MINUTES TO SECONDS, MAXIMIZING INCREMENTALITY

Cédric Champeau (@CedricChampeau), Gradle

WHO AM I

```
speaker {
    name 'Cédric Champeau'
    company 'Gradle Inc'
    oss 'Apache Groovy committer',
    successes 'Static type checker',
              'Static compilation',
              'Traits',
              'Markup template engine',
              'DSLs'
        failures Stream.of(bugs),
        twitter '@CedricChampeau',
        github 'melix',
        extraDescription '''Groovy in Action 2 co-author
Misc OSS contribs (Gradle plugins, deck2pdf, jlangdetect, ...)'''
}
```

Happy

AGENDA

- Incremental builds
- Compile avoidance
- Incremental compilation

INCREMENTAL BUILDS

- Gradle is meant for incremental builds
- `clean` is a waste of time
- So declare your inputs/outputs!

EXAMPLE: BUILDING A SHADED JAR

```
task shadedJar(type: ShadedJar) {
   jarFile = file("$buildDir/libs/shaded.jar")
   classpath = configurations.runtime
   mapping = ['org.apache': 'shaded.org.apache']
}
```

- What are the task inputs?
- What are the task outputs?
- What if one of them changes?

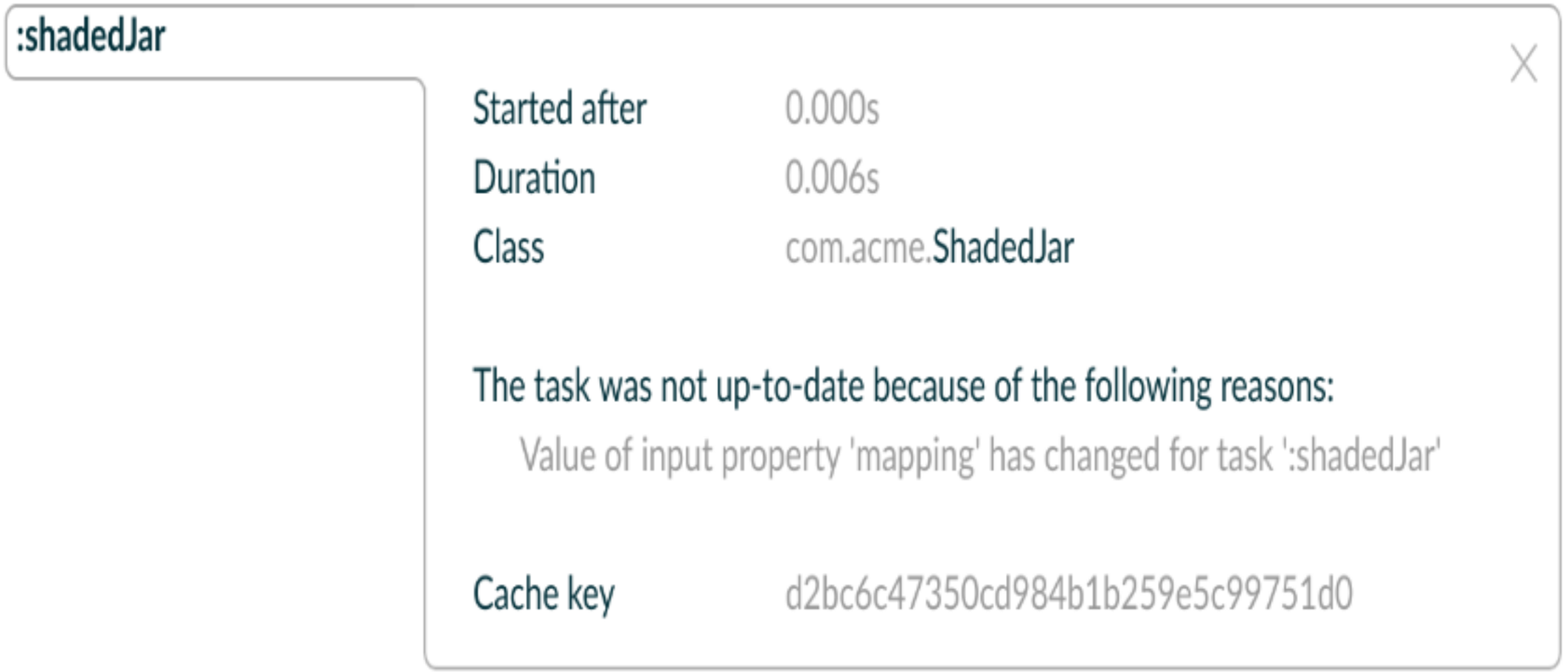
DECLARING INPUTS

```
public class ShadedJar extends DefaultTask {
    ...
    @InputFiles
    FileCollection getClasspath() { ... }

    @Input
    Map<String, String> getMapping() { ... }
}
```

DECLARING OUTPUTS

```
public class ShadedJar extends DefaultTask {
    ...

    @OutputFile
    File getJarFile() { ... }
}
```

KNOW WHY YOUR TASK IS OUT-OF-DATE



INCREMENTAL TASK INPUTS

- Know precisely *which* files have changed
- Task action can perform the minimal amount of work

