Primitive Data and Objects

The programmer computes on data. Data in Java is divided into primitive data and non-primitive data.

`int` is primitive, `String` is not. `double` is primitive, arrays are not.

(Wrapper classes allow primitive data to be treated like objects. The advantage of using wrapper classes is that all data can be treated uniformly. The disadvantage is some extra overhead.)

Definition

A primitive data type in Java is one of the eight: `boolean`, `char`, `byte` , `short` , `int` , `long` , `float` and `double`.

These types are predefined, available everywhere in the program, unstructured, and have values with simple, short machine representations.

Definition

An object in Java is a value of that is not primitive, including arrays and strings.

Primitive Data and Reference Data

The main characteristics of primitive types:

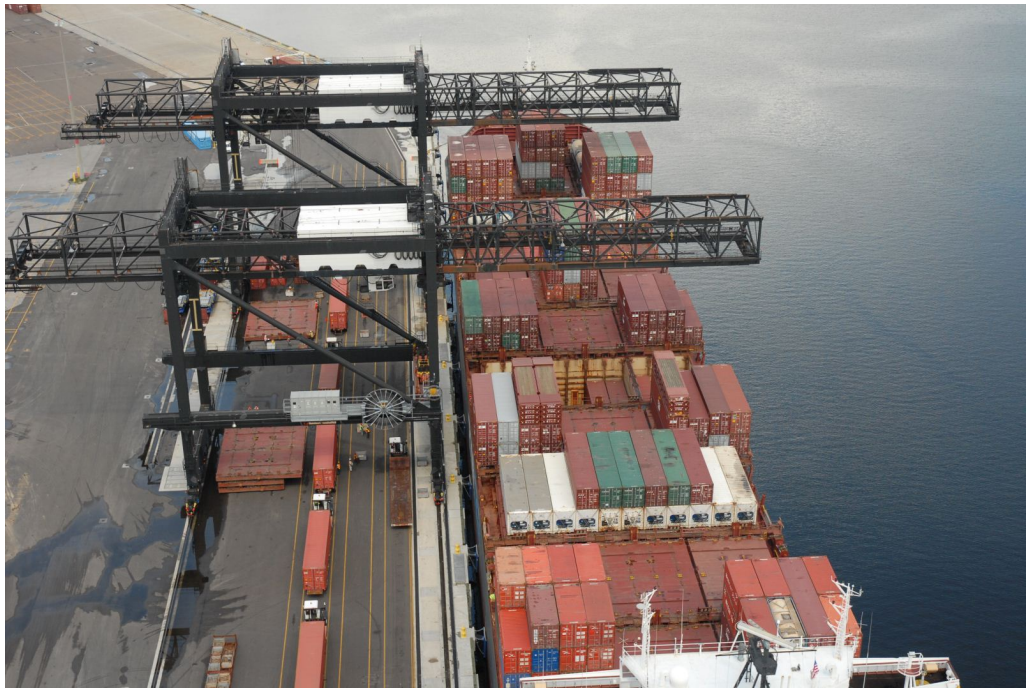
- available everywhere in any program
- uncomplicated (no substructure)
- all values fit in the machine's word size
- operations on the data are supported in the hardware

Instances of classes are allocated in a managed storage area called the heap and variables in the program refer indirectly to the instances. We call these instances *objects*, but referenced data might be a less overused term.

Primitive Data and Reference Data

The majority of data the programmer wants is not primitive. Java supports this vast, endless variety of data by allowing programmer to define new data types and by implementing some in the libraries. A large collection of data types is found in the extensive, standard Java libraries. User-defined data types are created using records and classes.

Data Abstraction

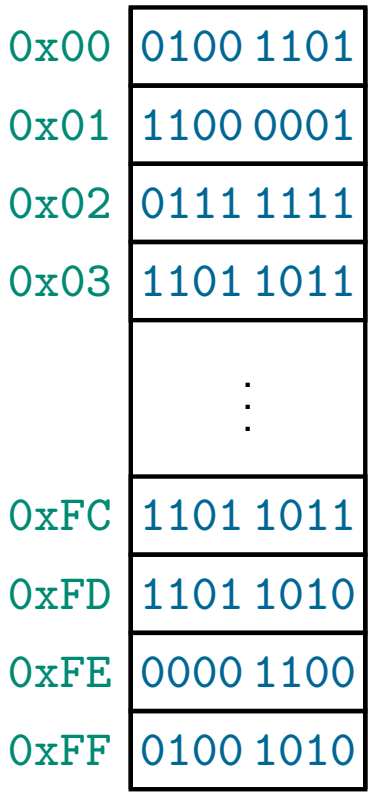


Good design enables easy handling and high volume

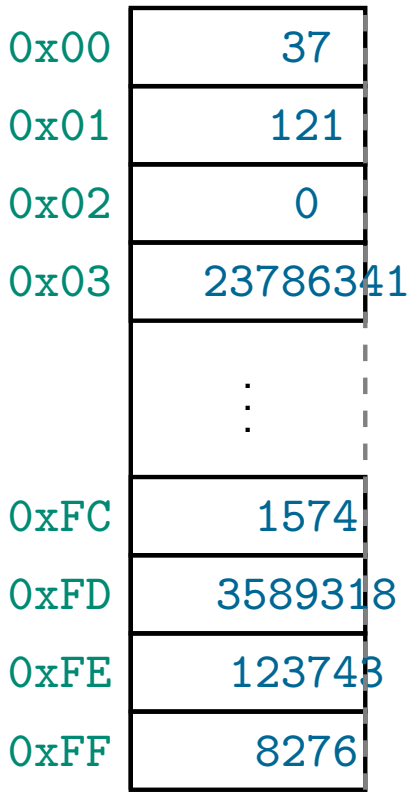
Data Abstraction

Programmer not only designs the flow of control (loops, etc), but also designs the value to compute with.
This may be the most important design task the programmer has, since the control flow could spring naturally from that data if the design is good

Storage–Finite Collection of Machine Words

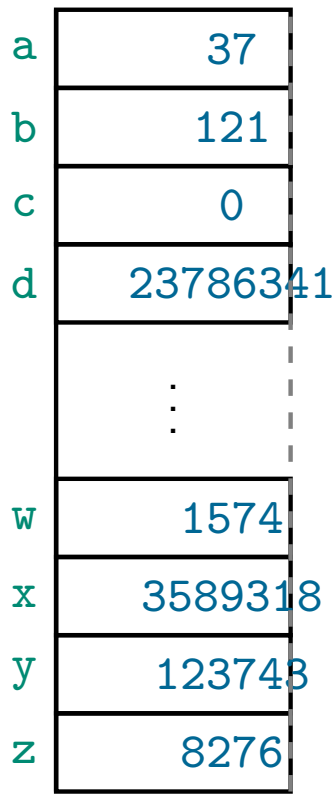


Storage–Finite Collection of Cells With Infinite Values



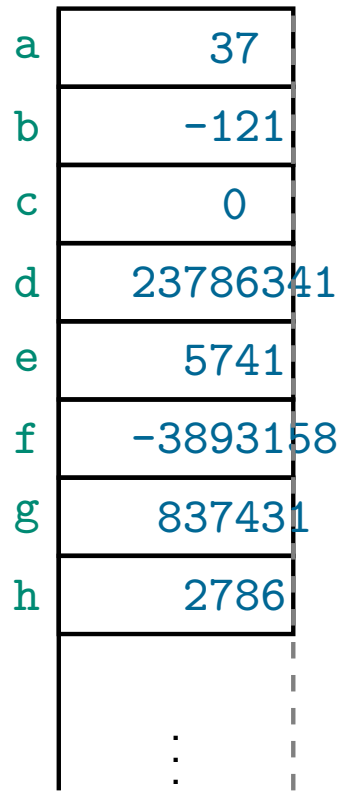
We sometimes pretend the cells hold arbitrarily large integers, but, of course, they can't.

Storage



Naming and labeling is necessary psychologically for using and organizing information. A key feature of a high-level programming language is power to label data (and other things).

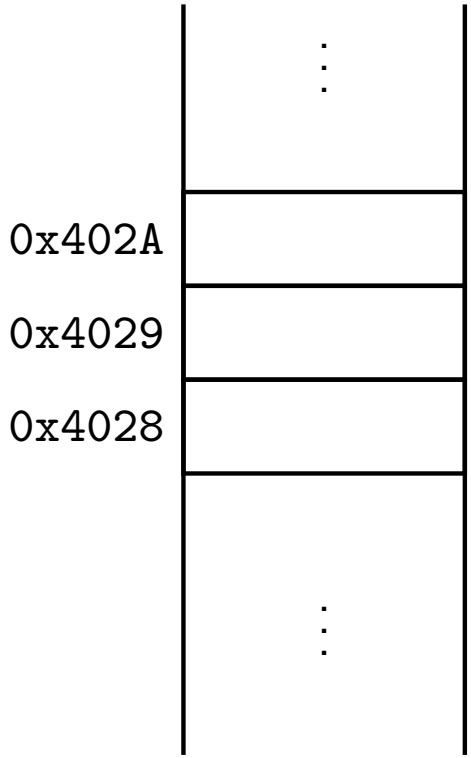
Storage



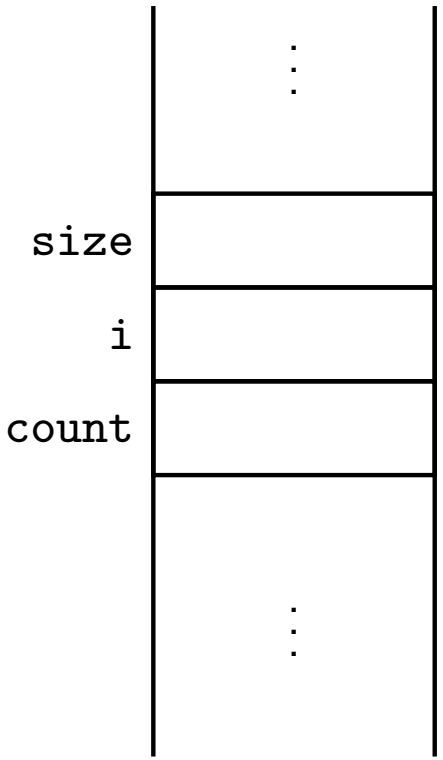
We sometimes pretend there are no end to the number of cells in the computer, but, of course, they aren't.

Storage

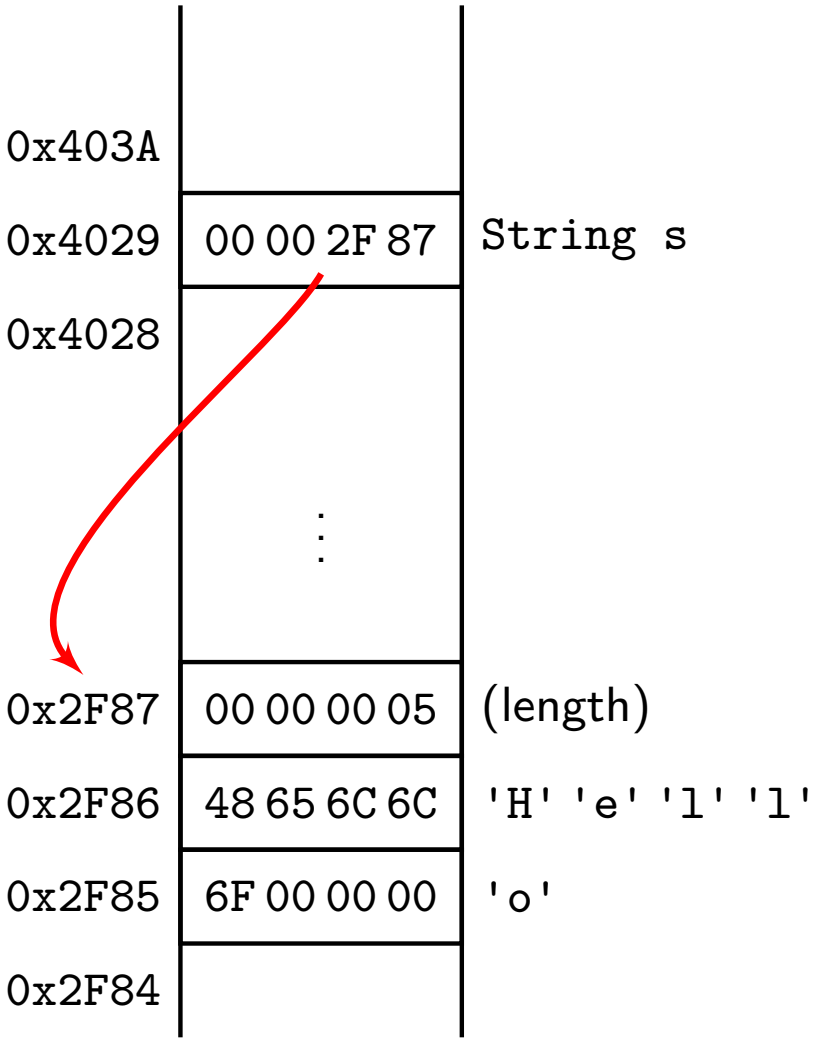
During execution the entire program is laid out in (virtual) storage, which we can envisioned as a gigantic array of words indexed by a (virtual) address. All the data appears somewhere in storage.



With Labels



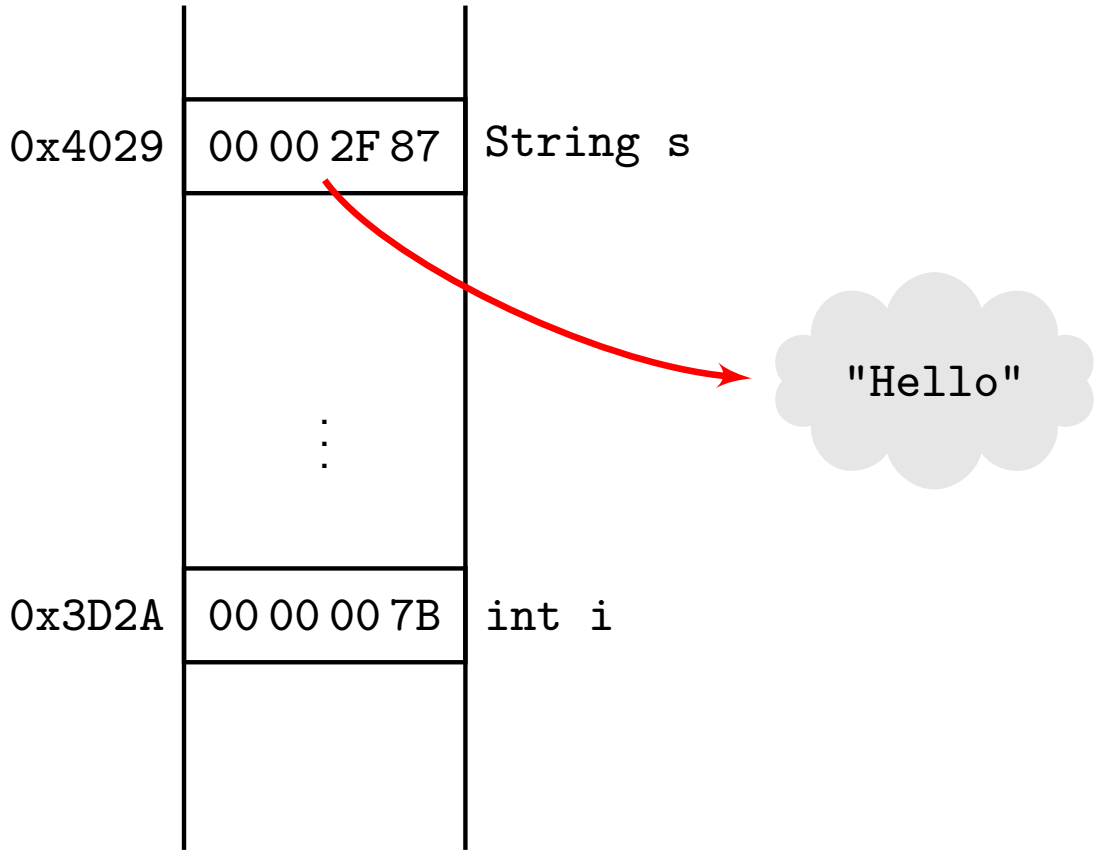
Layout of Program in Memory



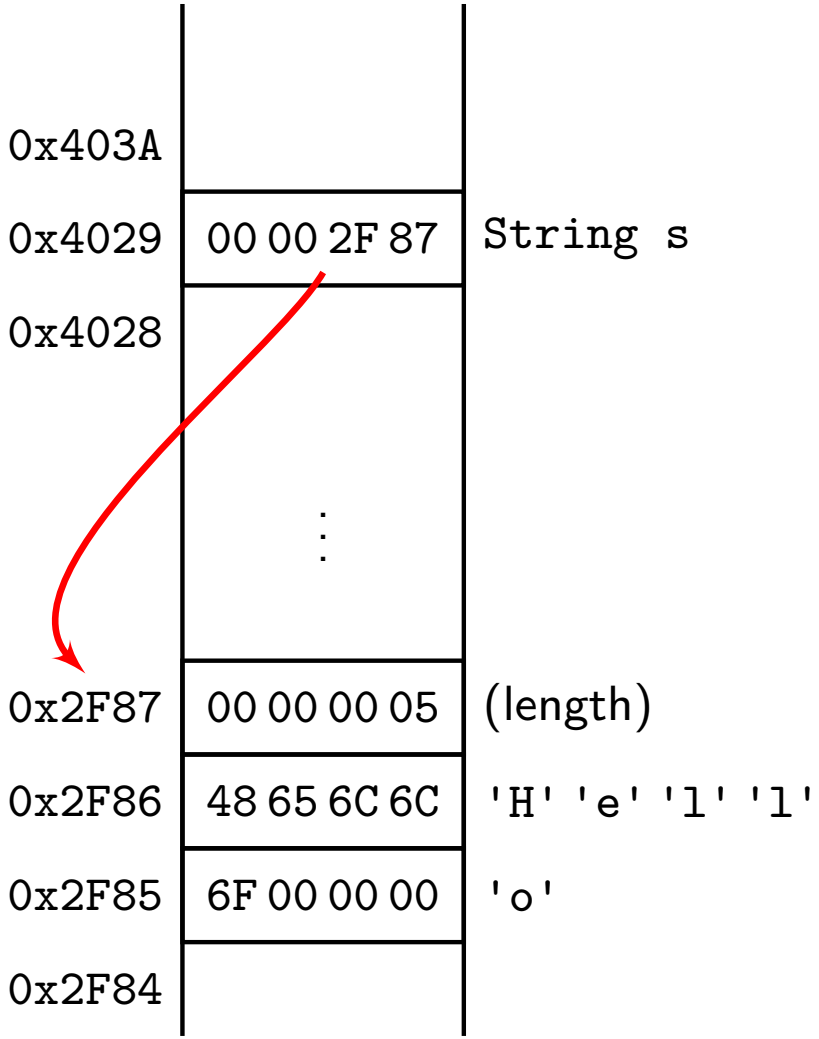
Primitive Data and Objects

Primitive data values are stored directly ("unboxed") and objects are stored indirectly ("boxed"). For example,

```
int i = 123;   String s = "Hello";
```



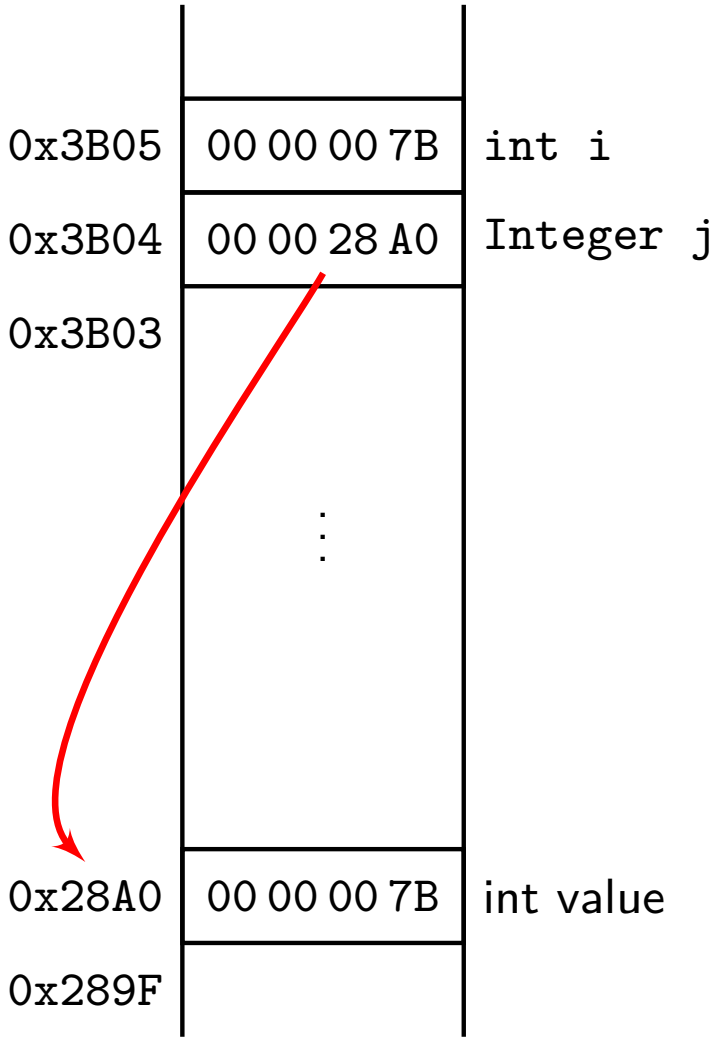
The space for objects is found in the heap. Of course, the heap must be found somewhere in storage.



