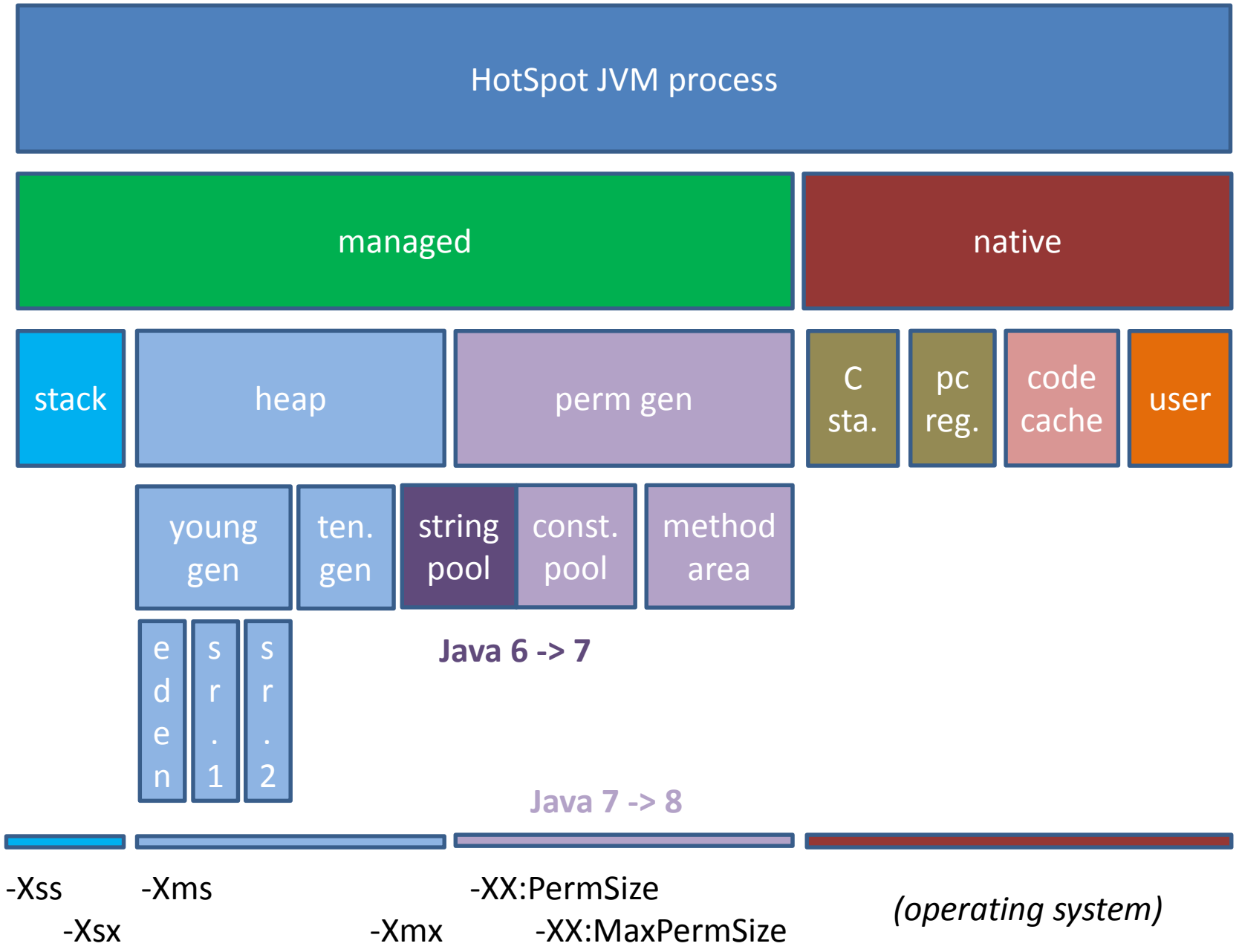
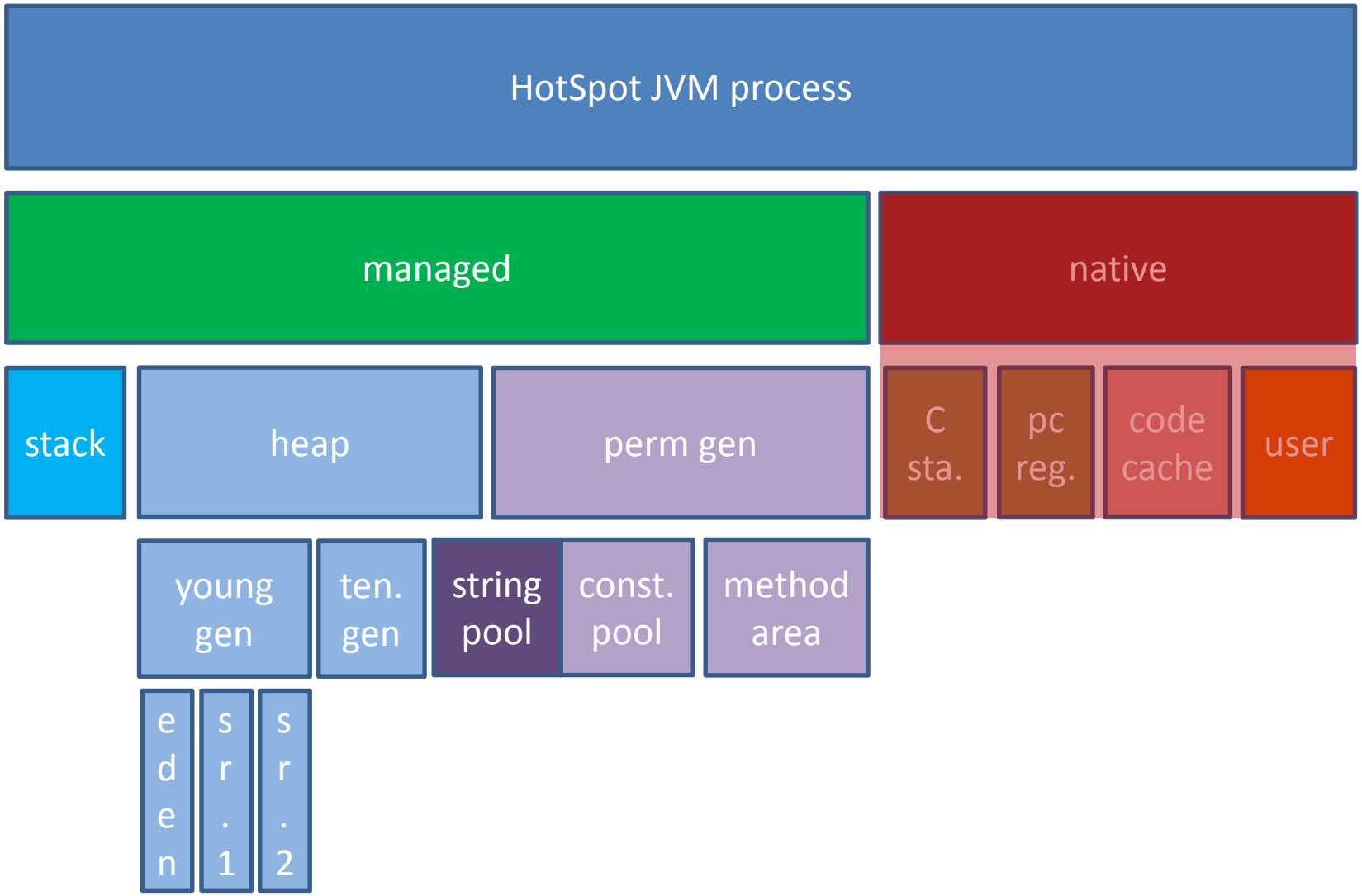
A topology of memory leaks on the JVM