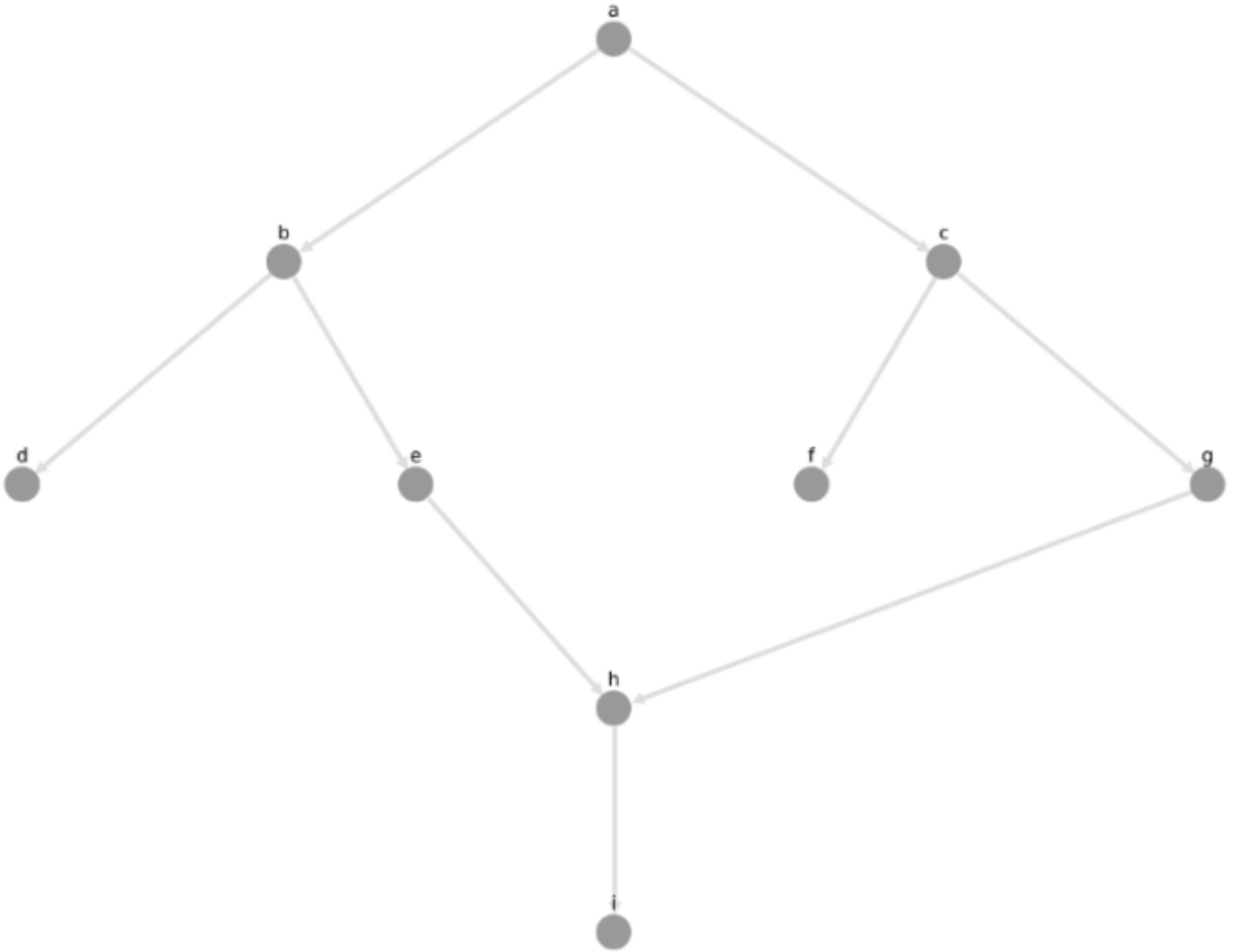
INCREMENTAL TASK INPUTS

```
@TaskAction
public void execute(IncrementalTaskInputs inputs) {
   if (!inputs.isIncremental()) {
      // clean build, for example
      // ...
   } else {
      inputs.outOfDate(change ->
         if (change.isAdded()) {
            ...
         } else if (change.isRemoved()) {
            ...
         } else {
            ...
         }
      });

   }
}
```