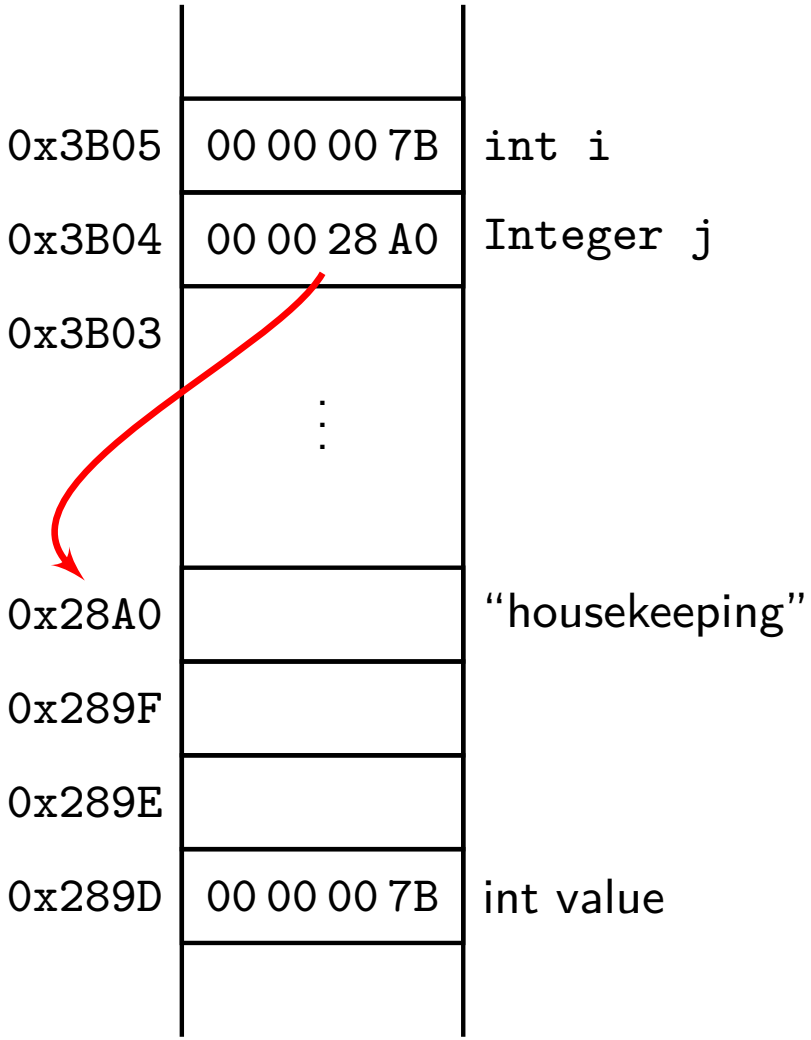
Other Languages

Other languages, like Haskell and C# do not require the programmer to distinguish between boxed and unboxed data. But both languages have boxed and unboxed values, integers, for instance. But in these languages the programmers only have one data type for integers. Unboxed is more efficient for computation and boxed is more uniform. Haskell and C# go back and forth between the two implementations automatically. Java too, goes back and forth automatically. But for each primitive data type there are two distinct types in the language: for example, `int` and `Integer`.

Comparing Integer Wrapper Class With Primitive `int`



Comparing Integer Wrapper Class With Primitive `int`



Creating Objects

When you declare a variable for a value of a primitive data type, an address is assigned which *must* hold a value of the primitive type (int, etc.)
When you declare a variable for objects, a box is assigned which can hold a reference to an instance of that type. No object/instance is created. An object/instance can only be created by `new`.
All non-primitive data objects are created (directly or indirectly) by executing `new`.
The syntax of the new expression:

```
new <class name> ( [ arguments ] )
```

it creates and returns an object of type `class name`.

There is an implicit `new` in special cases: strings, arrays, wrapper classes.

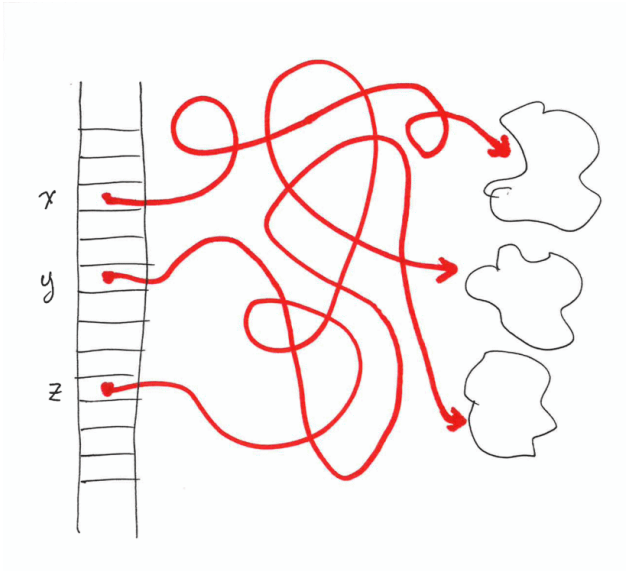
```
String s = new String("abc");  // redundant
Integer i = new Integer (123);
int[] a = new int[] {1,2,3};
```

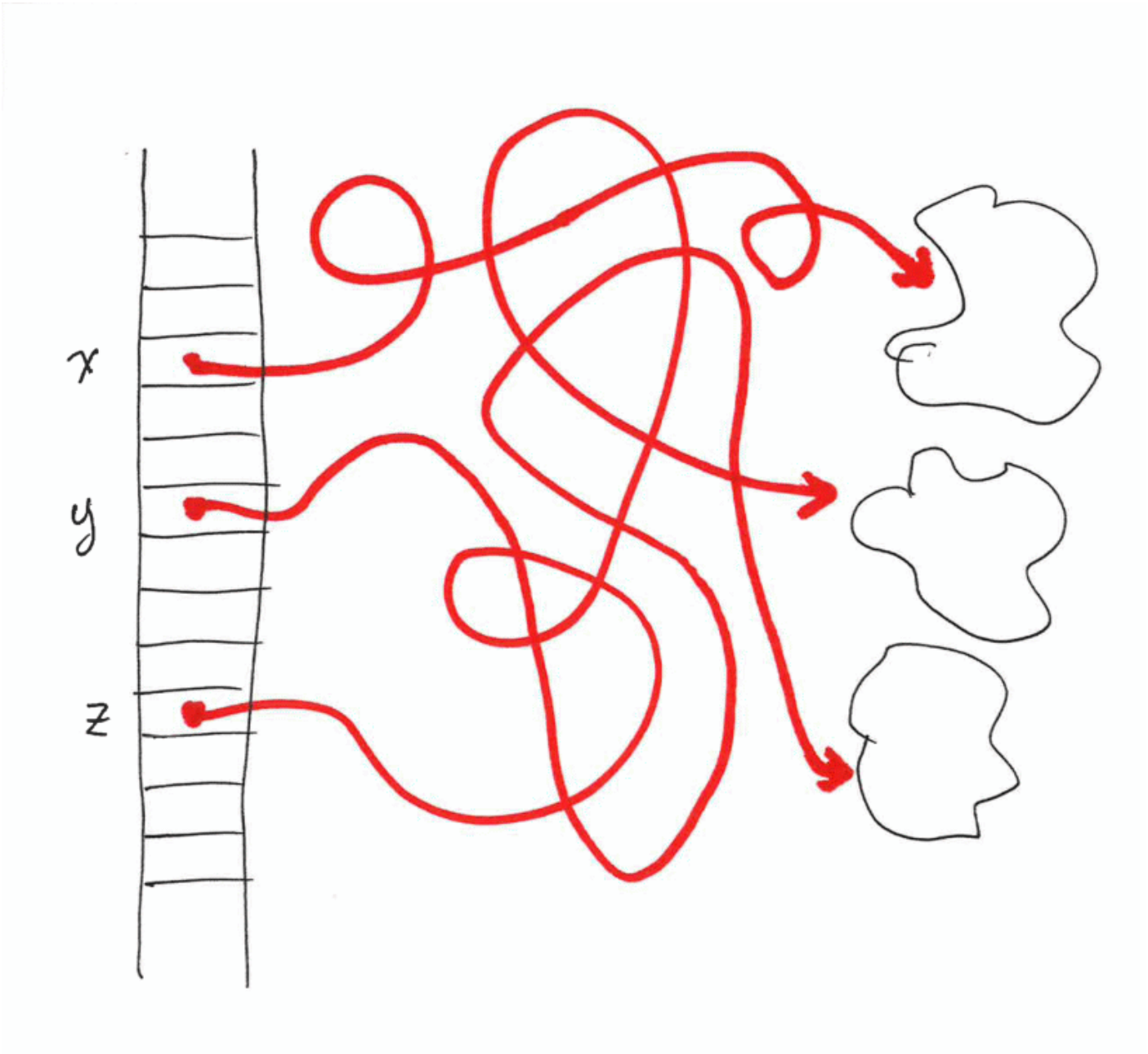
```
String s = "abc";
Integer i = 123;    // Auto-boxing!
int[] a = {1,2,3}; // new is optional in decl
```

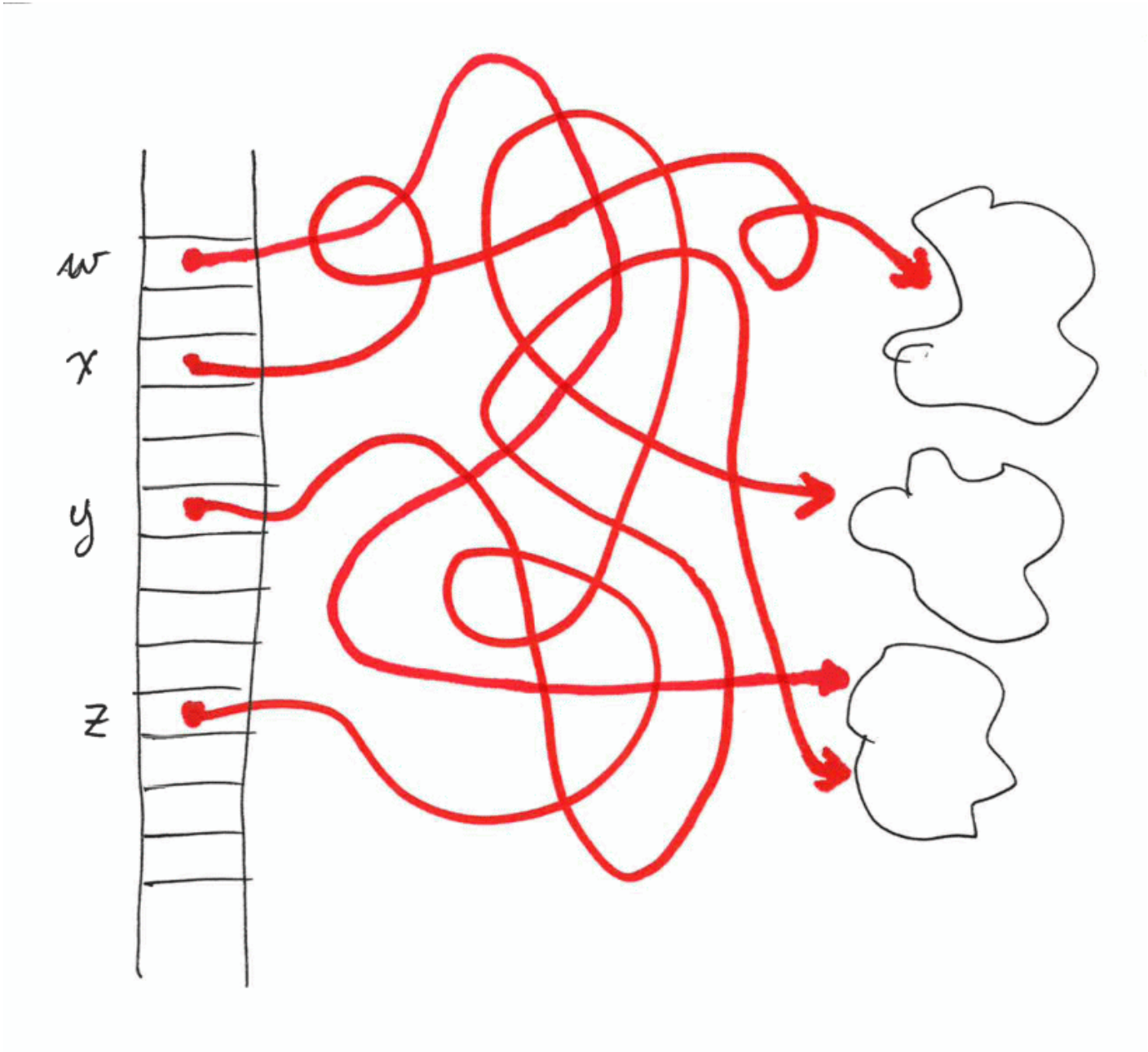
```
new java.lang.Object ()
new java.lang.StringBuilder (s)
new java.math.BigDecimal (203.99)
new java.awt.Color (r,g,b)
new java.util.Scanner (System.in)
new java.util.Locale (lang, country)
new java.io.File (dir, name)
new java.io.URL (protocol, host, file)
new java.util.ArrayList<String> ()
new javax.swing.JApplet ()
```

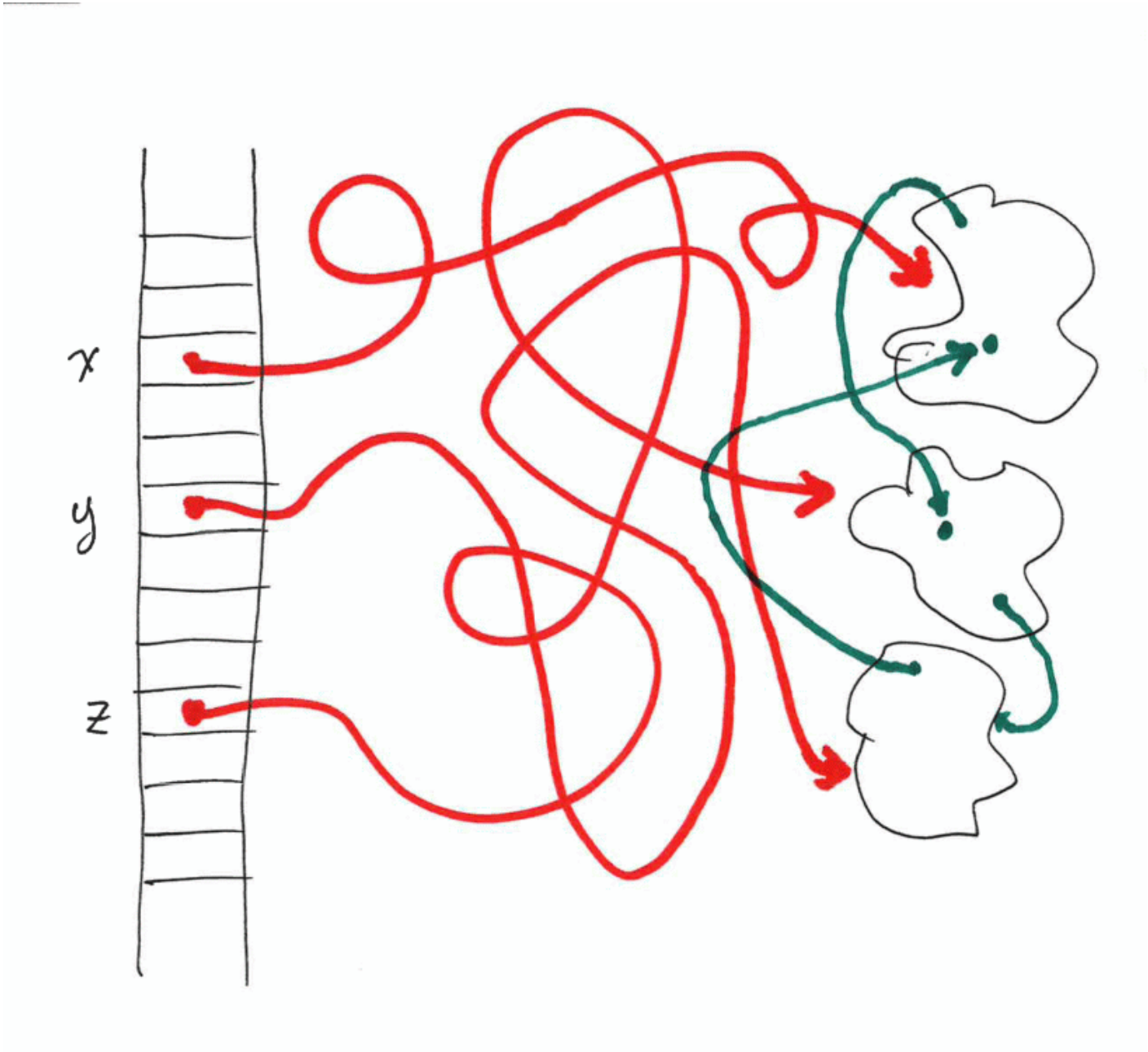
- 0, 1, or more arguments
- package name may be omitted if one uses an `import`
- generic instantiation
- some classes are never instantiated, i.e, `Math`
- other ways to get objects: "factory methods," methods which create the object for you.

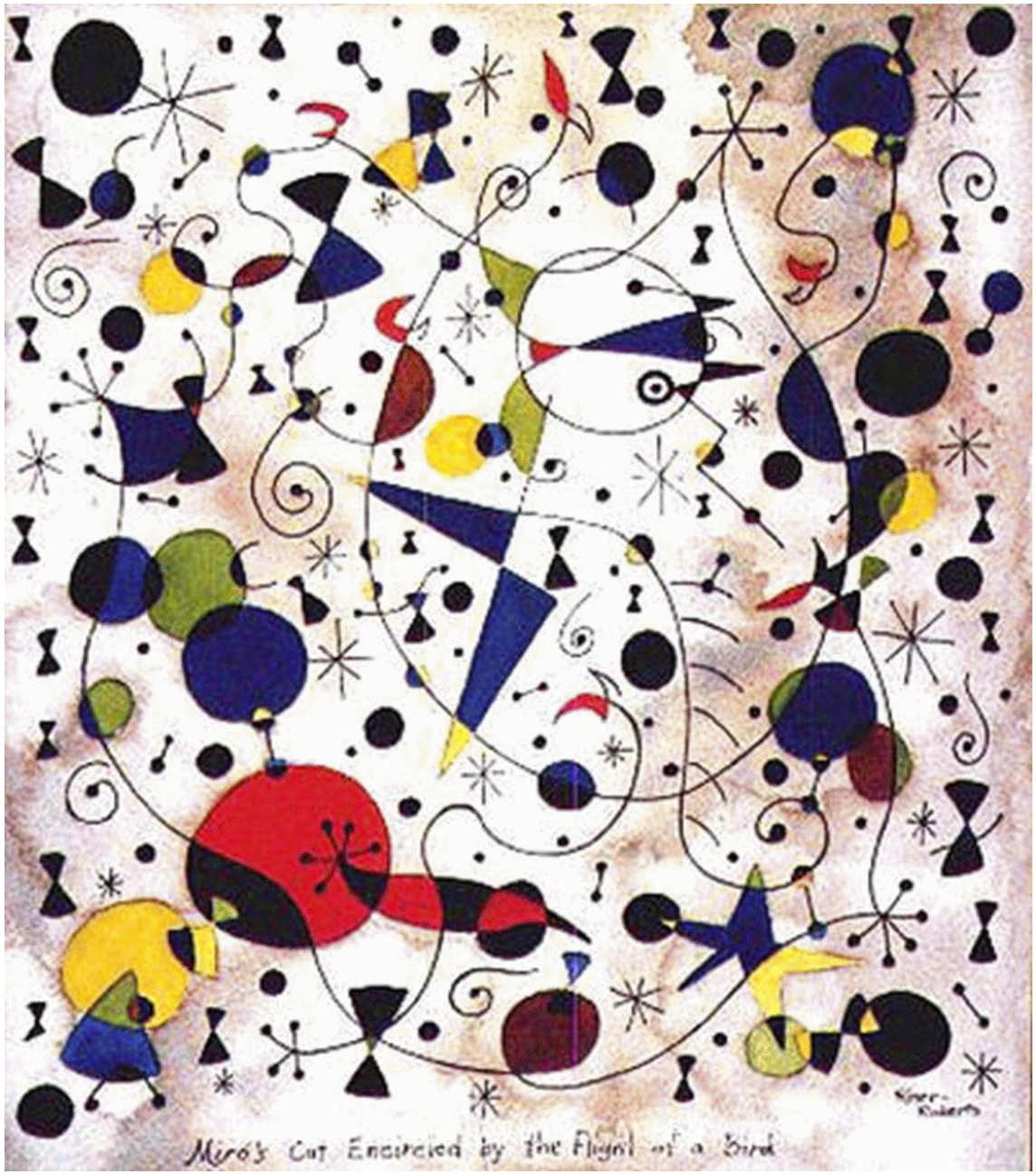
It is (regrettably) necessary to have a clear picture of boxed and unboxed values in your mind, in order to program correctly in Java.
Let me try to illustrate what I think is in my head.




