Intermediate Java

WELCOME!

Develop Intelligence

Simon Roberts

Copyright Dancing Cloud Services LLC & DevelopIntelligence LLC

WELCOME

Mark Your Attendance:

- Go to the Course Calendar Invite
- Click on the Course Event Link
- Press Check-In

Copyright Dancing Cloud Services LLC & DevelopIntelligence LLC

Join Us in Making Learning Technology Easier

Our mission...

Over 16 years ago, we embarked on a journey to improve the world by making learning technology easy and accessible to everyone.

...impacts everyone daily.

And it’s working. Today, we’re known for delivering customized tech learning programs that drive innovation and transform organizations.

In fact, when you talk on the phone, watch a movie, connect with friends on social media, drive a car, fly on a plane, shop online, and order a latte with your mobile app, you are experiencing the impact of our solutions.

Over The Past Few Decades, We've Provided

Over **62,300,000** expert-led learning hours

In 2019 Alone, We Provided

Training to over **13,500** engineers

Over **120** active trainers, with an average of over two decades of experience each.

Programs in **30** countries

Copyright Dancing Cloud Services LLC & DevelopIntelligence LLC
Copyright DevelopIntelligence LLC

Upskilling and Reskilling Offerings

Develop Intelligence

Intimately customized learning experiences just for your teams.

Workshop	2-3 day upskilling experiences
Fast Track	5-day reskilling experiences
Learning Spike	1-day technology overviews
Target Topics	90-minute instructor-led micro-learnings
Hack-a-thon	Learn and build an MVP in 2-3 days

BACK END DEVELOPMENT

BIG DATA

CLOUD COMPUTING

DEVOPS

FRONT END DEVELOPMENT

MACHINE LEARNING

MOBILE APP DEVELOPMENT

SOFTWARE ENGINEERING

SYSTEM ADMINISTRATION

AND MANY OTHER TRENDING TECHNOLOGIES

World Class Practitioners

Copyright Dancing Cloud Services LLC & DevelopIntelligence LLC

Note About Virtual Trainings

Develop Intelligence

What we want

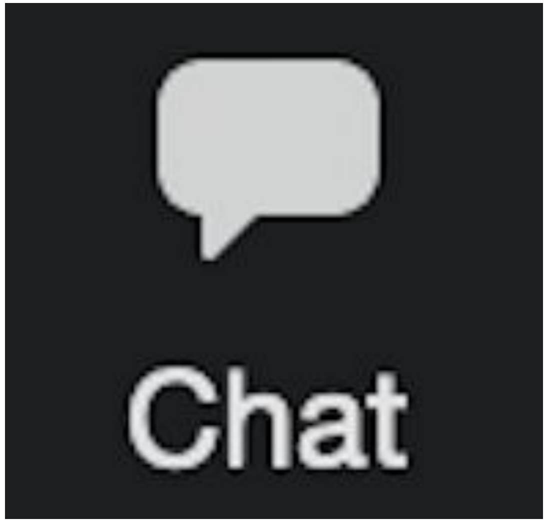
...what we've got

Copyright Dancing Cloud Services LLC & DevelopIntelligence LLC

Virtual Training Expectations for You

Arrive on time / return on time

Mute unless speaking



Use chat or ask questions verbally

Copyright Dancing Cloud Services LLC & DevelopIntelligence LLC

Virtual Training Expectations for Me

I pledge to:

- Make this as interesting and interactive as possible
- Ask questions in order to stimulate discussion
- Use whatever resources I have at hand to explain the material
- Try my best to manage verbal responses so that everyone who wants to speak can do so
- Use an on-screen timer for breaks so you know when to be back

Copyright Dancing Cloud Services LLC & DevelopIntelligence LLC

Prerequisites

- You should have written code in Java and be familiar with the majority of syntax that exists in Java 7.
- This course will introduce some of the features that are less well understood from Java 7 onwards.

Copyright Dancing Cloud Services LLC & DevelopIntelligence LLC

Objectives

At the end of this course you will be able to:

- Use try with resources and multi-catch constructs to handle exceptions.
- Use inner and nested classes to implement appropriate design patterns.
- Make good use of collections and streams apis.
- Use the java.time api.
- Create generic classes and methods
- Use Java's functional features including writing and using lambda expressions.
- Create annotations and write code that identifies and uses annotations at runtime.
- Use key features of the java.util.concurrent apis.

Copyright Dancing Cloud Services LLC & DevelopIntelligence LLC

About you

In 90 seconds!