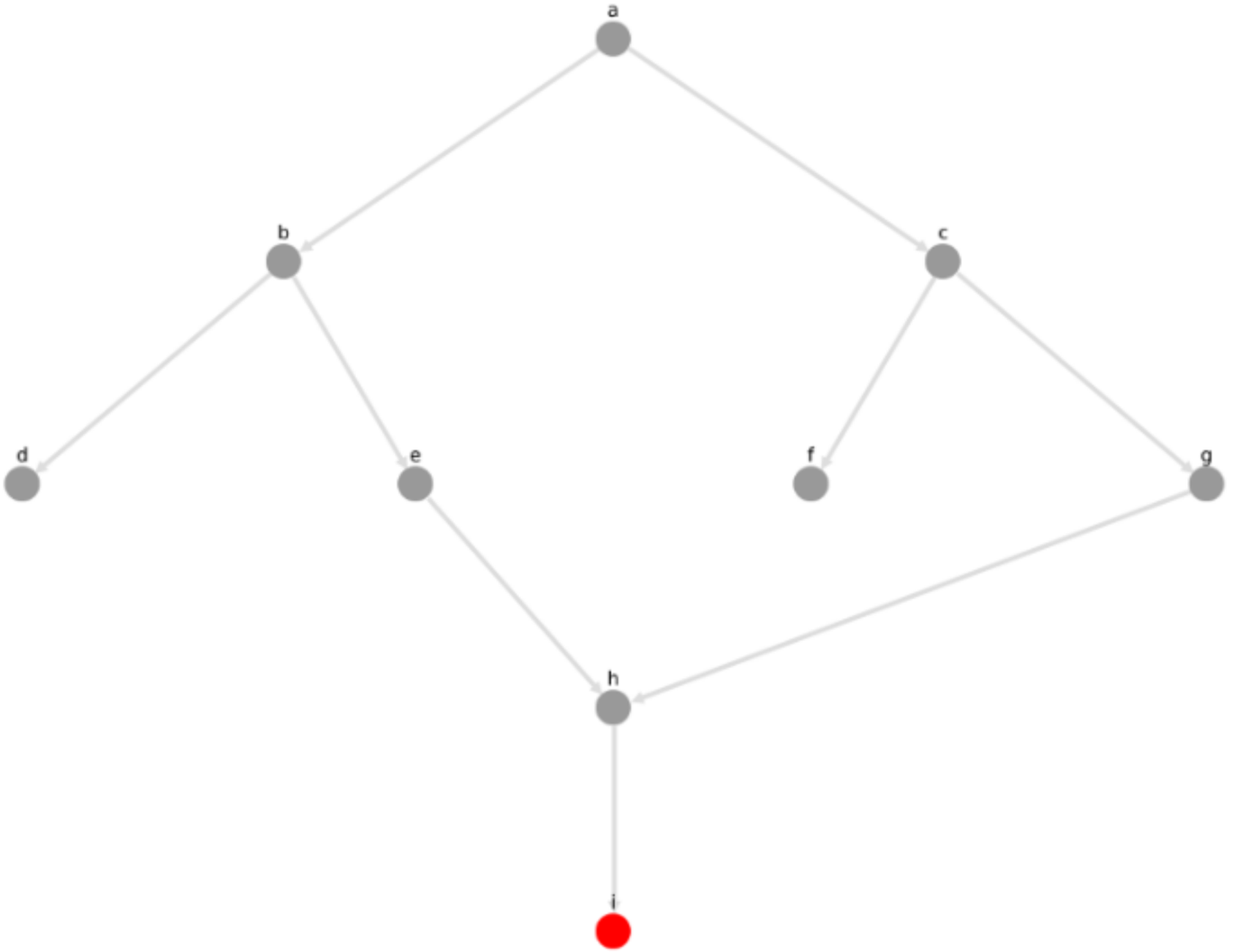
COMPILE AVOIDANCE

COMPILE CLASSPATH LEAKAGE

A TYPICAL DEPENDENCY GRAPH



CASCADING RECOMPILATION



SEPARATING API AND IMPLEMENTATION

EXAMPLE

```
import com.acme.model.Person;
import com.google.common.collect.ImmutableSet;
import com.google.common.collect.Iterables;

...

public Set<String> getNames(Set<Person> persons) {
   return ImmutableSet.copyOf(Iterables.transform(persons, TO_NAME))
}
```

BEFORE GRADLE 3.4

```
apply plugin: 'java'

dependencies {
   compile project(':model')
   compile 'com.google.guava:guava:18.0'
}
```

BUT...

