Byte code field report

or

Why we still heavily rely on Unsafe even in Java 13

What background do I base this talk/opinion based on?

In what areas are instrumentation and dynamic subclassing mainly used?

	behaviour changing	**behavior enhancing**
dynamic subclass	mocking e.g. Mockito	persistence proxy e.g. Hibernate
retransformation	security e.g. Sqreen	APM/tracing e.g. Instana

How to define and change byte code at runtime?

```
interface Instrumentation { // since Java 5
  void addClassFileTransformer(ClassFileTransformer cft);
}
```

```
class Unsafe { // since Java 11
  Class<?> defineClass(String name, byte[] bytes, ...) { ... }
}
```

```
class MethodHandle { // since Java 9
  Class<?> defineClass(byte[] bytes) { ... }
}
```

```
package jdk.internal;
class Unsafe { // since Java 9
  Class<?> defineClass(String name, byte[] bytes, ...) { ... }
}
```

Defining classes from Java agents

~~Transforming classes from Java agents~~

Defining classes from libraries

Transforming classes from libraries

Miscellaneous

Java agents also need to define classes.

```
class Sample {
  void method() {
    Api.invoke(new Callback() {
      @Override
      void callback() {
        System.out.println("called back");
      }
    });
  }
}

abstract class Callback {
  abstract void callback();
}

class Api {
  static void invoke(Callback c) { ... }
}
```

API enhancement proposal: JDK-8200559

```
interface ClassFileTransformer {

  interface ClassDefiner {
    Class<?> defineClass(byte[] bytes);
  }

  byte[] transform(
    ClassDefiner classDefiner,  // restricted to package of transformed class
    Module module,
    ClassLoader loader,
    String className,
    Class<?> classBeingRedefined,
    ProtectionDomain protectionDomain,
    byte[] classfileBuffer
  ) throws IllegalClassFormatException;
}
```

Injected classes with multiple use sites.

```
class Sender {
  void send(Receiver receiver) {
    Framework.sendAsync(receiver, new TaggedEvent());
  }
}

class Receiver {
  void receive(Event event) {
    if (event instanceof TaggedEvent) {
      TrackingAgent.record(((TaggedEvent) event).timestamp);
    }
    System.out.println(event);
  }
}

class TaggedEvent extends Event {
  long time = currentTimeMillis();
}
```

Why JDK-8200559 does not cover all use cases.

- A Java agent cannot rely on a given class loading order.
- The TaggedEvent class must be defined in the package of the first class being loaded.

```
class sender.Sender                    class receiver.Receiver
class sender.TaggedEvent               class receiver.TaggedEvent
class receiver.Receiver                class sender.Sender
```

- The mediator class might need to be defined in a different package to begin with.

```
package event;
public class Event {
  /* package-private */ void overridable() {
    // default behavior
  }
}
```

