Programming Paradigms
159.272

Advanced Java Features

Amjed Tahir
a.tahir@massey.ac.nz

Original author: Jens Dietrich

This section provide a brief overview over some more advanced Java features. This features are then covered in detail in other papers, such as:

159.251 Software Design and Construction
159.355 Concurrent Systems
159.352 Web Application Development
159.707 Advanced Software Design and Construction

Readings

1. Java tutorial, concurrency trail:
   http://docs.oracle.com/javase/tutorial/essential/concurrency/
2. Java Tutorial, reflection trail:
   http://docs.oracle.com/javase/tutorial/reflect/
3. Java tutorial, annotation lesson
   http://docs.oracle.com/javase/tutorial/java/annotations/

Overview

1. threading
2. reflection
3. annotation types
4. miscellaneous

Threads

- modern (multicore) processors and operating systems have built-in support for parallel processing
- modern programming languages need features to take advantage of this
- this can be used to speed up applications
- for this purpose, Java supports **threads**
- a thread is a sequence of programming statements
- multiple threads can be executed in parallel (at the same time)
- if Hardware and OS support this, multiple threads can be executed on multiple cores - leading to significant speedup

Example: Generating Large Prime Numbers

- write a program that generates large prime numbers
- large prime numbers are used in public key cryptography
  - **prime number:** is a natural number greater than 1 that cannot be formed by multiplying two smaller natural numbers.
- note that the example presented here is simplified - much better algorithms exist
- real world applications can use BigInteger.nextProbablePrime()
- Code:

Generating Random Prime Numbers

```
public class RandomPrimeNumberGenerator {
	private static Random random = new Random();
	public static int getNextRandomPrime(int max) {
		int n = -1;
		while (n==-1||!isPrime(n)) {
			n = random.nextInt(max);
		}
		return n;
	}

	private static boolean isPrime(int n) {
		for(int i=2; i < n ; i++){
			if (n % i == 0) return false;
		}
		return true;
	}}
```

keep on creating new numbers until a prime number is found

check whether a number is prime

Generate 10 Large Prime Numbers

```
List<Integer> primeNumbers = new ArrayList<Integer>();
for (int i=0;i<10;i++) {
	int max = 1_000_000_000;
	int nextPrimeNumber =
	RandomPrimeNumberGenerator.getNextRandomPrime(max);
}
```

running this took 20,552ms on Jens' MacBook with a 2-core Intel processor

running this took 12,963ms on Amjed' MacBook with a 4-core Intel processor (Eclipse Xmx=4096m)

Using Threads

- to use threads, the `Runnable` interface must be implemented
- `Runnable` defines a single method `run()`
- then a new thread can be created using `new Thread(Runnable)`, and then started with `start()`

Worker Implementation

```
public class GeneratePrimeNumbersWorker implements Runnable {
        private List<Integer> resultList = null;
        public GeneratePrimeNumbersWorker(List<Integer> resultList) {
                super();
                this.resultList = resultList;
        }

        @Override
        public void run() {
                while (resultList.size() < 10) {
                        int nextRandom = RandomPrimeNumberGenerator.
                getNextRandomPrime(1_000_000_000);
                resultList.add(nextRandom);
                }
                System.out.println("Result count is " + resultList.size());
}}
```

Starting Some Workers

```
List<Integer> primeNumbers = new Vector<Integer>();

int WORKER_COUNT = 2;
for (int i=0;i<WORKER_COUNT;i++) {
        GeneratePrimeNumbersWorker worker =
                new GeneratePrimeNumbersWorker(primeNumbers);
                new Thread(worker).start();
}
```

this took only **12,488ms** (instead of 20,552ms) on Jens' Mac Book with a 2-core Intel processor !

this took only **7,192ms** (instead of 13, 889ms) on Amjed' Mac Book with a 4-core Intel processor !

Synchronising Threads

- but something is wrong: when a thread computes the 10th prime, the other threads keep on going and finish computing another prime
- results: we have more primes than needed!
- solution: use another **guard condition** when results are added

Improved Worker Implementation

```
public class GeneratePrimeNumbersWorker implements Runnable {
        private List<Integer> resultList = null;
        public GeneratePrimeNumbersWorker(List<Integer> resultList) {
                super();
                this.resultList = resultList;
        }

        @Override
        public void run() {
                while (resultList.size() < 10) {
                        int nextRandom = RandomPrimeNumberGenerator.
                getNextRandomPrime(1000000000);
                        if (resultList.size() < Settings.COUNT) {
                                resultList.add(nextRandom);
                        }
                }
                System.out.println("Result count is " + resultList.size());
}}
```

perhaps another thread has already computed enough prime numbers !

