What's new in Java land

Michael Voelske
michael.voelske@uni-weimar.de

Bauhaus-Universitaet Weimar
Faculty of Media

August 7, 2012

Outline

Outline

Java SE 7

- Last release (Java SE 6) from December 2006
- Java SE 7 initially released in July 2011
- Many changes to language and platform
  - Core language changes ("Project Coin")
  - Additions to standard library
  - Some language features ("Project Lambda") left for Java SE 8

Outline

Eclipse Juno

- Version 4.2 (Juno) released June 27, 2012
- New features
  - Lua Development Tools
  - Code Recommenders
  - Xtend programming language
- Enhancements
  - UI Overhaul
  - JDT support for Java 7
  - Git integration

Outline

Redundancy in generic constructor calls

Problem: type parameter must be specified twice

```
1  // construct two lists with type parameter
2  List<String> stringlist = new ArrayList<String>();
3  List<Integer> intlist   = new ArrayList<Integer>();
```

Redundancy in generic constructor calls

Problem: type parameter must be specified twice

```
1  // construct two lists with type parameter
2  List<String> stringlist = new ArrayList<String>();
3  List<Integer> intlist   = new ArrayList<Integer>();
```

Type parameters can be omitted

```
4  // using the "raw type" constructor
5  intlist = new ArrayList();
```

Redundancy in generic constructor calls

Problem: type parameter must be specified twice

```
1   // construct two lists with type parameter
2   List<String> stringlist = new ArrayList<String>();
3   List<Integer> intlist   = new ArrayList<Integer>();
```

Type parameters can be omitted

```
4   // using the "raw type" constructor
5   intlist = new ArrayList();
```

But we lose compile-time checks

```
6   stringlist.add("foo");

8   // Copy constructor; legal at compile time:
9   intlist = new ArrayList(stringlist);

11  // ClassCastException at *run* time:
12  Integer item = intlist.get(0);
```

Enter the "diamond" operator

Diamond operator infers generic type parameters automatically

```
1  List<String> stringlist = new ArrayList <> ();
```

Enter the "diamond" operator

Diamond operator infers generic type parameters automatically

```
1  List<String> stringlist = new ArrayList <> ();
```

Retains compile-time checks

```
2  // This is a type mismatch error at *compile* time
3  List<Integer> intlist = new ArrayList <> (stringlist);
```

Outline

Strings in switch statements

Alternatives based on string values in Java 1.6

```
1  String hello(String string) {
2    if ("Alice".equals(string)) {
3      return "Hello, Alice!";
4    } else if ("Bob".equals(string)) {
5      return "Hello, Bob!";
6    } else {
7      return "Hello, Stranger!";
8    }
9  }
```

Strings in switch statements (2)

Cleaner syntax in Java 1.7

```
1  switch (string) {
2    case "Alice":
3    return "Hello, Alice!";
4    case "Bob":
5    return "Hello, Bob!";
6    default:
7    return "Hello, Stranger!";
8  }
```

Outline

New syntax for integer literals

```
1   // "0b"-prefix for base-2 integer literals
2   int four = 0b100;
3
4   // underscores for digit grouping
5   int billion = 1_000_000_000;
6   // anywhere *within* the digits
7   int million = 1_0_0_0___000;
8
9   // can be combined
10  int fortytwo = 0b0010_1010;
```

Outline

Automatically closing resources

Problem: clean up opened resources in the presence of exceptions

```
1  int firstChar(File f) throws IOException {
2    FileReader fr = new FileReader(f);
3    int c = fr.read() // may cause resource leak
4    fr.close()
5    return c;
6  }
```

Automatically closing resources

Problem: clean up opened resources in the presence of exceptions

```
1  int firstChar(File f) throws IOException {
2    FileReader fr = new FileReader(f);
3    int c = fr.read() // may cause resource leak
4    fr.close()
5    return c;
6  }
```

```
1  int firstChar(File f) throws IOException {
2    FileReader fr = null;
3    try {
4      fr = new FileReader(f);
5      return fr.read();
6    } finally {
7      if(fr != null) fr.close();
8    }
9  }
```

Try-with-resources and AutoCloseable

New syntax in Java 1.7

```
1  int firstChar(File f) throws IOException {
2    try (FileReader fr = new FileReader(f)) {
3      return fr.read();
4    }
5  }
```

Try-with-resources and AutoCloseable

New syntax in Java 1.7

```
1  int firstChar(File f) throws IOException {
2    try (FileReader fr = new FileReader(f)) {
3      return fr.read();
4    }
5  }
```

```
1  public interface AutoCloseable {
2    void close() throws Exception;
3  }
```

Multi-catch blocks

Several exceptions can be handled in one catch block

```
1   try (FileReader fr = new FileReader(file)) {
2     Thread.sleep(1);
3     return fr.read();
4   } catch (FileNotFoundException e) {
5     // handle error
6     System.err.println("File not found!");
7   } catch (IOException | InterruptedException e) {
8     // ignore
9     e.printStackTrace();
10  }
```

Outline

New file system API: Paths

- `java.nio.file` package supplements `java.io.File` class
- Motivation: better error handling, consistency across platforms and file systems

```
1  // before 1.7:
2  File file = new File("data/tmp/test.txt");
3  // now:
4  Path path = Paths.get("data","tmp","test.txt");
```

New file system API: Paths

- `java.nio.file` package supplements `java.io.File` class
- Motivation: better error handling, consistency across platforms and file systems

```
1  // before 1.7:
2  File file = new File("data/tmp/test.txt");
3  // now:
4  Path path = Paths.get("data","tmp","test.txt");
```

```
1  if(!file.delete()) {
2    // why didn't it work?...
3  }
```