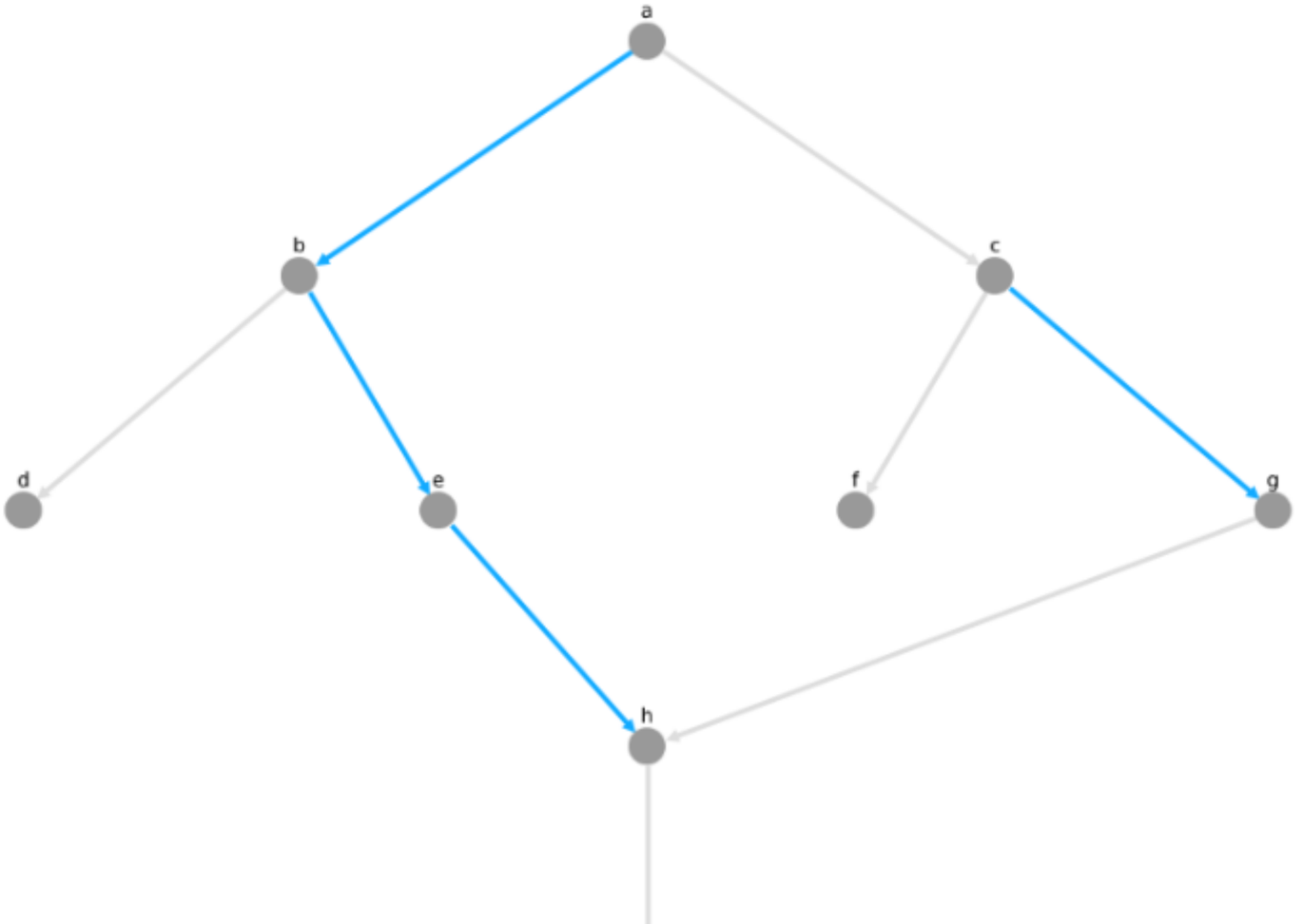
```
import com.acme.model.Person; // exported dependency
import com.google.common.collect.ImmutableSet; // internal dependency
import com.google.common.collect.Iterables; // internal dependency

...

public Set<String> getNames(Set<Person> persons) {
   return ImmutableSet.copyOf(Iterables.transform(persons, TO_NAME))
}
```

STARTING FROM GRADLE 3.4

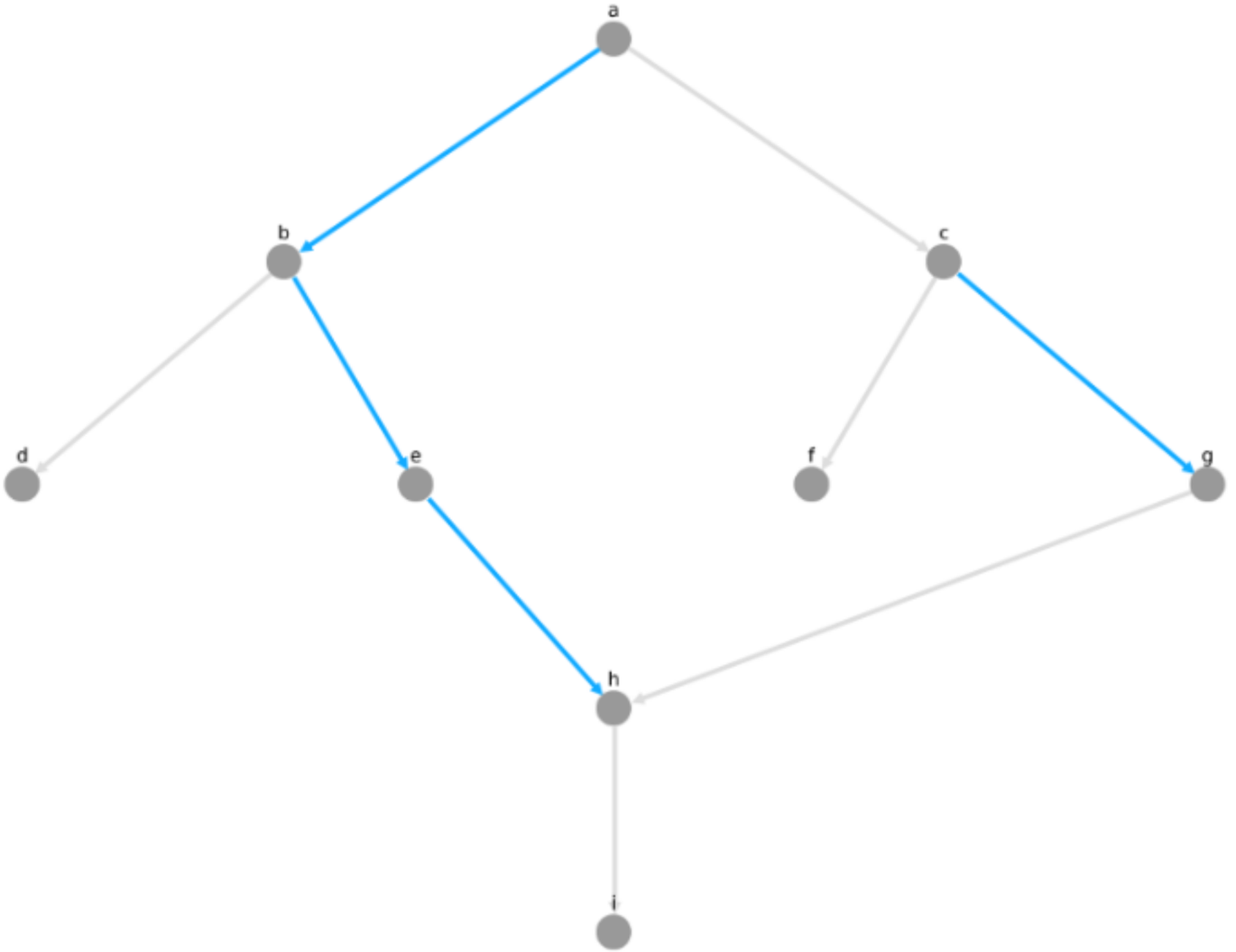
```
apply plugin: 'java-library' // This component has an API and an implementation

dependencies {
   api project(':model')
   implementation 'com.google.guava:guava:18.0'
}
```

