# What's new in Java 7?

# Improvements in Java 7

Syntax Improvements that help write cleaner code.

- Diamond <> operator
- `try` with resources
- Multi-exceptions `catch` blocks
- `String` arguments in `switch` statements

Java library new features for concurrency and IO.

- `LinkedTransferQueue` concurrent queue
- `java.nio.file` package
- `fork/join` framework

# Diamond Operator <>

- Use <> to avoid type repetition when constructing objects.

**Before Java 7**

```
// new needs to fully specify the Generic Parameters.

public TreeMap<String, LinkedList<String>> map =
        new TreeMap<String, LinkedList<String>>();
```

**Java 7**

```
// new can use the <> operator for shorter declaration.

public TreeMap<String, LinkedList<String>> map =
        new TreeMap<>();
```

# try-with-resources

**Before Java 7, use `finally` block to insure resources are released.**

```
BufferedReader in = null; // declare resource before try block
try {
    in = new BufferedReader(new FileReader(filename));
    for (String line = in.readLine(); line != null;
                                      line = in.readLine()) {
        process(line);
    }
    in.close();
    in = null;
} catch (IOException ex) {
    System.err.println("error: " + ex.getMessage());
} finally {
    if (in != null) { // resource allocation may have failed
        try {
            in.close();
        } catch (IOException ex) { // releasing resource may fail
            System.err.println("error: " + ex.getMessage());
        }
    }
}
```

# try-with-resources

**With Java 7, resource defined with `try()` are automatically released.**

```
// declare resources with try(...)

try (BufferedReader in =
                 new BufferedReader(new FileReader(filename))) {
    for (String line = in.readLine(); line != null;
                                      line = in.readLine()) {
        process(line);
    }
} catch (IOException ex) {
    System.err.println("error: " + ex.getMessage());
}

// resources are automatically released when try block is exited.
```

# try-with-resources

- Exceptions raised when releasing resources are suppressed
  - Typically what you want since released exceptions are ignored.
- If an Exception is allowed to escape the try-with-resource block, it will suppress any exception thrown during the auto-release.
  - Rationale: the exception thrown in the block is more important
  - The suppressed exception can be retrieved via `ex.getSuppressed()`

# try-with-resources

**`IOException` thrown in the `try` block masks any released exception.**

```
// block auto-releases Reader but let IOException bubble up.
try (BufferedReader
    in = new BufferedReader(new FileReader(filename))) {
    for (String line = in.readLine(); line != null;
                                      line = in.readLine()) {
        process(line);
    }
}
```

**Suppressed exception can be examined in the caller.**

```
} catch (IOException ex) {
    System.out.println("error when reading: " + ex.getMessage());
    for (Throwable se : ex.getSuppressed()) {
        System.out.println("suppressed error: " + se.getMessage());
    }
}
```

# try-with-resources

- Resources must implements `AutoCloseable` (compile error otherwise).
- Java library resource classes implement `AutoCloseable`
- Implement `AutoCloseable` in your Class to benefit from the try-with-resource idiom

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

# catching multiple exceptions

**Before Java 7, different exception types have separate catch blocks.**

```
try (BufferedReader in =
        Files.newBufferedReader(Paths.get(new URI(filename)),
                                Charset.defaultCharset())) {
    for (String line = in.readLine(); line != null;
                                      line = in.readLine()) {
        process(line);
    }
} catch (URISyntaxException ex) {
    System.err.println("error: " + ex.getMessage());
} catch (IOException ex) {
    System.err.println("error: " + ex.getMessage());
}
```

# catching multiple exceptions

**With Java 7, exceptions can be caught within one catch block.**

```
try (BufferedReader in =
        Files.newBufferedReader(Paths.get(new URI(filename)),
                                Charset.defaultCharset())) {
    for (String line = in.readLine(); line != null;
                                      line = in.readLine()) {
        process(line);
    }
} catch (URISyntaxException | IOException ex) {
    System.err.println("error: " + ex.getMessage());
}
```

# String arguments in switch

**Before Java7, only integer and enum arguments were allowed in switch statements.**

```
for (String arg : args) {
    if ("-help".equals(arg)) {
        displayHelp();
    } else if ("-verbose".equals(arg)) {
        setVerbose(true);
    } else if ("-recursive".equals(arg)) {
        setRecursive(true);
    } else {
        setFilename(arg);
    }
}
```

# String arguments in switch

**With Java 7, strings can also be used.**

```
for (String arg : args) {
    switch (arg) {
        case "-help":
            displayHelp();
            break;
        case "-verbose":
            setVerbose(true);
            break;
        case "-recursive":
            setRecursive(true);
            break;
        default:
            setFilename(arg);
    }
```