- What you hope to learn
- What your background level is

Copyright Dancing Cloud Services LLC & DevelopIntelligence LLC

Agenda

- Investigating the try-with-resources mechanism
- Reviewing Java syntax, design patterns, and using inner classes.
- Investigating collections and introducing functional ideas
- Building more functional syntax
- Using streams
- Building generic types and methods
- Working with annotations
- Using the concurrency api

Copyright Dancing Cloud Services LLC & DevelopIntelligence LLC

THANK YOU

Develop Intelligence

Copyright Dancing Cloud Services LLC & DevelopIntelligence LLC

Exception handling in Java 7

Java 7 added:

- Try-with-resources
- multi-catch

Try-with-resources

Prior to this feature Java's try/finally had significant issues when trying to ensure reliable closure of O/S resources (such as files).

- Variable scope rules made access to the resource in the finally block cumbersome
- The definite initialization rule requires null-checking every resource
- Many close() methods throw a checked exception, mandating try/catch in the finally block
- Perhaps worse only one live exception is possible per-thread in the JVM, which means significant exceptions can be lost if a close throws an exception.

Copyright Dancing Cloud Services LLC & DevelopIntelligence LLC

Try-with-resources

Java 7 try-with-resources addresses all of these problems:

```
try ( // declare+initialize AutoCloseable resources
  BufferedReader in =
    Files.newBufferedReader(Path.of("input.txt"));
  BufferedWriter out = Files.newBufferedWriter(
    Path.of("out.txt", StandardOpenOption.CREATE));
) {
  // regular try body.
} // No "finally" needed, resources automatically closed
```

Try-with-resources

- Multiple resources can be listed in the resources block
- Separate with semicolon
- "Terminating" semicolon permitted but not required
- Since Java 9, a final variable that is declared above the try may be listed, allowing greater scope, but still guaranteeing closure

Exercise

The try with resources mechanism makes a best effort to close all "resources" that are declared in the resources block. In this exercise you will build code that demonstrates the inner workings of this, and simultaneously allows you to practice the coding of a try-with-resources construct.

1) Create a class "MyResource" which implements the AutoCloseable interface

- Have the constructor take a String argument which is stored in the object
- Arrange for the constructor to print a message "initializing MyResource" followed by the String argument
- Arrange that the close method of MyResource prints a message "closing MyResource" followed by the initialzing String
- no other behavior is necessary in this class

Design Concepts

If a change from design A to design B is easy, and both satisfy the functional requirement, but a change from B to A is relatively hard, prefer design A unless there are other reasons to choose between them

- Make classes final until there is a determined need for inheritance, and the consequences are understood
- Prefer factories or builders over constructors
  - Constructors can only produce a new object of the exact type, or an exception
  - Any method can do this, but can do much more too
  - Removing constructors breaks clients, changing the implementation details of a method does not.

Copyright Dancing Cloud Services LLC & DevelopIntelligence LLC

Design concepts

Consider preferring immutable data and structures

- This can avoid "that can't happen" moments where other users of data altered them unexpectedly
- Immutable data can be shared, perhaps reducing the need for copying and compensating for the need to make new data to describe variations on old data
- Sometimes an immutable proxy (such as Collections.unmodifiableXxx) serves well to prevent clients changing something while allowing the "owner" to change it.

Copyright Dancing Cloud Services LLC & DevelopIntelligence LLC

Design concepts

Ensure that a business domain object is always "valid"

- Avoids unexpected states breaking client code
- Ensure any/all mutable fields are private
- Ensure that validity is verified prior to completing construction and before all changes
- Represent adjunct concepts (such as wire transfer and database storage) in secondary classes (e.g. a Data Transfer Object). Ensure the secondary class is dependent on the primary but not the other way round.

Copyright Dancing Cloud Services LLC & DevelopIntelligence LLC

Design concepts

Keep in mind that a Set is intended to reject duplicates, and might be preferable to a List in some situations. However:

- Sets reject duplicates simply by returning false from their add methods. This is easy to overlook.
- Sets require objects that implement equals/hashcode or ordering, and it's not always clear which is required if you simply have a java.util.Set.
- Most (but not all) core Java objects do not implement equals/hashcode unless they're immutable (String, primitive wrappers, java.time classes, for example)

Separation of concerns

Take care to separate unrelated concerns, and also concerns that change independently, for example:

To print a subset of a list involves 3 independent concerns:

- processing the list
- distinguishing which items are wanted
- printing the result

Copyright Dancing Cloud Services LLC & DevelopIntelligence LLC

Command pattern

Defining the selection mechanism is done (in object oriented languages) by defining a behavior (method) in an object, and passing that object as an argument for the purpose of the behavior it contains

Passing the behavior as in argument object is known (in the Gang of Four pattern catalog) as the "command" pattern

The concept is also also known as a higher order function in "functional programming"

Copyright Dancing Cloud Services LLC & DevelopIntelligence LLC

Lambda expressions

A lambda expression in Java defines an object in a context.