CONSEQUENCES ON CASCADING

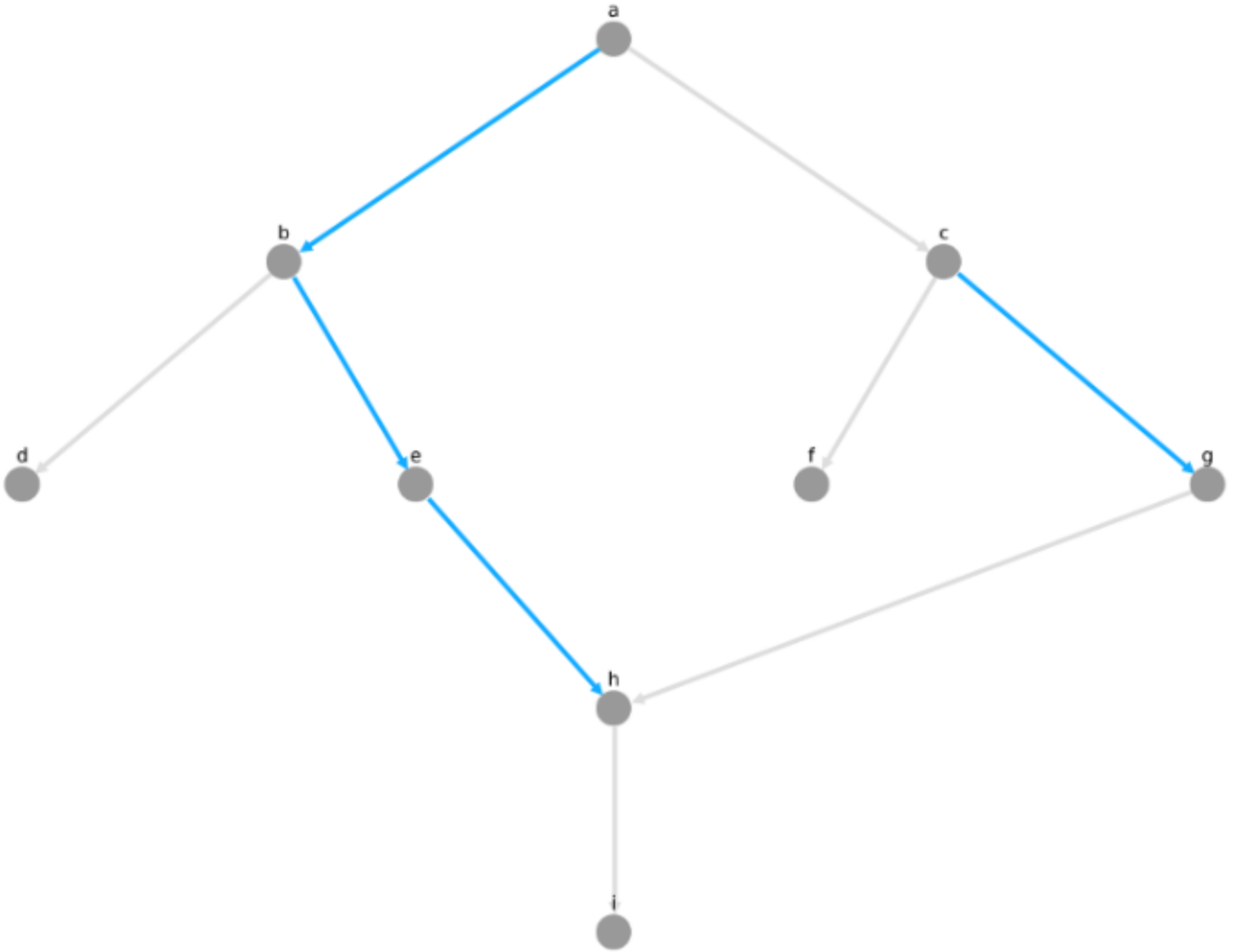
- blue : API dependences
- grey : implementation dependencies