HotSpot JVM process
managed
native
stack
heap
perm gen
C sta.
pc reg.
code cache
user
young gen
ten. gen
string pool
const. pool
method area
eden
sr.1
sr.2
Java 6 -> 7
Java 7 -> 8
-Xss
-Xsx
-Xms
-Xmx
-XX:PermSize
-XX:MaxPermSize
(operating system)

genuine leaks



Allocating native memory with *sun.misc.Unsafe*

1. Getting hold of “the unsafe“

```
Field unsafe = Unsafe.class
    .getDeclaredField("theUnsafe");
unsafe.setAccessible(true);
Unsafe theUnsafe = (Unsafe) unsafe.get(null);
```

2. Allocating native memory

```
final int intSize = 4;
long arraySize = (1L + Integer.MAX_VALUE) * intSize;
long index = unsafe.allocateMemory(arraySize);
Random random = new Random();
for(long l = 0L; l < arraySize; l++)
  unsafe.putInt(random.nextInt());
```

3. (Hopefully) releasing native memory

```
// Without me, the memory leaks
unsafe.freeMemory(index);
```

I don‘t do *sun.misc.Unsafe*

1. But some of your favorite tools do:
    1. Kryo (used by e.g. Twitter‘s Storm, Apache Hive)
    2. EhCache‘s “big memory“ (used by e.g. Hibernate)
    3. JDK (e.g. direct byte buffers)
    4. General purpose (e.g. GSON, Snappy)

2. *Unsafe* to become public API in Java 9 (competition with CLR?)

3. JNI leaks have the same effect

4. Java 8 removes “perm gen“ in favor of native memory “meta space“

symptoms of a genuine leak

No exception required by JVMS. However, JVM will be untypical slow when:

1. Creating a thread:
   Requires allocation of new stacks / pc register.

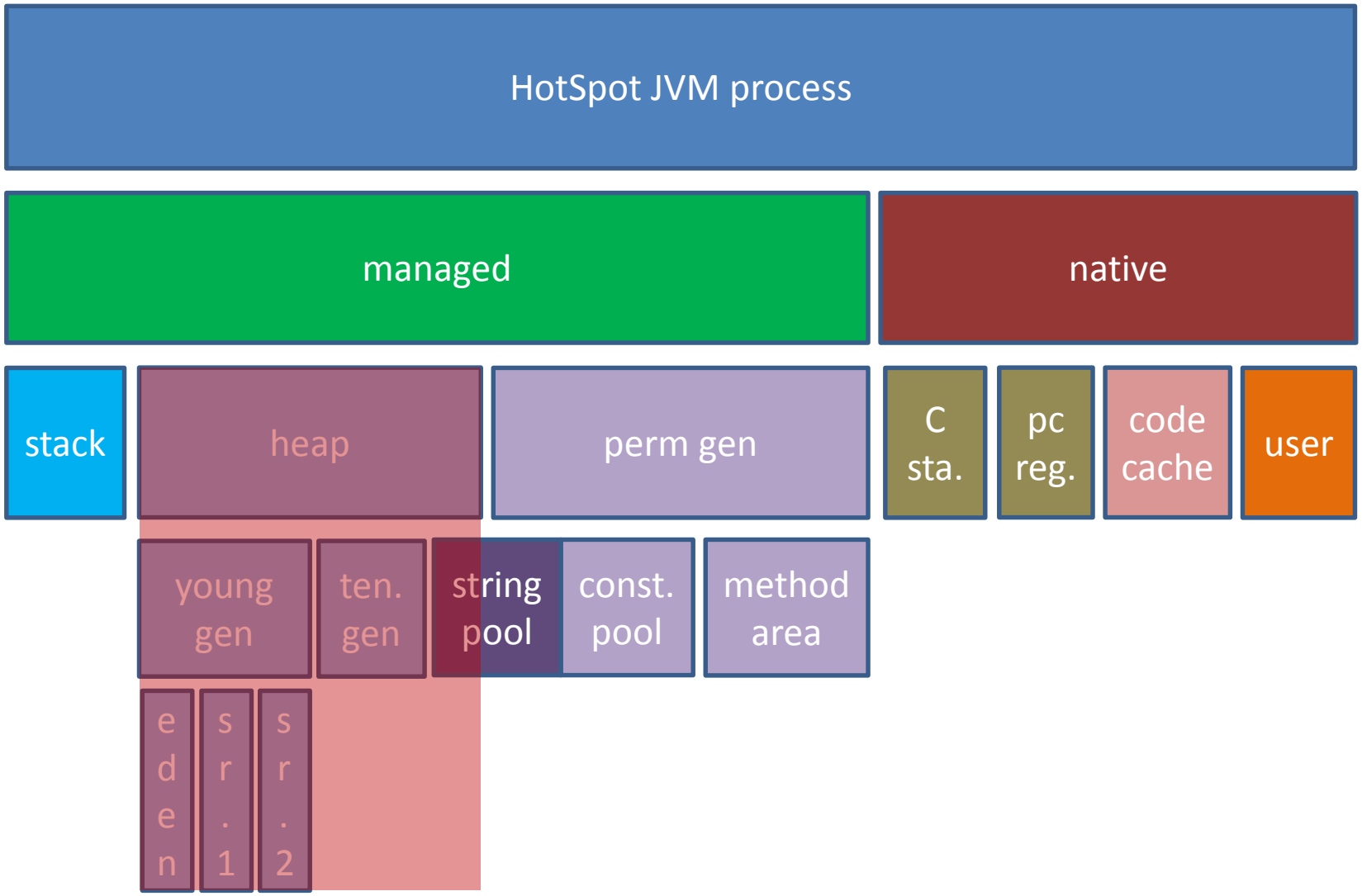
2. Loading a class / JIT:
   Requires native memory.

3. Doing I/O:
   Often requires native memory.

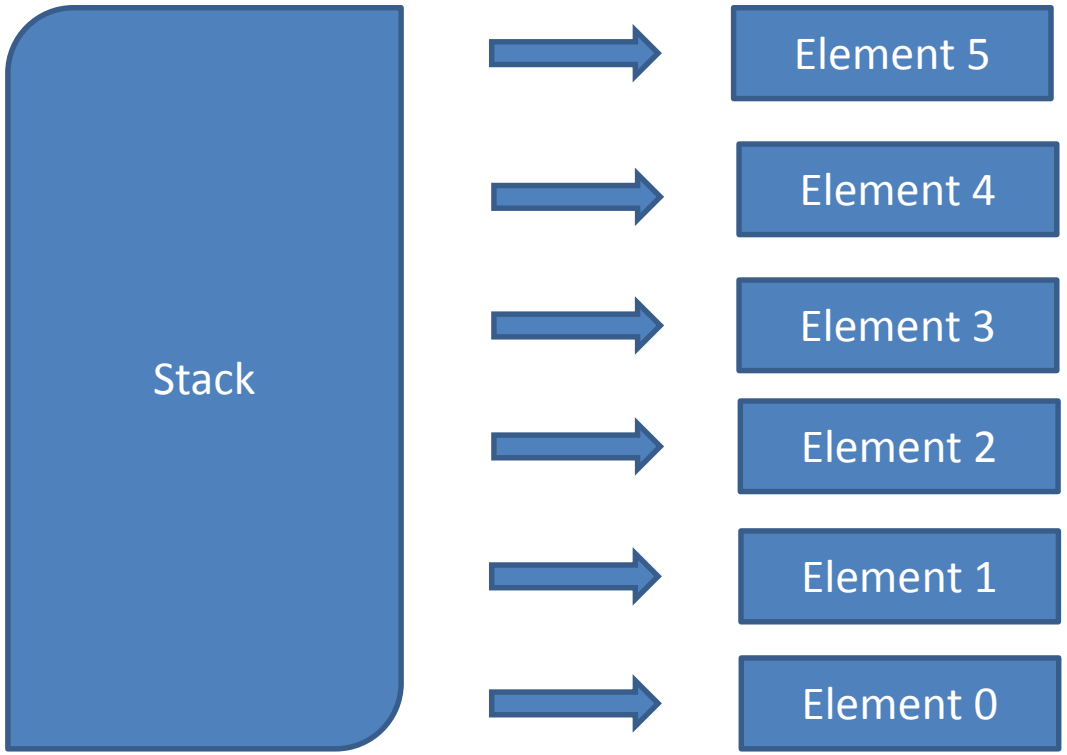
4. Calling a native method:
   Allocates resources on native stack / user space. Swapping can delay execution.

