-17.13.4.1

Този параметър е много полезен за извличане на информация, изпратена от POST в HTML форми.

Следният пример извлиза name параметъра на формата от POST form данните:

```
@POST
@Consumes("application/x-www-form-urlencoded")
```

```
public void post(@FormParam("name") String name) {
   // Store the message
}
```

За да получите map с имена и стойности на параметри за query и path параметри, ползвайте следния код:

```
@GET
public String get(@Context UriInfo ui) {
   MultivaluedMap<String, String> queryParams = ui.getQueryParameters();
   MultivaluedMap<String, String> pathParams = ui.getPathParameters();
}
```

Следният метод извлича имена и стойности на header и cookie параметри в map:

```
@GET
public String get(@Context HttpHeaders hh) {
   MultivaluedMap<String, String> headerParams = hh.getRequestHeaders();
   Map<String, Cookie> pathParams = hh.getCookies();
}
```

Обобщено, @Context може да бъде ползвана за да получим контекстните Java типове, свързани с request или response.

За form параметри бива възможно следното:

```
@POST
@Consumes("application/x-www-form-urlencoded")
public void post(MultivaluedMap<String, String> formParams) {
   // Store the message
}
```

http://docs.oracle.com/javaee/6/tutorial/doc/giepu.html

Spring Framework

Spring Framework представлява open source application framework и

inversion of control (IoC, дизайн при който ръчно написани порции получават контрол върху потока от обща преизползваща се библиотека) контейнер за Java платформата. Основните функции на този framework могат да бъдат ползвани от всяко Java приложение, но има разширения за изграждане на уеб приложения върху Java EE платформата. Въпреки че Spring Framework не налага специфичен програмен модел, той е доста популярен в Java обществото като алтернативен заместител и дори добавка към Enterprise JavaBean (EJB) модела.

http://en.wikipedia.org/wiki/Spring_Framework

http://en.wikipedia.org/wiki/Inversion_of_control

Spring ръководства:

http://spring.io/guides