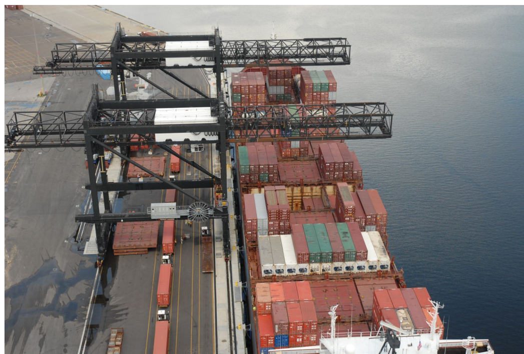
Primitive Data and Objects

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.)

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 desing task the programmer has, since the control flow could spring naturally from that data if the design is good

Primitive Data and Reference Data

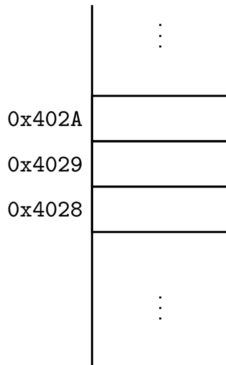
Usual characteristics of primitive data:

- small size, all values fit in the machine's word size
- uncomplicated (no substructure)
- operations on the data are supported in the hardware

The majority of data the programmer wants is not primitive: images, music, graphs, sequences, etc. Java supports this vast, endless variety of data by allowing programmer to define new data types and by implementing some in the libraries. User-defined data types are created using classes.
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.

Storage

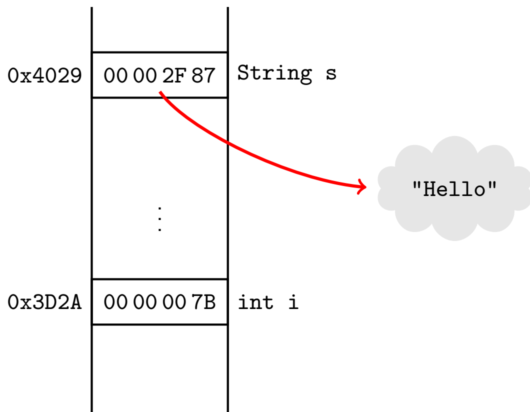
During execution the entire program is laid out in (virtual) storage, which we can envisioned as a gigantic array of words indexed by a (virtual) address. All the data appears somewhere in storage.



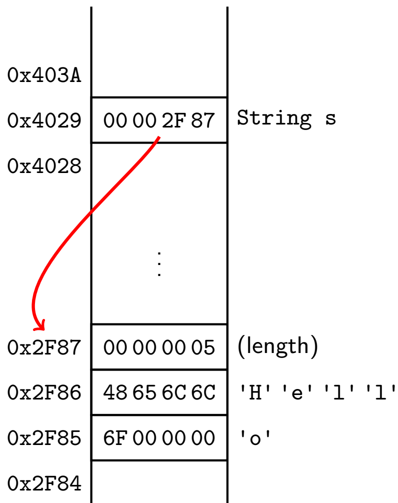
Primitive Data and Objects

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

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



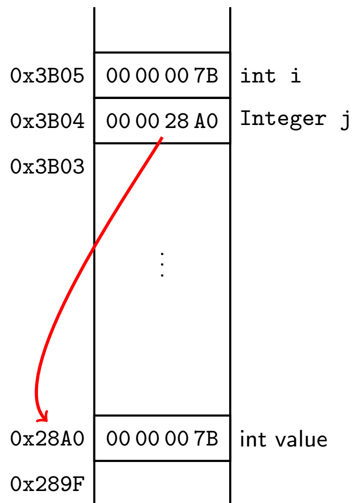
The space for objects is found in the heap. Of course, the heap must be found somewhere in storage.



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`.

The Integer Wrapper Class



Creating Objects

When you declare a variable for a values of a primitive data type, an address is assigned which can 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 the 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};
```

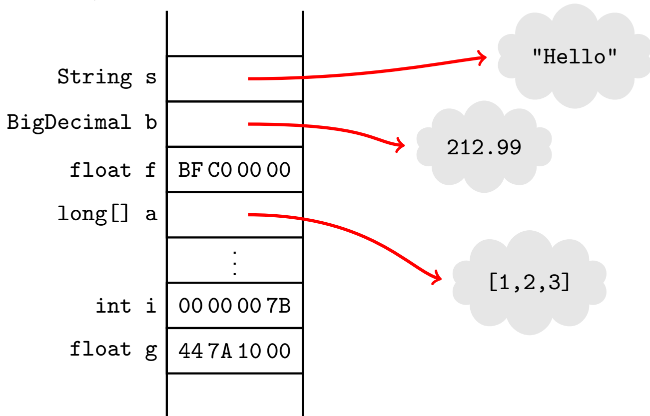
```
String s = "abc";
Integer i = 123;    // Auto-boxing!
int[] a = {1,2,3}; // new is optional in decl
```