# LinkedTransferQueue

- Class `LinkedTransferQueue` is a thread safe queue useful to communicate between threads.
- Important: to be thread safe, messages passed between the threads should be either:
  - read only
  - deep copy
  - such that sender doesn't keep handles on object

```
// Create queue and pass to both Producer and Consumer
LinkedTransferQueue<Message> queue = new LinkedTransferQueue<>();
Producer prod = new Producer(queue, 10);
Consumer cons = new Consumer(queue);
```

# LinkedTransferQueue

**Producer uses `put()` for a FIFO queue.**

```
// producer adds messages to the LinkedTransferQueue
public void run() {
    while (running) {
        for (int i = 1; i <= nmsgs; i++) {
            Message msg = createMessage(i);
            queue.put(msg);
            sleep(10);
        }
        running = false;
    }
}
```

# LinkedTransferQueue

**Consumer can use `take()` to read (blocking) from the FIFO queue.**

```
// consumer reads messages from the LinkedTransferQueue
public void run() {
    while (running) {
        try {
            // thread is blocked until an element is available.
            Message msg = queue.take();
            processMessage(msg);
        } catch (InterruptedException _) {
            // thread blocked on take() can be interrupted.
            // interrupting the thread is useful for the thread to
            // be able to check if running flag has changed.
        }
    }
}
```

# LinkedTransferQueue

**Consumer can use `poll()` to read (non-blocking) from FIFO queues.**

```
// consumer reads messages from multiple LinkedTransferQueue
// using the polling interface.
public void run() {
    while (running) {
        try {
            Command cmd;
            Message msg;
            if ((cmd = cmdQueue.poll()) != null) {
                processCommand(cmd);
            } else if ((msg = msgQueue.poll()) != null) {
                processMessage(msg);
            } else {
                Thread.sleep(50);
            }
        } catch (InterruptedException _) {
            // just ignore the interruption for this case.
        }
    }
}
```

# Fork/Join Framework

- New concurrency framework to take advantage of multiple cores/CPUs
- Designed for Divide-and-Conquer (recursive) problems
  - Use Class `ForkJoinTask` instances
  - Split a task with `fork()`
  - Wait for a forked task to complete with `join()`
  - Support tasks returning results with `RecursiveTask<E>`
  - Support resultless tasks with `RecursiveAction`

# Fork/Join Framework

**Example using Fork/Join with Tasks returning a result**

```
public class ParMaximum extends RecursiveTask<Double> {
...
public Double compute() {

    // if problem is small enough then solve sequentially
    if ((high - low) < THRESHOLD) {
        return computeDirectly();
    }

    // else (recursively) fork half the problem
    int split = low + (high - low) / 2;
    ParMaximum left = new ParMaximum(values, low, split, origin);
    left.fork();
    ParMaximum right = new ParMaximum(values, split, high, origin);

    return Math.max(right.compute(), left.join());
}
```

# Fork/Join Framework

**Example using Fork/Join with result-less Tasks**

```
public class ParQuicksort extends RecursiveAction {
...
public void compute() {
...
    // if the problem is big enough, and we have two branches
    // then solve in parallel.
    if (((high - low) > THRESHOLD) && (low < j) && (i < high)) {
        ParQuicksort sort = new ParQuicksort(numbers, low, j);
        sort.fork();
        new ParQuicksort(numbers, i, high).compute();
        sort.join();

    // else solve sequentially
    } else {
        if (low < j) {
            new ParQuicksort(numbers, low, j).compute();
        }
        if (i < high) {
            new ParQuicksort(numbers, i, high).compute();
        }
    }
```

# Fork/Join Framework

- Fork/Join uses a special ExecutorService : ForkJoinPool

```
// use a single ForkJoinPool per VM
static final ForkJoinPool pool = new ForkJoinPool();
...
ParQuicksort qs = new ParQuicksort(values, 0, values.length - 1);
pool.invoke(qs);
```

# java.nio.file

- Package `java.nio.file`

| Class | Usage |
| --- | --- |
| `Files` | Provides a set of static methods that operate on files, directories, or other types of files |
| `Paths` | Provides a set of static methods that return a Path by converting a path string or URI |
| `FileSystem` | Provides an interface to the file system and a factory for objects accessing files and other filesystem objects |
| `FileSystems` | Provides factory methods for file systems. This class defines the `getDefault()` method to access the default file system and factory methods to construct other types of file systems. |

# References

- Java 7 new features
  - O'Reilly, a look at Java7 new features
  - Oracle, Java SE Features and Enhancements
  - 10 JDK 7 Features to revisit before you welcome Java8
- Concurrency
  - When to use ForkJoinPool vs. ExecutorService
  - A java Fork/Join Calamity
  - Java Fork and Join using ForkJoinPool
  - Doug Lea's Workstation

# Attributions

- Duke's image is from Wikimedia "Duke: Java Mascot".
- This presentation is using the excellent remark framework.