MODULE

How objects work in Java

FIT2099 staff
Updated 28 April 2017

Variables and types

Compared to languages such as JavaScript and Python, Java is a **statically typed** language. This means that variables are given a type when they are declared, and they keep this type throughout their lifetimes. When you assign a value to a variable in Java, the compiler is able to check that it is of a compatible type. When you call a method on a variable in Java, the compiler is able to check that the variable is an instance of a class that implements that method.

This means that Java can catch type errors at compile time that Python can't catch until runtime. For example:

```
>>> def minusone(number):
...     return number - 1

>>> def callminusone():
...     print(minusone(4))
...     print(minusone("hello"))

>>> callminusone()
3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in callminusone()
  File "<stdin>", line 2, in minusone
TypeError: unsupported operand type(s) for -: 'str' and 'int'
```

Note that even though `minusone()` is declared before `callminusone()`, and `callminusone()` is declared before it is called, the Python interpreter does not notice that the second call to `minusone()` will result in a type error until an attempt is made to execute the offending line.

Compare this to equivalent code in Java:

```
public class TestMinus {
    public int minusOne(int number) {
        return number - 1;
    }

    public void callMinusOne() {
        System.out.println(minusOne(4));
        System.out.println(minusOne("hello");
    }
}
```

Trying to compile this code gives you the following error message:

```
TestMinus.java:8: error: method minusOne in class TestMinus cannot be applied to given t
        System.out.println(minusOne("hello"));
```

```
                                ^
  required: int
  found: String
  reason: actual argument String cannot be converted to int by method invocation conver
1 error
```

This error message means that the Java compiler has seen that you're trying to put a `String` where an `int` should go, and is telling you that you shouldn't have done that.

The fact that this happens at the time that the code is built rather than at the time that the code is run means that Java programs are much less likely to ship with type errors in them than Python programs are.

Classes are types

Java has several built-in primitive types that you can use:

- boolean: true or false
- char: for single Unicode characters
- byte, short, int, long: integers that are 8, 16, 32, and 64 bits wide, respectively
- float: IEEE 32-bit floating point number
- double: IEEE 64-bit floating point number

These primitive types aren't classes. They don't have methods or attributes. But Java does provide some classes that you can use. The one you'll probably use most often is `String`.

In many ways, a class is just like a type. You can use them to declare variables:

```
1 int i;          // i is of type int
2 MyClass mc;     // mc is an instance of class MyClass
```

Types and classes both determine what you can and can't do with a variable. For example, you can subtract from an `int` but not from a `boolean` or a String; you can only call `myVar.aMethod()` if `myVar` is an instance of a class that defines `aMethod()`; etc. In fact, if you think of classes as being types that you can define, that'll take you a long way towards understanding them.

They're not *exactly* the same under the hood, though. Classes are more complicated than primitive types. In the next section, you'll see why.

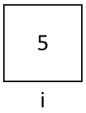
References and values

When you declare a variable, the Java virtual machine (JVM) sets some space aside for it. But primitive types and classes behave slightly differently.

If you declare an instance of a primitive type, what happens in memory is straightforward: the name of the variable gets associated with a particular location in memory, and the value of the variable gets stored there. Suppose we have this code:

```
int i = 5;
```

We can draw a picture of what that does:

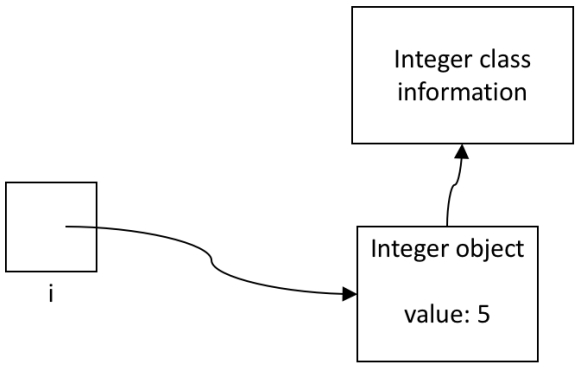


So far, this is straightforward. The variable `i` maps into a location in memory, which we've drawn as a box; the number 5 is stored at that location. But objects work a bit differently.

One of the classes that Java provides for you is one called `Integer` that, unsurprisingly, stores integers. It's used instead of `int` in places where Java expects to see an object rather than a primitive value. Suppose we have *this* code:

```
Integer i = new Integer(5);
```

Once again, we can draw this:



Memory diagram for Integer

Once again, we've got a memory location bound to the variable name `i`, but there isn't a 5 stored there. Instead, there's an address for *another* block of memory that contains all the information about this instance of `Integer`. For our purposes the important things that live here are:

- all the **instance variables** (attributes) for this object (you can see that the number 5 is stored in the attribute called `value`), and
- the address in memory of all the class information for `Integer`. (We'll look at this in more detail later on.)

