Object Oriented Programming

1. Java 7 untold
2. Java 8 untold

- Java 7

Some of the new things

Binary Literals

```
int mask = 0b101010101010;
aShort =
  (short)0b1010000101000101;
long aLong =
  0b101000010100010110100001010001
  01101000010100010110100001010001
  01L;
```

Underscores in Number Literals

- **Valid**

```
int mask = 0b1010_1010_1010;
long big = 9_223_783_036_967_937L;
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_BFFE;
```

- **Invalid**

```
float pi1 = 3_.1415F;               float pi2 = 3._1415F;
long ssn = 999_99_9999_L;
int x1 = _52;                       int x1 = 52_;
int x2 = 0_x52;                     int x2 = 0x_52;
```

Strings in switch statements

```
int monthNameToDays(String s, int year) {
  switch(s) {
    case "April": case "June":
    case "September": case "November":
      return 30;

    case "January": case "March":
    case "May": case "July":
    case "August": case "December":
      return 31;

    case "February”:
      ...
    default:
      ...
```

Automatic Resource Management

```
try (InputStream in = new FileInputStream(src),
     OutputStream out = new FileOutputStream(dest)) {
  byte[] buf = new byte[8192];
  int n;
  while (n = in.read(buf)) >= 0)
    out.write(buf, 0, n);
}
```

- New superinterface `java.lang.AutoCloseable`
- All `AutoCloseable` (throws `Exception`) and by extension `java.io.Closeable` (throws `IOException`) types useable with try-with-resources
- Anything with a `void close()` method is a candidate
- JDBC 4.1 retrofitted as `AutoCloseable` too

Supressed Exceptions

```
java.io.IOException
    at Suppress.write(Suppress.java:19)
    at Suppress.main(Suppress.java:8)
    Suppressed:  java.io.IOException
        at Suppress.close(Suppress.java:24)
        at Suppress.main(Suppress.java:9)
    Suppressed:  java.io.IOException
        at  Suppress.close(Suppress.java:24)
        at  Suppress.main(Suppress.java:9)
```

```
Throwable.getSupressed(); // Returns Throwable[]
Throwable.addSupressed(aThrowable);
```

Multi-catch

```
try {
   ...
} catch (ClassCastException e) {
  doSomethingClever(e);
  throw e;
} catch(InstantiationException |
             NoSuchMethodException |
             InvocationTargetException e) {
    // Useful if you do generic actions
  log(e);
    throw e;
}
```

Diamond operator works in many ways

- With diamond (<>) compiler infers type

```
List<String> strList = new ArrayList<>();
OR
List<Map<String, List<String>> strList =
   new ArrayList<>();
OR
Foo<Bar> foo = new Foo<>();
foo.mergeFoo(new Foo<>());
```

Why We Needed NIO2?

- Methods didn’t throw exceptions when failing
- Rename worked inconsistently
- No symbolic link support
- Additional support for meta data
- Inefficient file meta data access
- File methods didn’t scale
- Walking a tree with symbolic links not possible

Java NIO.2 Features

- Four key new helper Types new in Java 7
- Class java.nio.file.Paths
  - Exclusively static methods to return a Path by converting a string or Uniform Resource Identifier (URI)
- Interface java.nio.file.Path
  - Used for objects that represent the location of a file in a file system, typically system dependent
- Class java.nio.file.Files
  - Exclusively static methods to operate on files, directories and other types of files
- Class java.nio.file.FileSystem
- Typical use case:
  - Use Paths to get a Path. Use Files to do stuff.

Java NIO.2 Example of Helpers in Action

- File copy is really easy
  - With fine grain control

```
Path src = Paths.get(“/home/fred/readme.txt”);
Path dst = Paths.get(“/home/fred/copy_readme.txt”);
Files.copy(src, dst,
       StandardCopyOption.COPY_ATTRIBUTES,
       StandardCopyOption.REPLACE_EXISTING);
```

- File move is supported
  - Optional atomic move supported

```
Path src = Paths.get(“/home/fred/readme.txt”);
Path dst = Paths.get(“/home/fred/readme.1st”);
Files.move(src, dst, StandardCopyOption.ATOMIC_MOVE);
```

Java NIO.2 Features – Files Class

- Files helper class is feature rich:
  - Copy
  - Create Directories
  - Create Files
  - Create Links
  - Use of system “temp” directory
  - Delete
  - Attributes – Modified/Owner/Permissions/Size, etc.
  - Read/Write

Java NIO.2 Directories

- **DirectoryStream iterate over entries**
  - Scales to large directories
  - Uses less resources
  - Smooth out response time for remote file systems
  - Implements **Iterable** and **Closeable** for productivity
- **Filtering support**
  - Build-in support for glob, regex and custom filters

```
Path srcPath = Paths.get(“/home/jim/src”);
try (DirectoryStream<Path> dir =
    srcPath.newDirectoryStream(“*.java”)) {
  for (Path file : dir)
    System.out.println(file.getName());
}
```