x
y
z


w
x
y
z


x
y
z



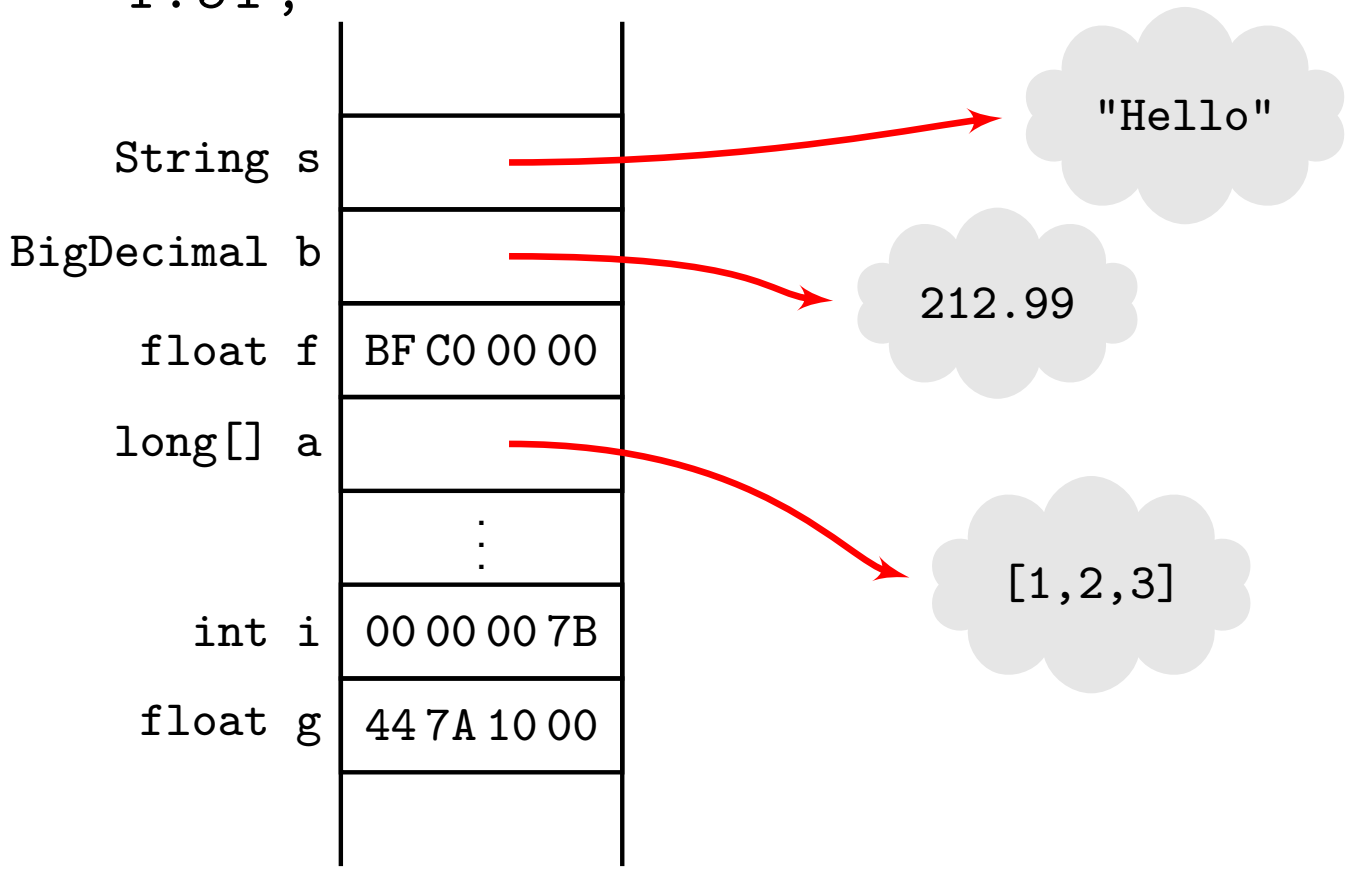
Joan Miró i Ferrà (1893–1983), *Frustrated Cat*

```
int i = 123;
short b = 8;
long a = 876L
float g = 1000.25f;
double s = 6.6d
float f = -1.5f;
```

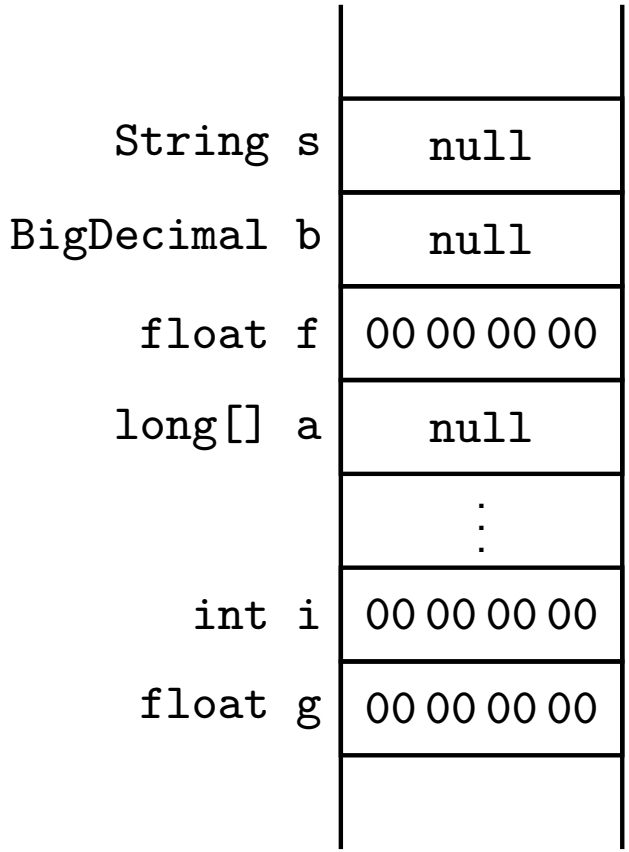
```
i : 123
b : 8
a : 876L
g : 1000.25f
s : 6.6d
f : -1.5f
```

```
int i = 123;
short b = 8;
long a = 876L;
float g = 1000.25f;
double s = 6.6d
float f = -1.5f;
a = 564L;
b = 1;
a = 488L;

i : 123
b : 8 / 1
a : 876L / 564L / 488L
g : 1000.25f
s : 6.6d
f : -1.5f;
```

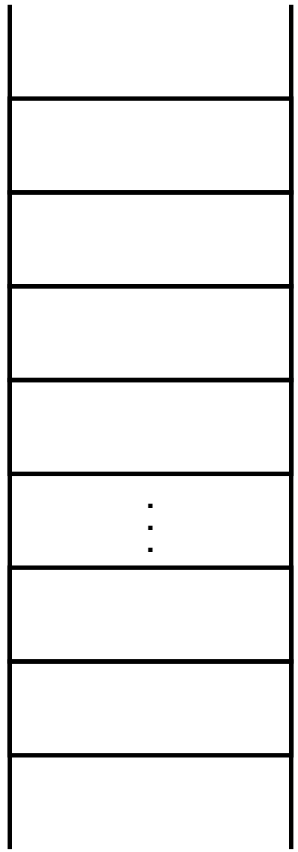
```
int i = 123;
BigDecimal b = new BigDecimal (212.99);
long[] a = new long[] {1,2,3};
float g = 1000.25f;
String s = "Hello";
float f = -1.5f;
```



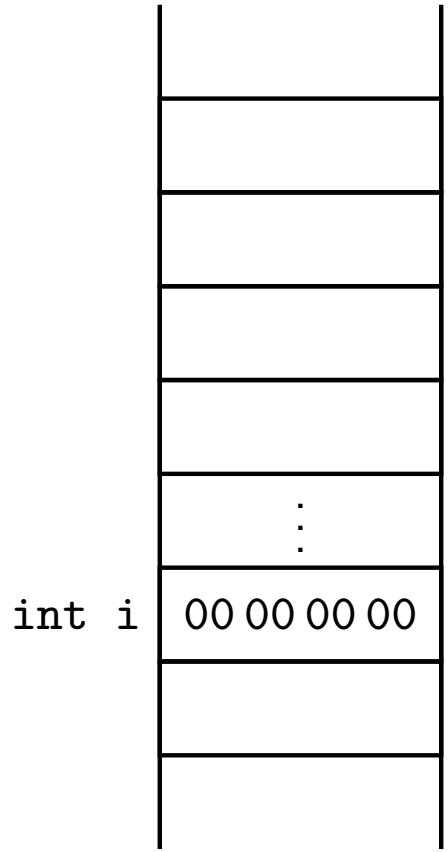
```
int i;
BigDecimal b;
long[] a;
float g;
String s;
float f;
```



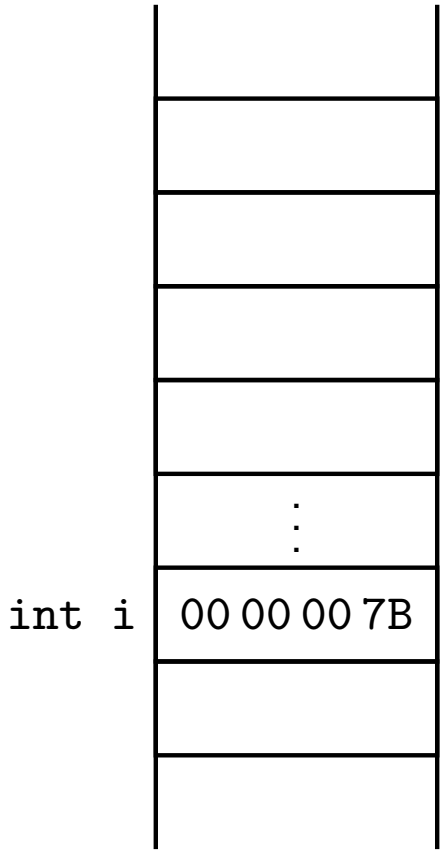
```
i = 123;
b = new BigDecimal (212.99);
a = new long[] {1,2,3};
g = 1000.25f;
s = "Hello";
f = -1.5f;
```



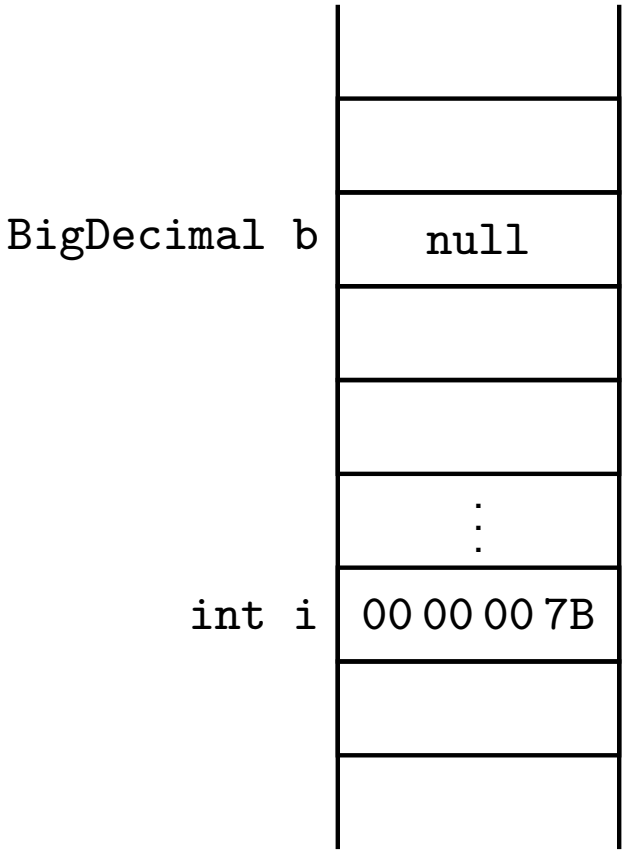
```
i = 123;
b = new BigDecimal (212.99);
a = new long[] {1,2,3};
g = 1000.25f;
s = "Hello";
f = -1.5f;
```



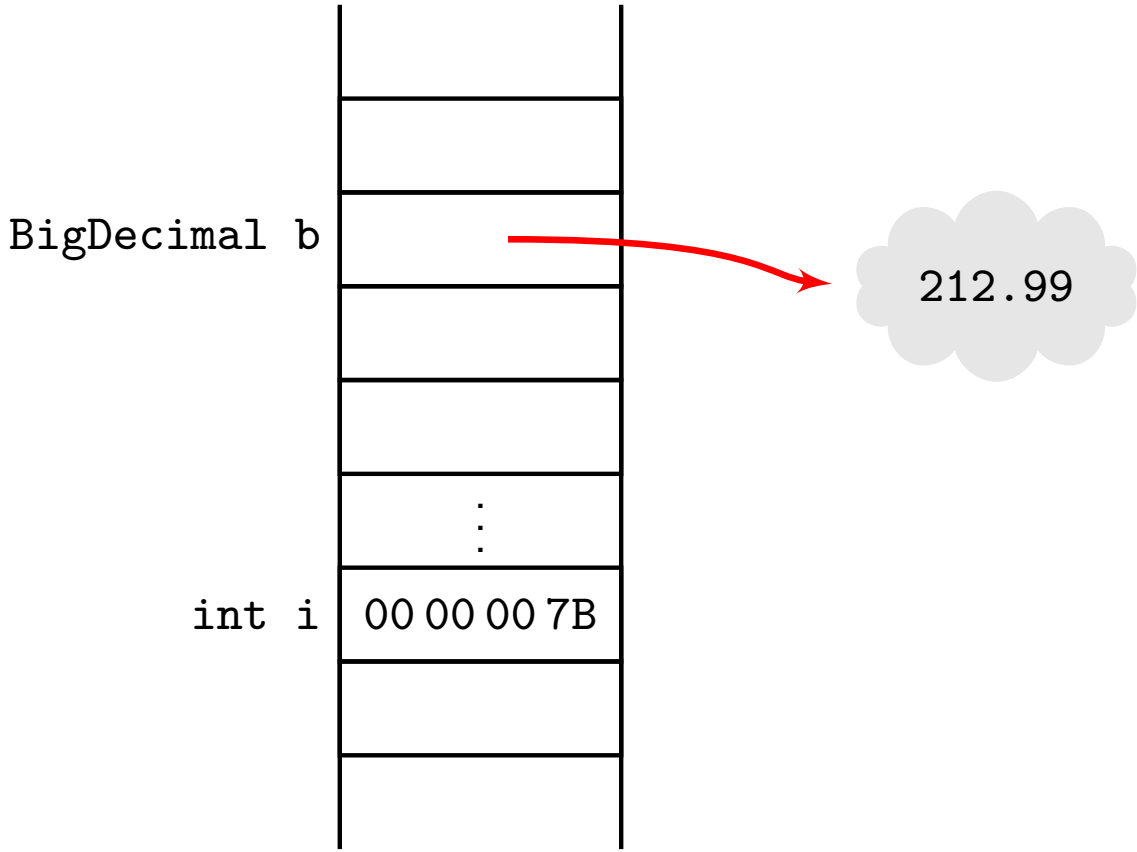
```
i = 123;
b = new BigDecimal (212.99);
a = new long[] {1,2,3};
g = 1000.25f;
s = "Hello";
f = -1.5f;
```



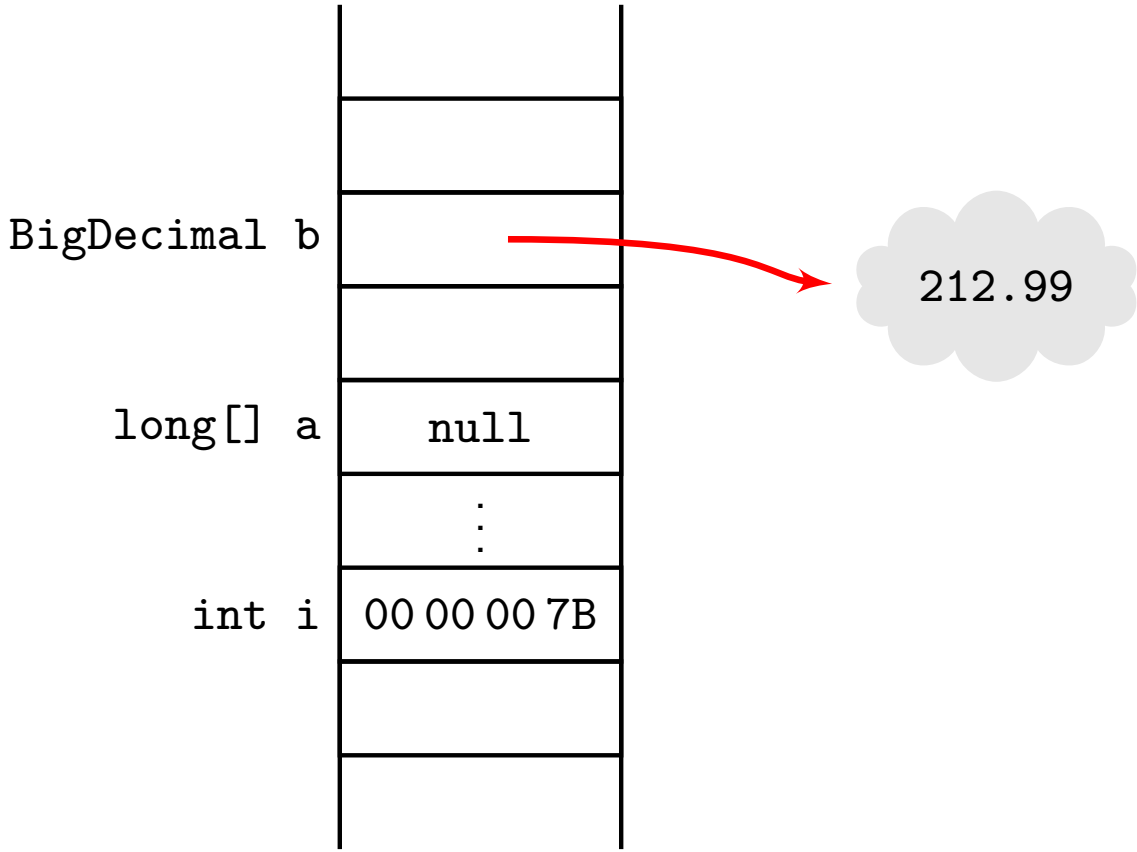
```
i = 123;
b = new BigDecimal (212.99);
a = new long[] {1,2,3};
g = 1000.25f;
s = "Hello";
f = -1.5f;
```



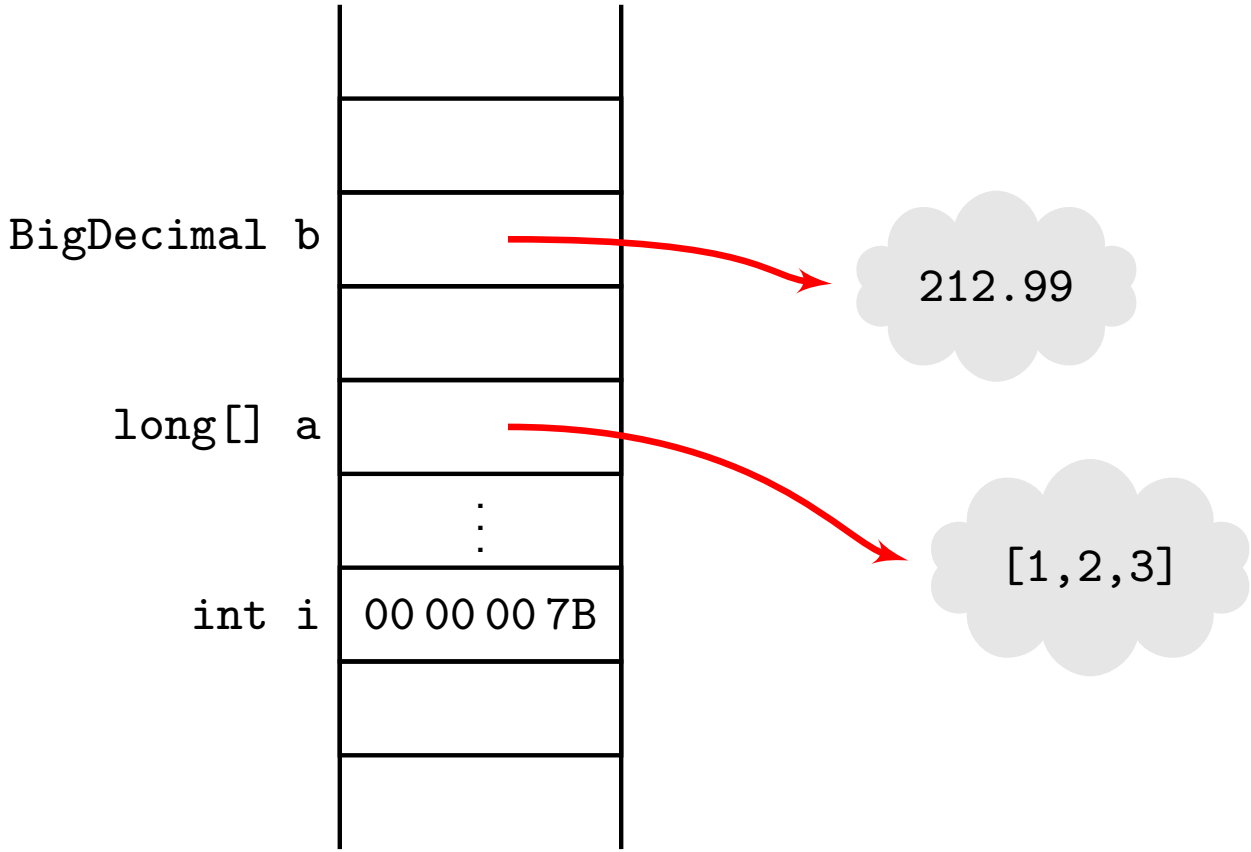
```
i = 123;
b = new BigDecimal (212.99);
a = new long[] {1,2,3};
g = 1000.25f;
s = "Hello";
f = -1.5f;
```