```
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 ()
```

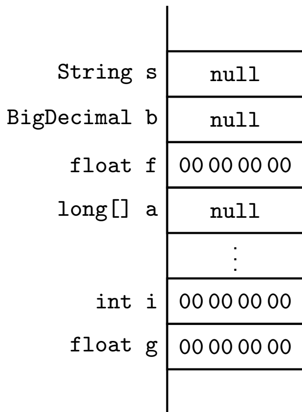
- 0, 1, or more arguments
- package name may be omitted if one uses an `import`
- generic instantiation
- some classes are never instantiated, i.e, `Math`
- other ways to get objects: "factory methods," methods which create the object for you.

It is (regretably) 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.
Then we see why this mental picture is important.

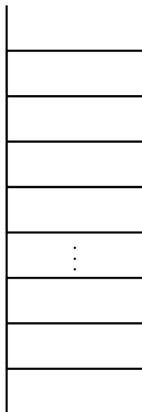
```
int i = 123;
BigDecimal b = new BigDecimal (212.99);
long[] a = new long[] {1,2,3};
float g = 1000.25f;
String s = "Hello";
float f = -1.5f;
```



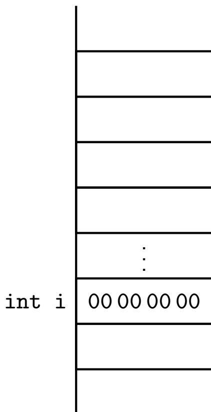
```
int i;
BigDecimal b;
long[] a;
float g;
String s;
float f;
```



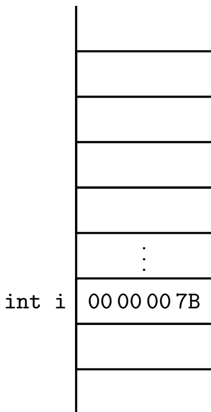
```
i = 123;