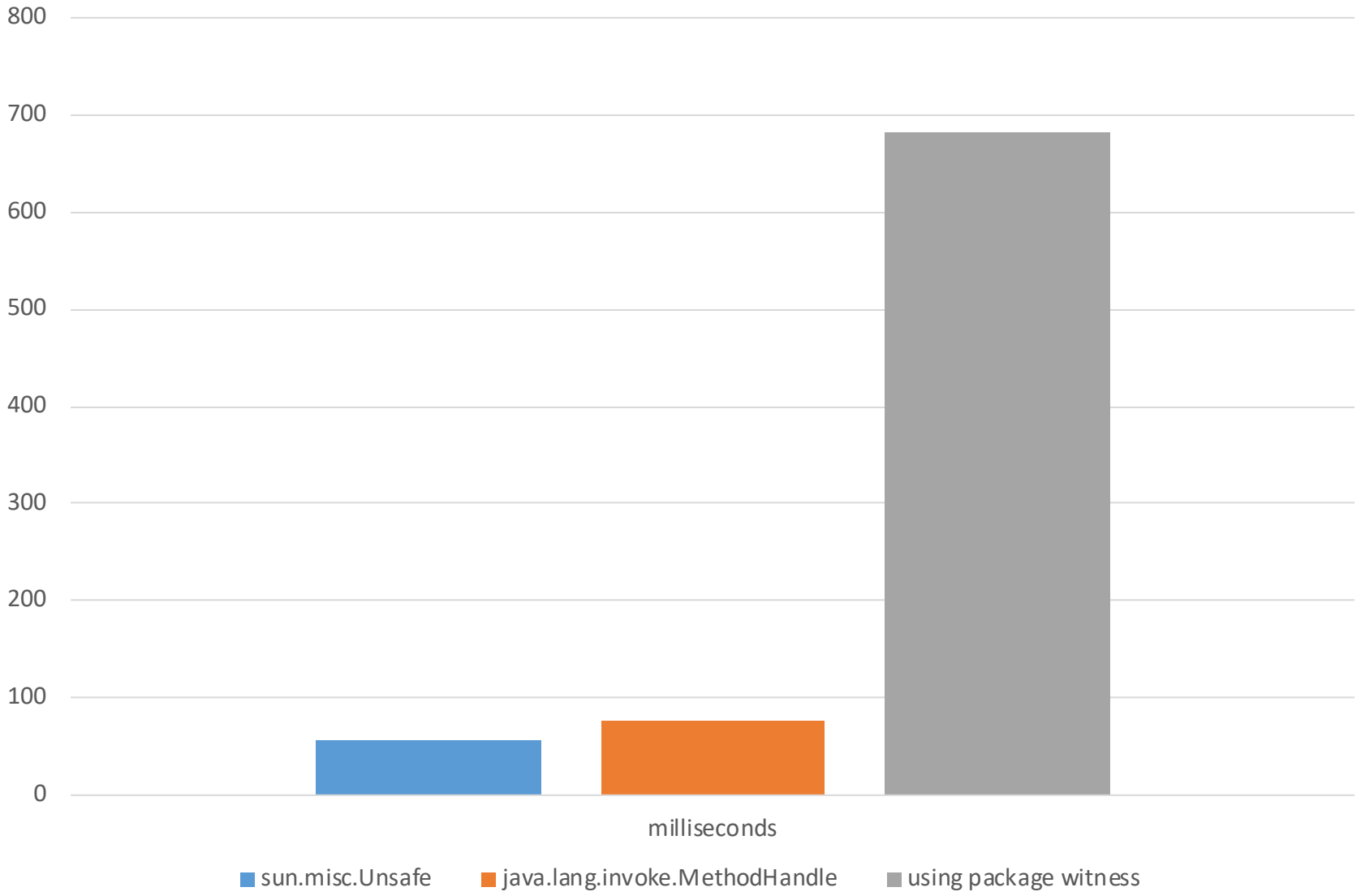
Emulating Unsafe.defineClass via JDK-8200559.

```
static void defineClass(Instrumentation inst, Class<?> pgkWitness, byte[] bytes) {
  ClassFileTransformer t =
  (definer, module, loader, name, c, pd, buffer) -> {
    if (c == pgkWitness) {
      definer.defineClass(bytes);
    }
    return null;
  };
  instrumentation.addClassFileTransformer(t, true);
  try {
    instrumentation.retransformClasses(pgkWitness);
  } finally {
    instrumentation.removeClassFileTransformer(t);
  }
}
```

Bonus: Hijacking the internal method handle to define classes in foreign packages.

```
class Lookup {
  static final Lookup IMPL_LOOKUP = new Lookup(Object.class, TRUSTED);
  static final Lookup PUBLIC_LOOKUP = new Lookup(Object.class, PUBLIC | UNCOND);
}
```

```
inst.retransformClasses(MethodHandles.class);

class MethodHandles {
  public static Lookup publicLookup() {
    if (Thread.currentThread().getId() == MY_PRIVILEGED_THREAD) {
      return Lookup.IMPL_LOOKUP;
    } else {
      return Lookup.PUBLIC_LOOKUP;
    }
  }
}
```


800
700
600
500
400
300
200
100
0
milliseconds
sun.misc.Unsafe
java.lang.invoke.MethodHandle
using package witness