5. Garbage collection:
   Tenured generation collection needs broad access such that OS must swap back the heap. Rule of thump: All JVM heaps must be swapped back into memory. (Leaked memory will stay swapped out.)

Heap leaks

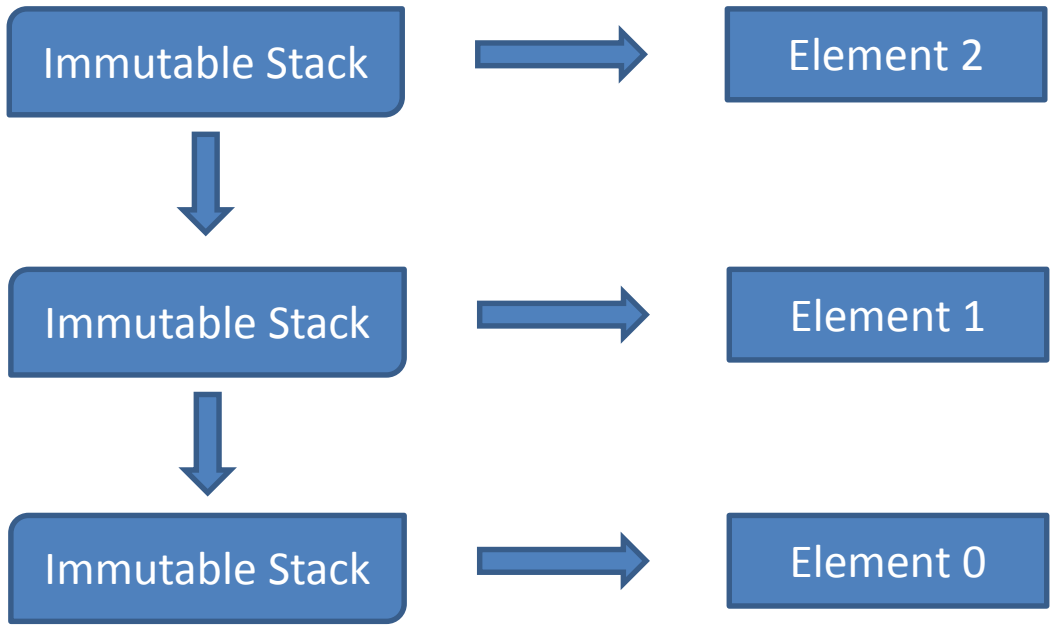

HotSpot JVM process
managed
native
stack
heap
perm gen
C sta.
pc reg.
code cache
user
young gen
ten. gen
string pool
const. pool
method area
eden
sr. 1
sr. 2

```
class FixedSizeStack {

  private final Object[] objectPool = new Object[10];
  private int pointer = -1;

  public Object pop() {
    if(pointer < 0)
      throw new NoSuchElementException();
    return objectPool[pointer--];
  }

  public Object peek() {
    if(pointer < 0)
      throw new NoSuchElementException();
    return objectPool[pointer];
  }

  public void push(Object object) {
    if(pointer > 8)
      throw new IllegalStateException("stack overflow");
    objectPool[++pointer] = object;
  }
}
```



All memory leaks in Java are linked to **reference** life cycles.

always prefer immutability

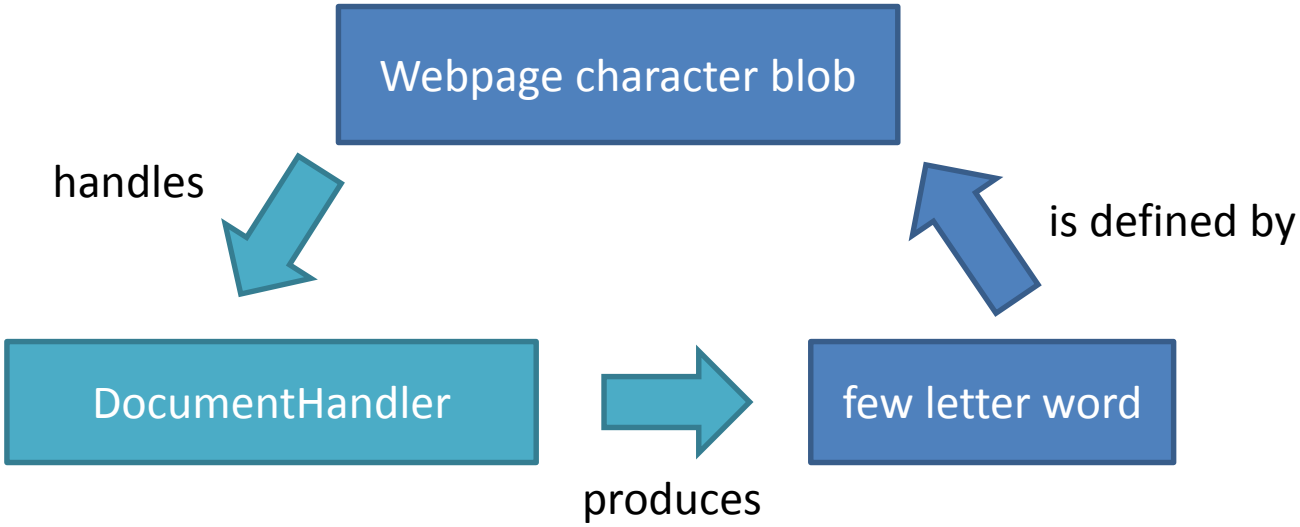


JDK 6

```
public String substring(int beginIndex, int endIndex) {
  // omitted argument validation
  return ((beginIndex == 0) && (endIndex == count))
      ? this
      : new String(offset + beginIndex,
                   endIndex - beginIndex,
                   value);
}

String(int offset, int count, char[] value) {
  this.value = value;
  this.offset = offset;
  this.count = count;
}
```

XML, anybody?

```
org.xml.sax.DocumentHandler {
  // several callback methods
  void characters(char[] ch, int start, int length);
}
```



JDK 7

make defensive copies instead of reusing super sets

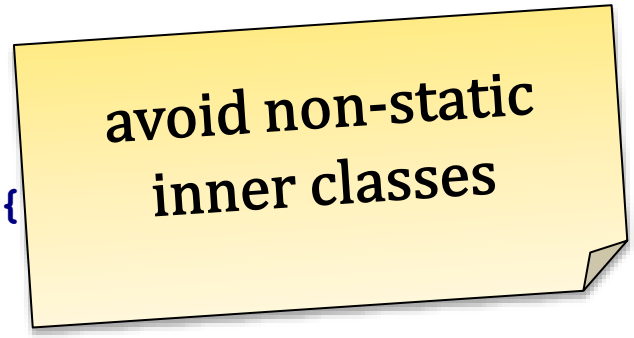
```
public String substring(int beginInde
  // omitted argument validation
  int subLen = endIndex - beginIndex;
  return ((beginIndex == 0) && (endIndex == value.length))
      ? this
      : new String(value, beginIndex, subLen);
}

public String(char[] value, int offset, int count) {
  // omitted argument validation
  this.value = Arrays.copyOfRange(value,
                                  offset,
                                  offset + count);
}
```