b = new BigDecimal (212.99);
a = new long[] {1,2,3};
g = 1000.25f;
s = "Hello";
f = -1.5f;
```



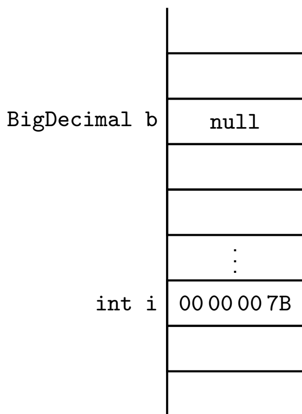
```
i = 123;
b = new BigDecimal (212.99);
a = new long[] {1,2,3};
g = 1000.25f;
s = "Hello";
f = -1.5f;
```



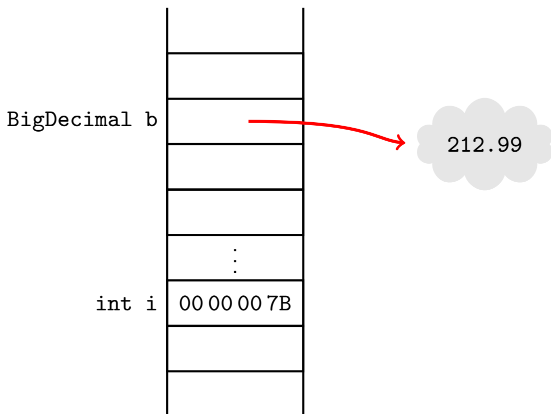
```
i = 123;
b = new BigDecimal (212.99);
a = new long[] {1,2,3};
g = 1000.25f;
s = "Hello";
f = -1.5f;
```



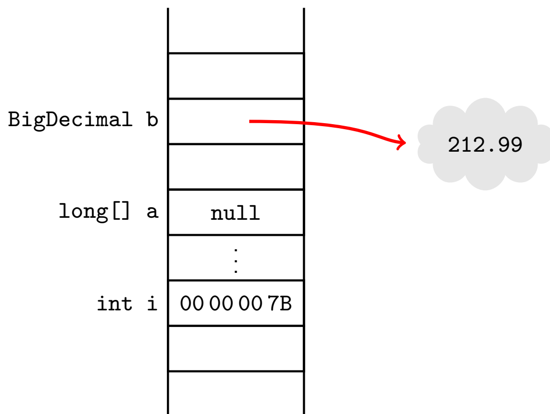
```
i = 123;
b = new BigDecimal (212.99);
a = new long[] {1,2,3};
g = 1000.25f;
s = "Hello";
f = -1.5f;
```



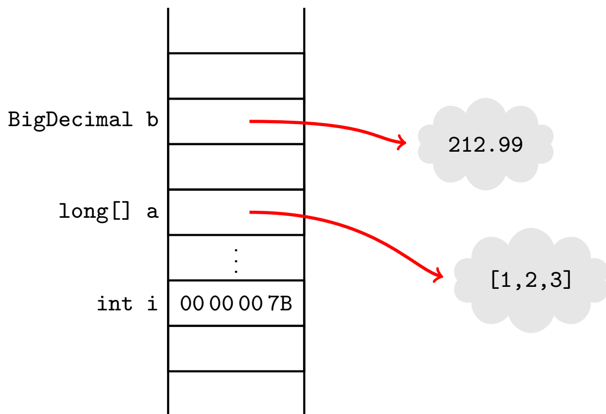
```
i = 123;
b = new BigDecimal (212.99);
a = new long[] {1,2,3};
g = 1000.25f;
s = "Hello";
f = -1.5f;
```



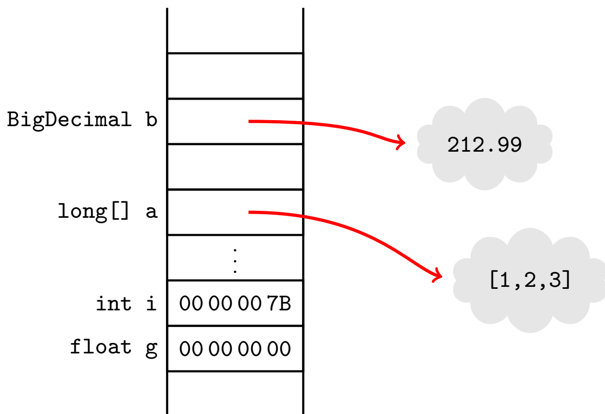
```
i = 123;
b = new BigDecimal (212.99);
a = new long[] {1,2,3};
g = 1000.25f;
s = "Hello";