Defining classes from Java agents

~~Transforming classes from Java agents~~

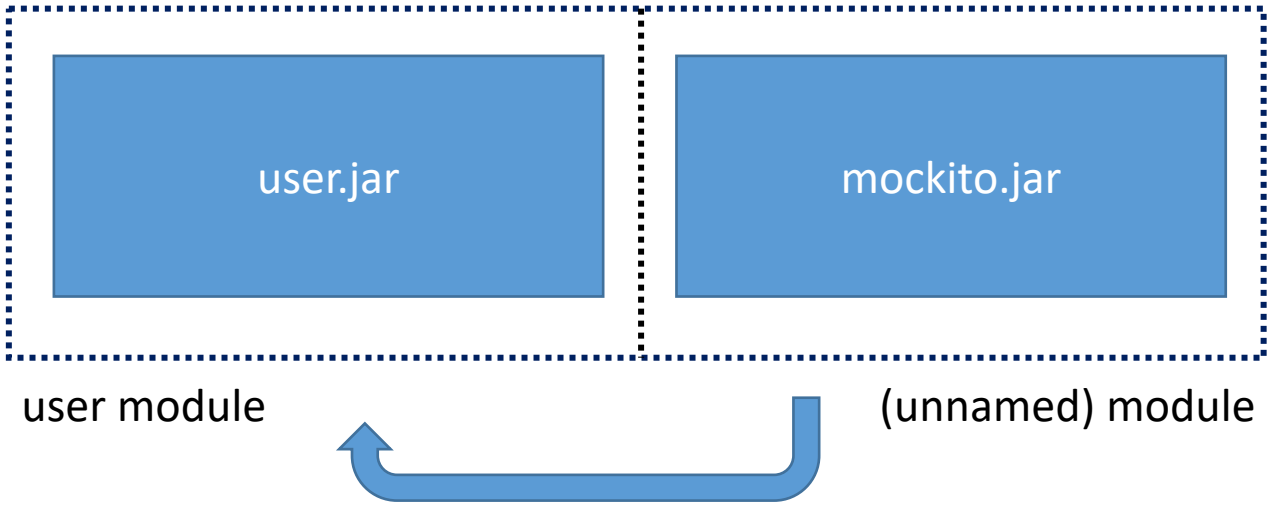
Defining classes from libraries

Transforming classes from libraries

Miscellaneous

Using unsafe class definition in testing context.

```
UserClass userClass = Mockito.mock(UserClass.class);
```




```
byte[] userClassMock = ...
methodHandle.defineClass(userClassMock);
```

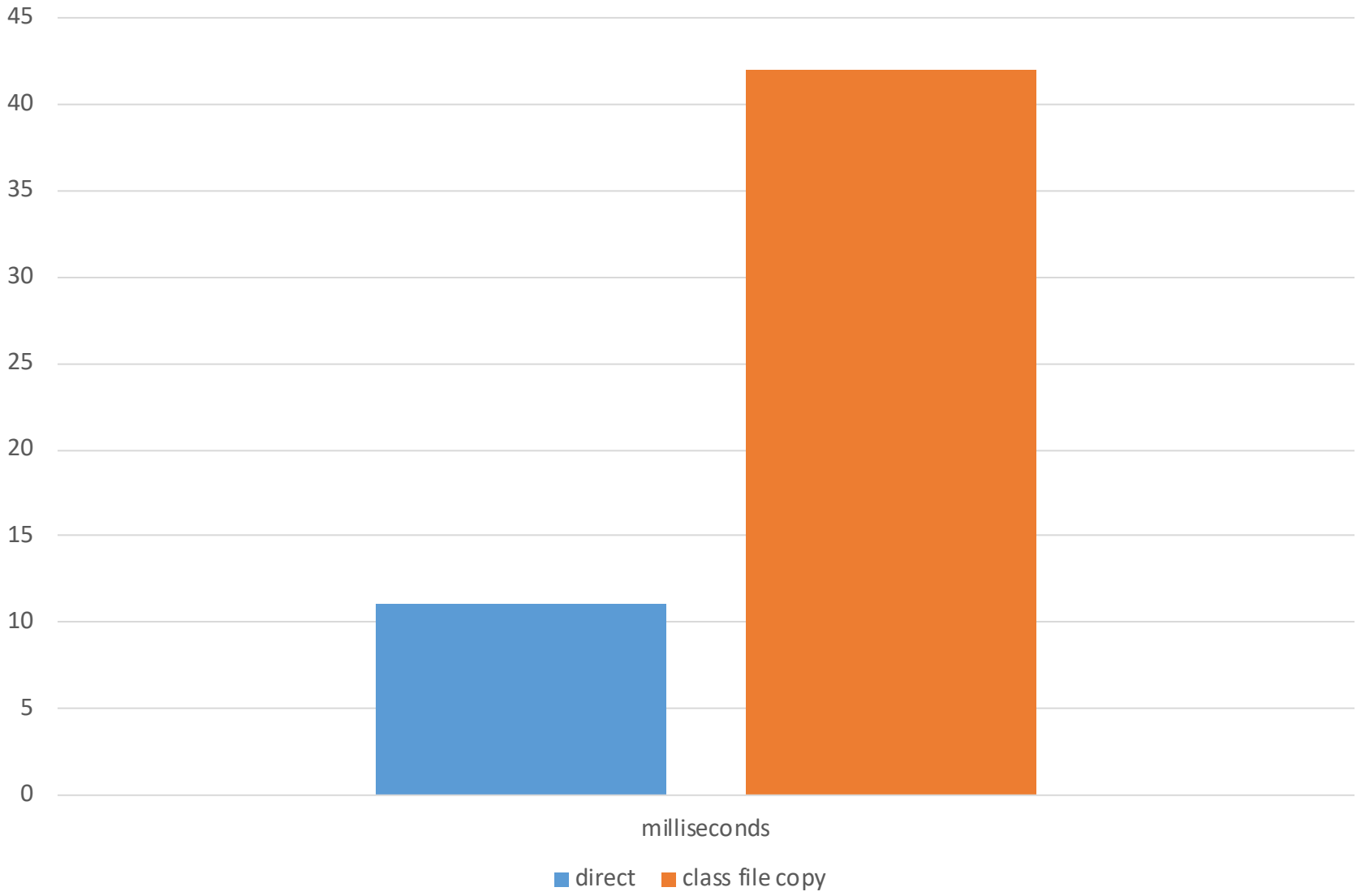
No standardized support for "test dependencies". It is impossible to open modules to Mockito which only exists in tests.