A classic Android leak

```
interface Message extends Serializable {
  String getInfo();
}

class ExampleActivity extends Activity {
  @Override public void onCreate(Bundle bundle) {
    startService(
        new Intent(this, getClass())
          .putExtra(
              "foo",
              new Message() {
                @Override public String getInfo() {
                  return "bar";
                }
           }));
  }
}
```

Non-static inner classes

```
new Message() {
  @Override public String getInfo() {
    return "bar";
  }
}
```



uncompiled

---

desugared

```
class ExampleActivity$1 implements Message {

  private ExampleActivity this$0;

  ExampleActivity$1(ExampleActivity this$0) {
    this.this$0 = this$0;
  }

  @Override public String getInfo() {
    return "bar";
  }
}
```

Java 8 lambda expressions

```
class Foo {
  void bar() {
    List<String> list = Arrays.asList("foo", "bar");
    list.forEach(s -> { System.out.println(s); });
  }
}
```

uncompiled

desugared

```
class Foo {
  void bar() {
    List<String> list = Arrays.asList("foo", "bar");
    list.forEach(LambdaMetafactory
        .INVOKEDYNAMIC(Foo::lambda$1)
        .make());
  }

  private static void lambda$1(String s) {
    System.out.println(s);
  }
}
```

Java 8 lambda expressions

```
class Foo {
  final String prefix; // constructor omitted
  void bar() {
    List<String> list = Arrays.asList("foo", "bar");
    list.forEach(s -> { System.out.println(
        prefix + ":" + s) });
  }
}
```

uncompiled

desugared

```
class Foo {
  final String prefix; // constructor omitted
  void bar() {
    List<String> list = Arrays.asList("foo", "bar");
    list.forEach(LambdaMetaFactory
        .INVOKEDYNAMIC(this::lambda$1)
        .make());
  }

  private void lambda$1(String s) {
    System.out.println(this.prefix + ":" + s);
  }
}
```

Java 8 lambda expressions

```
class Foo {
  final String prefix; // constructor om
  void bar() {
    List<String> list = Arrays.asList("f
    String prefix = this.prefix;
    list.forEach(s -> { System.out.println(
        prefix + ":" + s)});
  }
}
```

pay attention to your lambda expression's scope

uncompiled

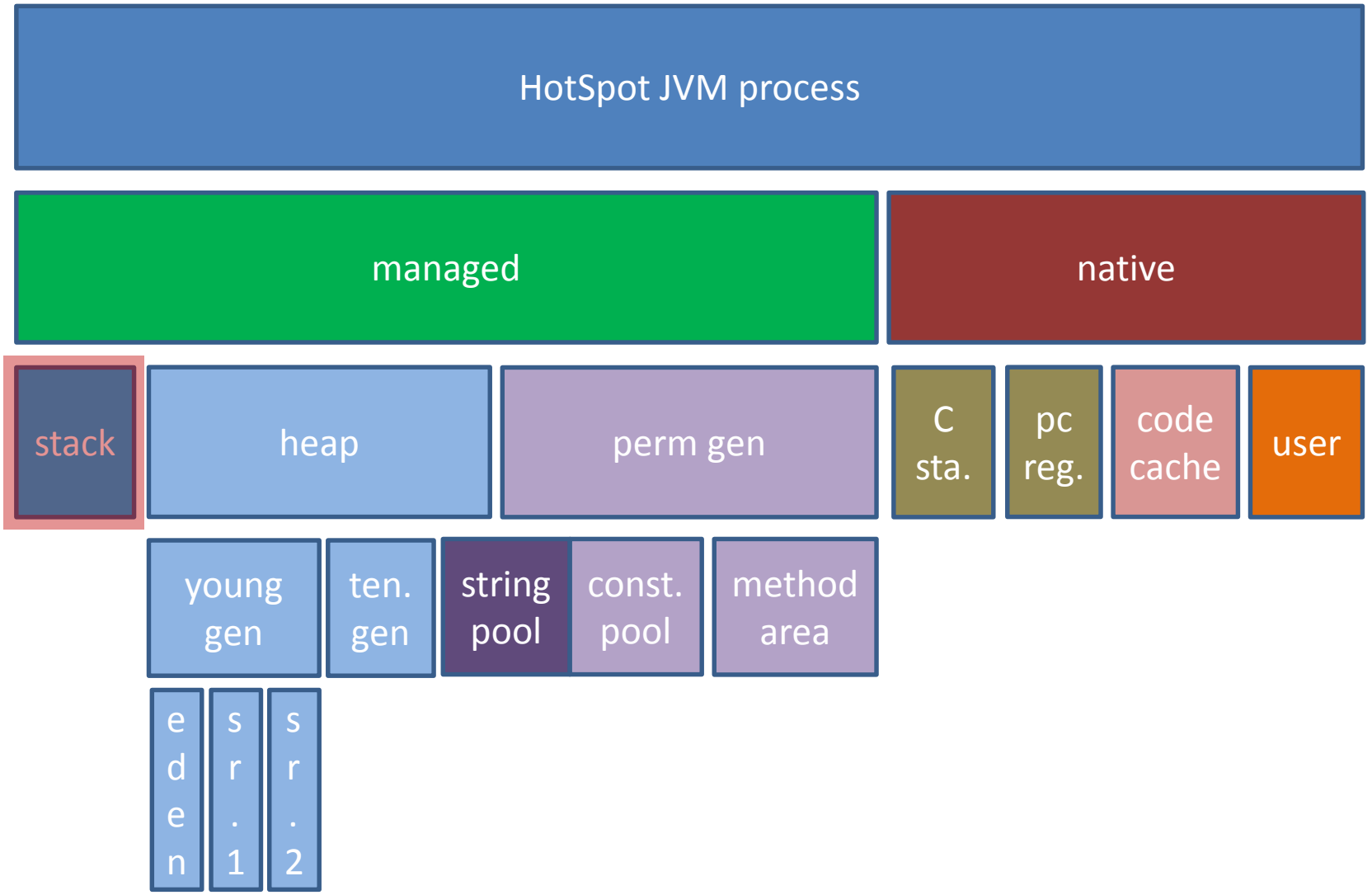
desugared

```
class Foo {
  final String prefix; // constructor omitted
  void bar() {
    List<String> list = Arrays.asList("foo", "bar");
    String prefix = this.prefix;
    list.forEach(LambdaMetafactory
        .INVOKEDYNAMIC(Foo::lambda$1)
        .make(prefix));
  }
  private static void lambda$1(String prefix, String s) {
    System.out.println(prefix + ":" + s);
  }
}
```