- The context must require an implementation of an interface
- The interface must declare EXACTLY ONE abstract method
- We must only want to implement that one abstract method
- We provide a modified method argument list and body with an "arrow" between them:

```
(Student s) -> {
  // function body
}
```

- The argument list and return type must conform to the abstract method's signature

Copyright Dancing Cloud Services LLC & DevelopIntelligence LLC

Lambda syntax variations

- Argument types can be omitted if they're unambiguous
  - This is "all or nothing"
- Since Java 10, argument types can be replaced with `var` if they're unambiguous
  - This is also "all or nothing"
- If a single argument carries zero type information the parentheses can be omitted
- If the method body consists of a single return statement, the entire body can be replaced with the expression that is to be returned.

Copyright Dancing Cloud Services LLC & DevelopIntelligence LLC

`@FunctionalInterface` annotation

If an interface is intended for use with lambdas, it must define exactly one abstract method.

The `@FunctionalInterface` annotation asks the compiler to verify this and create an error if this is not the case.

Predefined interfaces

Due to Java's strong static typing, and the restrictions preventing generics being used with primitives, different interfaces must be provided for a variety of different situations.

Key interface categories are:

`Function` - takes argument, produces result

`Supplier` - zero argument, produces result

`Consumer` - takes argument, returns void

`Predicate` - takes argument, returns boolean

`Unary/Binary Operator` - `Function` variants that lock args and returns to identical type

Predefined interface variations

For two arguments, expect a `Bi` prefix

For primitives:

- `Int`, `Long`, `Double` prefix usually means "primitive argument"
  - Note for `Supplier`, this prefix refers to return type (there are zero arguments.)
- `ToInt`, `ToLong`, `ToDouble` prefix means "primitive return"

Copyright Dancing Cloud Services LLC & DevelopIntelligence LLC

Method References

Four forms of lambda can be replaced with an alternate syntax called a method reference:

`(a, b, c) -> a.doStuff(b, c)` [for a of type MyClass, and any number of arguments b, c, but at least the argument a] can be replaced with:

```
MyClass::doStuff
```

`(a, b, c) -> MyClass.doStuff(a, b, c)` for any number of arguments can be replaced with:

```
MyClass::doStuff
```

Copyright Dancing Cloud Services LLC & DevelopIntelligence LLC

Method References

`(a, b, c) -> new MyClass(a, b, c)` for any number of arguments can be replaced with:

```
MyClass::new
```

`(a, b, c) -> <obj.expr>.doStuff(a, b, c)` for any number of arguments can be replaced with:

```
<obj.expr>::doStuff
```

If more than one method exists that would map correctly, the compilation fails

Copyright Dancing Cloud Services LLC & DevelopIntelligence LLC

Method References

Method references cannot be used unless the arguments of the lambda and the arguments of the target function:

- are in the correct order
- can be used without modification

Method references might sometimes have more than a single target method that would match the translation. In this case, the method reference form cannot be used.

Method references should be used to focus the reader on the functionality to be used, and when the arguments aren't considered important.

Copyright Dancing Cloud Services LLC & DevelopIntelligence LLC

The Monad (and Functor) patterns

A data structure that has an operation traditionally called "map", which applies a caller-provided function to each of its contained elements and produce a new data structure of the same overall type, containing the same number of elements (but perhaps of a different type) which are the results of the applied function is often called a "Functor"

A related structure that has an operation traditionally called "flatMap", which applies a caller-provided function to each argument, but where that caller-provided function can itself return another instance of the overall structure, is often called a Monad.

Copyright Dancing Cloud Services LLC & DevelopIntelligence LLC

The Monad (and Functor) patterns

Monads can do everything that a Functor can do

The combination of filter, map, and flatMap can create powerful and expressive code that focuses on the operations to be applied to data, rather than being cluttered with the "how" of applying those operations.

In fact, the internal implementation can change (potentially using a lazy approach or a multi-threaded approach) with no difference in how the client code is written.

The approach also works best without mutating any data (and thus is particularly suited to immutable objects).

Stream API

Stream is one of about three "Monad-like" APIs in Java.

It provides filter, map, and flatMap and many more operations

It differs from the "SuperIterable" example in two critical implementation ways:

- Stream is "lazy", items are "pulled" down the pipeline one at a time, so it does not create a complete set of intermediate results at any point
- Stream can "shard" the data across multiple threads, allowing for increased throughput utilizing multiple physical CPUs (but unlike some systems, e.g. Apache Spark, it does not shard across physically distinct, networked, machines.)