f = -1.5f;
```



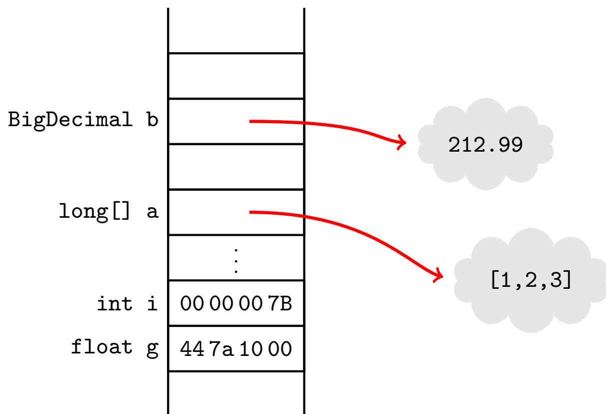
```
i = 123;
b = new BigDecimal (212.99);
a = new long[] {1,2,3};
g = 1000.25f;
s = "Hello";
f = -1.5f;
```



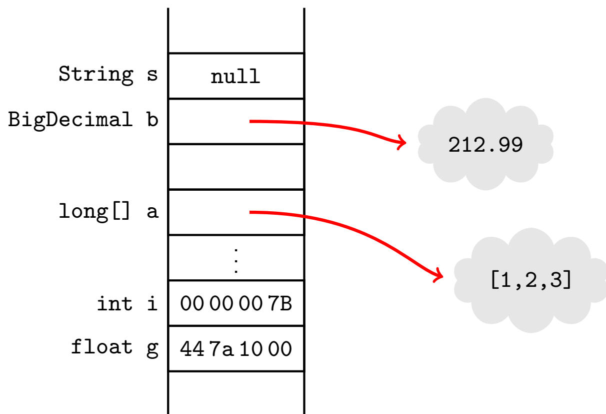
```
i = 123;
b = new BigDecimal (212.99);
a = new long[] {1,2,3};
g = 1000.25f;
s = "Hello";
f = -1.5f;
```



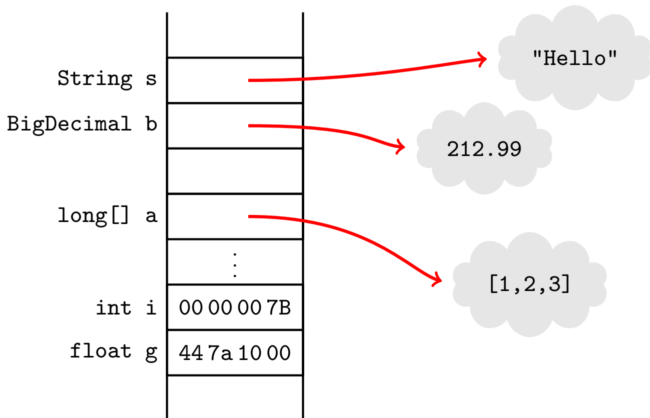
```
i = 123;
b = new BigDecimal (212.99);
a = new long[] {1,2,3};
g = 1000.25f;
s = "Hello";
f = -1.5f;
```



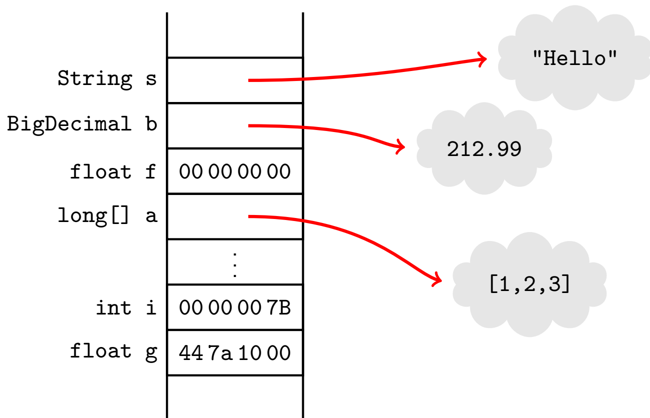
```
i = 123;
b = new BigDecimal (212.99);
a = new long[] {1,2,3};
g = 1000.25f;
s = "Hello";
f = -1.5f;
```



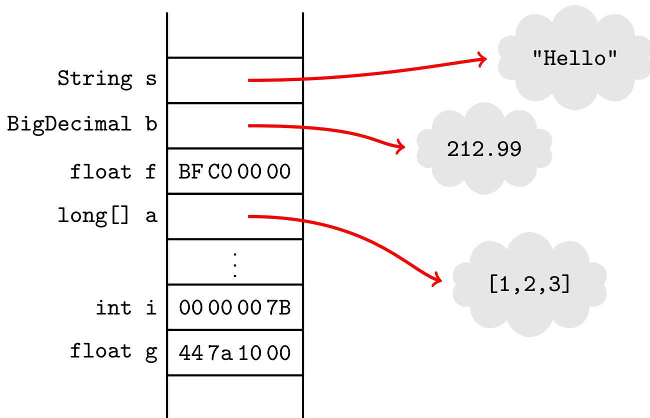
```
i = 123;
b = new BigDecimal (212.99);
a = new long[] {1,2,3};
g = 1000.25f;
s = "Hello";
f = -1.5f;
```



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