CHANGING AN IMPLEMENTATION DEPENDENCY



CHANGING AN API DEPENDENCY



CLASSPATH SNAPSHOTTING

CLASSPATH SNAPSHOTTING

- Compute a hash of inputs

CLASSPATH SNAPSHOTTING

- Compute a hash of inputs
- If hash hasn't changed, task is up-to-date

CLASSPATH SNAPSHOTTING

- Compute a hash of inputs
- If hash hasn't changed, task is up-to-date
- Is a compile classpath equivalent to runtime classpath?

COMPILE CLASSPATH

What does a **compiler** care about?

COMPILE CLASSPATH

What does a **compiler** care about?

- Input: jars, or class directories

COMPILE CLASSPATH

What does a **compiler** care about?

- Input: jars, or class directories
- Jar: class files

COMPILE CLASSPATH

What does a **compiler** care about?

- Input: jars, or class directories
- Jar: class files
- Class file: both API and implementation

COMPILE CLASSPATH

What we provide to the compiler

```
public class Foo {
    private int x = 123;

    public int getX() { return x; }
    public int getSquaredX() { return x * x; }
}
```

COMPILE CLASSPATH

What the compiler cares about:

```
public class Foo {
    public int getX()
    public int getSquaredX()
}
```

COMPILE CLASSPATH

But it could also be

```
public class Foo {
    public int getSquaredX()
    public int getX()
}
```

only public signatures matter

COMPILE CLASSPATH

COMPILE CLASSPATH

- Compute a hash of the signature of class : **`aedb00fd`**

COMPILE CLASSPATH

- Compute a hash of the signature of class : **`aedb00fd`**
- Combine hashes of all classes : **`e45bdc17`**

COMPILE CLASSPATH

- Compute a hash of the signature of class : **`aedb00fd`**
- Combine hashes of all classes : **`e45bdc17`**
- Combine hashes of all input on classpath: **`4500fc1`**

COMPILE CLASSPATH

- Compute a hash of the signature of class : **`aedb00fd`**
- Combine hashes of all classes : **`e45bdc17`**
- Combine hashes of all input on classpath: **`4500fc1`**
- Result: hash of the compile classpath

COMPILE CLASSPATH

- Compute a hash of the signature of class : `aedb00fd`
- Combine hashes of all classes : `e45bdc17`
- Combine hashes of all input on classpath: `4500fc1`
- Result: hash of the compile classpath
- Only consists of what is *relevant* to the `javac` compiler

RUNTIME CLASSPATH

What does the runtime care about?

RUNTIME CLASSPATH

What does the runtime care about:

```
public class Foo {
    private int x = 123;

    public int getX() { return x; }
    public int getSquaredX() { return x * x; }
}
```

At runtime, **everything** matters, from classes to resources.