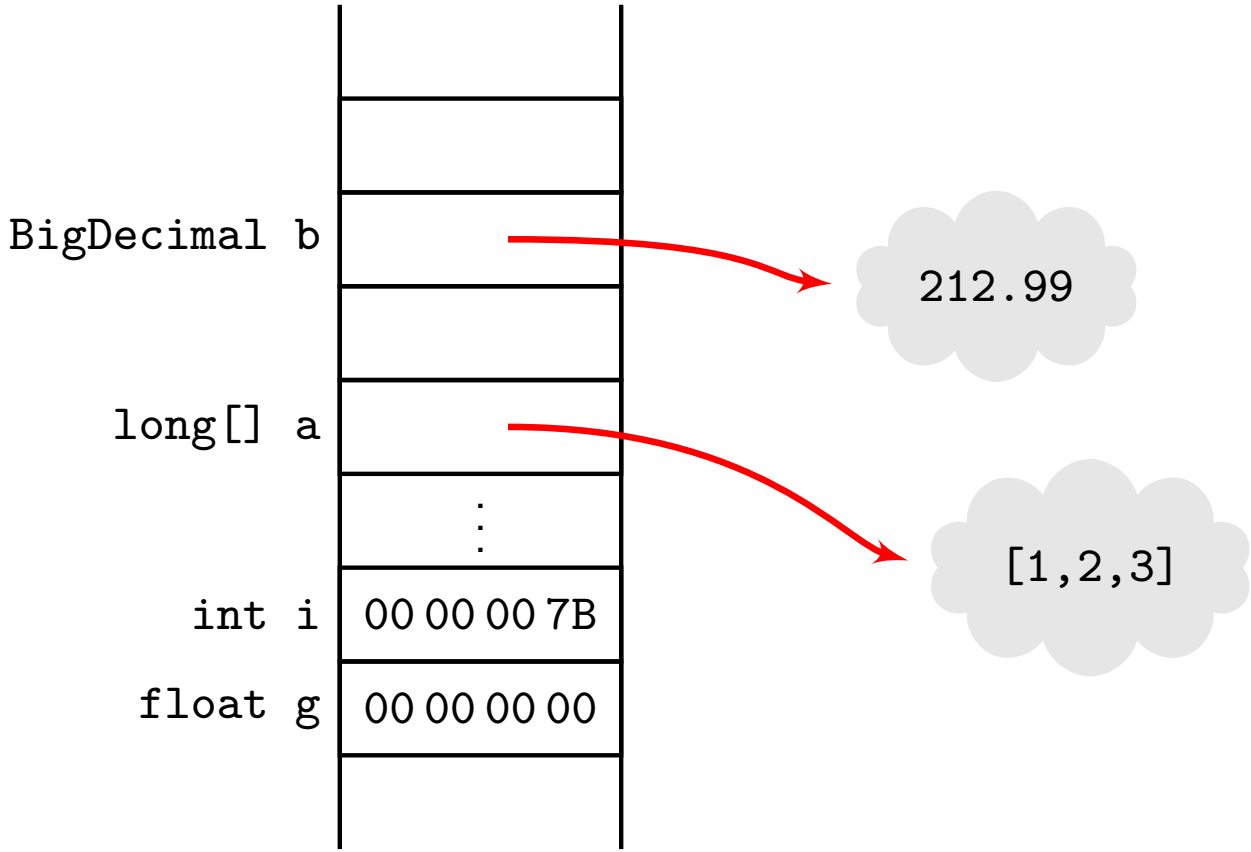
```
i = 123;
b = new BigDecimal (212.99);
a = new long[] {1,2,3};
g = 1000.25f;
s = "Hello";
f = -1.5f;
```



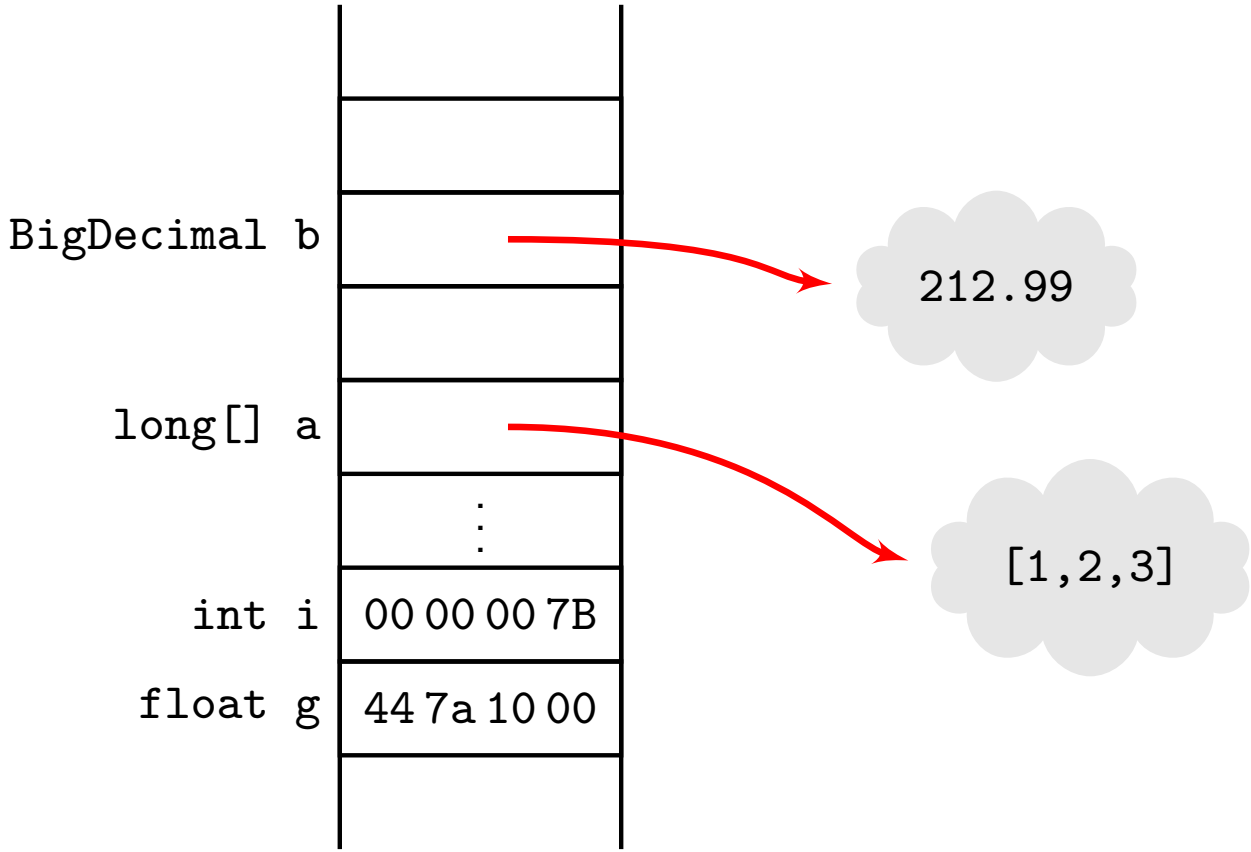
```
i = 123;
b = new BigDecimal (212.99);
a = new long[] {1,2,3};
g = 1000.25f;
s = "Hello";
f = -1.5f;
```



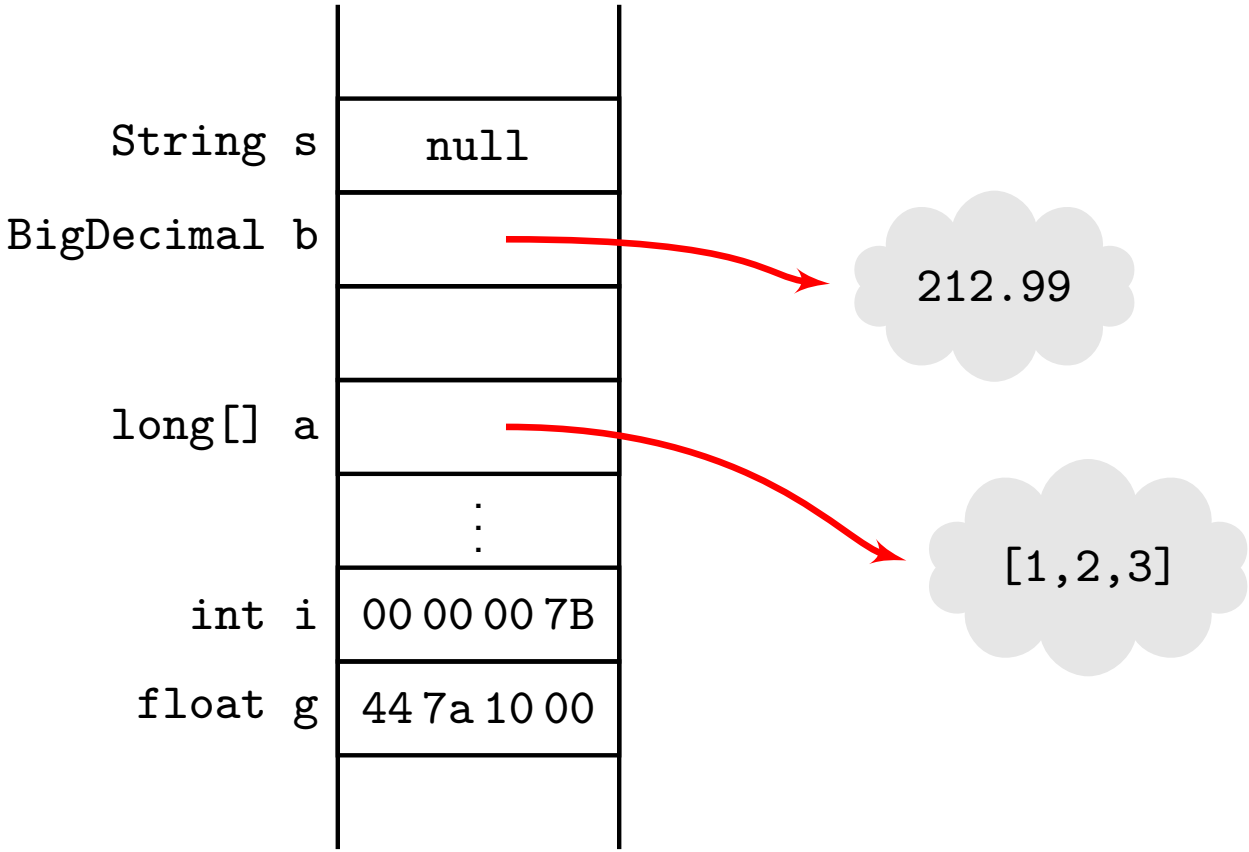
```
i = 123;
b = new BigDecimal (212.99);
a = new long[] {1,2,3};
g = 1000.25f;
s = "Hello";
f = -1.5f;
```



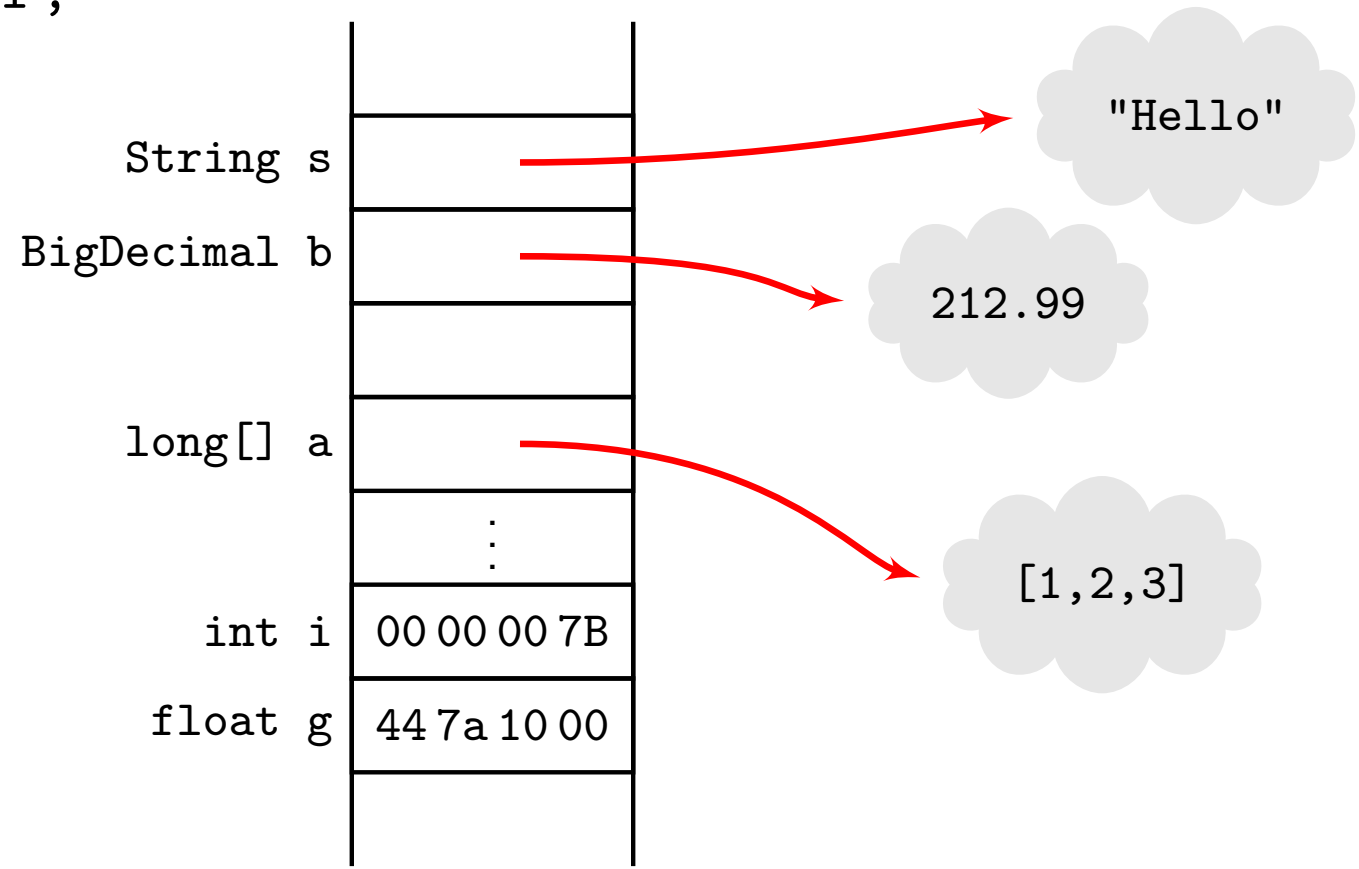
```
i = 123;
b = new BigDecimal (212.99);
a = new long[] {1,2,3};
g = 1000.25f;
s = "Hello";
f = -1.5f;
```



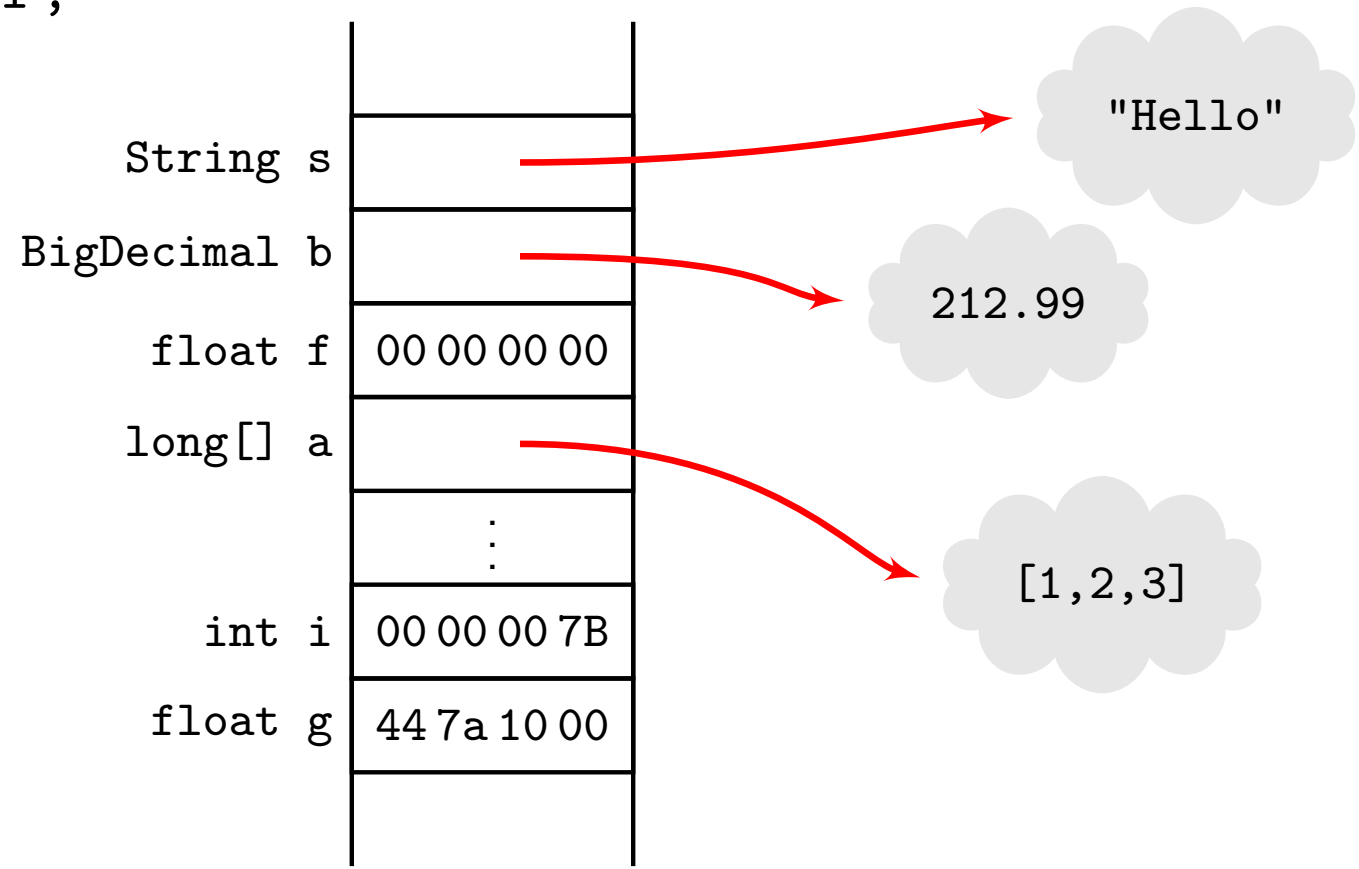
```
i = 123;
b = new BigDecimal (212.99);
a = new long[] {1,2,3};
g = 1000.25f;
s = "Hello";
f = -1.5f;
```



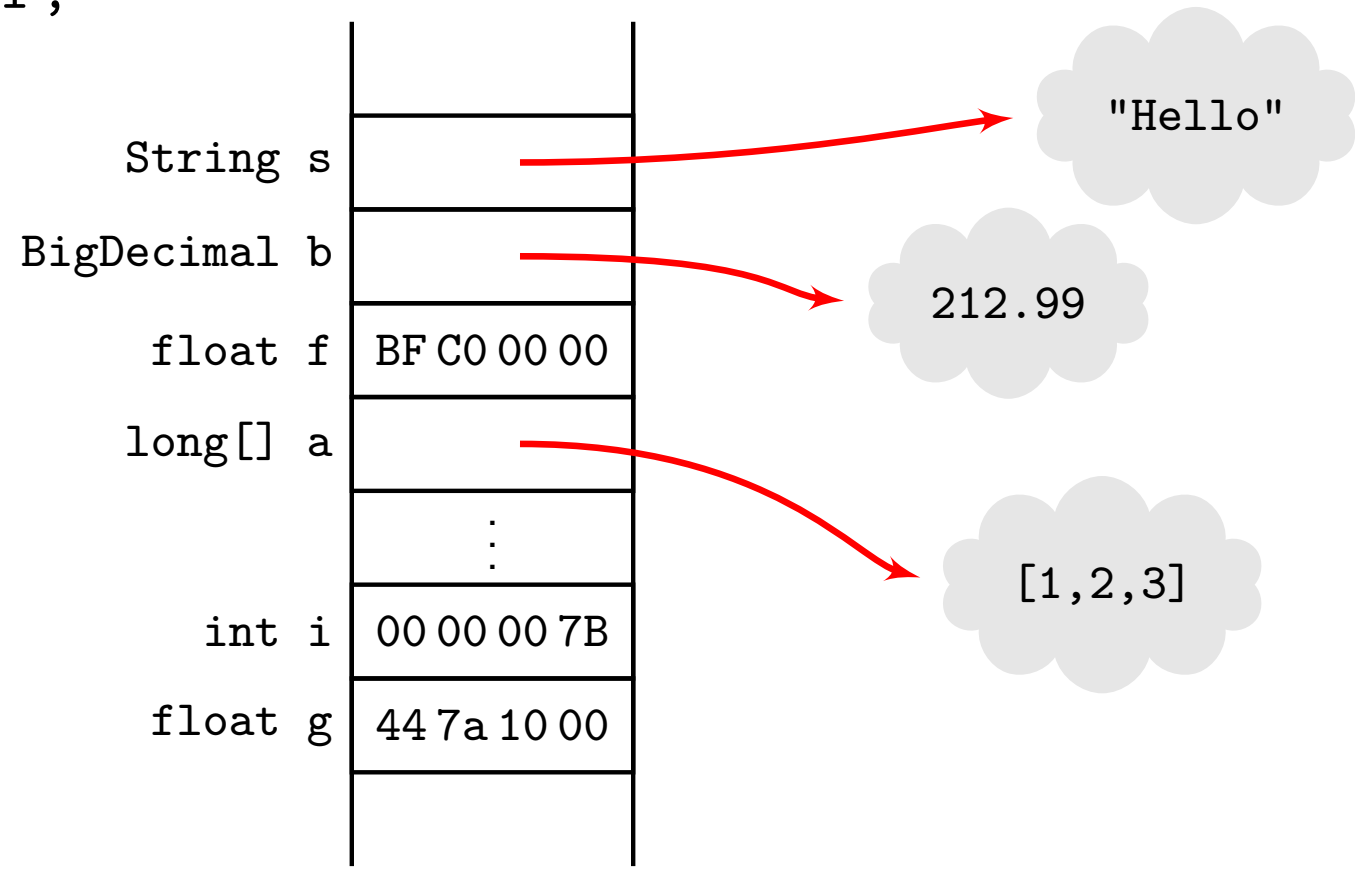
```
i = 123;
b = new BigDecimal (212.99);
a = new long[] {1,2,3};
g = 1000.25f;
s = "Hello";
f = -1.5f;
```



```
i = 123;
b = new BigDecimal (212.99);
a = new long[] {1,2,3};
g = 1000.25f;
s = "Hello";
f = -1.5f;
```



```
i = 123;
b = new BigDecimal (212.99);
a = new long[] {1,2,3};
g = 1000.25f;
s = "Hello";
f = -1.5f;
```



Null

Non-primitive variable are initialized to the special value null.
Null is a legal value for all non-primitive types.
Unintentionally accessing a null object is a very common problem which results in a `NullPointerException`.

```
String s;
long [] a;
char c = s.charAt(17);
long l = a[39];
```

Java catches some (but not all) initialization errors.

```java
public class Main {
    public static void main (String[] args) {
        String s;
        long [] a;
        char c = s.charAt(17);
        long l = a[39];
    }
}
```

```
> javac Main.java
Main.java:5: variable s might not have been initialized
      char c = s.charAt(17);
               ^
Main.java:6: variable a might not have been initialized
      long l = a[39];
               ^
2 errors
```

```
public class Fields {
    static String s;
    static long [] a;
    public static void main (String[] args) {
        char c = s.charAt(17);
        long l = a[39];
    }
}
```

```
> java Fields
Exception in thread "main" java.lang.NullPointerException
        at Fields.main(Fields.java:5)
```

> *I call it my billion-dollar mistake. It was the invention of the null reference in 1965. At that time, I was designing the first comprehensive type system for references in an object oriented language (ALGOL W). My goal was to ensure that all use of references should be absolutely safe, with checking performed automatically by the compiler. But I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement. This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years.*

Sir Charles Anthony Hoare

Optional

OVERVIEW MODULE PACKAGE CLASS USE TREE DEPRECATED INDEX HELP

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

Module java.base
Package java.util

Class Optional<T>

java.lang.Object
    java.util.Optional<T>

Type Parameters:

T - the type of value

```
public final class Optional<T>
extends Object
```

A container object which may or may not contain a non-null value. If a valu

Additional methods that depend on the presence or absence of a contained v

This is a value-based class; programmers should treat instances that are equ

API Note:

`Optional` is primarily intended for use as a method return type where there
`Optional` instance.