```java
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.
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];
    }
}
```

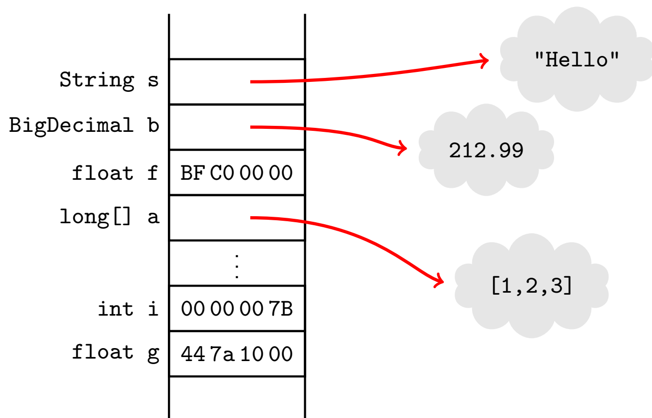
```
> java Fields
Exception in thread "main" java.lang.NullPointerException
        at Fields.main(Fields.java:5)
```

> *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.*

Tony Hoare

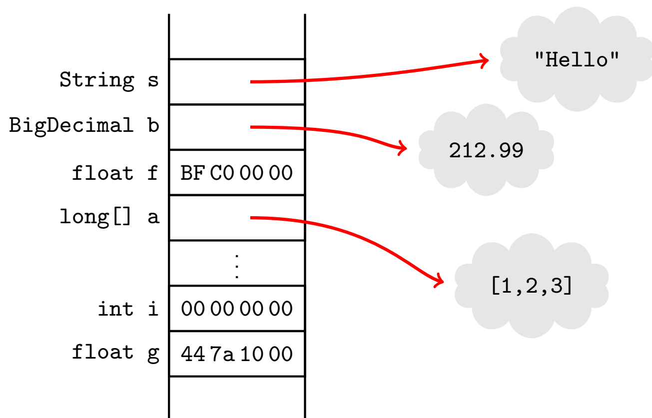
Garbage

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



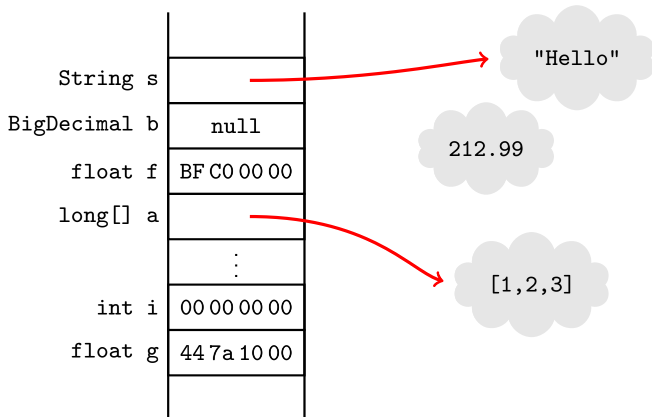
Garbage

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



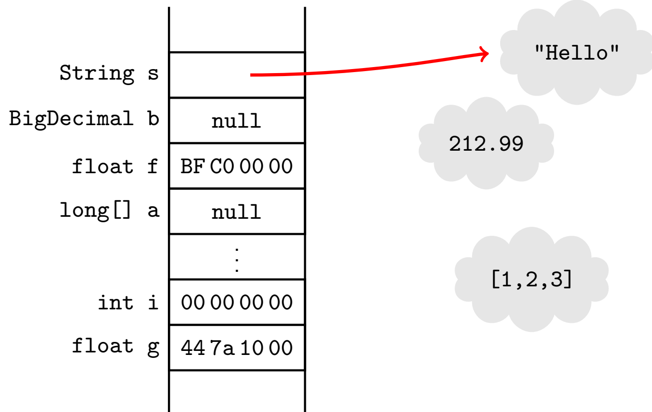
Garbage

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



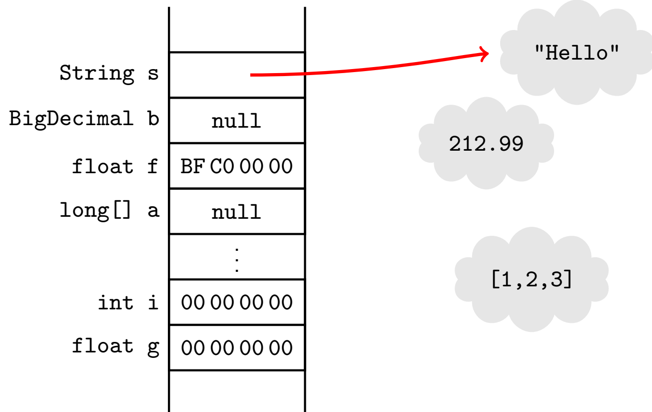
Garbage

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



Garbage

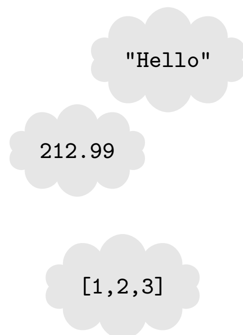
```
i = 0;
b = null;
a = null;
g = 0.0f;
s = null;
f = 0.0f;
```



Garbage

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

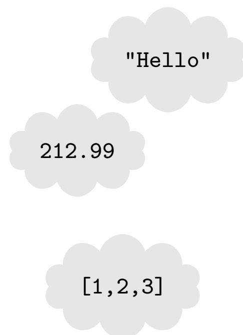
String s	null
BigDecimal b	null
float f	BF C0 00 00
long[] a	null
	⋮
int i	00 00 00 00
float g	00 00 00 00



Garbage

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

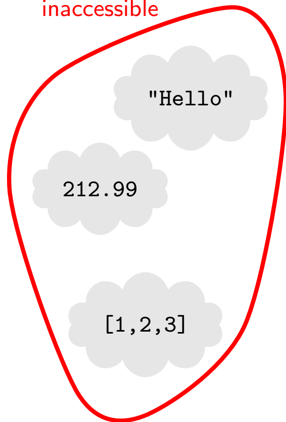
String s	null
BigDecimal b	null
float f	00 00 00 00
long[] a	null
	⋮
int i	00 00 00 00