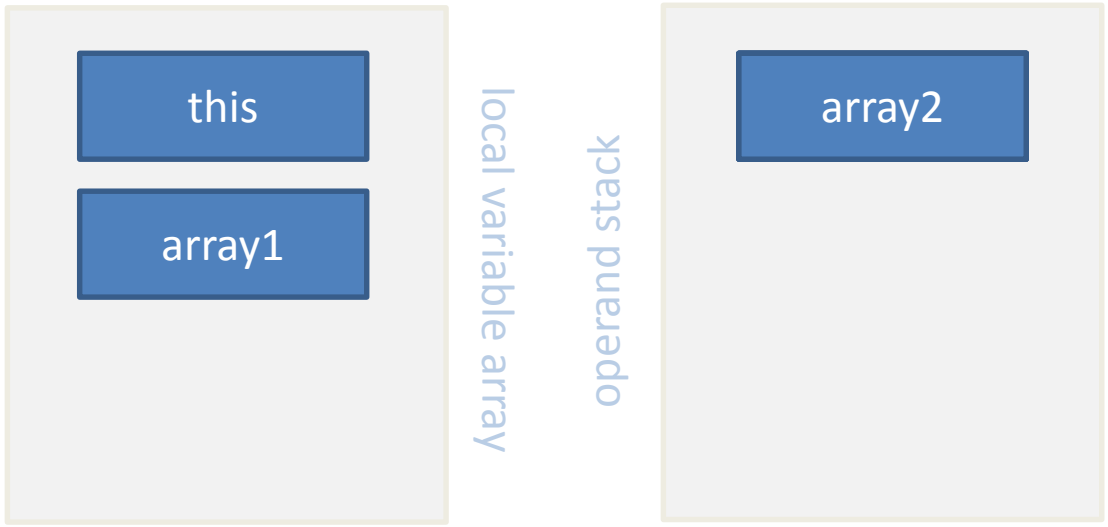
Other typical causes of heap leaks

- Functional expressions in e.g. Scala / Groovy
- Serialization leaks (e.g. Apache Wicket)
- Singletons / enums
- Context frameworks (e.g. DI like Spring)

Stack leaks

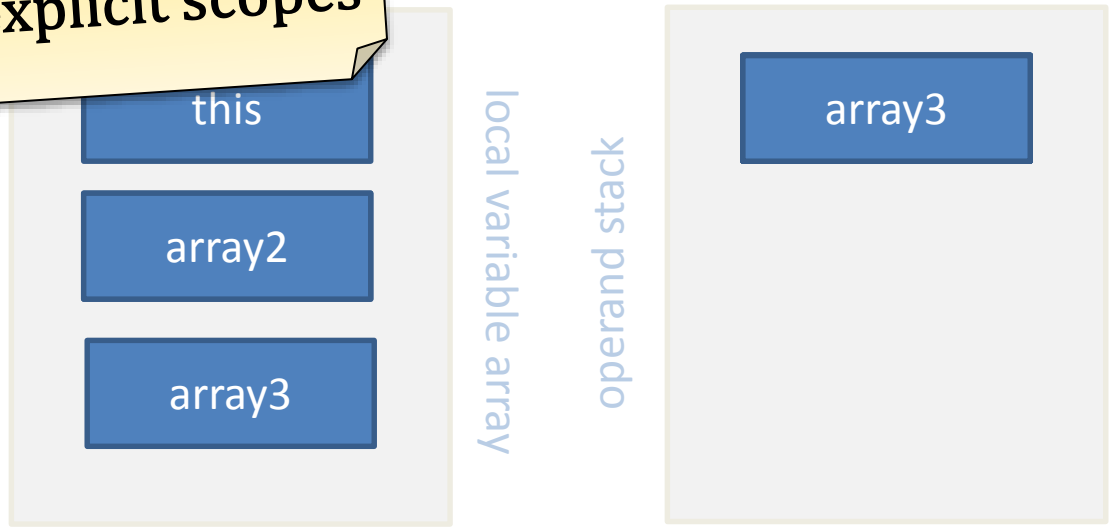

HotSpot JVM process
managed
native
stack
heap
perm gen
C sta.
pc reg.
code cache
user
young gen
ten. gen
string pool
const. pool
method area
eden
sr.1
sr.2

```
class StackLeak {
  int SIZE = (int) (0.5 * Runtime
                        .getRuntime()
                        .maxMemory());

  void foo() {
    {
      byte[] array1 = new byte[SIZE];
    }
    byte[] array2 = new byte[SIZE];
  }

  void bar() {
    {
      byte[] array1 = new byte[SIZE];
    }
    byte[] array2 = new byte[10];
    byte[] array3 = new byte[SIZE];
  }
}
```



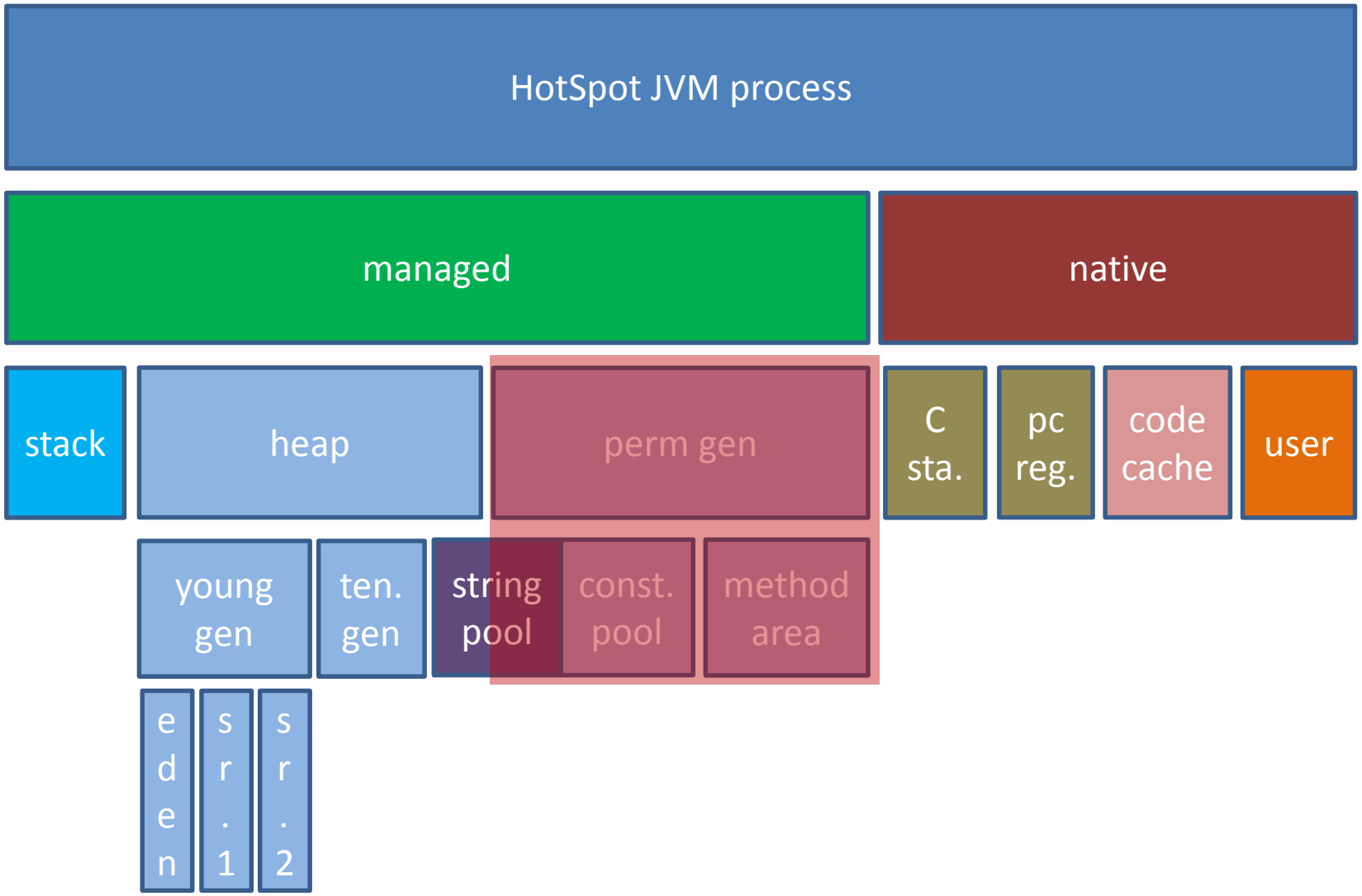

```
void foo() {
  {
    byte[] array1 = new byte[SIZE];
  }
  byte[] array2 = new byte[SIZE];
}
```