Copyright Dancing Cloud Services LLC & DevelopIntelligence LLC

Terminal operations

Once individual items have been selected and processed, it might be necessary to produce "a single result".

The result could be:

- a printout of the results
- a data structure containing the items
- a single data structure representing some aggregate result computed from all the intermediate results

The final step is called the "terminal operation"

In the Stream API, no processing is performed until the terminal operation "pulls" data through the pipeline (this is the lazy aspect).

Copyright Dancing Cloud Services LLC & DevelopIntelligence LLC

Terminal operations

The Stream API provides several built in, and several supporting terminal operations including:

- forEach
- allMatch, anyMatch, noneMatch
- three overloaded reduce methods
- two overloaded collect methods
- and a variety of implementations of the Collector interface that perform operations such as building List, Set, and Map structures, and possibly post-processing the items selected into a map

Copyright Dancing Cloud Services LLC & DevelopIntelligence LLC

Reduction and collection operations

The most general form of terminal operation might be "reduce".

- This operation must be provided with a BinaryOperator that accepts two of the data type in the stream and produces a new one representing the result of combining those two.
- This is applied repetitively to produce a single result
- The most basic form of reduce produces an Optional<T> result which is a monad-like container of zero or one element (avoiding the use of null)

Copyright Dancing Cloud Services LLC & DevelopIntelligence LLC

Java's reduce methods

In Java, there are 3 overloaded reduce operations

- Each requires a binary operator of the stream data type
- One returns an Optional if the stream is empty, the other two require an "identity" object (of the stream type), and if the stream is empty they return that value
- One allows a result type (and requires and identity of the result type) that is different from the stream type. This version requires a BiFunction that takes the result type and the stream type and creates a new result. This version also requires a BinaryOperator of the result type to perform a final merging process if the stream is running parallel.
- In all reduce methods, a new intermediate object must be created at every step. This can sometimes adversely impact performance.

Copyright Dancing Cloud Services LLC & DevelopIntelligence LLC

Collect operations

To mitigate the potential performance impact of creating many intermediate objects, Java provides a variant reduction operation called "collect"

- A collect mutates a single (per thread) result object, rather than creating a new one every time
- The collect operation therefore needs to be able to create its own result objects
- It also needs an operation that mutates the result object with each stream item
- It needs a way to add the contents of one result object from one thread to another result object in the case of parallel execution

Copyright Dancing Cloud Services LLC & DevelopIntelligence LLC

Collect arguments

The three arguments to a collect operation are:

- Supplier<ResultType>
- BiConsumer<ResultType, StreamType>
- BiConsumer<ResultType, ResultType>

Note that the "output" result is the first argument in these latter two cases. Avoid confusing these in the third argument, or data will be ignored.

Copyright Dancing Cloud Services LLC & DevelopIntelligence LLC

The Collector interface

The Collector interface allows constructing a class that encapsulates the three pieces of functionality used in a collect operation.

- It also adds a "finisher" which can convert from the result of the main collection to a different type (for example, converting an Average to a double.
- It also changes the combiner that takes intermediate "per-stream" results and uses a BinaryOperator model, avoiding the chance of passing arguments in the wrong order.
- In addition, it provides "characteristics" which can address whether the finisher should be used, whether the collection considers item order to be significant, and whether the collection uses a threadsafe result object.

Copyright Dancing Cloud Services LLC & DevelopIntelligence LLC

Prebuilt collectors

The Collectors class contains factory methods for a variety of useful collection operations.

Perhaps the most useful are the various overloads of groupingBy

- These collectors are given a function--called a classifier--that creates a key representing the stream item
- The key is the key in a map structure
- The value, in the simplest form of groupingBy is a list of all the stream items that produced that key.
- Overloaded variants of groupingBy support a "downstream" collector which can perform stream-like operations on each item that matches the key.
- Using these forms, the items can be mapped, flatMapped, filtered, and collected into structures other than a list, or reduced to other values, such as the count

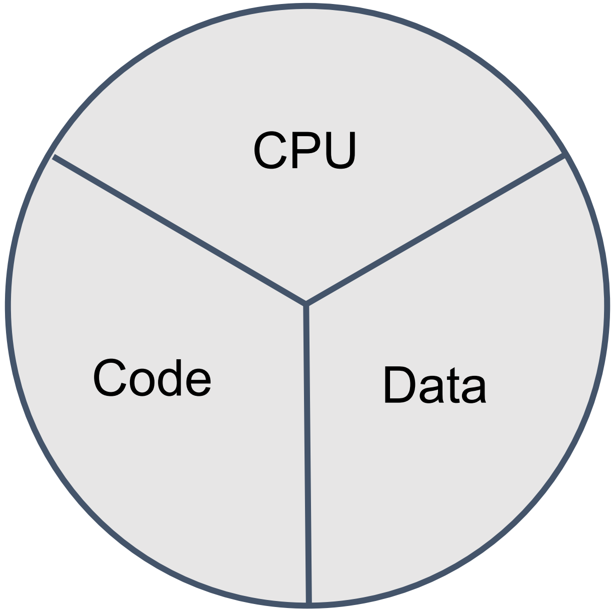
Copyright Dancing Cloud Services LLC & DevelopIntelligence LLC