How to use Unsafe with the jdk.unsupported module being unrolled?

```
Field f = sun.misc.Unsafe.class.getDeclaredField("theUnsafe");
f.setAccessible(true);
Unsafe u = (Unsafe) f.get(null);
u.defineClass( ... );
```

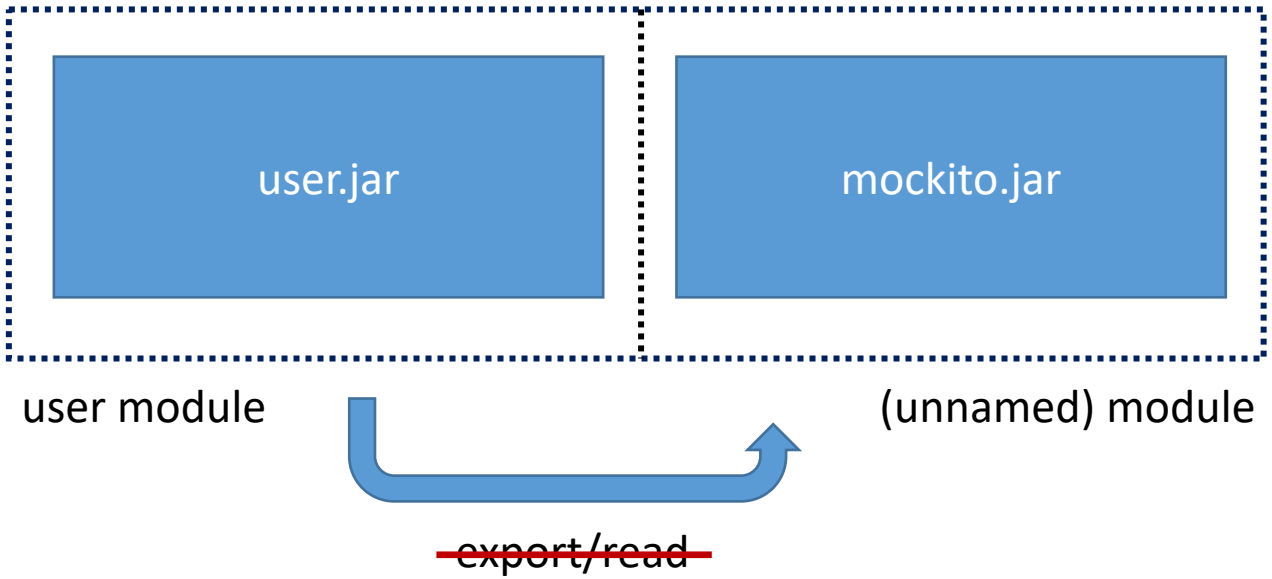
```
Field f = jdk.internal.misc.Unsafe.class.getDeclaredField("theUnsafe");
f.setAccessible(true); // only possible from agent (redefineModules) or cmd
Unsafe u = (Unsafe) f.get(null);
f.defineClass( ... );
```

```
static void makeAccessible(Unsafe unsafe, Field target) {
  Field f = AccessibleObject.class.getDeclaredField("override"); // since Java 12
  long offset = unsafe.getObjectFieldOffset(f);
  u.putBoolean(target, offset, true);
}
```


45
40
35
30
25
20
15
10
5
0
milliseconds
direct
class file copy

Handling proxies in a modularized application

```
class UserClass
```




```
interface MockitoMock
```

```
class ModuleProbe {
  static { UserClass.class.getModule()
       .addReads(MockitoMock.class
       .getModule()); }
}
```