write short methods,
mistrust explicit scopes

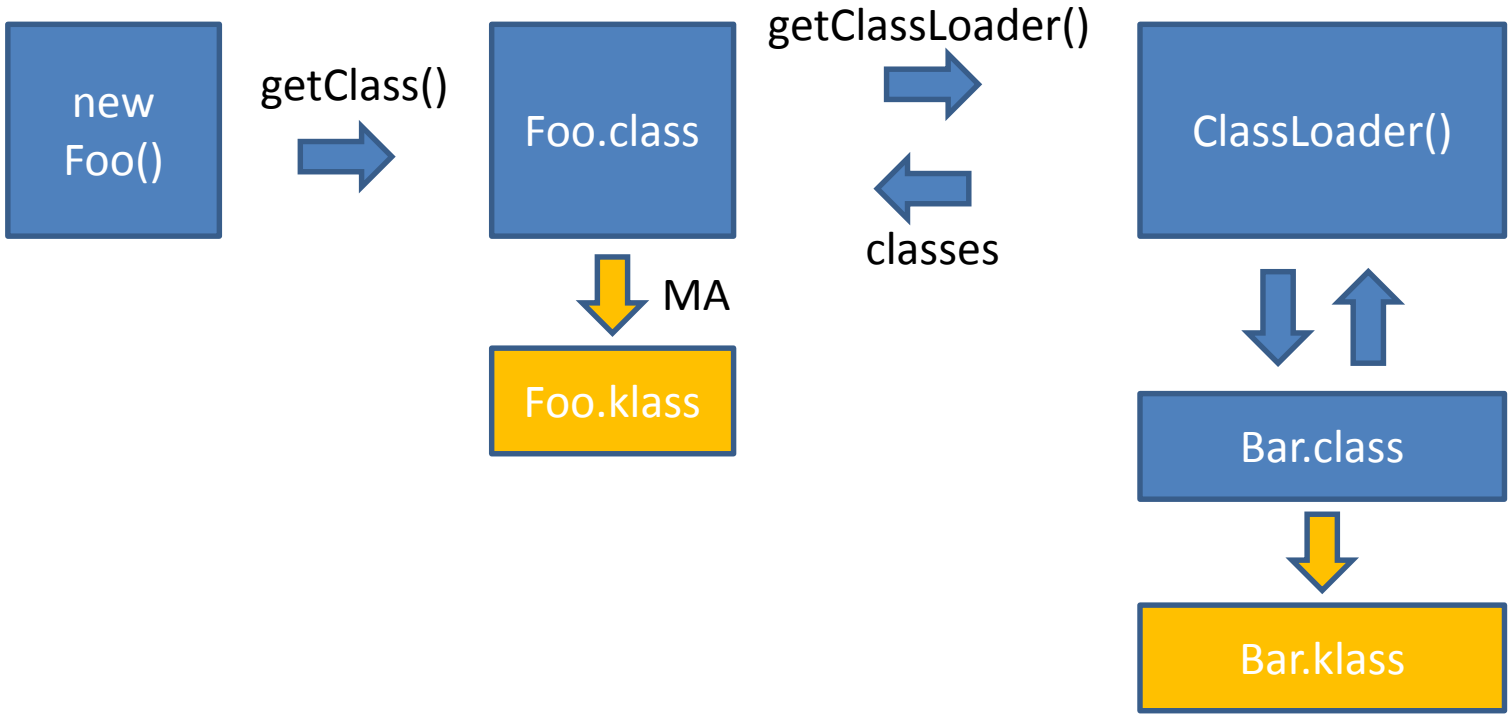



```
void bar() {
  {
    byte[] array1 = new byte[SIZE];
  }
  byte[] array2 = new byte[1];
  byte[] array3 = new byte[SIZE];
}
```