Additional Stream utilities

- Files.lines can create a Stream directly from a file
- Streams implement AutoCloseable to facilitate proper closing when there is a file as backing store
- Pattern (a regular expression) can split a line of text directly to a Stream
- Other Files utilities can give streams from the file system, such as a stream of Path names

Elements of a thread

Silicon, or a
thread as a "virtual
CPU"



Methods/static methods in objects and classes

Entry point of program:
`public static void main(String[] args)`

Entry point of thread:
`public void run()`
(in an instance of `Runnable`)

Method locals, on the "stack"
or
Objects in the heap

Three problems

1) Visibility
   - Cache and compiler optimizations mean you must ***not*** assume that after one thread writes to a variable, other threads will see the changed data.
   - Do ***not*** assume you understand how the implementation works
2) Transactional integrity
   - read-modify-write cycles
   - partial update of structured data
3) Timing
   - Avoid reading data before it has been published
   - Avoid overwriting data until after other threads have read it
   - Avoid "busy waiting" or "spin waiting" (in the general case)

Copyright Dancing Cloud Services LLC & DevelopIntelligence LLC

Visibility is defined by a happens-before relationship

Two actions can be ordered by a ***happens-before*** relationship. If one action *happens-before* another, then the first is visible to the second.

A happens-before relationship ***does not mean*** "*executes prior*", and if the second action does not observe the effect of the first, any assumptions are likely wrong.

Given two actions x and y, hb(x, y) indicates that x happens-before y.

1. If x and y are actions of the ***same thread*** and x comes before y in ***program order***, then hb(x, y).
2. If hb(x, y) and hb(y, z), then hb(x, z).

Copyright Dancing Cloud Services LLC & DevelopIntelligence LLC

More happens-before relationships

3. If x writes a volatile variable and y is a subsequent read of the same variable, then hb(x,y).
4. If x unlocks a monitor and y is a subsequent lock of the same monitor, then hb(x,y).
5. If x starts a thread and y is the first action of the thread, then hb(x,y)
6. If x is the last action of a thread and y is another thread observing that the first thread is dead, then hb(x,y).
7. If x interrupts another thread, and y is that thread observing the interrupt, then hb(x,y).
8. If x is the last action of a constructor and y is the first action of the finalizer thread acting on the same object, then hb(x,y).

Copyright Dancing Cloud Services LLC & DevelopIntelligence LLC

Producer-consumer concurrency model

The primitives are hard to use reliably, and it's hard to reason about all the potential consequences of decisions in the general case.

If threads are to cooperate, they must share data in some coordinated way.

A BlockingQueue implementing the "producer-consumer" model provides an architecture that solves all three concurrency problems in a way that is easy to understand implement correctly. Note that the approach has several good architectural qualities, but is often not absolute most efficient.

Copyright Dancing Cloud Services LLC & DevelopIntelligence LLC

Producer-consumer essentials

The model can be likened to a production line. At any given instant the work product is either wholly owned by one worker thread, or is inaccessible to all threads and is "in transit" down the production line.

One thread, called the producer prepares data that is wholly owned by that thread (ensure that no thread-shared data are used in this). Because the data are confined to one thread, no concurrency problem can exist.

At the moment the producer has completed its work, it puts the data into a queue and nulls-out its reference to it. At this point, the data is inaccessible to all threads.

Later, another thread (the consumer) takes the data item from the queue. The data are again confined to a single thread, so no concurrency problems can exist.

Copyright Dancing Cloud Services LLC & DevelopIntelligence LLC

Behavior of a `BlockingQueue` producer-consumer

When a producer-consumer architecture is implemented correctly using a BlockingQueue the system exhibits certain important properties:

- An object inserted into the queue will be taken by exactly one thread (unless it remains in the queue because nothing tries to take it). This is true even in the face of multiple concurrent producers and consumers.
- If inserting the object into the queue is action x, and taking it from the queue is action y, then hb(x,y). So, *the consumer is guaranteed proper visibility of the data in the object*.