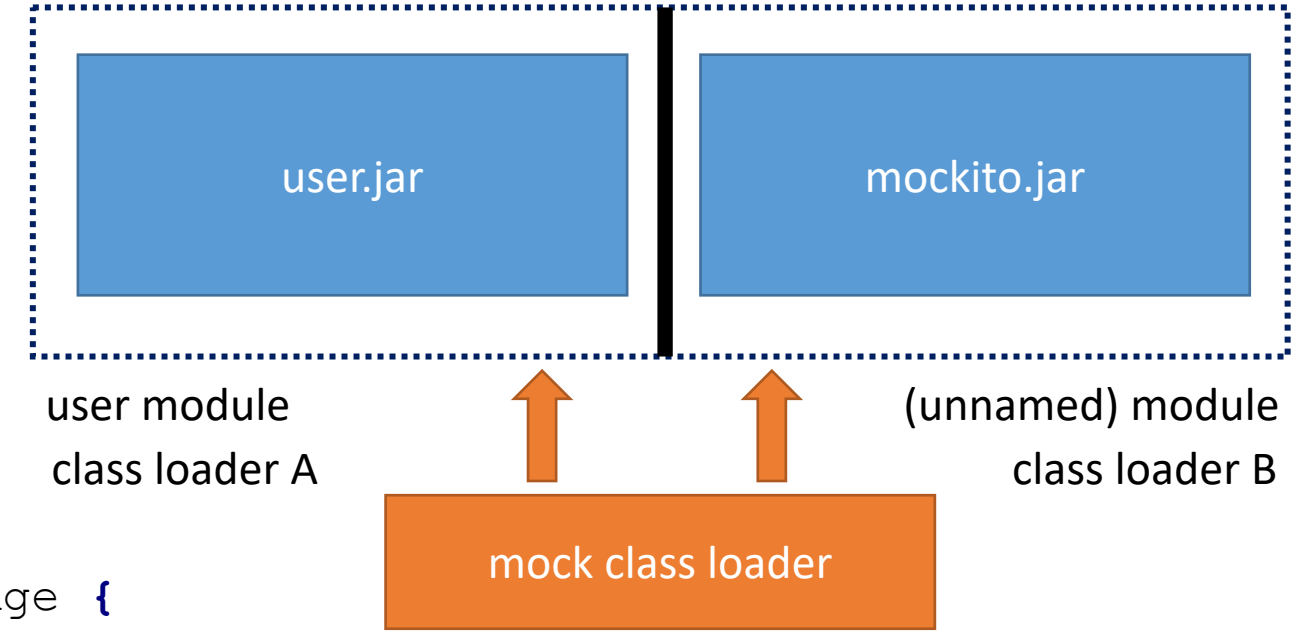
```
UnsafeHelper.defineClass(...);

Class.forName(
  ModuleProbe.class.getName(),
  true, cl);
```

```
class UserClass$MockitoMock
  extends UserClass
  implements MockitoMock
```

```
UnsafeHelper.defineClass(...);
```