“Perm gen“ leaks

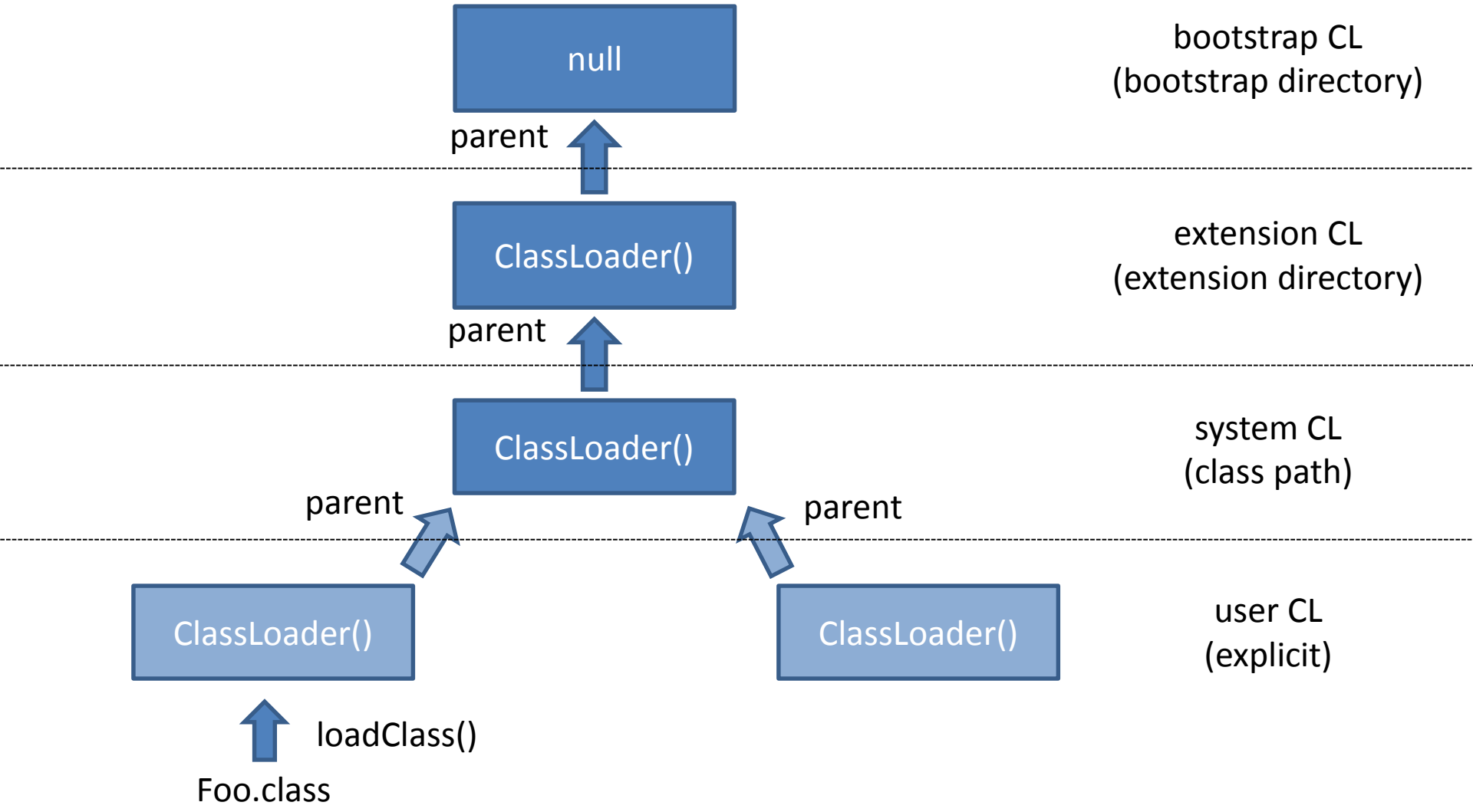


Classes and HotSpot

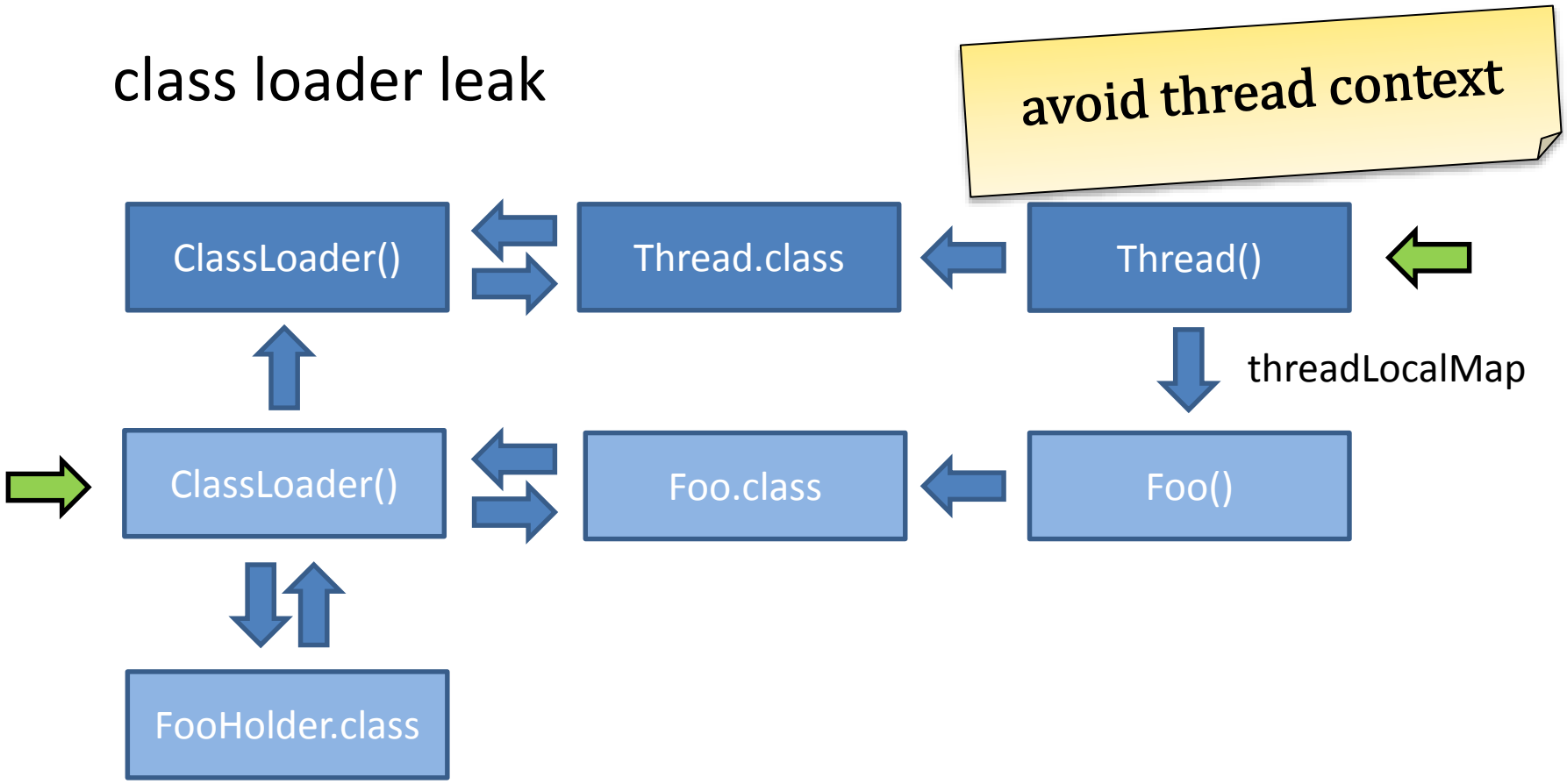



```
Field field = ClassLoader.class.getDeclaredField("classes");
field.setAccessible(true);
Vector<Class<?>> classes = (Vector<Class<?>>)
    field.get(ClassLoader.getSystemClassLoader());
```

ClassLoaders and HotSpot



class loader leak




```
class FooHolder {
  static ThreadLocal<Foo> tlFoo = new ThreadLocal<Foo>();
  static { tlFoo.set(new Foo()); }
}
```

Other typical causes o

avoid redeployment,
use one container per app

- Shut down hooks
- Use of thread context class loaders (e.g. OSGi)
- Service provider interfaces (SPIs)
- JDBC drivers (JDK *DriverManager*)
- Security frameworks (e.g. JDK *Policy*)
- Class loader magic (e.g. instrumentation)

Why do modern applications require so much perm gen memory?

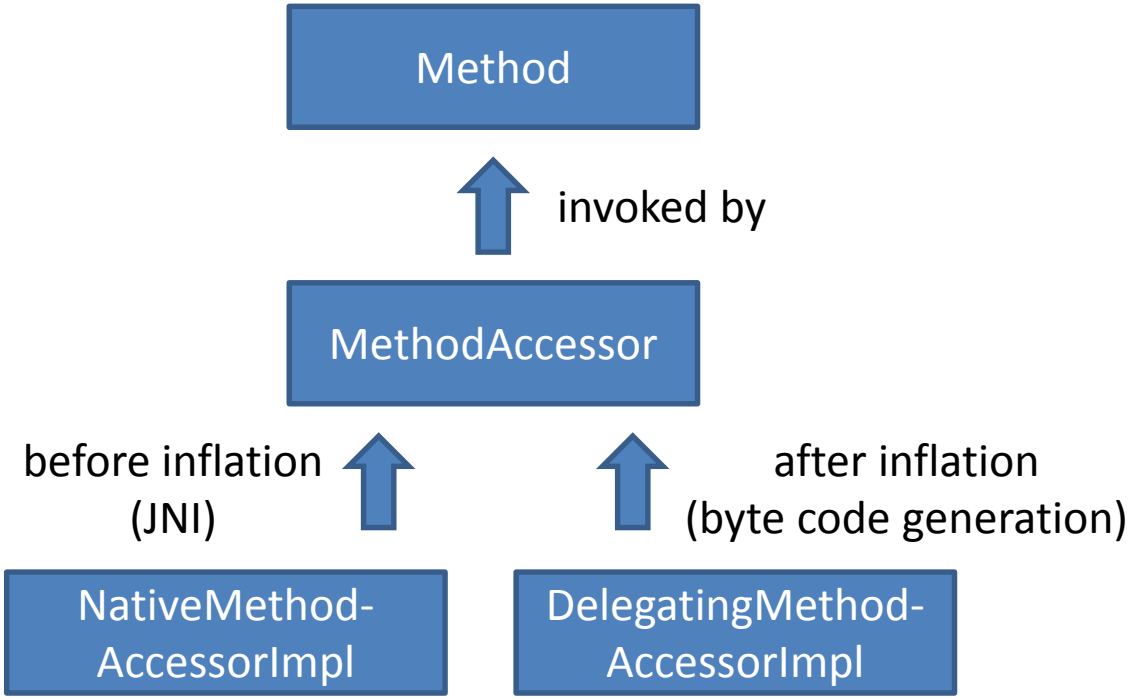
```
class Foo {
  public Object bar(Object o) { return o; }
}

Enhancer enhancer = new Enhancer();
enhancer.setSuperclass(Foo.class);
enhancer.setCallback(new MethodInterceptor() {
  @Override public Object intercept(Object obj,
      Method method,
      Object[] args,
      MethodProxy proxy) throws Throwable {
    return proxy.invokeSuper(obj, doSomethindWith(args));
  }});
Foo enhancedFoo = (Foo) enhancer.create();
```