Copyright Dancing Cloud Services LLC & DevelopIntelligence LLC

Behavior of a `BlockingQueue` producer-consumer

- If a thread attempts to put an item into the queue when it is full, it will be descheduled and wait for space to be available.
- If a thread attempts to take from an empty queue it will be descheduled and wait for available data.
- Read data items become the exclusive property of the consumer and are not overwritten.
- The combined effect is that *all timing concerns are handled by this architecture*.

Behavior of a `BlockingQueue` producer-consumer

- Data are neither duplicated nor lost (so long as consumers continue to run)
- Items from a single producer will be removed from the queue in the order they were added (though this might be hidden if multiple consumers are running)
- Data are only ever visible to a single thread at any one time, all updates must be completed before the data is transferred through the queue to another thread.
- Because changes are completed entirely in one thread, *all transactional concerns are handled by the architecture*.

The ExecutorService

Creating threads in current versions of Java is expensive, because a java.lang.Thread object maps to an operating system thread.

Using a thread for a single, short-lived, task is wasteful

Instead, a thread can read tasks from a BlockingQueue, execute the task and then loop round to get another task.

ExecutorService is an interface wrapped around this idea, and the core Java libraries provide several features of this kind.

The class Executors provides a variety of commonly used types of ExecutorService implementation with differing queue and thread-count behaviors

Copyright Dancing Cloud Services LLC & DevelopIntelligence LLC

Using an ExecutorService with Runnable

Create the service using, for example

```
es = Executors.newFixedThreadPool(4)
```

submit a Runnable object for execution:

```
es.execute(myRunnable)
```

Runnable's run() method returns void and cannot throw checked exceptions

Copyright Dancing Cloud Services LLC & DevelopIntelligence LLC

Using Callable

The Callable interface is generic:

```
interface Callable<T> {
  T call() throws Exception;
}
```

This allows defining a task (in a fashion similar to a Runnable)

However, the task can return a value, and could throw a checked exception

These "results" must be communicated back to the task creator

Copyright Dancing Cloud Services LLC & DevelopIntelligence LLC

Submitting a Callable

The ExecutorService provides a submit method which accepts a Callable

It returns an instance of Future, which serves as a "handle" on the job

The handle can be used to:

- determine if the job is complete
- request cancelation
- obtain the results of the job when complete

Copyright Dancing Cloud Services LLC & DevelopIntelligence LLC

Submitting a Callable

```
Callable<String> myJob = …
ExecutorService es = ....
Future<String> handle = es.submit(myJob);
if (handle.isDone()) {
  try { // get will block if job is not done
    System.out.println("result: " + handle.get());
  } catch (ExecutionException ee) {
    System.out.println("Job threw: " + ee.getCause();
  } // other exceptions are possible from infrastructure
}
```

Terminating a task

A task should never be killed (Thread.stop() is deprecated)

Doing so might leave things in inconsistent state

Instead, send a "please shut down" request.

This is conventionally done with the Thread.interrupt()

The interrupt is effectively a boolean flag visible to the target thread

Interrupt will cause blocking methods to unblock with and throw InterruptedException some very old methods have bugs in this respect

Can also poll the Thread.interrupted() method

Respond to the interrupt by shutting down tidily

Copyright Dancing Cloud Services LLC & DevelopIntelligence LLC

Canceling a task in an ExecutorService

A Future can request cancelation of a job

- If the job has not started, it is removed from the input queue
- A job that is running can optionally be sent sent an interrupt

Write long-running jobs so they will respond to interrupt by shutting down promptly, otherwise your servers might not shut-down cleanly

Concurrent utilities

Concurrent utilities typically provide thread safety, high scalability, or perhaps both.

Be sure which type of behavior a utility provides and use it appropriately.

Prefer library provided utilities over home-brewed ones; it's far too easy to make a small mistake, and debugging concurrency problems is exceptionally hard, at least in part because they tend to be non-deterministic (not repeatable) in nature.

Copyright Dancing Cloud Services LLC & DevelopIntelligence LLC

Atomic types

Atomic types provide indivisible read-modify-write cycles at a library API level.

The package `java.util.concurrent.atomic` is "lock free" which ais to provide very high scalability.

A few sample operations on `AtomicInteger` illustrates general concepts. Most create a *happens-before* relationship as though the variable were `volatile`:

- `get()` : returns the current value
- `addAndGet(int x)` : add `x` to the current value, and return the updated value.
- decrementAndGet() : reduce value by 1 and return the result.

Atomic array types

Provide atomic operations along with *happens-before* relationships on array elements.

The array may be created with a length, or by duplicating an existing array.

Example operation:

```
int accumulateAndGet(int i, int x, IntBinaryOperator accFn)
```

Atomically updates `array[i]` with the result of `accFn.apply(array[i], x)`

Accumulators

If a value is subject to concurrent updates, but very rare writes, and the update operations are entirely independent of one another (for example, simple increments) then it's possible to gain scalability by giving each thread a thread-local variable, so it can have unrestricted updates.

