Types in Java

In computer programming, every variable and piece of data has a **type** that describes what values a variable of that type can hold, and what operations are supported on values of that type. For example, the `int` type can hold the integer values from -2^{31} to $2^{31} - 1$ inclusive, and the type supports operations like `+`, `-`, `*`, `/`, `%`, and more.

A value of some type is called an **instance** of that type. 10 is an instance of the `int` type, `3.14` is an instance of the `double` type, etc. Whereas a type is a description, an instance is an actual value that matches that description.

Java has two broad categories of types – **primitive** types and **object** types – but we will call them by different (better IMO) names: **value** types and **reference** types, which translate better to other programming languages.

Variables

When a variable is declared in a computer program, the programming language collaborates with the operating system to assign a memory address for that variable to live at. The exact number of bytes given to the variable, and exactly *what* is saved at the assigned address (the variable's **value**), depends on the type of the variable.

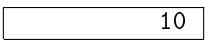
Value (Primitive) types

A value type variable stores its instance directly as the variable's value. The variable is given a certain amount of memory space that is known at compile time based on the variable's type – enough to store the binary form of the instance. We draw pictures of value types somewhat like this:

Given:

```
int x = 10;
```

We get:



where the rectangle represents the bytes set aside for the variable's instance (`10`).

When value types are copied – assigned to other variables, passed as a parameter to a function, returned from a function – they are copied "by value": by saving a **copy/duplicate** of the old variable's instance in the memory location of the new variable. Each of the variables involved in the copy are completely distinct in memory, and a future change to one of the variables will not change the other.

Exercise 1. What is the value of the variable `y` after this program executes?

```
int x = 10;
int y = x;
x = 20;
```

Primitive types in Java:

Java uses the term "primitive types" for its value types, of which the language supports **exactly 8**:

Type	Size (bytes)	Range	Purpose
`boolean`	1	`{false, true}`	Yes/no values
`byte`	1	$[-2^7, 2^7 - 1]$	Signed integer value with small range
`char`	2	N/A	Single character value
`short`	2	$[-2^{15}, 2^{15} - 1]$	Signed integer value with moderate range
`int`	4	$[-2^{31}, 2^{31} - 1]$	Signed integer value with large range; default "integer type"
`long`	8	$[-2^{63}, 2^{63} - 1]$	Signed integer value with vary large, but not infinite, range
`float`	4	N/A	Signed floating point value with limited precision
`double`	8	N/A	Signed floating point value with greater precision; default "floating point type"

All primitive types in Java can be assigned **literals**: a text representation of a value. Examples:

Literal	Type
`10`	`int`
`12345678910L`	`long`
`3.14159`	`double`
`3.14159f`	`float`
`true`	`boolean`
`'A'`	`char`

Besides creating them with literals and copying them, the only thing we can do do with primitive types is pair them with operators to perform arithmetic and logic. The "standard" set of arithmetic operators from Python can be used with all integer and floating point types.

Exercise 2. What are the values of the variables in the following program?

```
int x = 10;
int y = (x * 3 + 7) % 5;
int z = y / 2;
double w = y / 2.0;
```

The different values for `z` and `w` exhibit the difference in integer vs floating point division. Whenever two different types are involved in an operator, Java performs "promotes" the less-precise type by **coercing** it to the more-precise type; the result of the operator is in the more-precise type. `y / 2.0` therefore coerces the `int` type `y` to a `double`, and then performs division without truncating the result.

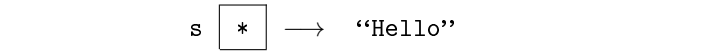
Reference (Object) types

When a variable of a reference type is declared, the computer assigns it an amount of memory that is **always** 8 bytes.[1] Those 8 bytes do **not** store the instance of the variable, in fact, they store a *reference* to another memory location at which an instance of that type can be found. We draw pictures of reference types differently than value types.

Given:

```
String s = "Hello";
```

We get:



`s`'s value is not the string "Hello", it is a *reference* to a string instance, and that instance happens to be "Hello". `s`'s value (the actual number stored in the memory assigned to `s`, represented in the picture as a `*`) is not a string, it is the memory address of another location in memory that holds a string. This is subtle but very important; we must distinguish between the variable (whose value is a memory address) and the instance it refers to.

Reference types are also copied **by value**: when a reference type variable is assigned to another, the value (memory address) of the first variable is copied to the second. Unlike value types, this does **not** create a copy *of the actual instance* (in this case `“Hello”`)... instead, it creates a copy of the memory address, thereby making the second variable refer to the same instance as the first.

Exercise 3. In the following program, how many distinct string instances exist in the computer's memory? How many string *variables* exist?

```
String s = "Hello";
String t = s;
```

Contrast your answer with how many integer instances and integer variables existed in Exercise 1.

Reassigning references:

When we assign a reference type a new value, we simply make that variable "point to" a new instance in a different location in memory. Any other variable that referred to the old instance is unaffected, and continues to happily point at their "old" instance.

Exercise 4. What instance does `t` refer to after this program executes?

```
String s = "Hello";
String t = s;
s = "Goodbye";
```

[1] It actually depends on whether you have a 32- or 64-bit processor, whether your program was compiled as a 32- or 64-bit executable, and other esoteric facts that 99% of the time comes out to be "8 bytes".

Object types in Java:

Java calls reference types "Object types", and **anything that isn't a primitive type is an object type**. This includes `String`, `Scanner`, `Random`, and millions more. Java refers to the instance as *the object*, and the variable as *the reference*. This will be **very important** later.

Unlike primitive types (that can only be assigned and applied to operators), object types can have **methods** called on them. A method is a function that manipulates or otherwise uses an object instance to do something in terms of that instance's value. For example, the `String` type has a method called `toUpperCase()`; when the method is called on a string instance, it creates and returns a new string instance by converting each English letter in the original instance to uppercase:

```
String s = "Hello";
String t = s.toUpperCase();
// t is now "HELLO"; s is still "Hello".
```

Methods cannot be called on primitive types. In exchange, operators cannot be applied to object types, so we cannot add `Scanner` objects together, or multiply a `String` by a `Random`, etc. There are two exceptions to this rule:

1. `String` instances can be concatenated together using the `+` operator; doing so creates a new string instance. Unlike Python, *any* type of value can be added to a `String` in Java, with the non-string being converted to a string representation:

   ```
   String s = "Hello";
   String t = s + "Goodbye" + 100;
   // t is now "HelloGoodbye100"
   ```

2. All objects can be compared for equality using `==` and `!=`. However, this type of equality uses **reference equality** which is only true if the two variables **refer to the same instance:**

   ```
   String s = "Hello100";
   String t = "Hello" + 100;
   if (s == t) // this is always FALSE :(
   ```

   All objects in Java therefore have a method called `equals()` which accepts another object and determines if the two instances are equal in value. This comparison is done in a way that makes sense for each object type:

   ```
   String s = "Hello100";
   String t = "Hello" + 100;
   if (s.equals(t)) // TRUE! :)
   ```