Handling proxies in a modularized application

```
class UserClass
```

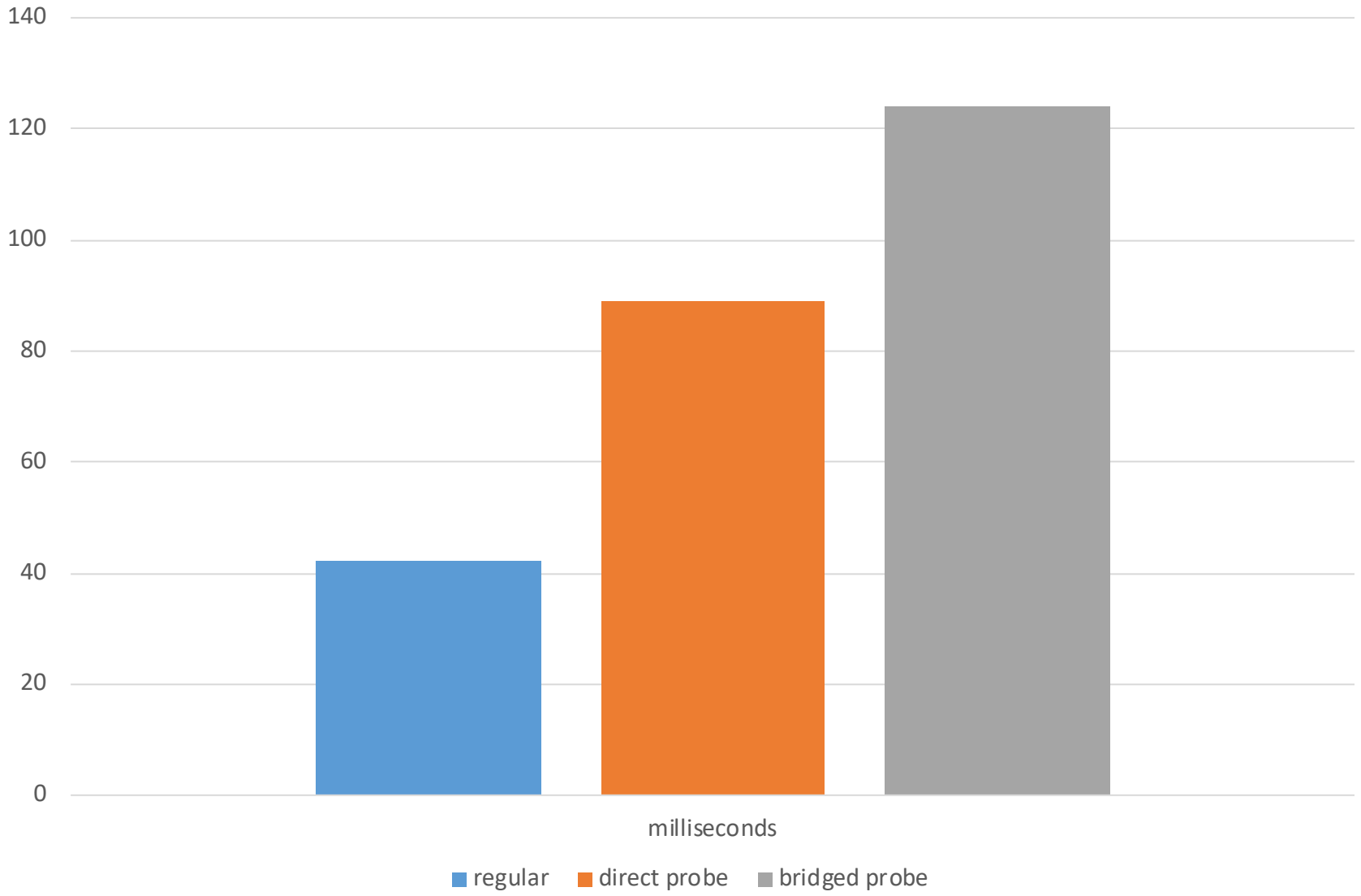



```
interface MockitoMock
```

```
class MockitoBridge {
  static Module module;
}
class ModuleProbe {
  static { UserClass.class.getModule()
      .addReads(MockitoBridge.class
      .getModule());
    UserClass.class.getModule()
      .addExports(MockitoBridge.module);
  }
}
```

```
UnsafeHelper.defineClass(...);
MockitoBridge.module = mcl
    .getUnnamedModule();
```

```
UnsafeHelper.defineClass(...);
Class.forName(
  ModuleProbe.class.getName(),
  true, cl);
```