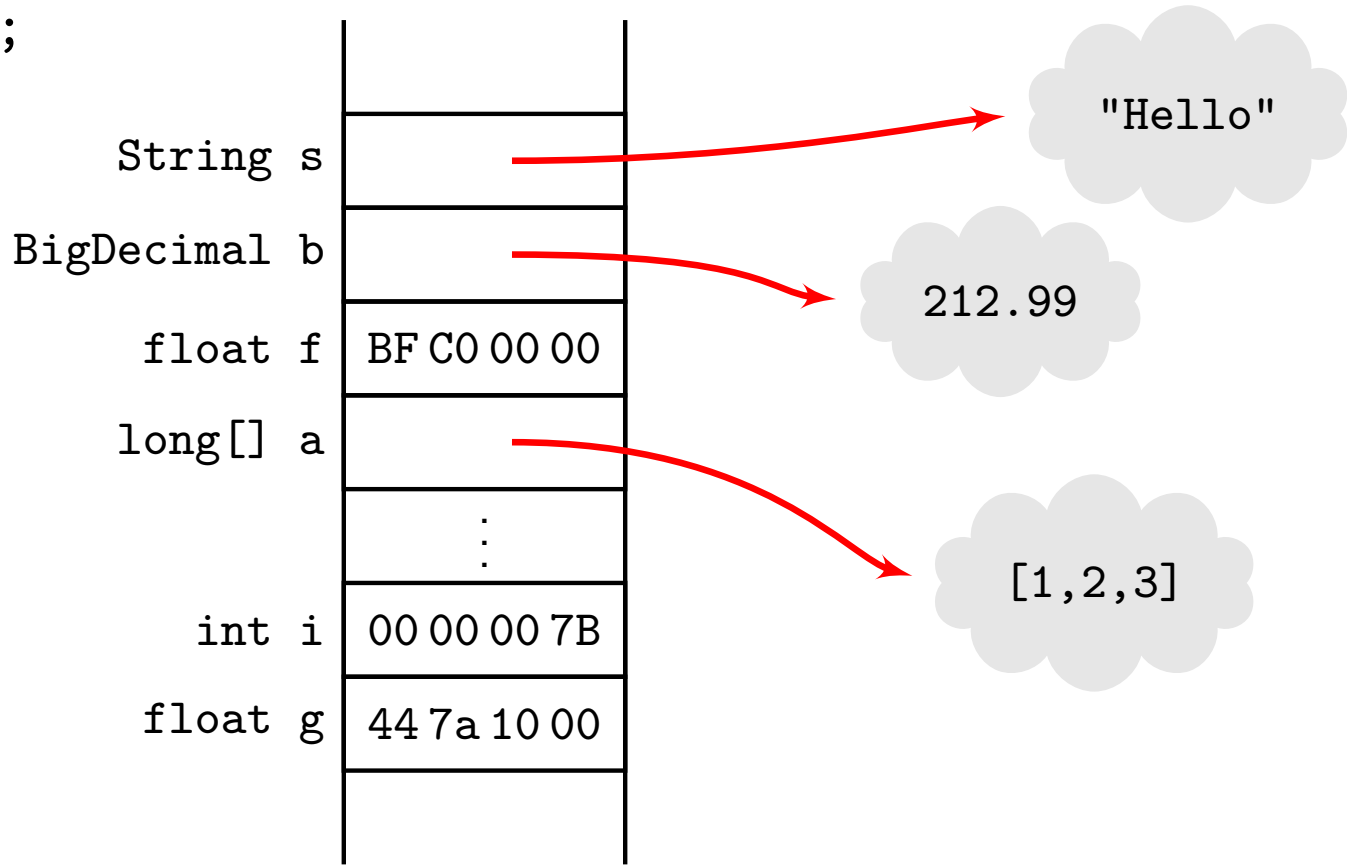
Since:

1.8

`Objects.requireNonNullElse`

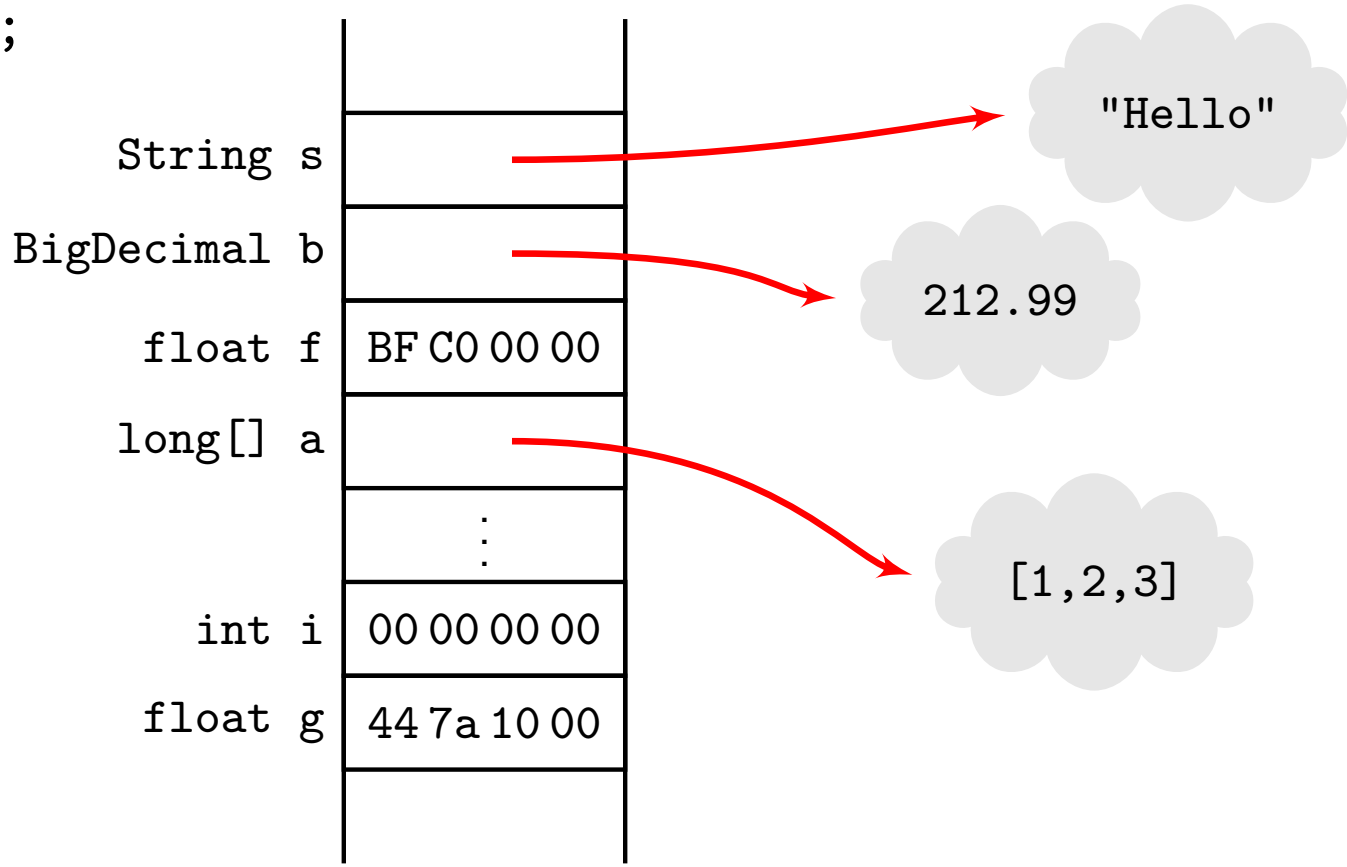
Garbage

```
i = 0;
b = null;
a = null;
g = 0.0f;
s = null;

f = 0.0f;
```



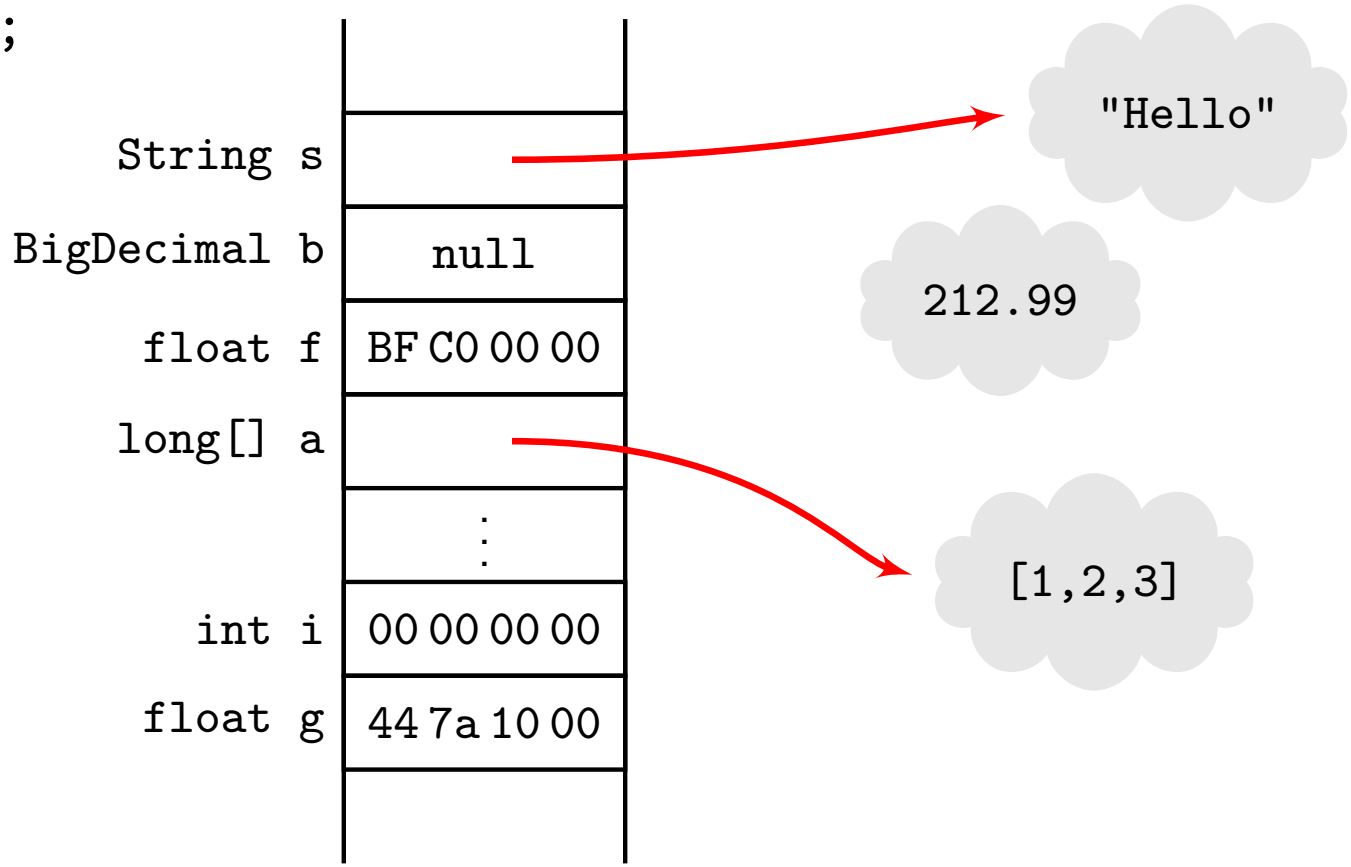
Garbage

```
i = 0;
b = null;
a = null;
g = 0.0f;
s = null;

f = 0.0f;
```



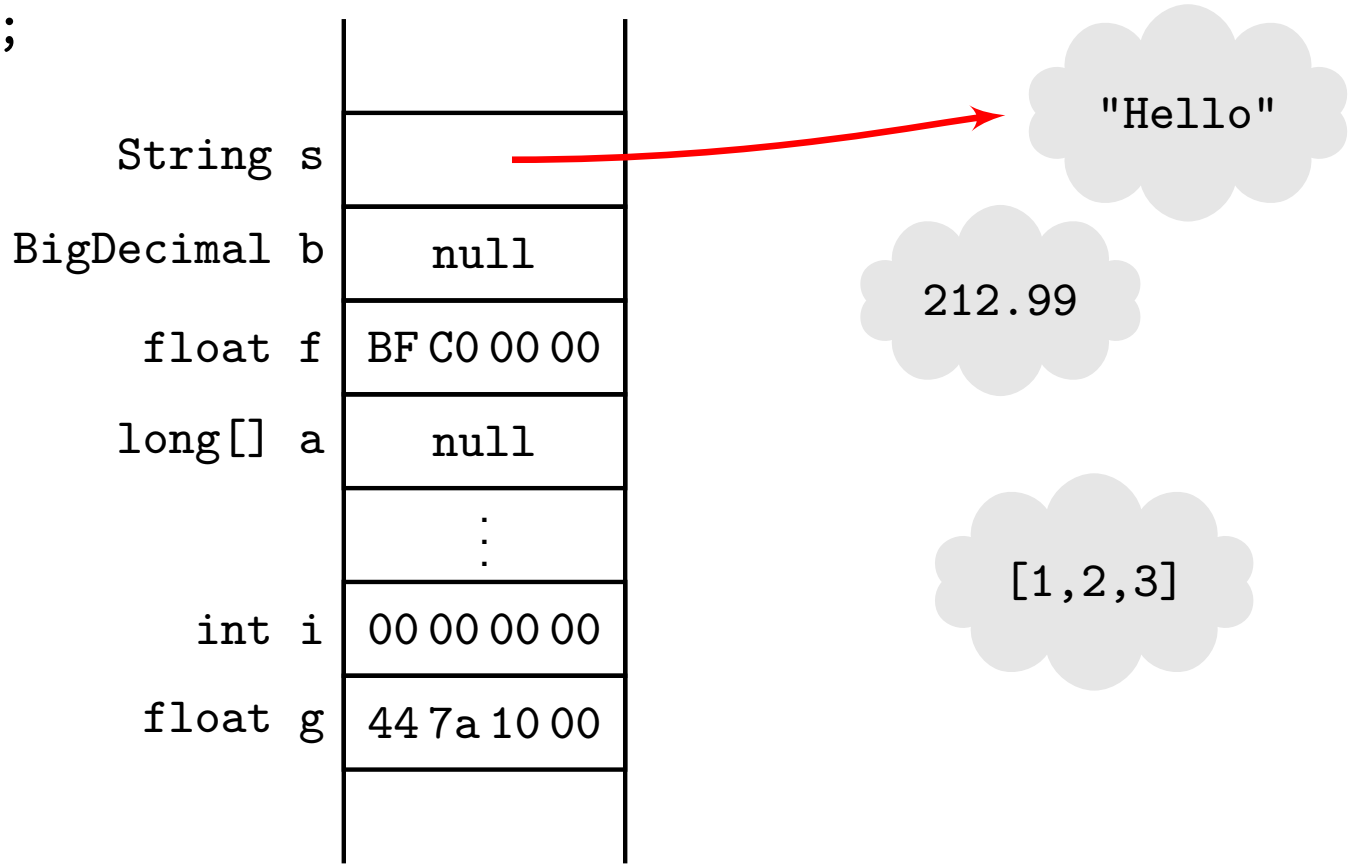
Garbage

```
i = 0;
b = null;
a = null;
g = 0.0f;
s = null;

f = 0.0f;
```



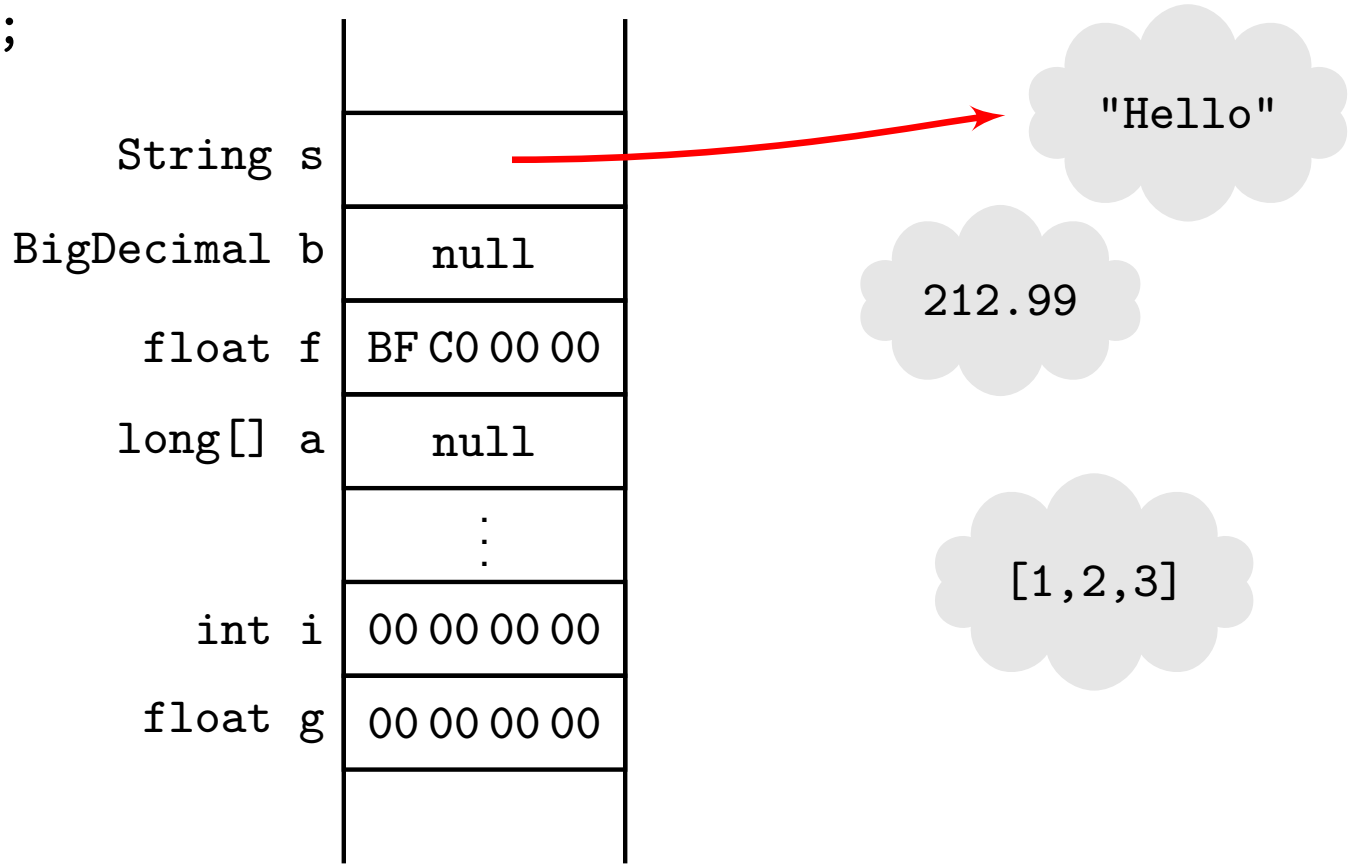
Garbage

```
i = 0;
b = null;
a = null;
g = 0.0f;
s = null;

f = 0.0f;
```



