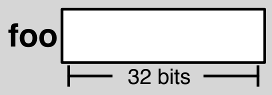
The difference between primitive and object types in Java

Variables of primitive types (int, double, boolean, char) are different than non-primitive (Object) variables in Java. The key is to understanding what happens at compile-time and run-time in terms of allocating memory for your variables.

code **memory**

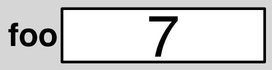
When you declare an int variable called foo the compiler can create a container big enough for an int (32 bits) and stick a label on it so you can refer to it later in your code.

```
int foo;
```



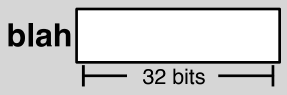
Next, if you assign a value to foo, during run-time it can just stick the number you assign into the container that was reserved for it (assuming the integer can be expressed in 32 bits)

```
foo = 7;
```



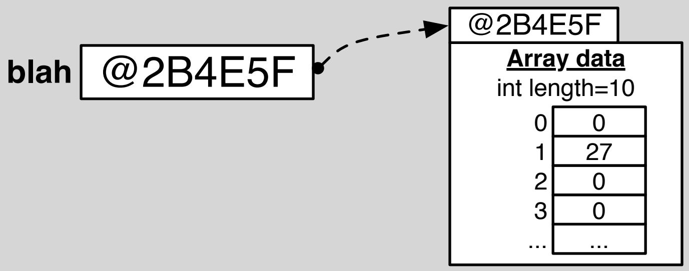
Now what happens when you declare an Object variable? How much memory should be set aside? Well, it's actually impossible to know, since different objects require different amounts of actual memory during run-time, and even might change during run-time. So the solution is that an object variable is actually created as a 32-bit container as well....

```
Object blah;
```



Next, if you instantiate an object (or an array in the example below) with "new", think of that as meaning "allocate new memory" for the object. During run-time then, some working memory is reserved for the type and size of object that was actually created and the address of where the new memory for the object was created (sort of like a tracking number**) is returned and stored in the variable.

```
blah = new int[10];
blah[1] = 27;
```



So what's stored in the 32-bit Object variable is actually just another integer - the memory address of where the object is located. We often call this number a "memory reference" and draw diagrams with an arrow from the variable container to the object in memory. That way we don't have to write complicated (arbitrary) numbers all the time. This has implications for when you pass variables to methods in other objects or functions....

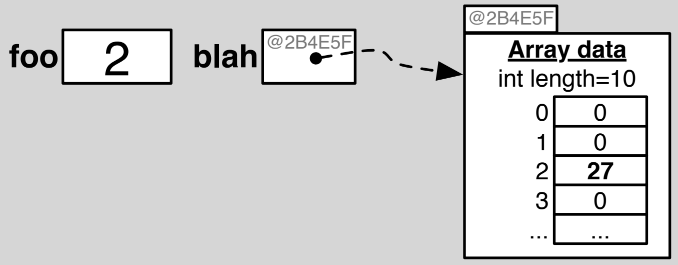
[Con'd next page...]

The difference between primitive and object types in Java

code **memory**

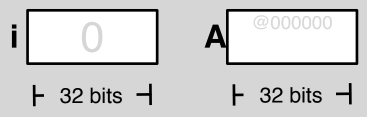
Let's look at a method that alters the values in some variables and what happens in memory. Here's the setup continuing from the previous page.

```
foo=2;
blah[foo] = 27;
```



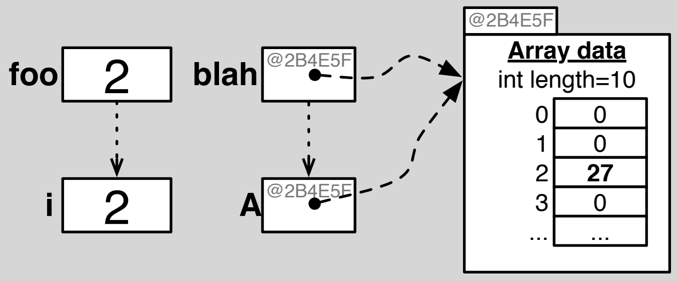
Now let's say we have this method below. The method parameters are like variable declarations in any other piece of code, but they only live as long as the method is running. Memory must be reserved for these variables even though we can't know the values until the program is running. Since there is an int and an array, the compiler will reserve 32 bits for the int, and 32 bits for a reference to an array...

```
void modifyArray(int i, int[] A){
     A[i] = A[i]/2;
     i = -1;
}
```