140
120
100
80
60
40
20
0
milliseconds
regular
direct probe
bridged probe

Defining classes from Java agents

~~Transforming classes from Java agents~~

Defining classes from libraries

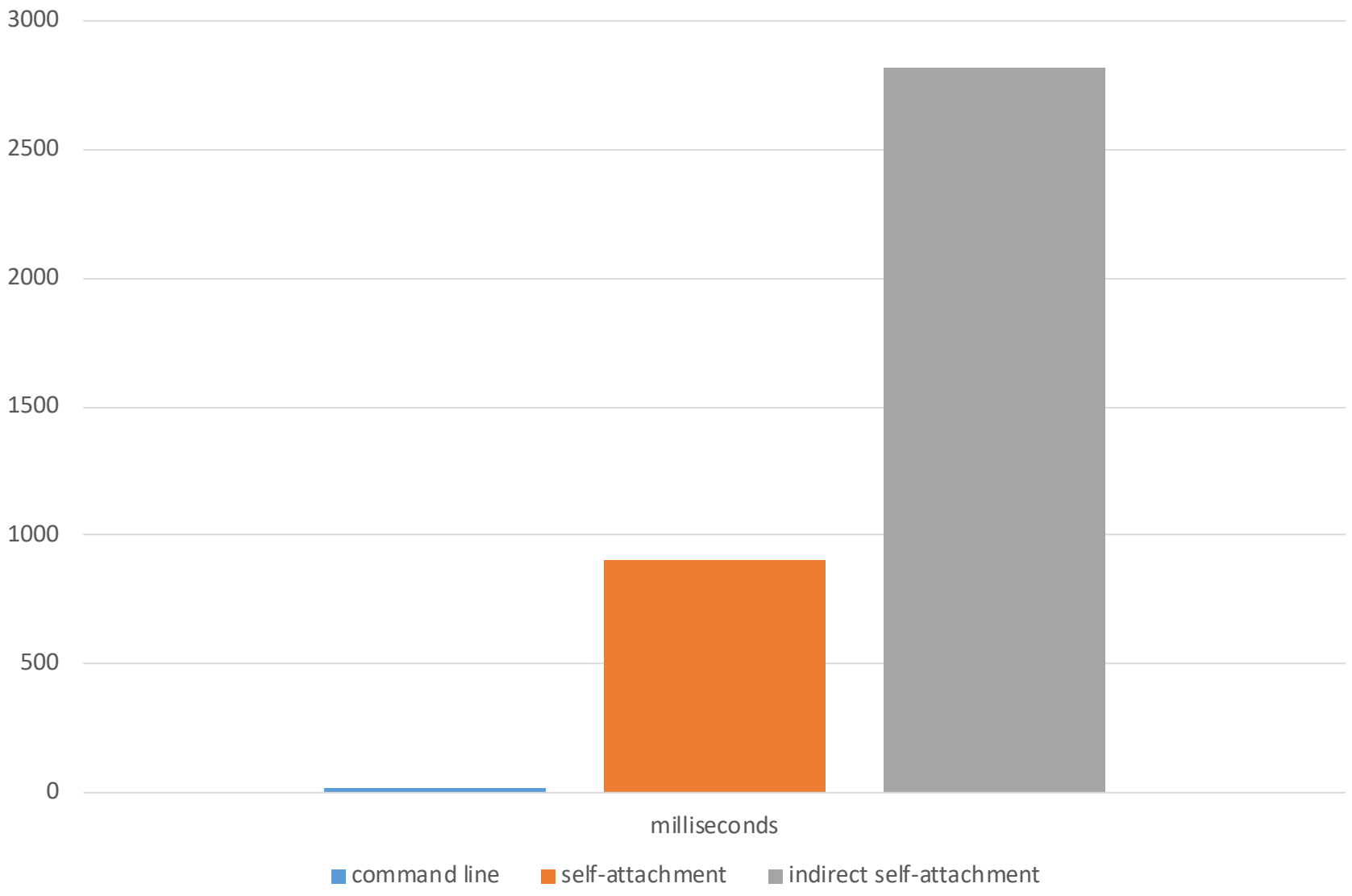
Transforming classes from libraries

Miscellaneous

Instrumenting code without attaching a Java agent.

```
FinalUserClass finalUserClass = Mockito.mock(FinalUserClass.class);

class InstrumentationHolder {
  static Instrumentation inst;
  public static void agentmain(String arg, Instrumentation inst) {
    InstrumentationHolder.inst = inst;
  }
}

static Instrumentation inst() {
  long processId = ProcessHandle.current().pid();
  String location = InstrumentationHolder.class.getProtectionDomain()
    .getCodeSource()
    .getURL()
    .toString();
  VirtualMachine vm = VirtualMachine.attach(String.valueOf(processId));
  vm.loadAgent(location, "");
  return InstrumentationHolder.inst;
}
```

Controlled by the jdk.attach.allowAttachSelf option which is false by default.


3000
2500
2000
1500
1000
500
0
milliseconds
command line
self-attachment
indirect self-attachment

Using self-attach for emulating Unsafe.allocateInstance in Mockito.

```
UserClass userClass = Mockito.mock(UserClass.class);

class UserClass {
  UserClass() { // some side effect }
}   if (!MockitoThreadLocalControl.isMockInstantiation()) {
      // some side effect
    }
  }
}
```

Dealing with the security manager in unit tests and agents.

```
class AccessControlContext {
  void checkPermission(Permission perm) throws AccessControlException {
    SecurityManagerInterceptor.check(this, perm);
  }
}
```

Not all security managers respect a policy file what makes instrumentation even more attractive.