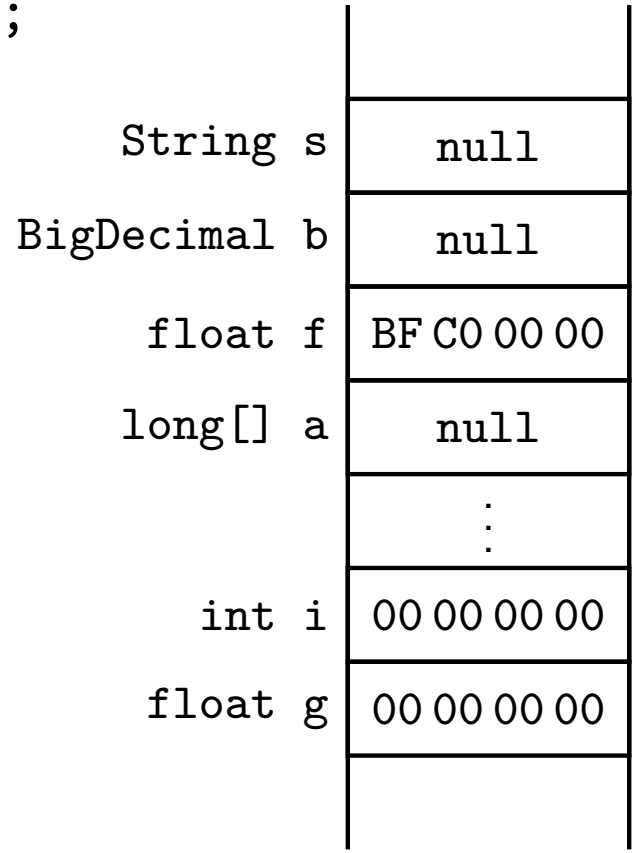
Garbage

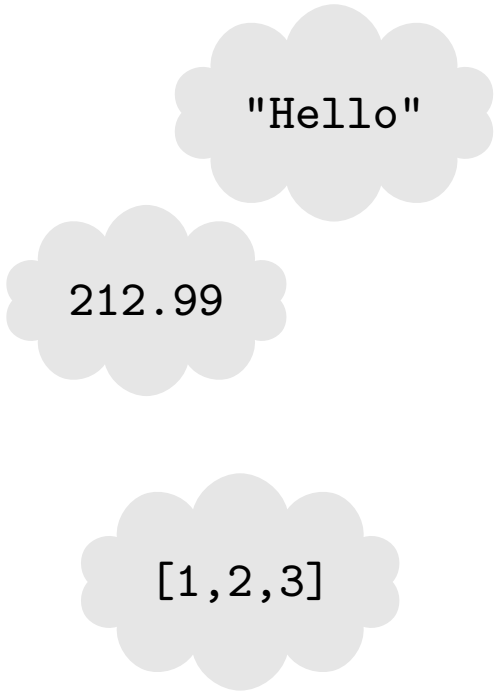
```
i = 0;
b = null;
a = null;
g = 0.0f;
s = null;

f = 0.0f;
```



Garbage

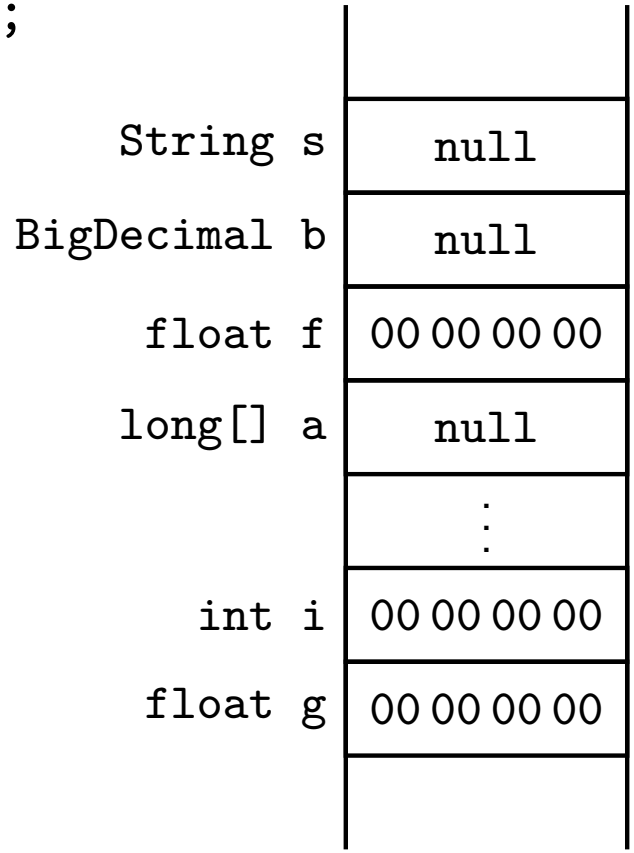
```
i = 0;
b = null;
a = null;
g = 0.0f;
s = null;

f = 0.0f;
```

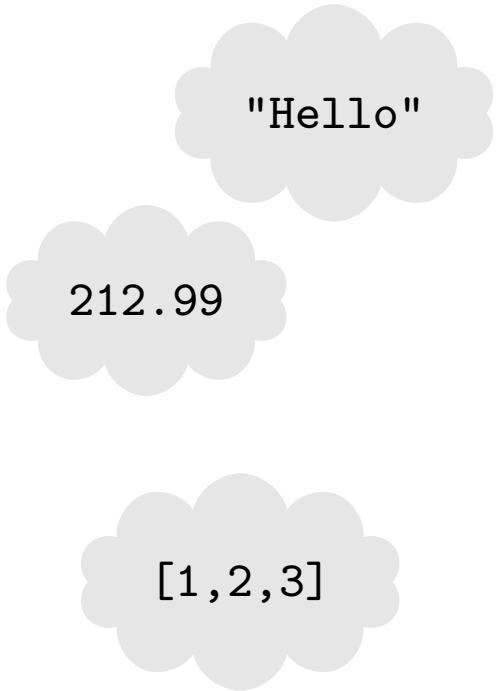




Garbage

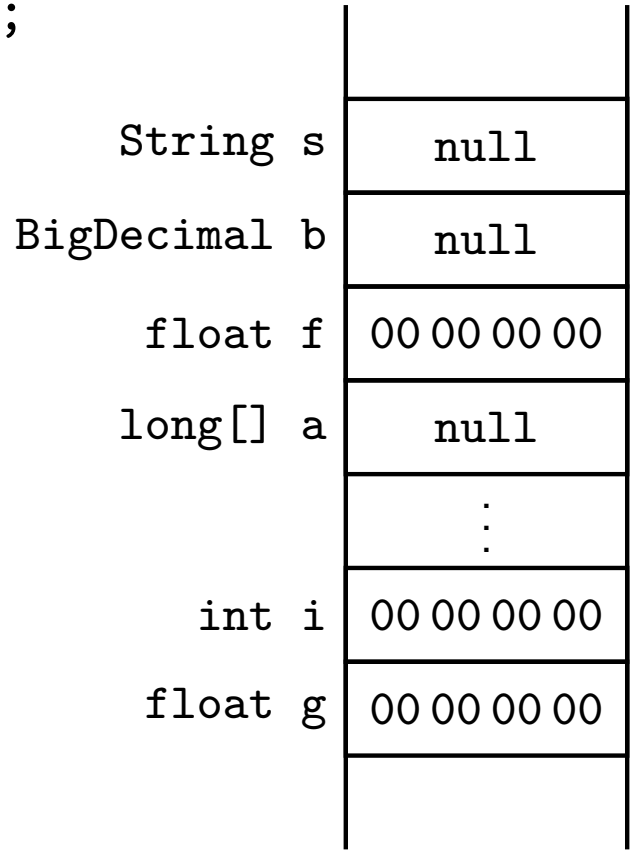
```
i = 0;
b = null;
a = null;
g = 0.0f;
s = null;

f = 0.0f;
```

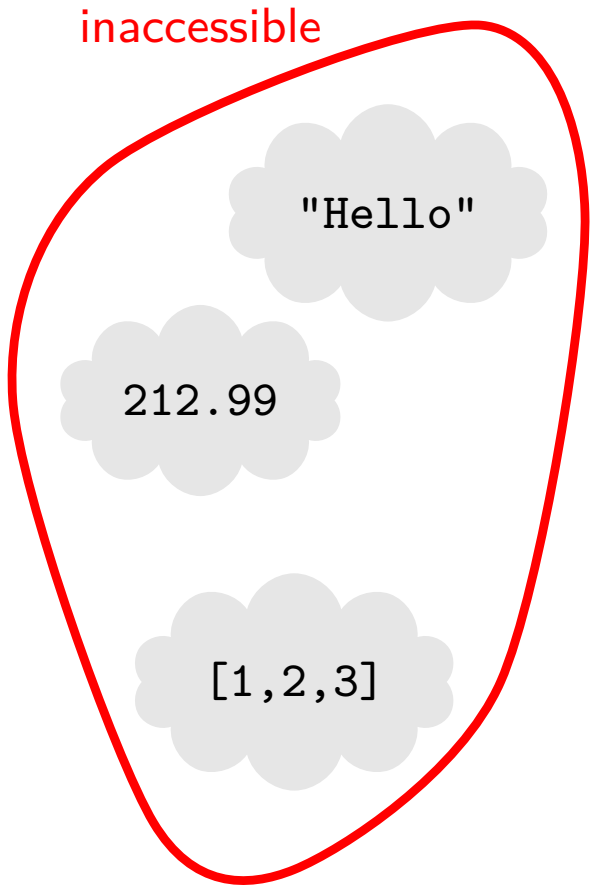


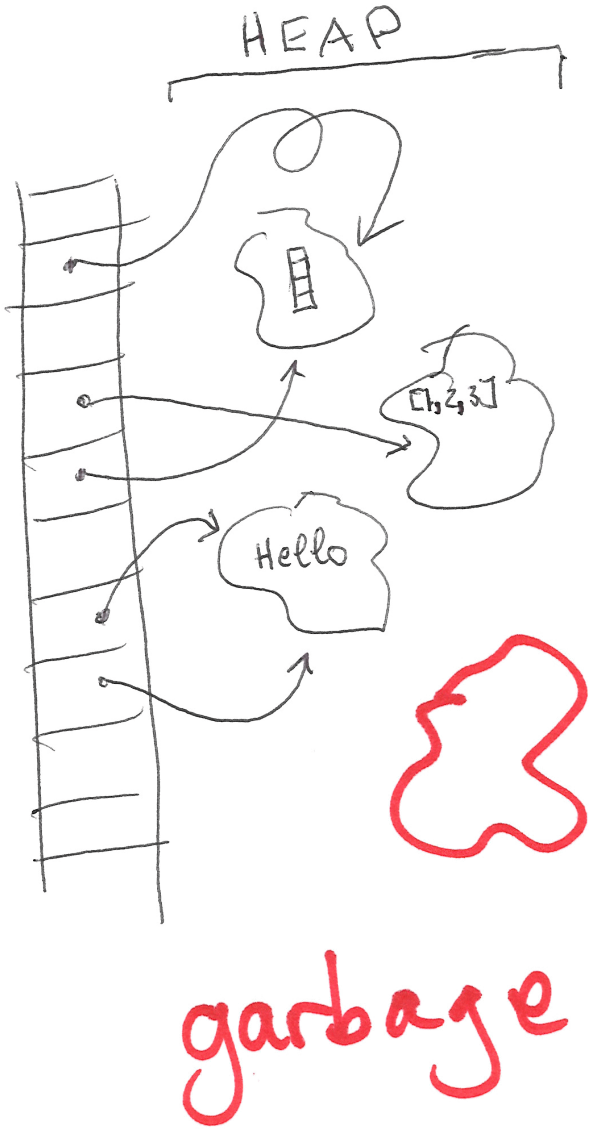


Garbage

```
i = 0;
b = null;
a = null;
g = 0.0f;
s = null;

f = 0.0f;
```






HEAP
[1,2,3]
Hello
garbage

Definition

Objects allocated on the heap which cannot accessed by the program are said to be garbage.

Definition

Garbage collection is an autonomous service during program execution which returns inaccessible objects in the heap in order to use the space for allocation of objects in the future.

Java and Garbage Collection

Java does garbage collection.

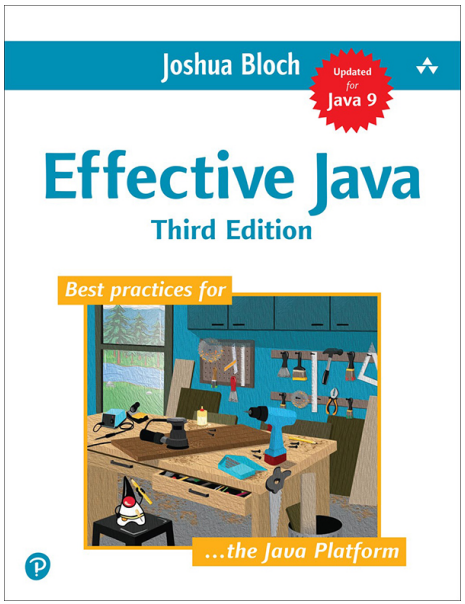
> *One of Java's most significant features is its ability to automatically manage memory. The idea is to free the programmers from the responsibility of managing memory by keeping track of orphaned objects and returning the memory they use to a pool of free memory.*

Sedgwick & Wayne, CS: An Interdisciplinary Approach, page 357

Immutability

In *Effective Java*, Joshua Bloch makes this important recommendation:

> *Classes should be immutable unless there's a very good reason to make them mutable. If a class cannot be made immutable, limit its mutability as much as possible.*



Immutability

Mutability

Mutability Leads To Disaster

22 Pages Today

The Boston Post

EXTRA

THURSDAY, JANUARY 16, 1919

HUGE MOLASSES TANK EXPLODES IN NORTH END; 11 DEAD, 50 HURT

Giant Wave of 2,300,000 Gallons of Molasses, 50 Feet High, Sweeps Everything Before It—100 Men, Women and Children Caught in Sticky Stream—Buildings, Vehicles and L Structure Crushed

35 STATES ON DRY LAW LIST

Amendment Ratified by Five Yesterday—One More Needed—Predict Nation Dry July 1

SECRECY IN PEACE CONGRESS

NORTH END PARK PLAYGROUND

Search for More Victims During the Night

No Escape From Gigantic Wave of Fluid

INTERNAL EXPLOSION WAS CAUSE, SAYS STATE CHEMIST

http://www.nytimes.com/2016/11/26/science/boston-molasses-flood-science.html

Mutability

Definition

A mutable object is an object whose state can be modified after it is created.

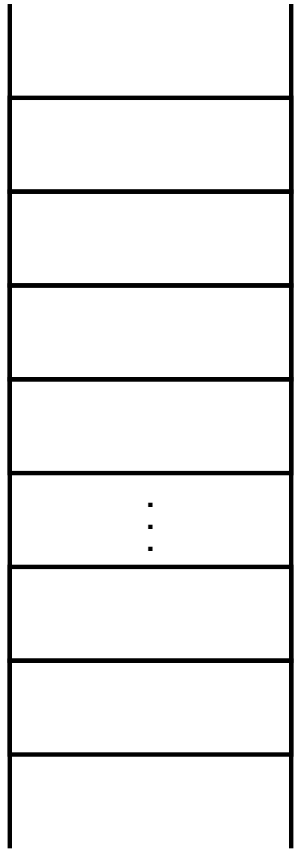
Mutability

- There is no direct support of immutability in the Java programming language. (Records are a help, but they are immutable only in the first level—"shallow.")
- Therefore, one must learn the mental discipline of immutable data structures and take advantage as best one can of the features of the Java programming language.

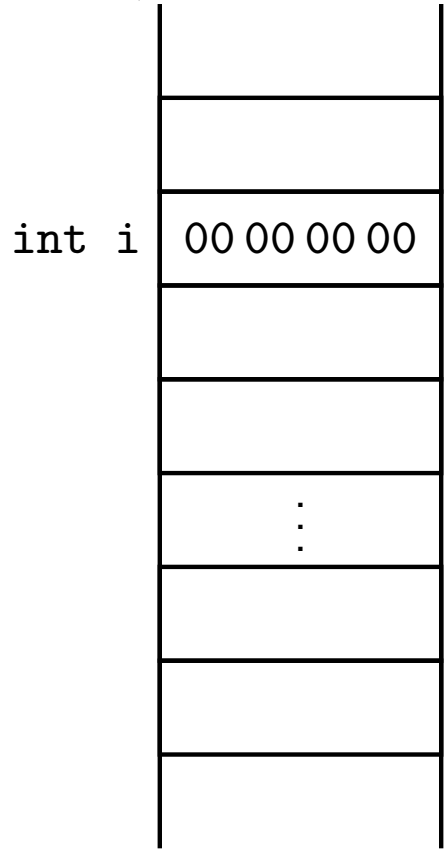
Mutable Objects

Objects in Java, like all data, can be divided into two types: mutable and immutable.
An immutable object is one that no operation can change. For example, an object of type `String`. A good example of a mutable object is an array (of any type). Changing an element of an array, changes the state of the array. Thus an array is a mutable object, it has a state which may be modified without changing the identity of the object.
Do not confuse immutable with the term *constant*. An identifier is said to be *constant* if it always refers to the same object (immutable or not). An object is said to be *immutable* if no operations can change it.
All primitive types are immutable. Not all objects are mutable.

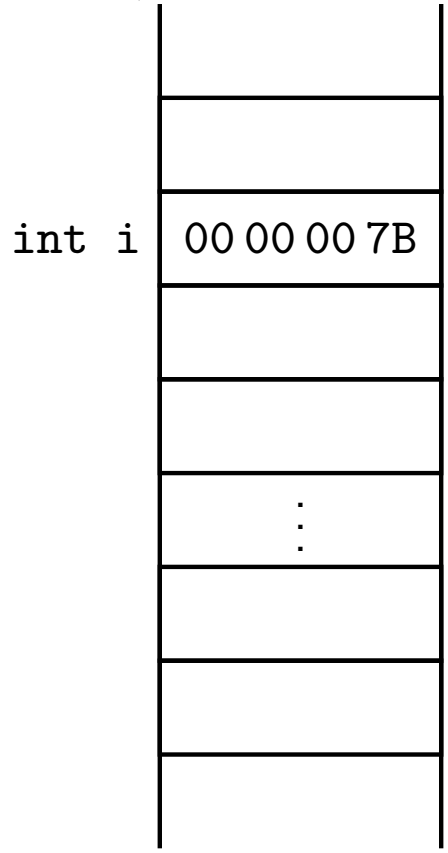
```
int i = 123;
StringBuilder b = new StringBuilder ("world");
StringBuilder f = b
int j = i;
String s = "Hello";
String t = s;
f.deleteCharAt (3);
```



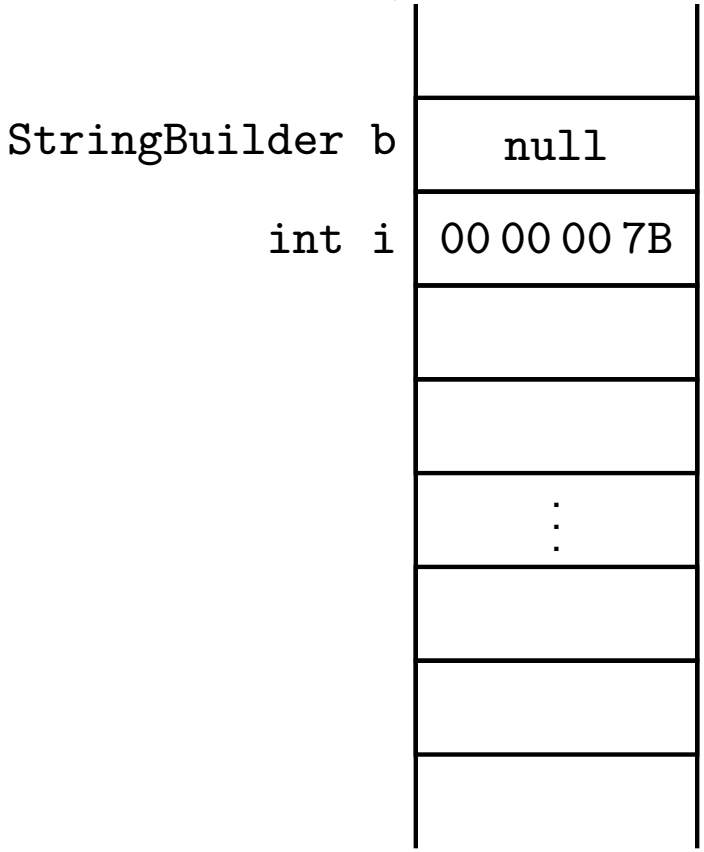
```
int i = 123;
StringBuilder b = new StringBuilder ("world");
StringBuilder f = b
int j = i;
String s = "Hello";
String t = s;
f.deleteCharAt (3);
```



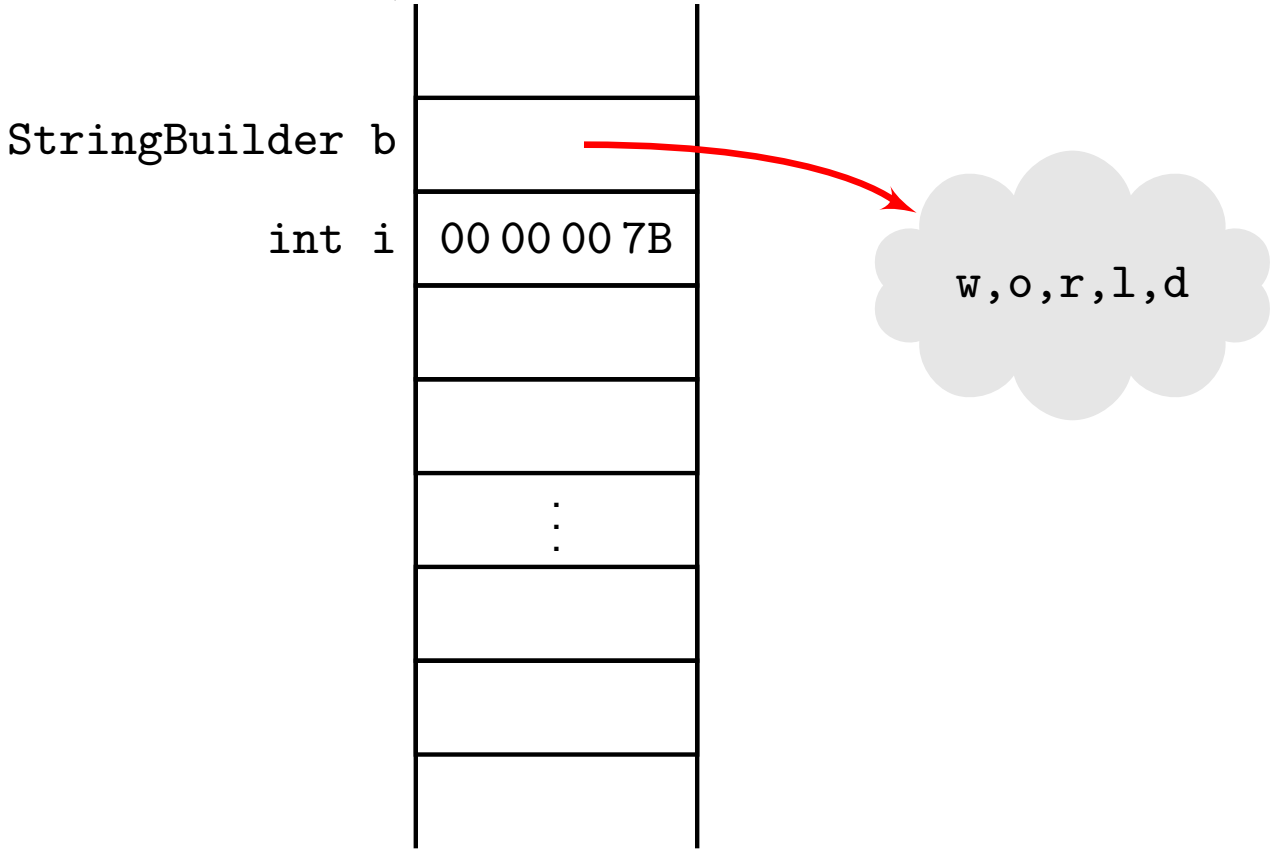
```
int i = 123;
StringBuilder b = new StringBuilder ("world");
StringBuilder f = b
int j = i;
String s = "Hello";
String t = s;
f.deleteCharAt (3);
```