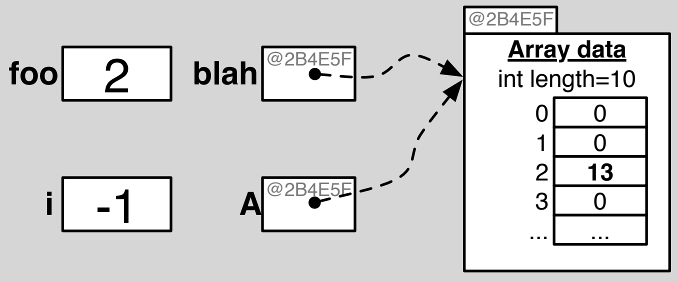
...but those values don't get filled in until the method is actually called. And what gets passed along to the method parameter variables is a ***copy of the value that is in the container.*** For primitive types, that means the *value itself is copied* and passed to the method. For objects it means that a *copy of the reference to the object* is passed.

```
 modifyArray(foo, blah);
```



After the method has executed its code (but just before it quits and returns) the state of the world looks like this. Notice that *i* and *foo* are separate variables whose contents are independent of each other, changing one does not impact the other. *blah* and *A,* by the same token, are independent copies...but they are copies of an object reference, so they refer to the same thing.

```
void modifyArray(int i, int[] A){
     A[i] = A[i]/2;
     i = -1;
}
```



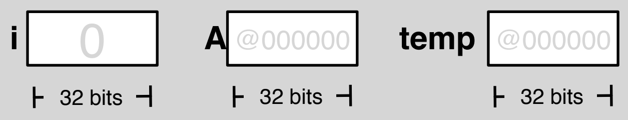
The difference between primitive and object types in Java

code

memory

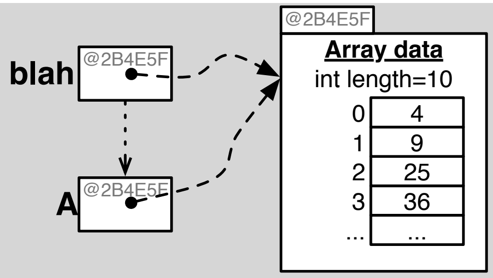
Here is a common mistake that programmers can make. Let's say you want to write a method to change the length of an array to be twice its current length (making sure to copy all the values). This method uses several variables, so when you compile space is reserved for them.

```
void resize(int[] A){
  int[] temp = new int[A.length*2];
  for(int i=0;i<temp.length; i++){
     temp[i] = A[i];
  }
  A = temp;
}
```



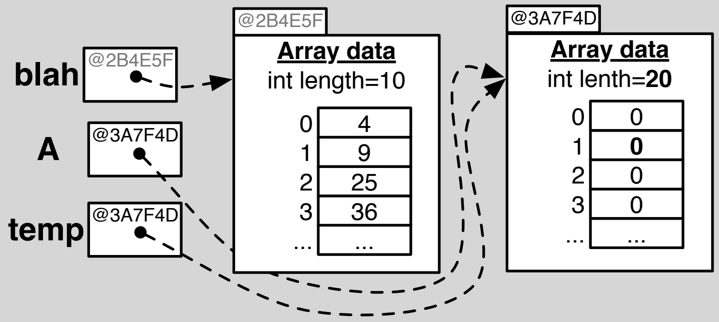
Now let's take the same setup as before and look at memory right at the moment is method is called.

```
blah=new int[10];
//blah filled with values somehow
reszie(blah);
```



Now let's look after the code is executed but just before the method returns...temp is assigned a reference to new memory for the array, the values from the old array are copied into the new one, then the code reassigns A to a refer to the same array as temp. NOTICE: the end result of this method is basically nothing, because A and temp are local variables to the method, they go away once the method is finished (and the new array along with it). This leaves the original array unaffected - blah has not been resized.

```
void resize(int[] A){
  int[] temp = new int[A.length*2];
  for(int i=0;i<temp.lenght; i++){
     temp[i] = A[i];
  }
  A = temp;
}
```



If you actually want alter blah, then the method must return a reference to the new array it created, and the code that calls the method can re-assign blah.