Depending on the exact JVM you're using, there may also a bunch of bookkeeping stuff that only people who develop JVMs are likely to care about. Oracle doesn't specify exactly how the JVM must format its object data, only how those objects should behave during program execution.

We say that `Integer` is a **reference type**. That means that `Integer` variables, unlike `int`s, don't simply store a value. They store a reference (i.e. pointer) to a block of memory that stores attributes.

Declaration, instantiation, and initialization

In Java, there are three phases to the declaration of a variable:

- **declaration**, in which you tell the compiler the type of the variable,
- **instantiation**, in which the variable is created, and

- **initialization**, in which the variable receives its initial value.

Learning this terminology will help you understand some of the error messages and warnings that your Java IDE and compiler will occasionally come up with. Here's a quick guide to help you tell them apart:

> Declarations have the syntax *`VariableType variableName`*.
>
> Instantiation uses the keyword `new`.
>
> Initialization is done with the assignment operator, `=`.
>
> All three of these things can happen on one line, but they don't have to.

All variables in Java *must* be declared, but initialization is optional. If they're not initialized, then primitive types default to the value of `0` (if they're integers or chars) or `0.0` (if they're floating point). Reference type variables default to `null`. That's a special "reference" that doesn't refer to anything, which is why Java will give you a `NullPointerException` if you try to access a reference variable that hasn't been initialized.

Reference variables need to be made to point to objects in order to be able to be used. Often, you do this by creating a new instance of a class – that is, a new object. This creation is called **instantiation**. The code in the previous section initializes the `Integer i` by creating a new instance of the `Integer` class and then assigning it to `i`.

You can also initialize a reference by making it point to an existing object of a compatible type. For example,

```
1 Integer i = new Integer(5);
2 Integer j = i;
```

Rather than creating a new `Integer` for `j` to point to, this code simply makes `j` point to the same object in memory as `i` does. It saves having to type out the instantiation code again, but be careful – `i` and `j` are now two different names for the same object. If you like, you can think of `j` as an **alias** for `i`. Sometimes this is exactly what you want, but if you want these two variables to be able to store different values, you'll need to make sure that `j` is assigned a different object at some stage.

Constructors

There's a reason that `new Integer(5)` looks kind of like a method call. Something *is* being called: one of `Integer`'s constructors.

A constructor gets called every time you create an object. First, the VM allocates enough memory to store the new object's attributes, and sets them to default values. Then, Java looks at the types of the constructor's parameters (if any), finds a constructor that has parameters of compatible types, and calls it.