What is missing for a full migration away from Unsafe?

```
interface Instrumentation {
  Class<?> defineClass(byte[] bytes, ClassLoader cl);
  // ...
}


class TestSupport { // module jdk.test
  static Instrumentation getInstrumentation() { ... }
  static <T> T allocateInstance(Class<T> type) { ... }
  static void setSecurityManagerUnsafe(SecurityManager sm) { ... }
}
```

The jdk.test module would:

- not be bundled with a non-JDK VM distribution
- it would print a console warning when being loaded
- allow to mark test-scope libraries not to load in production environments
- be resolved automatically by test runners like Maven Surefire

Defining classes from Java agents

~~Transforming classes from Java agents~~

Defining classes from libraries

Transforming classes from libraries

Miscellaneous

How do agents inject state into classes without changing their shape?

```
Callback callback = ...;
Dispatcher.vals.put("unique-name", callback);
ClassFileTransformer t = (definer, module, loader, name, c, pd, buffer) -> {
  // how to make the callback instance accessible to an instrumented method?
}

void foo() {
} Callback c = (Callback) Dispatcher.vals.get("unique-name");
  c.invoked("foo");
}

class Dispatcher { // inject into a well-known class loader
  ConcurrentMap<String, Object> vals = new ConcurrentHashMap<>();
}
```

How do agents inject state into classes without changing their shape?

```
State state = ...;
Init.vals.put("unique-name", state);
ClassFileTransformer t = (definer, module, loader, name, c, pd, buffer) -> {
  // how to inject non-serializable state into an instrumented class?
}


class UserClass {
  static final State state;
} static {
    state = (State) Init.vals.get("unique-name");
  }
}


class Init { // inject into a well-know class loader
  static ConcurrentMap<String, Object> vals = new ConcurrentHashMap<>();
}
```

Working with "well-known" class loaders.

Well-known (across all Java versions): system class loader, boot loader

```
interface Instrumentation {
  void appendToBootstrapClassLoaderSearch(JarFile jar);
  void appendToSystemClassLoaderSearch(JarFile jar);
}
```

Change in behavior:

- Java 8 and before: URLClassLoader checks appended search path for any package.
- Java 9 and later: BuiltInClassLoader checks appended search path for unknown packages.

Working with "well-known" modules.

```
interface Instrumentation {
  void redefineModule(
    Module module,
    Set<Module> extraReads,
    Map<String,Set<Module>> extraExports,
    Map<String,Set<Module>> extraOpens,
    Set<Class<?>> extraUses,
    Map<Class<?>,List<Class<?>>> extraProvides
  );
}
```

Not respected by other module systems (OSGi/JBoss modules) which are harder to adjust.

Solutions:

- Adjust module graph via instrumentation + instrument all class loaders to whitelist agent dispatcher package.
- Put dispatcher code into a known package that all class loaders and the VM accept: java.lang@java.base.

The latter is only possible via Unsafe API since Java 9 and later.

Most dynamic code generation is not really dynamic.

Dynamic code generation is

- mainly used because types are not known at library-compile-time despite being known at application compile-time.
- should be avoided for production apllications (reduce start-up time) but is very useful for testing.

```
@SupportedSourceVersion(SourceVersion.latestSupported())
@SupportedAnnotationTypes("my.SampleAnnotation")
public class MyProcessor extends AbstractProcessor {
  void init(ProcessingEnvironment env) { }
  boolean process(Set<? extends TypeElement> annoations, RoundEnvironment env) { }
}
```

Downside of using annotation processors:

- Bound to the Java programming language.
- Cannot change bytecode. (Only via internal API as for example in Lombock.)
- No general code-interception mechanism as for Java agents.

How to write a "hybrid agent" using build tools.

```
<dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate-maven-plugin</artifactId>
  <version>LATEST</version>
</dependency>
```

Unified concept in Byte Buddy: agents, plugins and subclass proxies:

```
interface Plugin {
  DynamicType.Builder<?> apply(
    DynamicType.Builder<?> builder,
    TypeDescription typeDescription,
    ClassFileLocator classFileLocator);
}
```

```
interface Transformer {
  DynamicType.Builder<?> transform(
    DynamicType.Builder<?> builder,
    TypeDescription typeDescription,
    ClassLoader classLoader,
    JavaModule module);
}
```

Remaining downside of build plugins:
Difficult to instrument code in the JVM and third-party jar files.
An agent-like compile-time transformation API would be a great edition to AOT-based Java, e.g. Graal.

Memory-leaks caused by hybrid agents: lack of ephomerons

```
class UserClass {
  void m() { /* do something */ }
}
```

static

dynamic

```
class UserClass {
  AgentDispatcher dispatcher;
  void m() {
    dispatcher.handle("m", this);
  }
}
```

```
public class BootDispatcher {
  public static WeakMap<Object, Dispatcher>
      dispatchers;
}

class UserClass {
  void m() {
    BootDispatcher.dispatchers
        .get(this)
        .handle("m", this);
  }
}
```

hard reference

http://rafael.codes
@rafaelcodes

http://documents4j.com
https://github.com/documents4j/documents4j

http://bytebuddy.net
https://github.com/raphw/byte-buddy