float g	00 00 00 00



Garbage

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

String s	null
BigDecimal b	null
float f	00 00 00 00
long[] a	null
	⋮
int i	00 00 00 00
float g	00 00 00 00



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.*

page 357

Mutable Objects

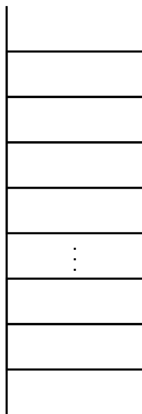
Objects in Java, like all data, can be divided into two types: mutable and immutable.

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, an object that "has a state which may be modified by certain operations without changing the identity of the object."

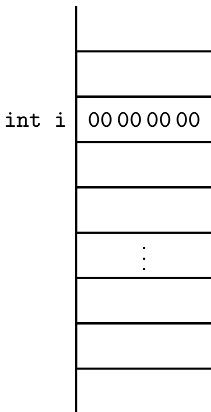
Do not confuse immutable with the term *constant*. An identifier is said to be *constant* if it always refers to the same object (immutable or not). An object is said to be *immutable* if no operations can change it.

All primitive types are immutable. Not all objects are mutable.

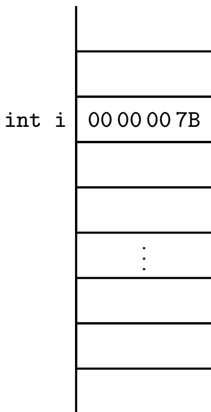
```java
int i = 123;
StringBuilder b = new StringBuilder ("world");
StringBuilder f = b
int j = i;
String s = "Hello";
String t = s;
f.deleteCharAt (3);
```



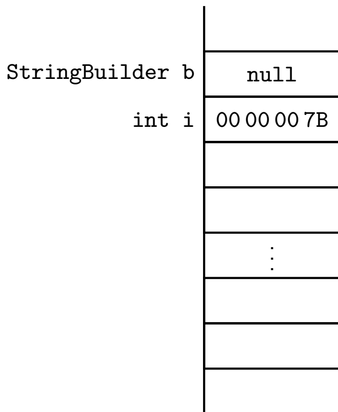
```
int i = 123;
StringBuilder b = new StringBuilder ("world");
StringBuilder f = b
int j = i;
String s = "Hello";
String t = s;
f.deleteCharAt (3);
```



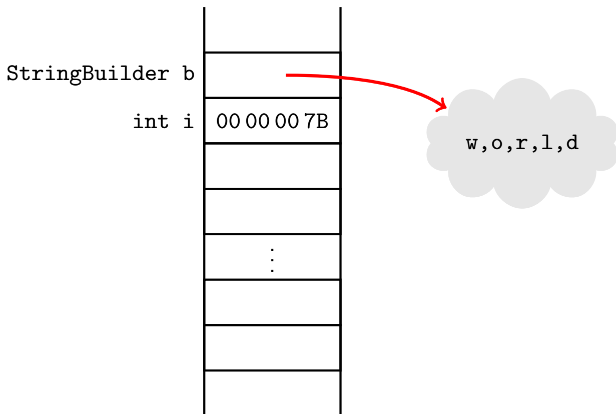