Created classes: 2 + #methods * 2
[FastClass, FastMethod, MethodProxy]

avoid instrumentation

Modern JDK's inflation works similarly

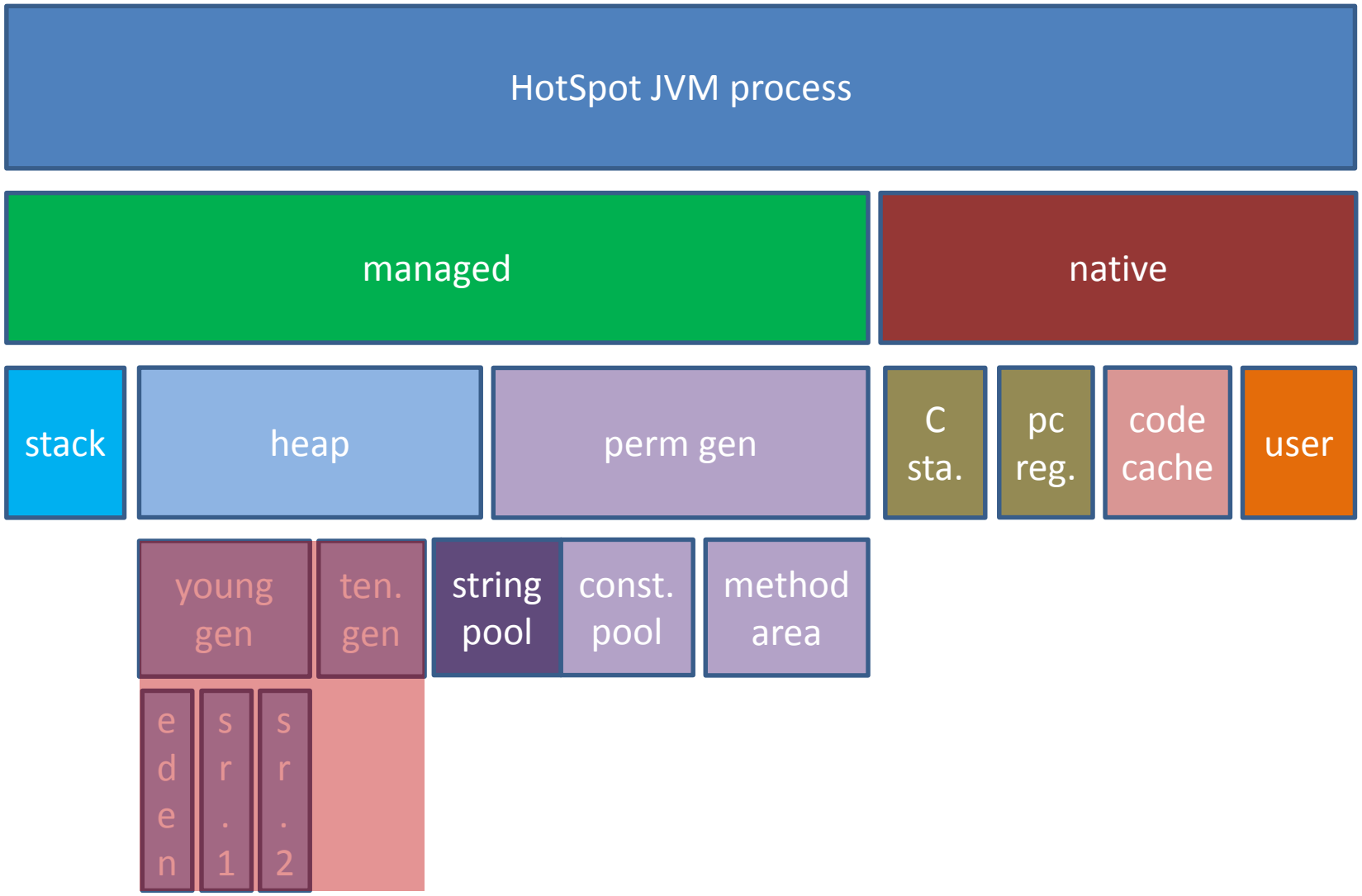
remember inflation



Java 8 meta space

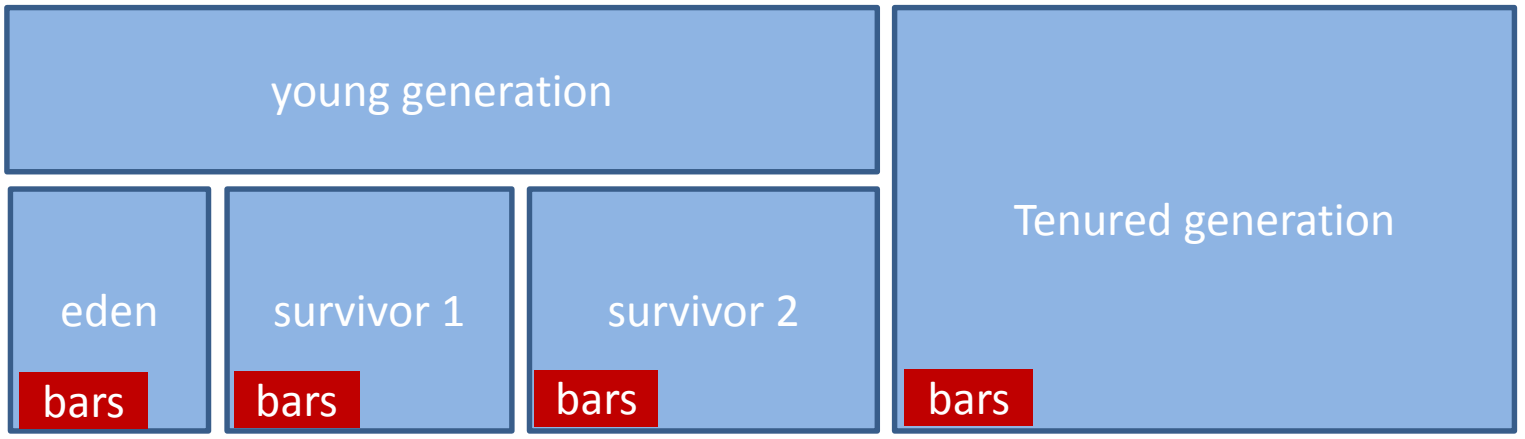
- Permanent generation rebranded (and probably approximation to JRockit)
- Meta space is allocated in native memory
- No space limit by default (*MaxMetaspaceSize*)
- Today‘s debugging tools will not be able to read meta space

Tenuring leaks



don‘t be scared of “new“,
avoid object pooling,
JIT knows best

```
public void foo() {
  for(int i = 0; i < 100; i++) {
    Set<Bar> bars = new HashSet<Bar>();
    for(int j = 0; j < 100; j++) {
      bars.add(makeBar(i, j));
    }
    doSomethingWith(bars);
  }
}
```



collection of survivor 2

Implicit memory leaks

- Leaked **threads**: Require fixed amount of memory for call stacks / pc register / general allocation
- Leaked **handles** (files, sockets, databases): Usually require JVM memory resources

debugging / profiling

```
jmap -dump:live,format=b,file=<...> <pid>

java -XX:+HeapDumpOnOutOfMemoryError
     -XX:+HeapDumpPath=<...> <...>

jhat <hprof file>
     select a from [I a where a.length >= 256
```

GUI tools: MAT (Eclipse RCP) / JVisualVM / HeapWalker

tools that create heap dumps by instrumentation require memory to run

http://rafael.codes
@rafaelcodes

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

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