Java NIO.2 Symbolic Links

- Path and Files are “link aware”
- **createSymbolicLink(Path, Path, FileAttribute<?>)**

```
Path newLink = Paths.get(. . .);
Path existingFile = Paths.get(. . .);
try {
    Files.createSymbolicLink(newLink, existingFile);
} catch (IOException x) {
    System.err.println(x);
} catch (UnsupportedOperationException x) {
      //Some file systems or some configurations
             //may not support links
    System.err.println(x);
}
```

Java NIO.2 More on Symbolic Links

- Hard Links
- Detect a Symbolic Link
- Find the Target of the Link

```
try {
    Files.createLink(newLink, existingFile);
} catch (IOException | UnsupportedOperationException x) {
    System.err.println(x);
}
boolean isSymbolicLink =
    Files.isSymbolicLink(file);
Path link = ...;
Files.readSymbolicLink(link));
```

- A `FileVisitor` interface makes walking a file tree for search, or performing actions, trivial.
- `SimpleFileVisitor` implements

```
preVisitDirectory(T dir, BasicFileAttributes attrs);
visitFile(T dir, BasicFileAttributes attrs);
visitFileFailed(T dir, IOException exc);
postVisitDirectory(T dir, IOException exc);
```

- SAMPLE:

```
Path startingDir = ...;
PrintFiles pf = new PrintFiles(); // SimpleFileVisitor sub
    // visitFile(Path p, BasicFileAttributes bfa) {
  //
    System.out.println(file.getFileName());}
Files.walkFileTree(startingDir, pf);
```

Java NIO.2 Watching a Directory

- Create a `WatchService` "watcher" for the filesystem
- Register a directory with the watcher
- "Watcher" can be polled or waited on for events
  - Events raised in the form of Keys
  - Retrieve the Key from the Watcher
  - Key has filename and events within it for create/delete/modify
- Ability to detect event overflows

Java NIO.2 Custom FileSystems

- **FileSystems class is factory to great FileSystem (interface)**
- **Java 7 allows for developing custom FileSystems, for example:**
  - Memory based or zip file based systems
  - Fault tolerant distributed file systems
  - Replacing or supplementing the default file system provider
- **Two steps:**
  - Implement `java.nio.file.spi.FileSystemProvider`
    - URI, Caching, File Handling, etc.
  - Implement `java.nio.file.FileSystem`
    - Roots, RW access, file store, etc.

NIO.2 Filesystem Provider for zip/jar Archives

- A fully-functional and supported NIO.2 filesystem provider for zip and jar files

```
Map<String, String> env = new HashMap<>();
       env.put("create", "true");
       // locate file system by using the syntax
       // defined in java.net.JarURLConnection
       URI u= URI.create("jar:file:/foo/zipfs/zipfstest.zip");
       try (FileSystem z = FileSystems.newFileSystem(u, env)) {
              Path externalTxtFile =
Paths.get("/foo/zipfs/Sample.txt");
              Path pathInZipfile = z.getPath("/Sample.txt");
       // copy a file into the zip file
       externalTxtFile.copyTo(pathInZipfile);
}
```

Mapping java.io.File to java.nio.file

`java.io.File`	`java.nio.file.Path`
`File.canRead, canWrite, canExecute`	`Files.isReadable, Files.isWritable, and Files.isExecutable.`
`File.isDirectory(), File.isFile(), and File.length()`	`Files.isDirectory(Path, LinkOption...), Files.isRegularFile(Path, LinkOption...), and Files.size(Path)`
`File.lastModified() and File.setLastModified(long)`	`Files.getLastModifiedTime(Path, LinkOption...) and Files.setLastModifiedTime(Path, FileTime)`
`File methods: setExecutable, setReadable, setReadOnly, setWritable`	`Files methods: setAttribute(Path, String, Object, LinkOption...).`
`new File(parent, "newfile")`	`parent.resolve("newfile")`

There is no one-to-one correspondence between the two APIs

Mapping java.io.File to java.nio.file

`File.renameTo`	`Files.move`
`File.delete`	`Files.delete`
`File.createNewFile`	`Files.createFile`
`File.deleteOnExit`	`DELETE_ON_CLOSE option in createFile`
`File.exists`	`Files.exists and Files.notExists`
`File.compareTo and equals`	`Path.compareTo and equals`
`File.getAbsolutePath and getAbsoluteFile`	`Path.toAbsolutePath`
`File.getCanonicalPath and getCanonicalFile`	`Path.toRealPath or normalize`
`File.isHidden`	`Files.isHidden`
`File.mkdir and mkdirs`	`Path.createDirectory`
`File.listRoots`	`FileSystem.getRootDirectories`