```java
int i = 123;
StringBuilder b = new StringBuilder ("world");
StringBuilder f = b
int j = i;
String s = "Hello";
String t = s;
f.deleteCharAt (3);
```



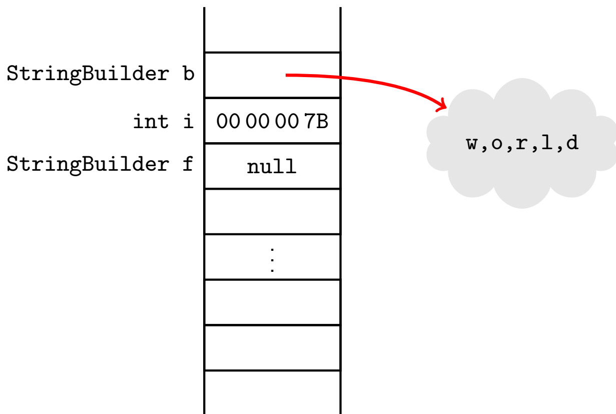
```
int i = 123;
StringBuilder b = new StringBuilder ("world");
StringBuilder f = b
int j = i;
String s = "Hello";
String t = s;
f.deleteCharAt (3);
```



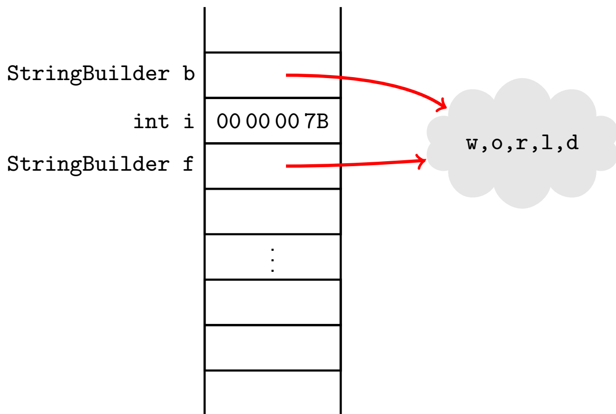
```
int i = 123;
StringBuilder b = new StringBuilder ("world");
StringBuilder f = b
int j = i;
String s = "Hello";
String t = s;
f.deleteCharAt (3);
```



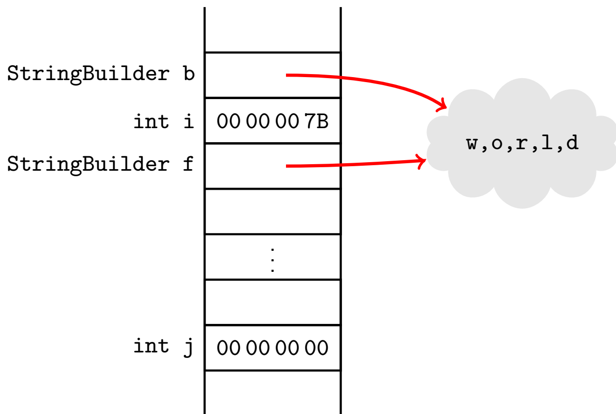
```
int i = 123;
StringBuilder b = new StringBuilder ("world");
StringBuilder f = b
int j = i;
String s = "Hello";
String t = s;
f.deleteCharAt (3);
```



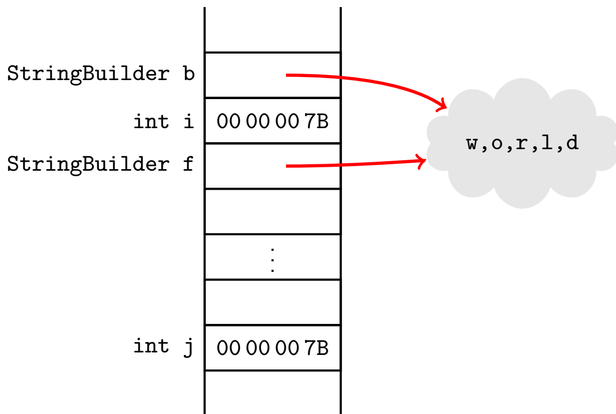
```java
int i = 123;
StringBuilder b = new StringBuilder ("world");
StringBuilder f = b
int j = i;
String s = "Hello";
String t = s;
f.deleteCharAt (3);
```



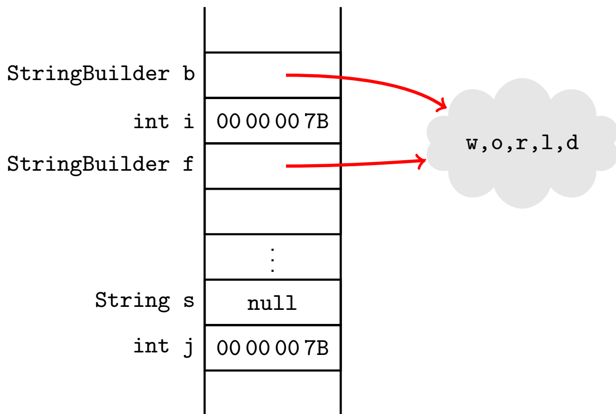
```
int i = 123;
StringBuilder b = new StringBuilder ("world");
StringBuilder f = b
int j = i;
String s = "Hello";
String t = s;
f.deleteCharAt (3);
```



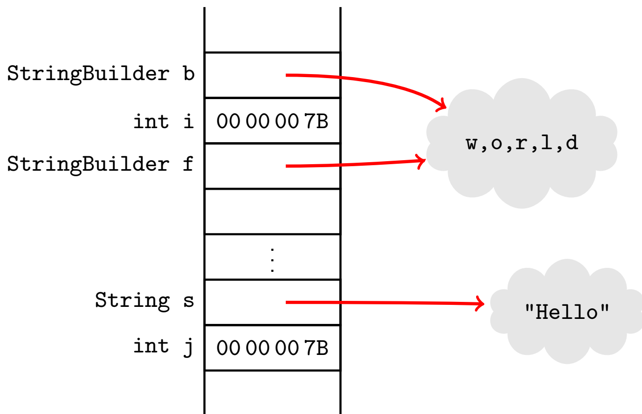
```
int i = 123;
StringBuilder b = new StringBuilder ("world");
StringBuilder f = b
int j = i;
String s = "Hello";
String t = s;