On reading the multiple values must be collected, aggregated, and returned.

Initialize an accumulator with a binary operator suited to the data type. This is used to apply the updates.

The operation should be commutative, associative, and free of side-effects, or the behavior of the accumulator will likely be unpredictable.

Copyright Dancing Cloud Services LLC & DevelopIntelligence LLC

Concurrent data structures

These data structures provide thread safety, but are primarily focused on scalability, minimizing or eliminating locks in their operation.

`ConcurrentHashMap`, `ConcurrentSkipListMap`, `ConcurrentSkipListSet`, `CopyOnWriteArrayList`, and `CopyOnWriteArraySet`

ConcurrentHashMap locks only one "bucket" of the map rather than the entire map.

When significant contention is expected concurrent data structures are preferred, for non-contented concurrent situations, synchronized structures are likely better. Entirely single-threaded access favors normal structures.

Copyright Dancing Cloud Services LLC & DevelopIntelligence LLC

CopyOnWriteArrayList

This structure is intended for heavily concurrent reading with occasional updates.

Read operations are lock free, however writing is accomplished in two steps:

- Read and duplicate the entire array (this is clearly expensive)
- Modify the copy
- Redirect all reads to the new version allowing the old array to be garbage collectable.

Synchronized structures

If contention is rarely expected, a simple lock is likely the best approach.

The `Collections` class provides static factory methods for proxy objects that wrap around the main interfaces of the collections API, such that access through that proxy is serialized using simple locking.

E.g.

```
List<String> syncList =
  Collections.synchronizedList(new ArrayList<String>());
```

Copyright Dancing Cloud Services LLC & DevelopIntelligence LLC

Java's `CompletableFuture` API

Java's CompletableFuture implements the popular "Promise Pipeline" concept.

Promises use a monadic approach that allows us to define the computation that should be applied to input data, and to send the result to another processing stage, but allows the computation to be performed in another thread, often taken from a pool.

In particular, long running processes that support a "callback" style completion can be supported directly without the need for hard-to-maintain nested blocks.

Promise pipelines differ from normal monads in that they provide two data paths, one for success results, and another for exception or failure propagation, along with recovery mechanisms to get back onto the "happy path".

Copyright Dancing Cloud Services LLC & DevelopIntelligence LLC

Starting the pipeline

Unlike a Stream, a `CompletableFuture` processes a single data item, it's also a single use structure.

Pipeline might start with an API operation that returns a `CompletableFuture`, or with a "supply" method:

```
CompletableFuture.supplyAsync(Supplier<U> supplier)
```

Methods with the suffix "`async`" cause the work to be performed by a thread from a pool. Those without are executed in the thread that completes the `CompletableFuture`.

By default, the pool is the `ForkJoinPool.commonPool()`, but can be a caller specified pool.

Operations on the `CompletableFuture`

An operation equivalent to a monadic "map" as `thenApply` / `thenApplyAsync`

There is no equivalent of "filter"

A "forEach" method is available as `thenAccept` / `thenAcceptAsync` (remember only one data item runs down a single `CompletableFuture`)

The "flatMap" method is called `thenCompose` / `thenComposeAsync`. Remember that the function provided to a flatMap must return a monad of the same type. This method is how we chain functions that complete asynchronously after immediately returning a `CompletableFuture`.

Copyright Dancing Cloud Services LLC & DevelopIntelligence LLC

Teeing pipelines

CompletableFuture pipelines can have as many subsequent pipelines as we desire, and the data produced will be propagated down each.

```
CompletableFuture<String> cfs =
   CompletableFuture.supplyAsync(()->"Hello");
cfs.thenAcceptAsync(
     m -> System.out.println("Message is " + m));
cfs
   .thenApplyAsync(m -> m.toUpperCase())
   .thenAcceptAsync(System.out::println);
```

Rejoining pipelines

Multiple CompletableFutures can be collected together allowing combining of their output data. It's also possible to select the first to complete successfully:

```
cf1.acceptEither(cf2, consumer)
cf1.thenAcceptBoth(cf2, biConsumer)
cf1.applyToEither(cf2, function)
cf1.thenCombine(cf2, biFunction)
cf1.runAfterEither(cf2, runnable)
```

Copyright Dancing Cloud Services LLC & DevelopIntelligence LLC

Handling exceptions

Java's functional interfaces do not permit throwing checked exceptions, but unchecked exceptions can be thrown and cause the pipeline to jump to the second pipeline.