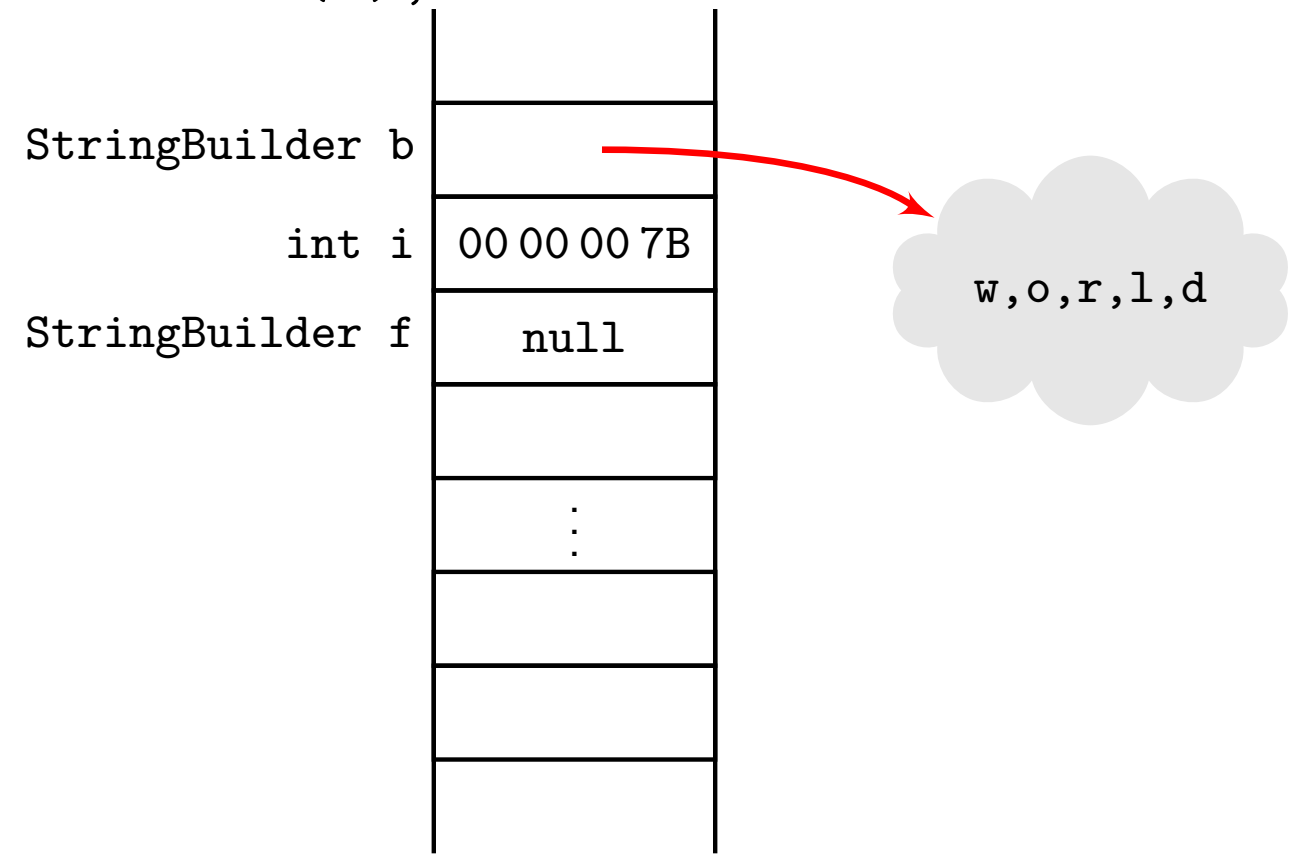
```
int i = 123;
StringBuilder b = new StringBuilder ("world");
StringBuilder f = b
int j = i;
String s = "Hello";
String t = s;
f.deleteCharAt (3);
```



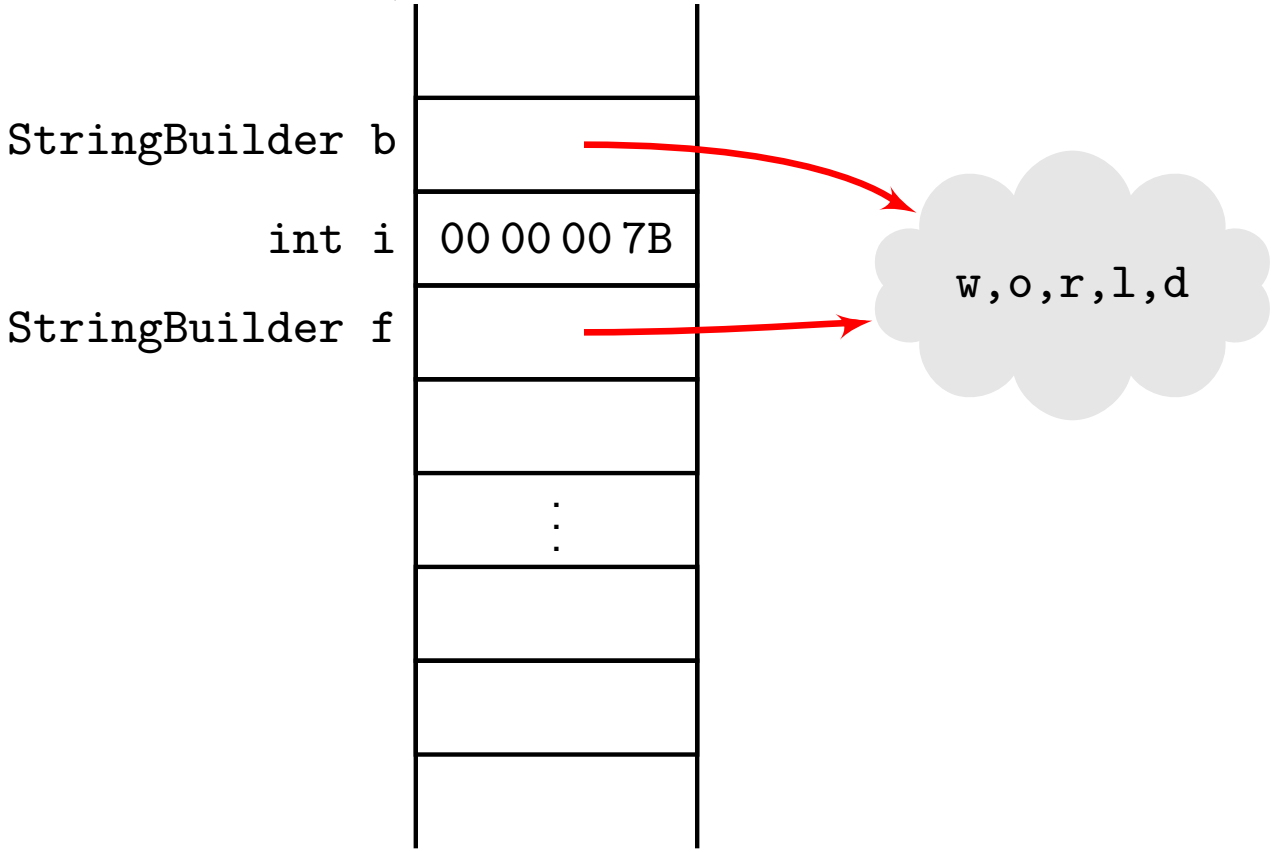
```
int i = 123;
StringBuilder b = new StringBuilder ("world");
StringBuilder f = b
int j = i;
String s = "Hello";
String t = s;
f.deleteCharAt (3);
```



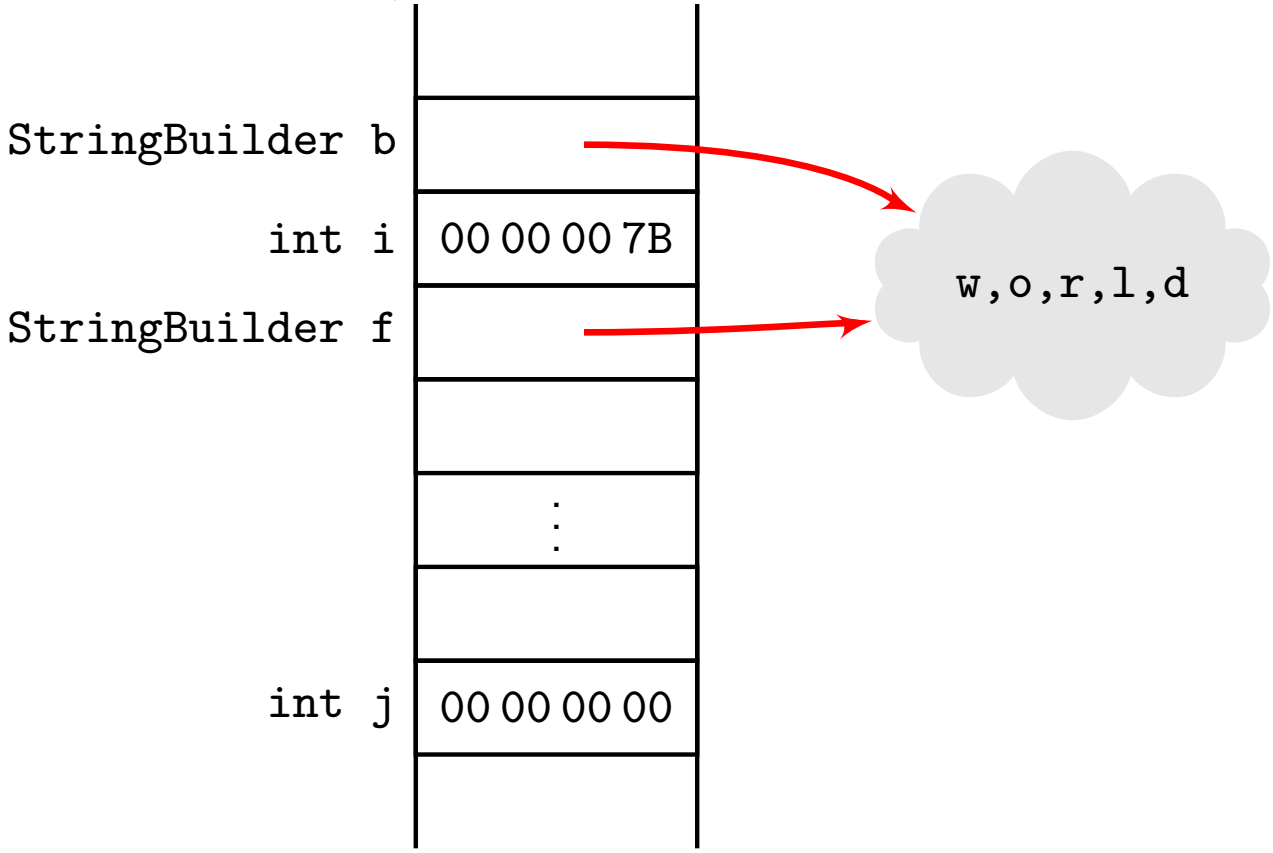
```
int i = 123;
StringBuilder b = new StringBuilder ("world");
StringBuilder f = b
int j = i;
String s = "Hello";
String t = s;
f.deleteCharAt (3);
```



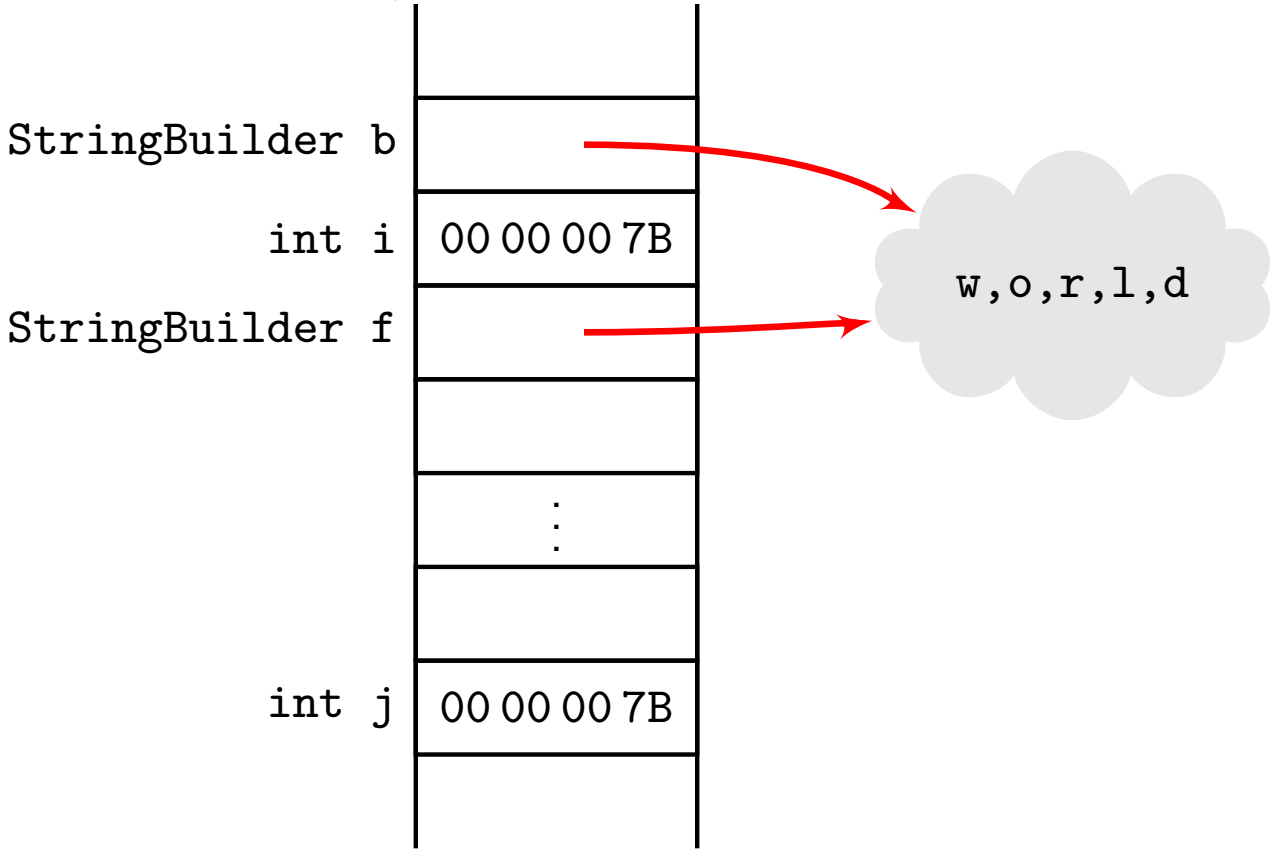
```
int i = 123;
StringBuilder b = new StringBuilder ("world");
StringBuilder f = b
int j = i;
String s = "Hello";
String t = s;
f.deleteCharAt (3);
```



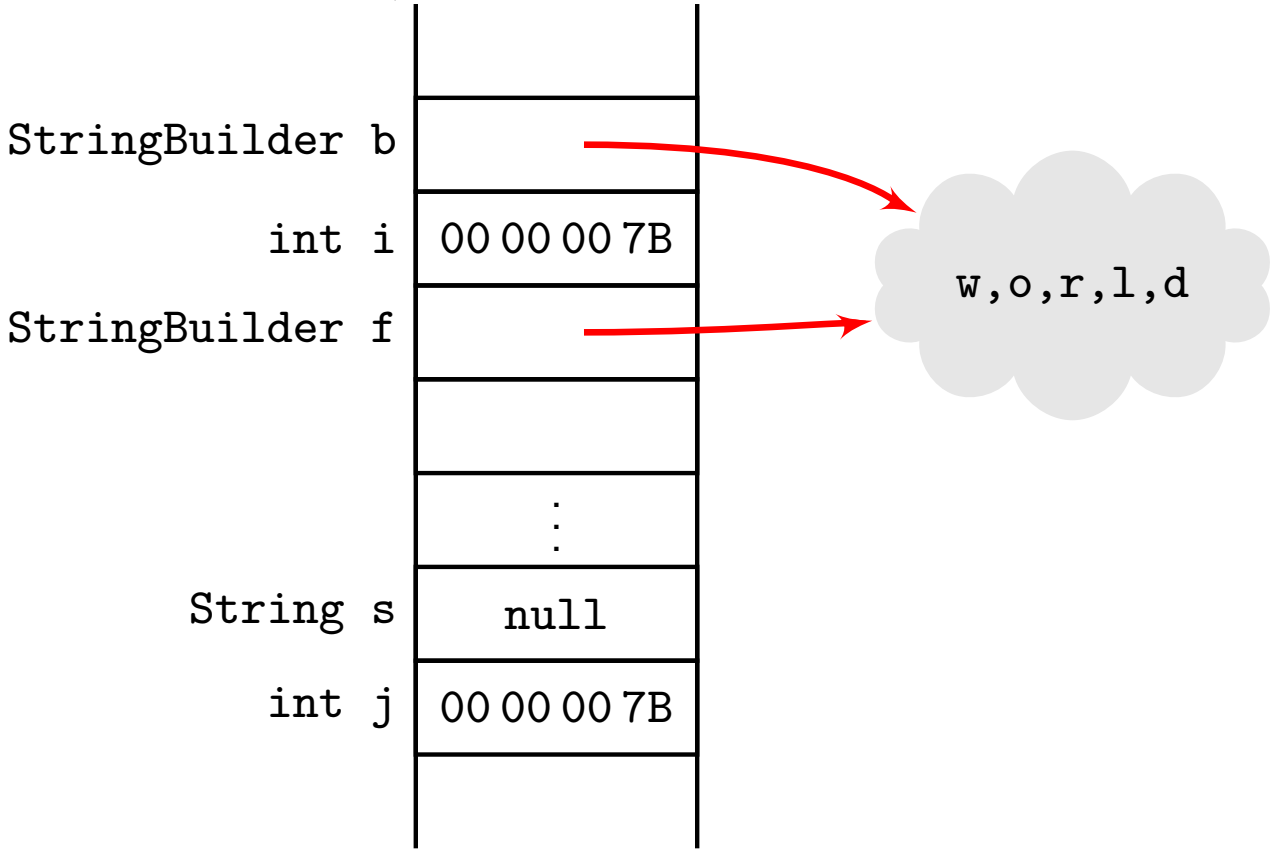
```
int i = 123;
StringBuilder b = new StringBuilder ("world");
StringBuilder f = b
int j = i;
String s = "Hello";
String t = s;
f.deleteCharAt (3);
```