Synchronising Threads ctd

- while this ensures that only 10 numbers end up in the list, the other threads still waste resources to create one more number, it is just not added!
- this is more difficult to control (e.g., by using an object indicating whether the computation has been cancelled that is checked within the workers)

The Synchronisation Problem

- this points to a bigger problem: ***synchronisation***
- it is possible that different threads concurrently manipulate the state of one object (example: resultList)
- this can lead to unexpected side effects, and inconsistent object state
- often it is required that a thread has exclusive access to an object to manipulate this

- this is similar to transactions in banking: when making a transfer, both the **deposit** and the **withdrawal** have to run in a transaction (all or nothing), and the account has to be locked while a transaction is performed

Locking Objects

- for this reason, threads can **lock** (and unlock) objects
- i.e., they can gain exclusive access to objects by locking them
    - Similar to Databases!
- the easiest way to do this is to declare methods with the **`synchronized`** keyword: when a thread executes such a method, no other thread can execute any synchronized method on the same objects

- more advanced concurrency features exist (synchronized code blocks, semaphores etc), this is outside the scope of this paper!

Locking Objects ctd

- some data structures are **thread-safe**, others are not
- sometimes, parallel implementations of similar data types exist (e.g. `java.util.Vector` is a thread safe list implementation, `java.util.ArrayList` is similar but not thread safe)
- Only one threat can call methods in `Vector`
- synchronising itself is expensive
  - e.g., `ArrayList` is slightly faster than `Vector`
- `java.util.Collections` provides several static utility methods to wrap data structures like lists, sets and maps with synchronised "wrappers"
- functional programming techniques are considered to be more suitable for parallel programming as they don't require synchronisation of access to the state of objects

Reflection

- **reflection** is the ability of a program to inspect and modify itself

- It makes it possible to inspects classes, fields, methods at runtime without knowing the actual names!

- reflection is important for generic programming: writing general-purpose software that solves common problems without depending on specific properties of software
- examples:
  - store parts of the object graph
  - build user interfaces dynamically with forms suitable to edit arbitrary objects
  - Running programs in different OS/environments

Classes as Objects

- every object supports the method `getClass()`
- this method returns an instance of `java.lang.Class`
- i.e., classes can be represented as objects !
- `Class` has methods to retrieve information such as:
  - the super classes
  - implemented interfaces
  - the package
- Classes can be dynamically instantiated using the `newInstance()` method in `Class`

Members as Objects

- `Class` also supports several methods that can be used to retrieve members
- use:
  - `getConstructor` - to access constructors
  - `getMethod` - to access constructors
  - `getField` - to access constructors
- these objects instantiate the classes `Constructor`, `Method`, `Field`, respectively in the package `java.lang.reflect`
- methods can be dynamically executed, and fields can be dynamically accessed !

Executing a Method Dynamically

```
Student student = new Student();

Class clazz = student.getClass();
Method method = clazz.getMethod("getName",new Class[]{});
Object returnValue = method.invoke(object,new Object[]{});
```

this is an array of parameter types - getName() has no parameters !

This is the receiver of the method (name of the class)

method parameters

- note that this is the same as invoking `student.getName()`
- but the code works for arbitrary objects that have a `getName()` method!

Java Beans

- the **JavaBeans framework** uses reflection to dynamically build user interfaces
- in particular, the JavaBeans framework can inspect properties of objects (combinations of getters and setters)
- this is used by tools like user interface builders to generate editors for components dynamically
- this can significantly reduce the expenses of software systems
- example: `XMLDecoder` and `XMLEncoder` (in `java.beans`) utilities to save and store object graphs
- a new generation of tools that are based on "convention over configuration" rely on reflection

Reflection Based Persistency: Save

```
Student s = new Student();
s.setName("Taylor");
s.setFirstName("Tom");
OutputStream out =
   new FileOutputStream("student.xml");
XMLEncoder encoder =
   new XMLEncoder(out);
encoder.writeObject(s);
encoder.close();
```

Source: here

Reflection Based Persistency: Load

```
FileInputStream in =
   new FileInputStream("student.xml")
XMLDecoder decoder = new XMLDecoder(in);
Student student =
   (Student)decoder.readObject();
decoder.close();
```

source:
Here

Reflection Based Persistency File Format

```
<?xml version="1.0" encoding="UTF-8"?>
<java version="1.6.0_43" class="java.beans.XMLDecoder">
 <object class="reflection.Student">
  <void property="dob">
   <object class="java.util.Date">
    <long>1365998942374</long>
   </object>
  </void>
  <void property="firstName">
      <string>Tom</string>
  </void>
  <void property="name">
   <string>Taylor</string>
  </void>
 </object> </java>
```