New file system API: Paths

- `java.nio.file` package supplements `java.io.File` class
- Motivation: better error handling, consistency across platforms and file systems

```
1  // before 1.7:
2  File file = new File("data/tmp/test.txt");
3  // now:
4  Path path = Paths.get("data","tmp","test.txt");
```

```
1  if(!file.delete()) {
2    // why didn't it work?...
3  }
```

```
1  try {
2    path.delete()
3  } catch(FileNotFoundException e) {
4    // handle error
5  }
```

New file system API: WatchService

`WatchService` monitors file system changes

Outline

Asynchronous channel API

- Package `java.nio.channels`
- Non-blocking, asynchronous transfer of byte streams over network sockets or files
- `AsynchronousChannel` object represents a connection; performs non-blocking I/O operations using background threads

```
1 package java.nio.channels;
2 public abstract class AsynchronousSocketChannel
3  implements AsynchronousByteChannel, NetworkChannel
4 {
5  public abstract Future<Integer> read(ByteBuffer dst);
6  public abstract Future<Integer> write(ByteBuffer src);
7  /* ... */
8 }
```

Recap: Futures

- Asynchronous operations return `Future` objects that
  - Inspect state
  - Wait for completion
  - Retrieve results

  of the background thread

```
1  package java.util.concurrent;
2  public interface Future<V> {
3    /* ... */
4    V get() throws InterruptedException,
5      ExecutionException;
6    boolean isDone();
7    /* ... */
8  }
```

Outline

Fork/Join pools

New parallelization framework, especially well suited to recursive divide-and-conquer algorithms

```
1  package java.util.concurrent;
2  public class ForkJoinPool extends
3    AbstractExecutorService {
4
5    public ForkJoinPool(int parallelism);
6
7    public <T> T invoke(ForkJoinTask<T> task);
8    /* ... */
9  }
```

Fork/Join pools

New parallelization framework, especially well suited to recursive divide-and-conquer algorithms

```
1  package java.util.concurrent;
2  public class ForkJoinPool extends
3    AbstractExecutorService {
4
5    public ForkJoinPool(int parallelism);
6
7    public <T> T invoke(ForkJoinTask<T> task);
8    /* ... */
9  }
```

```
1  public abstract class RecursiveTask<V> extends
2    ForkJoinTask<V> {
3
4    protected abstract V compute();
5    /* ... */
6  }
```

Outline

Lambda functions

- Many modern programming languages have syntax for function literals
- Example: `lambda` keyword in Python

```
1  list1 = [1,2,3,4,5]
2  list2 = map(lambda x: x * x, list1)
3  # list2 now contains [1,4,9,16,25]
4
5  # equivalent named function:
6  def square(x):
7    return x * x
8  list2 = map(square, list1)
```

Use case for lambda functions in Java 8

Can't pass function, but often "functional interfaces"

```
1  public class File {
2    /* ... */
3    public File[] listFiles(FileFilter filter);
4  }
```

```
1  public interface FileFilter {
2    boolean accept(File pathname);
3  }
```

Use case for lambda functions in Java 8

Can't pass function, but often "functional interfaces"

```
1  public class File {
2    /* ... */
3    public File[] listFiles(FileFilter filter);
4  }
```

```
1  public interface FileFilter {
2    boolean accept(File pathname);
3  }
```

Often implemented as anonymous inline class

```
1  File[] files = myDir.listFiles(new FileFilter(){
2      @Override
3      public boolean accept(File pathname) {
4        return pathname.canExecute();
5      }
6  });
```

Possible syntax for Java 8 lambdas I

Main use for lambda expressions will be implementing functional interfaces

```
1 // statement syntax
2 FileFilter exists = (File f) -> f.exists();
3 File[] files = myDir.listFiles( exists );

5 // block syntax
6 files = myDir.listFiles( (File f) -> {
7   return f.canWrite();
8 }
```

Possible syntax for Java 8 lambdas II

Many standard library interfaces would fit this "functional" pattern

```
1  // interface:
2  // int Comparator<String>.compare(String s1, String s2)
3  Comparator<String> cmp =
4    (s1, s2) -> s1.compareToIgnoreCase(s2);
5
6  // interface:
7  // void Runnable.run()
8  new Thread( () -> {
9    performLongComputation();
10   System.out.println("Done!");
11 } ).start()
```

Conclusions

- Several new language features
  - Diamond operator
  - Switch statement with strings
  - Numeric literals
  - Exceptions
- Standard library additions
  - Paths/Filesystems API
  - Asynchronous channels
  - Fork/Join pools
- Project Lambda will be released with version 8
  - Expected in Summer 2013
  - Will include bulk parallel collections APIs (Filter/Map/Reduce)
- Eclipse Juno

Further Reading I

OpenJDK
Project Coin
http://openjdk.java.net/projects/coin/

Oracle Corporation
Java Programming Language Enhancements
http://docs.oracle.com/javase/7/docs/technotes/guides/language/enhancements.html#javase7

Madhusudhan Konda
A look at Java 7's new features
O'Reilly Radar, September 2, 2011
http://radar.oreilly.com/2011/09/java7-features.html

Alan Bateman et al
JSR 203: More New I/O APIs for the Java Platform ("NIO.2")
http://jcp.org/en/jsr/detail?id=203

Further Reading II


Catherine Hope and Oliver Deakin
An NIO.2 primer, Part 1: The asynchronous channel APIs
http://www.ibm.com/developerworks/java/library/j-nio2-1/index.html


Brian Goetz et al.
JSR 335: Lambda Expressions for the Java Programming Language
http://www.jcp.org/en/jsr/detail?id=335


Brian Goetz
State of the Lambda
http://cr.openjdk.java.net/~briangoetz/lambda/lambda-state-4.html