Normal pipeline elements will be skipped over by an exception, but the first handler will be invoked.

```
cf.handle(BiFunction<E, Throwable, R> fn)
```

fn is called with either a value, or an exception. The "missing" element is a null value. The result continues down the pipeline.

Copyright Dancing Cloud Services LLC & DevelopIntelligence LLC

Using `thenCompose`

If a utility method returns a CompletableFuture, it runs asynchronously, likely using a callback type approach internally. Include it in the CompletableFuture pipeline with the method thenCompose:

```
cf1.thenCompose(x ->
functionReturningCompletableFuture(x))
```

Wrapping callback functions

Given a callback-style asynchronous behavior, it can be wrapped to work with a CompletableFuture pipeline.

Assuming an asynchronous method makes a callback to some function fn, we can wrap this in a function that returns a CompletableFuture by a wrapper function that immediately returns a new CompletableFuture, and then calls the async operation with a callback that completes the CompletableFuture, either normally, or with an exception.

Wrapping a callback to a `CompletableFuture`

Simulated callback function:

```
public static void callback(String s, Consumer<String> op) {
  new Thread(() -> {
    try {
      Thread.sleep(1000 + (int)(Math.random() * 1000));
      op.accept(s.toUpperCase());
    } catch (InterruptedException ie) {}
  }).start();
}
```

Wrapping a callback to a `CompletableFuture`

Wrapping function:

```
public static CompletableFuture<String> doAsync(String x) {
   CompletableFuture<String> cfs = new CompletableFuture<>();
   callback(x, s -> cfs.complete(s));
   return cfs;
}
```

The callback lambda can, if needed, handle exceptions by calling `cfs.completeExceptionally(exception)`

Wrapping a callback to a `CompletableFuture`

Calling the function that returns a CompletableFuture in a pipeline:

```
CompletableFuture.supplyAsync(() -> "hello")
        .thenApplyAsync(x -> x + " world!")
        .thenComposeAsync(x -> doAsync(x))
        .thenAcceptAsync(System.out::println)
        .join();
```

The `thenComposeAsync` method can be called with any method that returns a `CompletableFuture`, so when APIs are available that are built around `CompletableFuture`, this will be a key way to build code in those APIs.

Creating annotations

Annotations:

- Behave like "sticky notes", they can be attached to syntax elements, and can carry data "attributes", but do not have any behavior
- Are defined by a special declaration "@interface"
- Can be annotated with a RetentionPolicy (SOURCE, CLASS, or RUNTIME) that determines how long the annotation "sticks"
- Can be annotated with a Target to indicate which syntax elements they can be applied to
- Can declare methods as accessors for the data attributes they carry
- Can have default values for those attributes
- Attribute types can only be primitive, String, Class, annotation, and arrays of the preceding (but not multi-dimensional arrays).

Example annotation declaration

```
@Target({ElementType.METHOD, ElementType.CONSTRUCTOR})
@Retention(RetentionPolicy.RUNTIME)
public @interface MyAnnotation {
  String value() default "unknown";
  int count();
}
```

Applying annotations

The annotation is prefixed with @ and the combination precedes a suitable target syntax element

If there are data attributes declared for the annotation for which no default is defined, they must have a specified value

Value of an attribute is generally specified as a key=value pair

Multiple attributes are specified in a comma separated list

Multiple values of an array type are enclosed in {}, but an array attribute with a single element my omit the curly braces

An attribute with the special name "value" can be specified without the key= part provided that it is the only attribute that is specified in the source

Copyright Dancing Cloud Services LLC & DevelopIntelligence LLC

Example annotations

- `@NoAttributeAnnotation`
- `@AnnotationWithOnlyValue("Use this string for value")`
- `@AnnotationWithSeveralAttributes(name="Alf", count=3)`
- `@ArrayValue({1, 2})`
- `@ArrayValueWithSingleElement(1)`
- `@MoreAttributes(names={"Fred", "Jim"}, count=2)`

Copyright Dancing Cloud Services LLC & DevelopIntelligence LLC

Finding annotations at runtime

Obtain the java.lang.Class object that defines the object that might be annotated:

```
Class<?> clazz = targetObject.getClass();
```

Get the elements that might carry the annotation:

```
Method[] methods = clazz.getDeclaredMethods();
```

Look for annotation "MyAnno"

```
for (Method m : methods) {
  MyAnno annot = m.getAnnotation(MyAnno.class);
  if (annot != null) // found annotation
}
```

Reading attributes from the annotation

Given:

```
@interface MyAnnot { String value(); }
```

And:

```
@MyAnnot("Excellent") public void doStuff(){}
```

```
MyAnnot annot = … // find above annotation
String value = annot.value(); // returns "Excellent"
```

Copyright Dancing Cloud Services LLC & DevelopIntelligence LLC