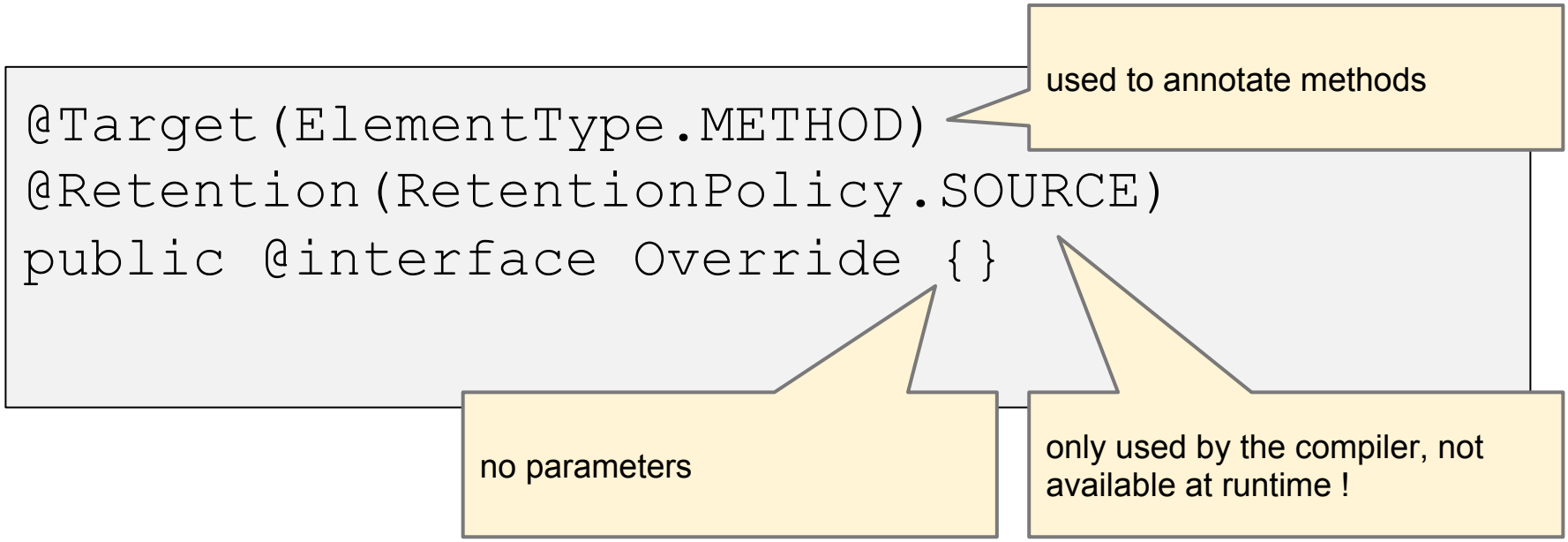
type: use Class.newInstance () when file is read to dynamically instantiate the class using this name

property: use dynamic access to getter to read, and to setter to write

Annotations

- annotations are meta information that are not part of the program itself, but provide useful information to the compiler, the JVM or other tools
- classes, fields, constructors, variables, parameter, methods and annotation types themselves can be annotated
- annotations can have parameters
- annotations represent an element of declarative programming in Java

Annotation Example 1
java.lang.Override - Definition



Annotation Example 1
java.lang.Override - Usage

```
@Override
public void foo() {
      ..
}
```

Annotation Example 2
org.junit.Test - Definition (simplified)

```
@Target({ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)

public @interface Test {
    Class<? extends Throwable> expected();
    long timeout();
}
```

used to annotate methods

used at runtime (by the junit test runner)

two parameters

Annotation Example 1
java.lang.Override - Usage

```
@Test
public void testSomething1() {..}

@Test(timeout=1000)
public void testSomething2() {..}

@Test(expected=NullPointerException.class)
public void testSomething3() {..}
```

simple test with asserts

simple test that fails if it lasts more than 1000 ms

test that is expected to throw a NullPointerException

Processing Annotations

- it is easy to write custom annotations
- it is more difficult to write code that processes them
- annotation processors can be implemented to process annotations
- for runtime annotations, it is possible to use the reflection API for processing

Remote Objects

- Java programs can access objects across the networks as they were local objects
- these objects can even be objects written in another programming language
- there are two technologies available to achieve this:
  - remote method invocation (RMI) - a Java specific technology
  - common object request broker architecture (CORBA) - a cross-language technology

Java Web Technologies: Client

- **Java Applets** are user interface components that can be embedded into web pages
- **JavaFX** is a new technology for flash-like animations
- **Java Web Start** is a client that can be used to distribute and update Java desktop applications over the web
- the **Google Web Toolkit** (GWT) is a library to use Java to program interactive web pages
- this code is then translated by the GWT compiler into JavaScript

Java Web Technologies: Server

- **Java Servlets** is a technology used to write web server scripts (to generate web pages from requests)
- **Java Server Pages** (JSP) are based on servlets but document driven - i.e., Java code is embedded into (HTML) documents
- there are multiple technologies used for enterprise level server-side programming, such as
  - **Enterprise Java Beans** (EJB) with support for transactional manipulation of objects
  - **Java Database Connectivity** (JDBC) with support for relational database
  - **Object-relational mapping software** (ORM) like Hibernate is often used on top of JDBC