f.deleteCharAt (3);
```



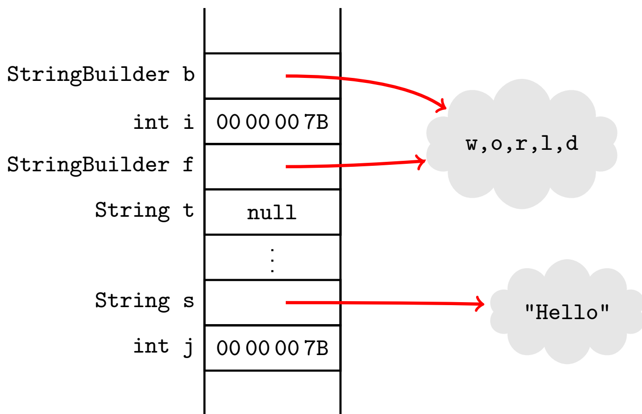
```java
int i = 123;
StringBuilder b = new StringBuilder ("world");
StringBuilder f = b
int j = i;
String s = "Hello";
String t = s;
f.deleteCharAt (3);
```



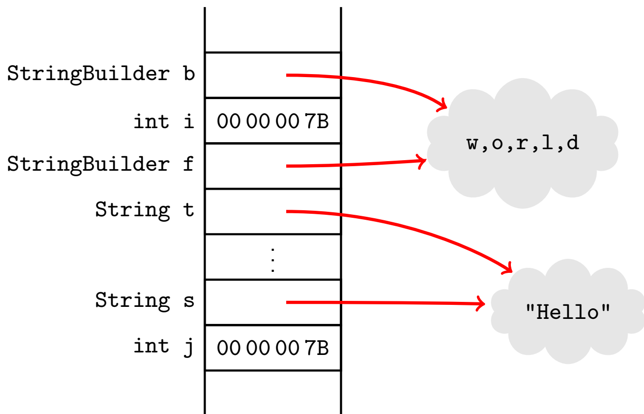
```java
int i = 123;
StringBuilder b = new StringBuilder ("world");
StringBuilder f = b
int j = i;
String s = "Hello";
String t = s;
f.deleteCharAt (3);
```



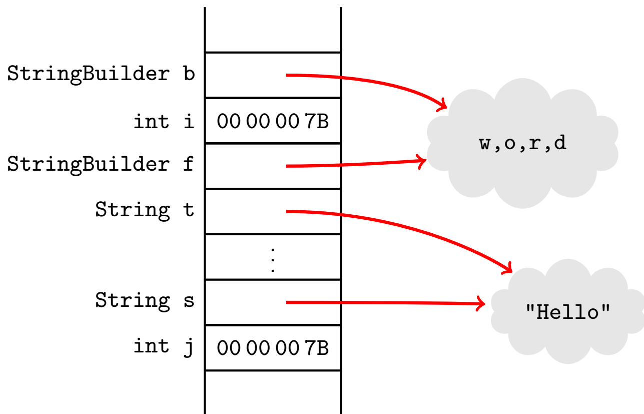
```
int i = 123;
StringBuilder b = new StringBuilder ("world");
StringBuilder f = b
int j = i;
String s = "Hello";
String t = s;
f.deleteCharAt (3);
```



```
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 we often overlook that that sharing comes from method calls. Parameter passing in Java (call by value) 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.

What if array list sort invoked the clear method?

```
ArrayList<Integer> list =

return Collections.unmodifiableList (list);
```

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.

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.

```
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[] ) {
      System.out.println (new Main().field);
   }
}

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

Parameter Passing

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