COMPILE VS RUNTIME CLASSPATH

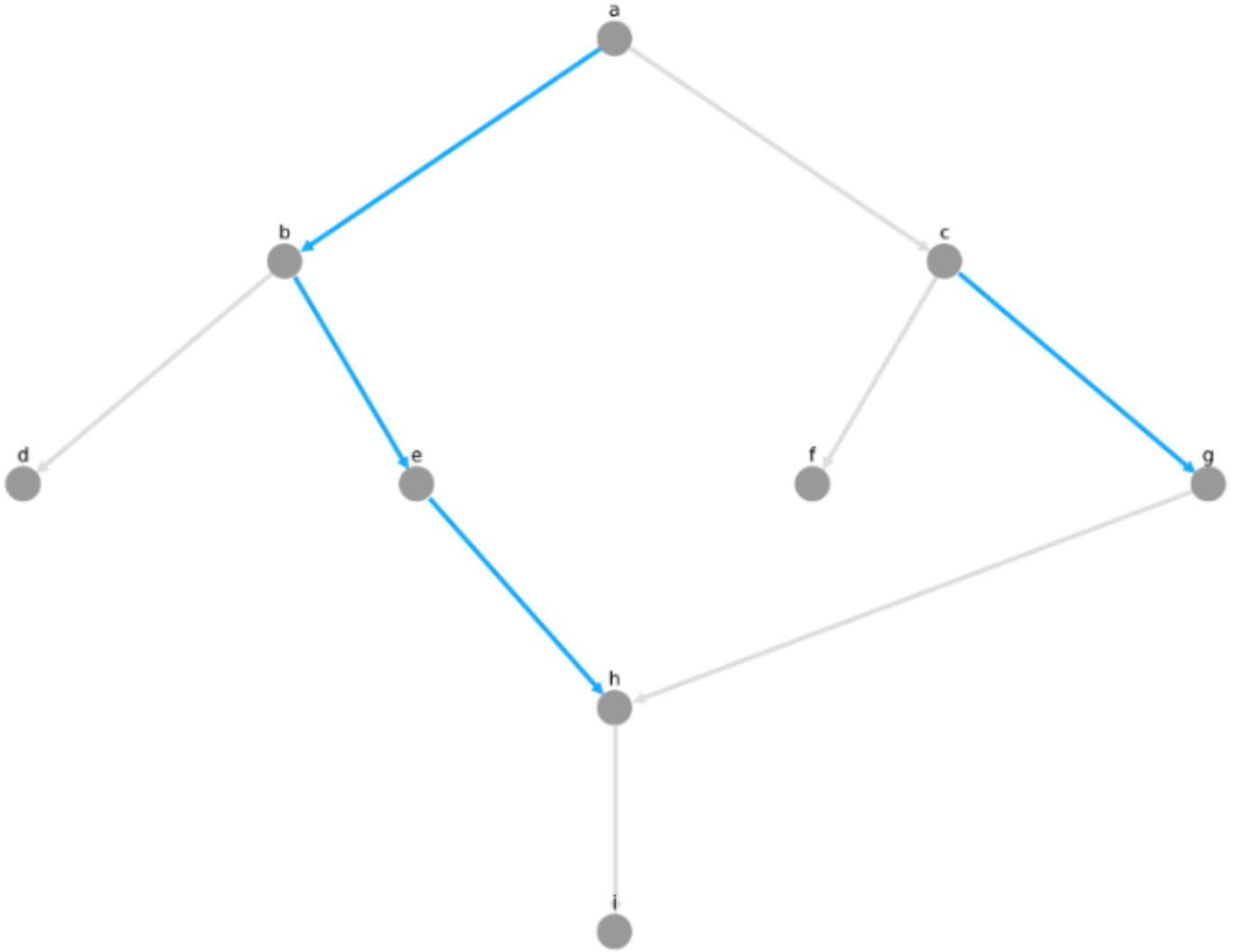
In practice:

```
@InputFiles
@CompileClasspath
FileCollection getCompileClasspath() { ... }

@InputFiles
@Classpath
FileCollection getRuntimeClasspath() { ... }
```

COMPILE AVOIDANCE

- Gradle makes the difference
- Ignores irrelevant (non ABI) changes to compile classpath

CONSEQUENCES ON CASCADING RECOMPILATIONS



ICING ON THE CAKE

- Upgrade a dependency from `1.0.1` to `1.0.2`
- If ABI hasn't changed, Gradle will *not* recompile
- Even if the name of the jar is different (`mydep-1.0.1.jar` vs `mydep-1.0.2.jar`)
- Because only *contents* matter

INCREMENTAL COMPILATION

- Given a set of source files
- Only compile the files which have changed…
- and their dependencies
- Language specific

GRADLE HAS SUPPORT FOR INCREMENTAL COMPILATION OF JAVA

```
compileJava {
    //enable incremental compilation
    options.incremental = true
}
```

- Kotlin plugin implements its own incremental compilation

IN PRACTICE

```
import org.apache.commons.math3.complex.Complex;

public class Library {
    public Complex someLibraryMethod() {
        return Complex.I;
    }
}
```

IN PRACTICE

```
import org.apache.commons.math3.complex.Complex;

public class Library {
    public Complex someLibraryMethod() {
        return Complex.I;
    }
}
```

- **`Complex`** is a dependency of **`Library`**

IN PRACTICE

```
import org.apache.commons.math3.complex.Complex;

public class Library {
    public Complex someLibraryMethod() {
        return Complex.I;
    }
}
```

- **`Complex`** is a dependency of **`Library`**
- if **`Complex`** is changed, we need to recompile **`Library`**

IN PRACTICE

```
import org.apache.commons.math3.complex.Complex;

public class Library {
    public Complex someLibraryMethod() {
        return Complex.I;
    }
}
```

- **`Complex`** is a dependency of **`Library`**
- if **`Complex`** is changed, we need to recompile **`Library`**
- if **`ComplexUtils`** is changed, no need to recompile

GOTCHA

```
import org.apache.commons.math3.dfp.Dfp;

public class LibraryUtils {
   public static int getMaxExp() {
      return Dfp.MAX_EXP;
   }
}
```

GOTCHA

```
import org.apache.commons.math3.dfp.Dfp;

public class LibraryUtils {
   public static int getMaxExp() {
      return Dfp.MAX_EXP;
   }
}
```

- **`Dfp`** is a dependency of **`LibraryUtils`**

GOTCHA

```
import org.apache.commons.math3.dfp.Dfp;

public class LibraryUtils {
   public static int getMaxExp() {
      return Dfp.MAX_EXP;
   }
}
```

- **`Dfp`** is a dependency of **`LibraryUtils`**
- so if **`MAX_EXP`** changes, we should recompile **`LibraryUtils`**, right?