There is no one-to-one correspondence between the two APIs

- Java 8

Lambdas and streams

Lambdas and streams

- Understand the syntax, semantics, and typechecking of lambdas in Java
- Write code effectively with lambdas in Java
- Use the Java stream library both sequentially and in parallel
- Use default methods to put reusable code in Java interfaces

Why Lambdas. An example

```
final String name = "Santa Klaus";
Runnable greeter = new Runnable() {
     public void run() {
          System.out.println(“Welcome " + name);
     }
};
```

```
// add functionality to the step button.
step.addActionListener(new ActionListener(){
  @Override
     public void actionPerformed(ActionEvent arg0) {
          worldPanel.step();
     }
});
```

- One line of code needs a lot of boilerplate code

Lambdas: Convenient Syntax for Single-Function Objects

```
final String name = "Santa Klaus";
Runnable greeter = new Runnable() {
    public void run() {
        System.out.println(“Welcome " + name
    }
};
// with Lambdas, can rewrite the code above like thi
String  name  =  " Santa Klaus ";
Runnable  greeter  =  ()  ->  System.out.println(“Welcome "  +
          name);
```

The name variable is used in the function; need not be final, but must be *effectively final*

The function can be assigned to a Runnable, because it has the same signature as **run()**

We use a lambda expression to define a function that takes no arguments

The function body just prints to standard out

Effectively Final Variables

```
final String name = "Santa Klaus";
Runnable greeter = new Runnable() {
     public void run() {
          System.out.println(“Welcome " + name);
     }
};
// with Lambdas, can rewrite the code above like this
String name = " Santa Klaus ";
Runnable greeter = () -> System.out.println(“Welvcome " + name);
```

The name variable is used in the function; need not be final, but must be *effectively final*

Lambdas can use local variables in outer scopes only if they are effectively final. A variable is *effectively final* if it can be made final without introducing a compilation error. This facilitates using lambdas for concurrency, and avoids problems with lambdas outliving their surrounding scope.

Replacing For Loops with Lambdas

```
// Java 7 code to print an array
List<Integer> intList = Arrays.asList(1,2,3);
for (Integer i in intList)
System.out.println(i)
// Java 8 provides a forEach method to do the same thing...
intList.forEach(new Consumer<Integer>() {
public void accept(Integer i) {
System.out.println(i);
}
});
// Java 8’s Lambda’s make forEach beautiful
intList.forEach((Integer i) -> System.out.println(i));
intList.forEach(i -> System.out.println(i));
```

This lambda expression takes one argument, i, of type Integer

Even cleaner…since intList.forEach() takes a Consumer<Integer>, Java infers that i’s type is Integer

Example adapted from Alfred V. Aho

Lambda Syntax Options

- Lambda Syntax

```
(parameters) -> expression
or (parameters) -> { statements; }
```

- Details
  - Parameter types may be inferred (all or none)
  - Parentheses may be omitted for a single inferred-type parameter
- Examples

```
(int x, int y) -> x + y  // takes two integers and returns their sum
(x, y) -> x - y  // takes two numbers and returns their difference
() -> 42  // takes no values and returns 42
(String s) -> System.out.println(s)  // takes a string, prints its value
x -> 2 * x  // takes a number and returns the result of doubling it
c -> { int s = c.size(); c.clear(); return s; }  // takes a collection,
// clears it, and returns its previous size
```

Functional Interfaces

- There are *no function types* in Java
- Instead, Java has *Functional Interfaces*
  - interfaces with only one explicitly declared abstract method
    - methods inherited from `Object`, like `equals()`, don't count
  - Optionally annotated with `@FunctionalInterface`
    - Helps catch errors if you intend to write a functional interface but don't
- Some Functional Interfaces

```
java.lang.Runnable: void run()
java.util.function.Consumer<T>: void accept(T t)
java.util.concurrent.Callable<V>: V call()
java.util.function.Function<T,R>: R apply(T t)
java.util.Comparator<T>: int compare(T o1, T o2)
java.awt.event.ActionListener: void actionPerformed(ActionEvent e)
```

- There are many more, especially in package `java.util.function`

Typechecking and Type Inference Using Expected Types

- A lambda expression must match its expected type
  - The type of the variable to which it is assigned or passed

```
intList.forEach(i -> System.out.println(i));
```

- Example: `forEach`
  - `intList.forEach` accepts a parameter of type `Consumer<Integer>,` so this is the expected type for the lambda
  - `Consumer<Integer>` has a function `void accept(Integer t)`, so the lambda's argument is inferred to be of type `Integer`

```
Runnable greeter = () -> System.out.println("Hi " + name);
```

- Example: `Runnable`
  - We are assigning a lambda to a variable of type `Runnable`, so that is the expected type for the lambda
  - `Runnable` has a function `void run(),` so the lambda expression must not take any arguments