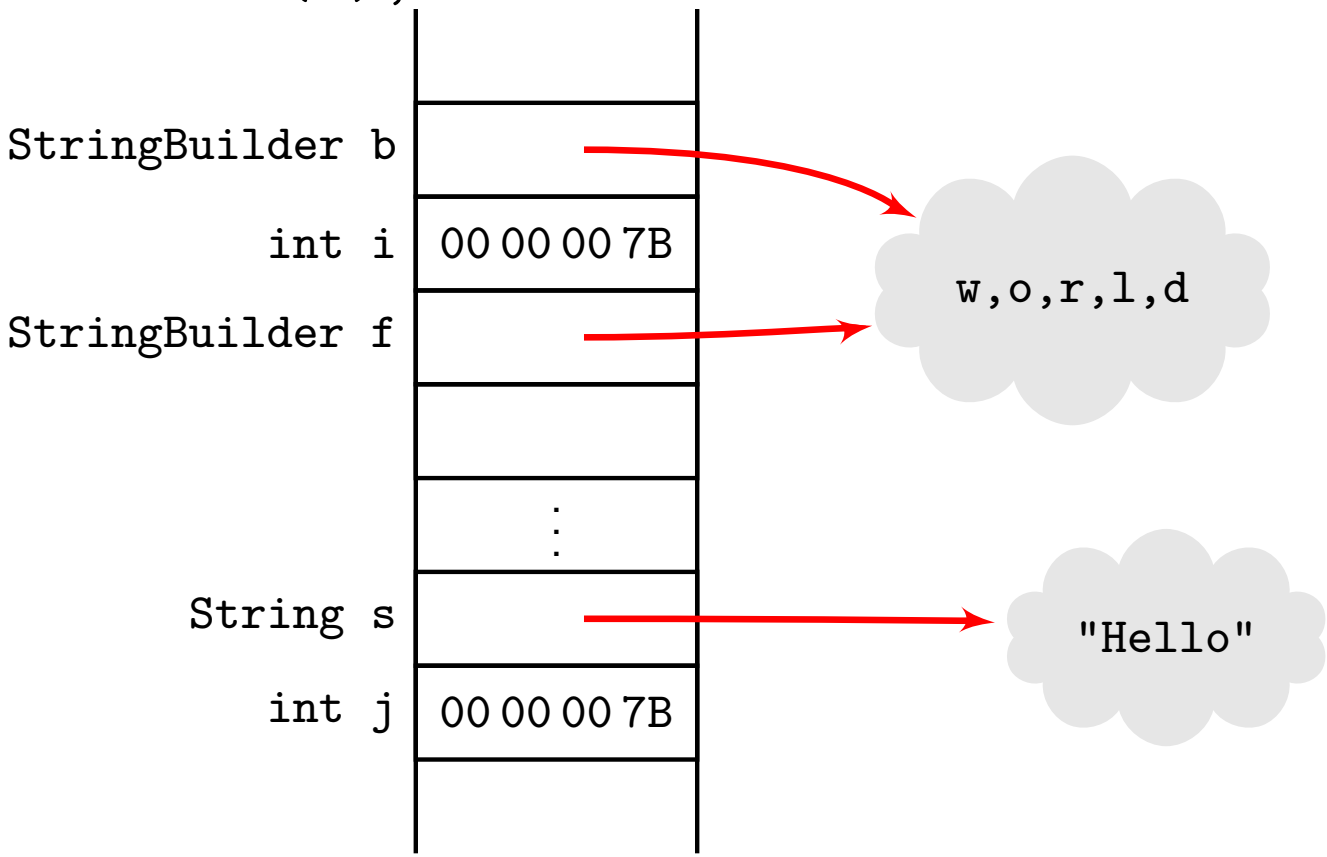
```
int i = 123;
StringBuilder b = new StringBuilder ("world");
StringBuilder f = b
int j = i;
String s = "Hello";
String t = s;
f.deleteCharAt (3);
```



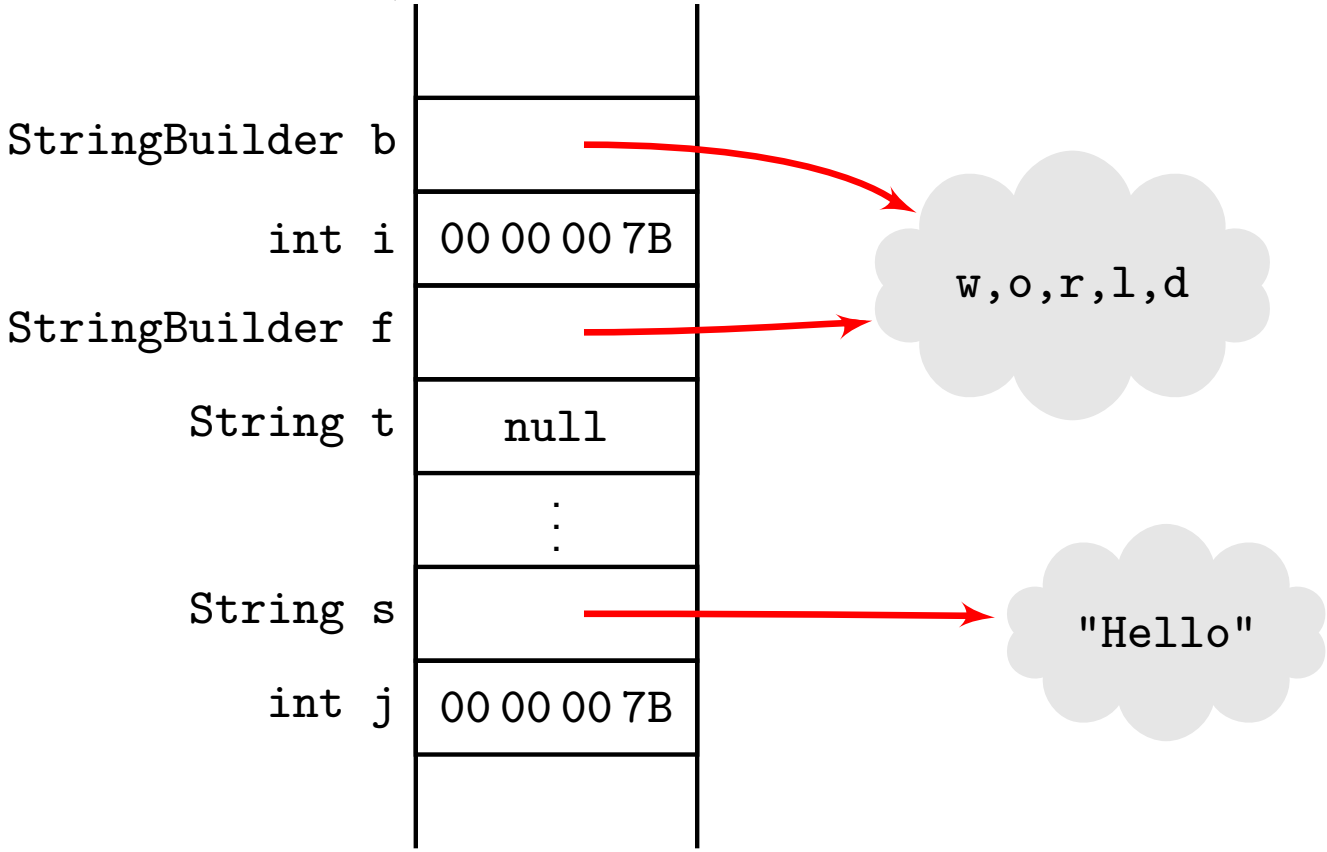
```
int i = 123;
StringBuilder b = new StringBuilder ("world");
StringBuilder f = b
int j = i;
String s = "Hello";
String t = s;
f.deleteCharAt (3);
```



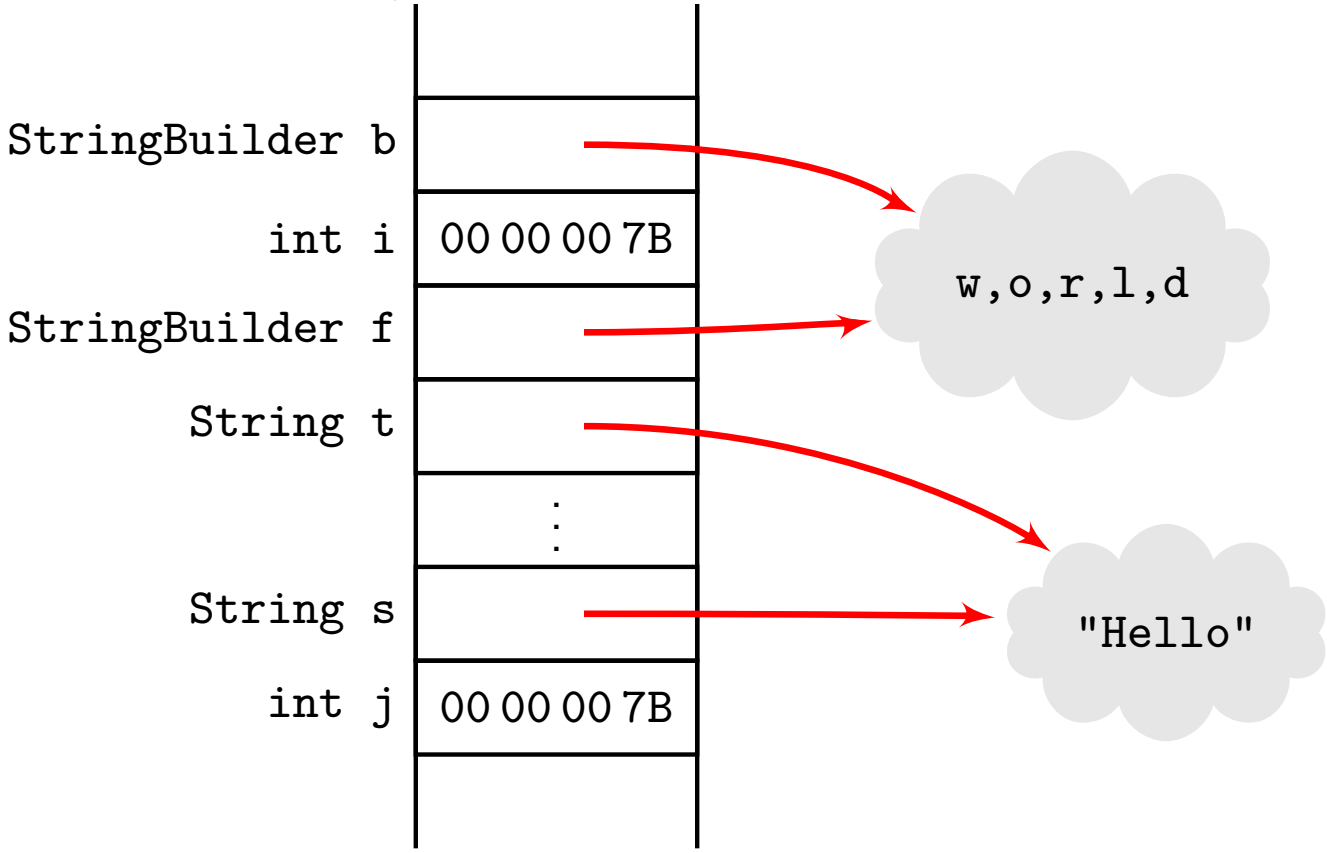
```
int i = 123;
StringBuilder b = new StringBuilder ("world");
StringBuilder f = b
int j = i;
String s = "Hello";
String t = s;
f.deleteCharAt (3);
```



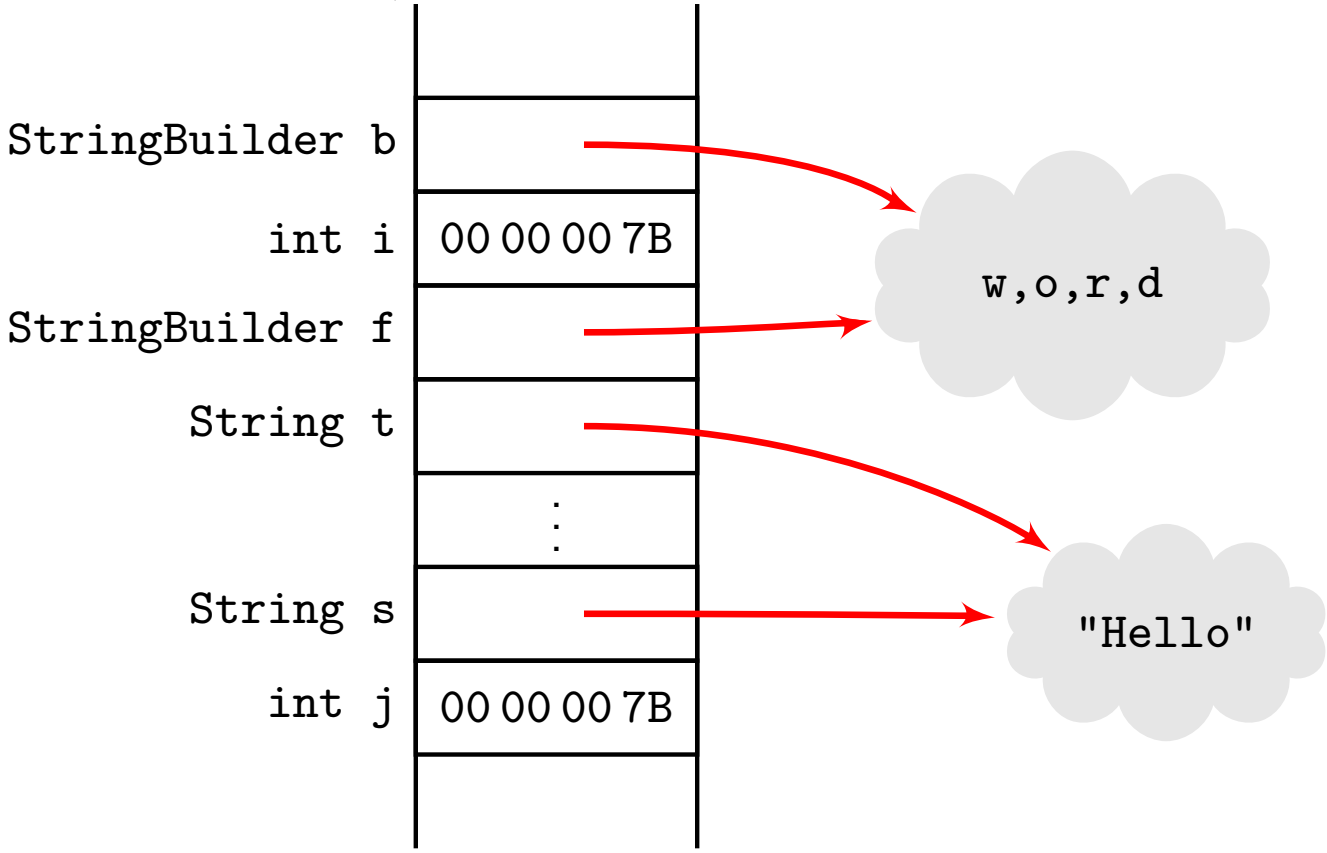
```
int i = 123;
StringBuilder b = new StringBuilder ("world");
StringBuilder f = b
int j = i;
String s = "Hello";
String t = s;
f.deleteCharAt (3);
```



```
int i = 123;
StringBuilder b = new StringBuilder ("world");
StringBuilder f = b
int j = i;
String s = "Hello";
String t = s;
f.deleteCharAt (3);
```



```
int i = 123;
StringBuilder b = new StringBuilder ("world");
StringBuilder f = b
int j = i;
String s = "Hello";
String t = s;
f.deleteCharAt (3);
```



Where does sharing come from? Obviously it comes from assignment. But what causes some of the biggest problems is sharing come from method calls. Parameter passing in Java is like an assignment to a local variable and it causes sharing for all non-primitive types.
The programmer has no choice in parameter passing and so must always be on the defensive when using mutable objects.

Mutability quietly erodes the protection parameter passing provides primitive value.
How do you pass mutable data to be used as input to a subprocedure, but guarantied not be be changed by the caller?

What if sort invoked the clear method on your data?!

```
ArrayList<Integer> list = Arrays.asList (1,2,3);
ImpudentClass.veryBadSort (list);  -- return lis
assert list.size()==3;
```

```
List<Integer> list = List.of (1,2,3); -- immutab
ImpudentClass.veryBadSort (list);  -- unsupporte
assert list.size()==3;
list.add (4);  -- unsupported operation!
```

But you cannot sort the elements of an immutable list anyway.
A means of protection:

```
return Collections.unmodifiableList (list);
```

Summary

one parameter passing mechanism (assignment)!
yes, two kinds of data (boxed and unboxed)
Caution: boxed data can be shared
Two kinds of data (mutable and immutable)!!

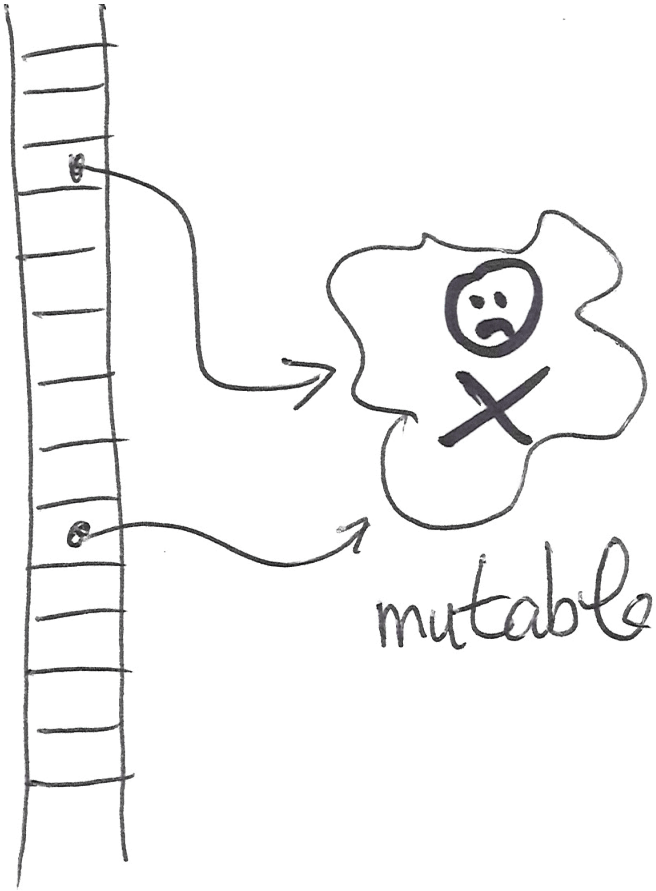
Summary

Sharing is cheap, but buggy; copying (large objects) is expensive, but safe. Immutable objects can be shared without problems, Always design for immutability; optimize later.

premature optimization is the root of all evil

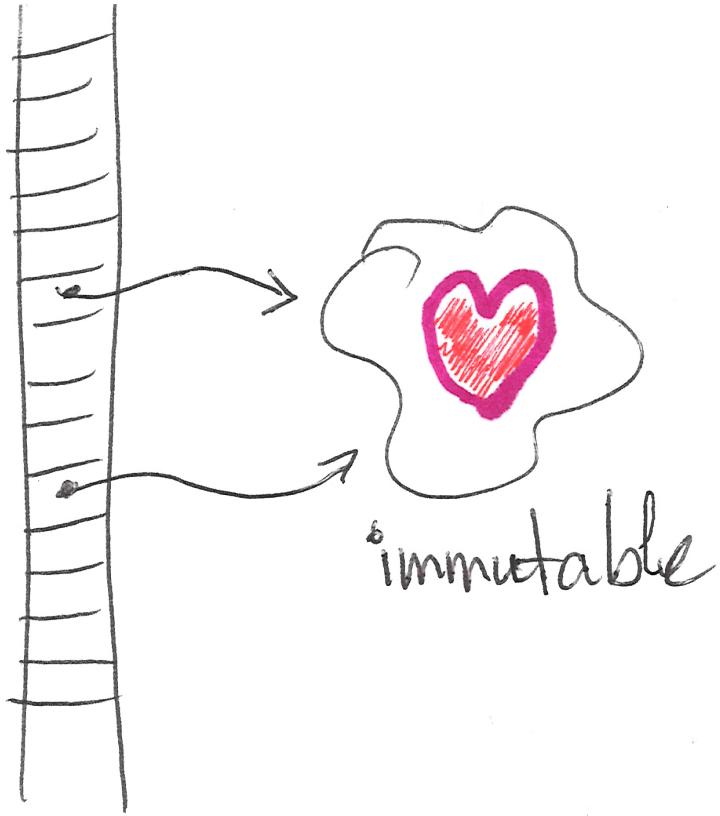
Knuth, 1974 Turing Award lecture

Sharing Mutable



sharing

Sharing Immutable



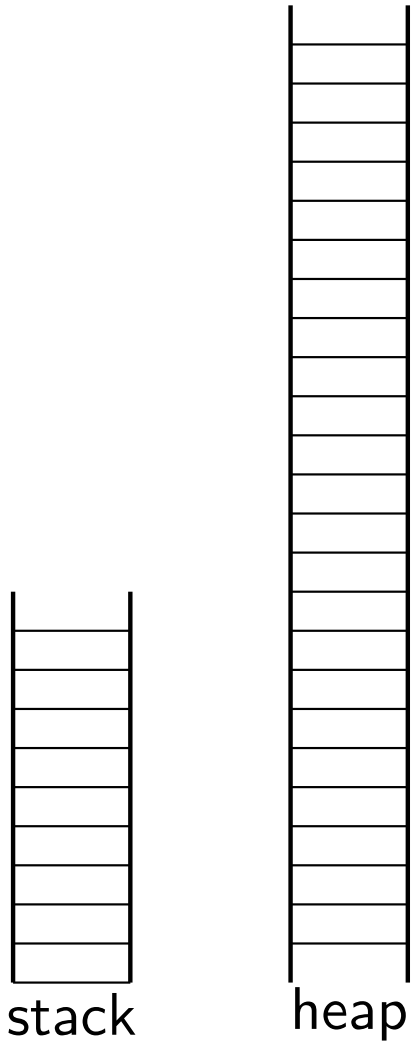
sharing

The Stansifer sayings:

It is easier to make a correct program more efficient than to make a buggy program more correct.

A program that does what you think it does is much better than a program that might do what you want it do.

Reference Counting



```
class name { members }
```

Members may be static or instance.
Members may be methods (subprocedures) or fields (data values).

	static	instance
methods		
fields		

Static member are accessed: `ClassName.member`. Instance members are accessed: `expressionDenotingAnInstance.member`. Very important to observe the capitalization convention and never to access static members like this:

```
expressionDenotingInstanceOfClass.member
```

(which is legal but bewilders the reader).

Instance members can access both static and instance members. (But don't take advantage of this.) Static members can only access static members.

```java
public class Main {
    static int field = 123; // static member
    public static void main (String[] a) {
        System.out.println (new Main().field);
    }
}
```

```java
public class Main {
    int field = 123;    // instance field
    public static void main (String[] a) {
        System.out.println (Main.field);
    }
}
```

Not caught by any checkstyle check, but is a translation warning:

```
javac -Xlint:static Main.java
Main.java:4: warning: [static] static variable should be
by type name, Main, instead of by an expression
```

Parameter Passing

Call by value is about protection. Information does not flow back it is not about objects nor 'final'.

- `PassByValue.java` – Java uses "call by value"
- `PassByWrapper.java` – Using wrapper class does *n*ot provide means of creating "out" parameters.
- If you want to return something, there are no "out" parameters; use a function

Returning information

Use functions to return values. Don't use secret dead drops.