WAIT A MINUTE...

```
javap -v build/classes/java/main/LibraryUtils.class
```

```
...
  public static int getMaxExp();
    descriptor: ()I
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
      stack=1, locals=0, args_size=0
         0: ldc           #3                  // int 32768
         2: ireturn
```

- reference to `Dfp` is gone!
- compiler *inlines* some constants
- JLS says compiler doesn't have to add the dependent class to constant pool

WHAT GRADLE DOES

WHAT GRADLE DOES

- Analyze all *bytecode* of all classes

WHAT GRADLE DOES

- Analyze all *bytecode* of all classes
- Record which constants are used in which file

WHAT GRADLE DOES

- Analyze all *bytecode* of all classes
- Record which constants are used in which file
- Whenever a producer changes, check if a *constant* changed

WHAT GRADLE DOES

- Analyze all *bytecode* of all classes
- Record which constants are used in which file
- Whenever a producer changes, check if a *constant* changed
- If yes, recompile *everything*

VARIANT AWARE DEPENDENCY MANAGEMENT

PRODUCER VS CONSUMER

PRODUCER VS CONSUMER

- A **`consumer`** *depends on* a **`producer`**

PRODUCER VS CONSUMER

- A `consumer` *depends on* a `producer`
- There are multiple requirements
  - What is required to compile against a `producer`?
  - What is required at *runtime* for a specific configuration?
  - What artifacts does the producer offer?
  - Is the `producer` a sub-project or an external component?

WHAT DO YOU NEED TO COMPILE AGAINST A COMPONENT?

- Class files
- Can be found in different forms:
  - class directories
  - jars
  - aars, ...

Question: do we need to build a jar of the producer if all we want is to compile against it?

DISCRIMINATE THANKS TO *USAGE*

> *Give me something that I can use to compile*
>
> — Consumer

DISCRIMINATE THANKS TO *USAGE*

> *Sure, here's a jar*
>
> — Producer

DISCRIMINATE THANKS TO *USAGE*

But we can be finer:

> *Sure, here's a class directory*
>
> — Producer

DISCRIMINATE THANKS TO *USAGE*

Or smarter:

> *mmm, all I have is an AAR, but don't worry, I know how to transform it to something you can use for compile*
>
> — Producer

THE JAVA LIBRARY PLUGIN

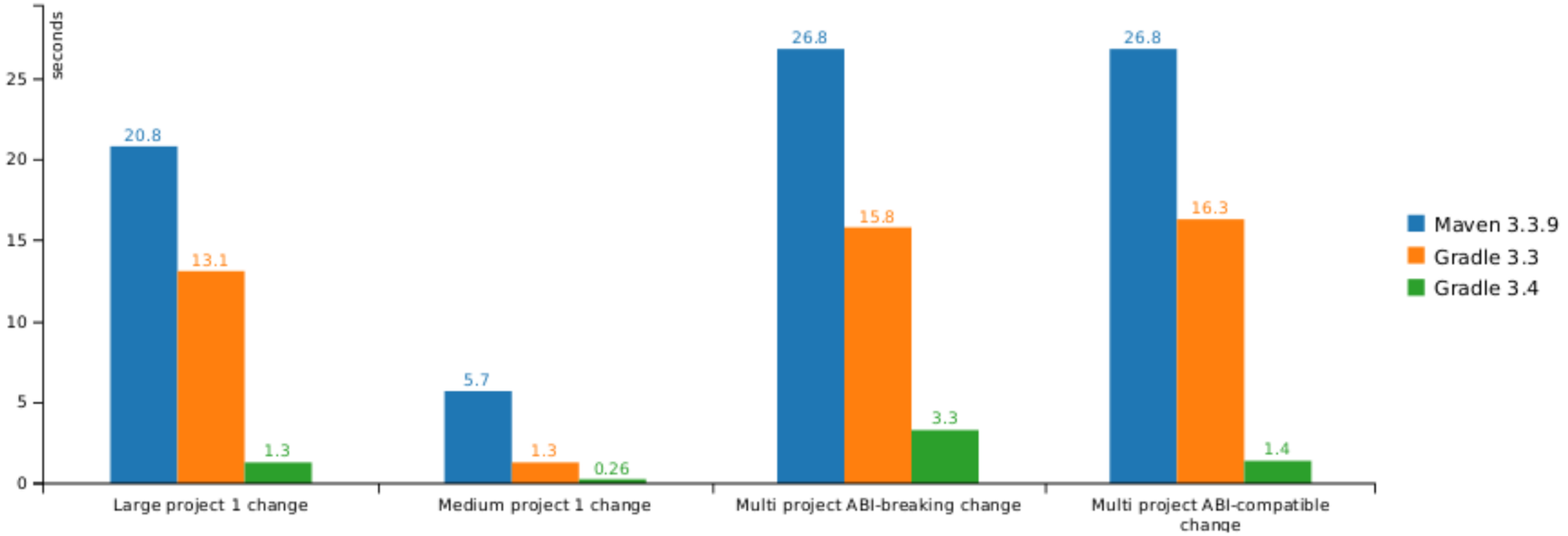
- will provide consumers with a *class directory* for compile
- will provide consumers with a *jar* for runtime

As a consequence:

- only `classes` task will be triggerred when compiling
- `jar` (and therefore `processResources`) only triggerred when needed at runtime

CONCLUSION

Use the Java Library Plugin!



Thank you

Gradle Summit 2017