Method References

```
// Recall Java 8 code to print integers in an array
List<Integer> intList = Arrays.asList(1,2,3);
intList.forEach(i -> System.out.println(i));
// We can make the last line even shorter!
intList.forEach(System.out::println);
```

- `System.out::println` is a method reference
  - Captures the `println` method of `System.out` as a function
  - The type is `Consumer<Integer>`, as required by `intList.forEach`
  - The signature of `println` must match (and it does)

Method Reference Syntactic Forms

- Capturing an *instance method* of a particular object

  Syntax: `objectReference::methodName`

  Example: `intList.forEach(System.out::println)`

- Capturing a *static method*

  Syntax: `ClassName::methodName`

  Example: `Arrays.sort(myIntegerArray, Integer::compare)`

- Capturing an *instance method*, *without* capturing the *object*
  - The resulting function has an extra argument for the receiver

  Syntax: `ClassName::methodName`

  Example: `Function<Object,String> printer = Object::toString;`

- Capturing a *constructor*

  Syntax: `ClassName::methodName`

  Example: `Supplier<List<String>> listFactory = ArrayList::<String>new;`

Collections Usage in Java

- *Bulk operations*: common usage pattern for Java collections
  - Read from a source collection
  - Select certain elements
  - Compute collections holding intermediate data
  - Summarize the results into a single answer
- Example: how much taxes do student employees pay?

```
List<PayStub> studentStubs = new ArrayList<PayStub>();
for (Employee e in employees)
  if (e.getStatus() == Employee.STUDENT)
   studentStubs.addAll(e.payStubs());
double totalTax=0.0;
for (PayStub s in studentStubs)  totalTax += s.getTax();
```

- Issues
  - Inefficient to create temporary collections
  - Verbose code
  - Hard to do work in parallel

Streams: A Better Way

```
double    totalTax=
  employees.parallelStream()
      .filter(e->e.getStatus()==Employee.STUDENT)
      .flatMap(e->e.payStubs().stream())
      .sum()
```

- Benefits
  - Shorter
  - More abstract – describes what is desired
  - More efficient – avoids intermediate data structure
  - Runs in parallel

Streams

- Definition: a possibly-infinite sequence of elements supporting sequential or parallel aggregate operations
  - *possibly-infinite*: elements are processed lazily
  - *sequential or parallel*: two kinds of streams
  - *aggregate*: operations act on the entire stream
    - contrast: iterators

- Some stream sources
  - Invoking `.stream()` or `.parallelStream()` on any `Collection`
  - Invoking `.lines()` on a `BufferedReader`
  - Generating from a function: `Stream.generate(Supplier<T> s)`

Streams

- Intermediate operations
  - Produce one stream from another
  - Examples: `map`, `filter`, `sorted`, …
- Terminal operations
  - Extract a value or a collection from a stream
  - Examples: `reduce`, `collect`, `count`, `findAny`
- Demos:
  - `GetWords`
  - `ComputeANumber`
  - `ComputeABigNumber`

Default Methods

- Java 8 just added several methods to `Collection` interfaces

```
Stream<E> stream()
Stream<E> parallelStream()
void forEach(Consumer<E> action)
Spliterator<E> spliterator()
boolean removeIf(Predicate<E> filter)
```

- If you defined a `Collection` subclass, did it just break?
- No! These were added as default methods
  - Declared in an interface with the `default` keyword
  - Given a body

```
interface Collection<E> {
 default Stream<E> stream() {
     return StreamSupport.stream(spliterator(), false);
 }
}
```

Default Methods: Semantics and Uses

- Semantics
  - A method defined in a class always overrides a default method
  - Default methods in sub-interfaces override those in super-interfaces
  - Remaining conflicts must be resolved by overriding
  - New syntax for invoking a default method from implementor
    - `A.super.m(...)`
    - Important because m may be defined in two implemented interfaces, so can’t use simply `super.m(...)`
- Benefits of default methods
  - Extending an interface without breaking implementors
  - Putting reusable code in an interface
    - can reuse default methods from several interfaces
    - known as ***traits*** in other languages (e.g. Scala)

Summary

- Java 8 has new features useful in program expression
  - Lambdas are a lightweight syntax for defining functions
    - Support shorter and more abstract code
  - Succinct manipulation of data through streams
    - Support for pipelining and parallelism
  - Default methods provide code reuse in interfaces

Sources

- Text adapted from
  - http://www.cs.cmu.edu/~charlie/courses/15-214/2015-fall/slides/07b-java8.pdf
- Maurice Naftalin's Lambda FAQ
  - http://www.lambdafaq.org/
- The Java Tutorials:
  - Lambda Expressions
    - https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html
  - Aggregate Operations
    - https://docs.oracle.com/javase/tutorial/collections/streams/index.html