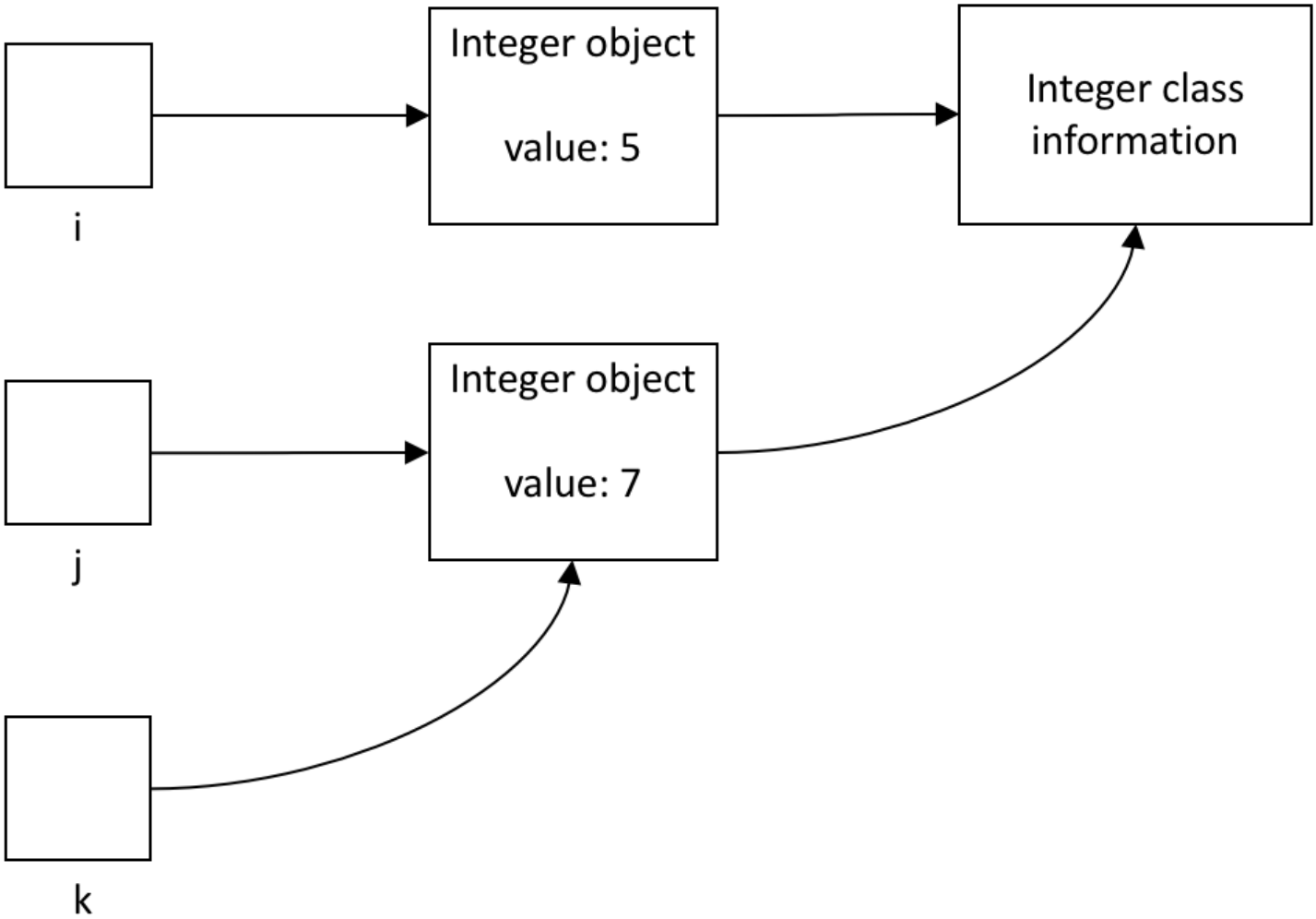
Every class has a zero-parameter constructor, even if you don't give it one. If you don't write your own zero-parameter constructor, Java will give you a default one that does nothing. That means you don't have to write trivial, do-nothing constructors.

Class memory

As well as its attributes, every instance of a class also stores the address of a block of memory that stores information about the class itself. There's only one block of this **class memory** per class, and it is shared between all instances of that class. Suppose your code looks like this:

```
1 Integer i = new Integer(5);
2 Integer j = new Integer(7);
3 Integer k = j;
```

If you run this code, you'll end up with memory that looks like this:



References, objects, and classes

As you can see, we have:

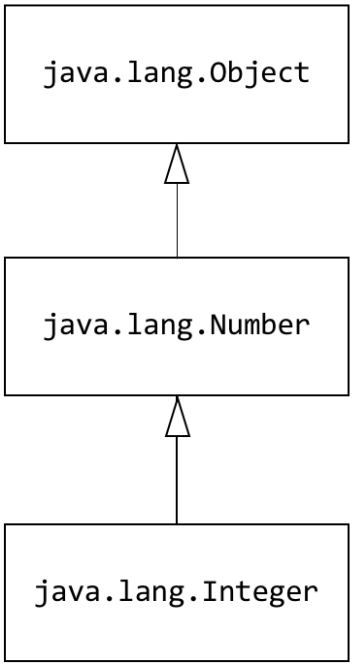
- three references to `Integer`, named `i`, `j`, and `k`;
- two `Integer` objects, storing the values 5 and 7;
- one block of class memory for `Integer`, shared between all instances of `Integer`.

The class memory contains:

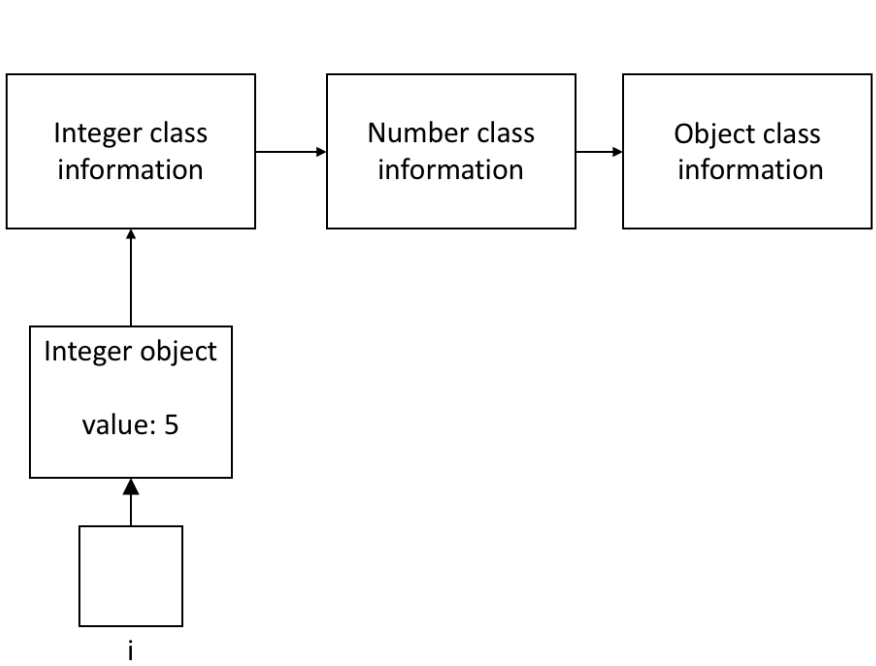
- a reference to the class memory of the parent class,
- the addresses of every method that the function has, along with their signatures (i.e. names and parameter types), and
- the values of all the class's static variables.

Inheritance and polymorphism

The class memory of every class has a reference to the class memory of its parent class. This forms a kind of linked list:



UML class diagram



Memory diagram,
`Integer i = new Integer(5);`

Let's examine the way that Java decides what code needs to run when a method is called. Suppose we call a method like this:

```
1  Integer i = new Integer(5);
2  String s = i.toString();
3  System.out.println(s);
```

The first thing that the virtual machine will do will be to follow `i`'s internal pointer to the class memory for `Integer`. There, it will search for a method named `toString` that takes no parameters. It will find one, because `Integer` does define its own `toString()` method with this signature.

If the VM does not find a method with a matching signature in the class, it will follow the class memory's pointer and look in the parent class. If there's still no matching signature, it'll keep going up the chain of inheritance until it either finds a match or runs out of parents – the end of every Java inheritance chain is always `java.lang.Object`. That's the only Java class that has no parent.

If there's no matching method anywhere in the inheritance hierarchy, the VM throws a `NoSuchMethodException`. It is rare for this to happen at runtime, because the compiler checks at the time of compilation that a suitable method will be present somewhere. However, it is possible to make it happen if you build and run on computers with different, incompatible Java libraries available. Java's good with portability, but it's definitely not perfect.

Reference type or object type?

Java learners often find themselves asking which version of a method will be executed at runtime -will it depend on the actual type of the object, or on the type of the reference you call the method on? For example, suppose you declare a reference to Number and make it point to an instance of Integer, like this:

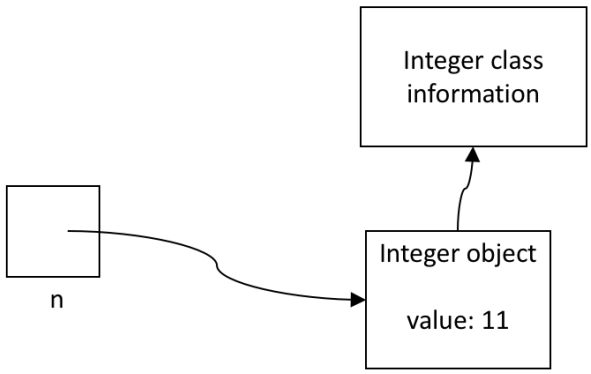
```
Number n = new Integer(11);
```

Now, this *will* compile and run correctly. An `Integer` is a `Number`, i.e. class `Integer` extends `Number`, so the assignment statement won't upset the compiler's type checking. But if we then call

```
n.toString();
```

which version of `toString()` will be executed? There's two versions to choose from, one in `Integer` and one in `Object`. (The `Number` class doesn't declare its own version of toString.)

This diagram might help you understand:



The type of the *reference* `n` is reference to `Number`. We can see this from the declaration, `Number n`. The compiler would not let you call any method in `n` that does not exist in `Number`, because even though it's *currently* pointing at an `Integer`, there's nothing to stop you from making it point at an instance of some other subclass of `Number` that might not have that method.

But it's the type of the *object* that determines which chunk of code gets run when a method is called, and when you tell Java to give you a `new Integer(11)`, you get an instance of `Integer`. Its pointer to class information points at `Integer`'s class memory, so the VM will look there first when it's searching for methods.

If you understand how this method searching mechanism works, you will find it easier to understand how method inheritance works in Java. You'll also understand how **polymorphism** works – you can declare a collection of a base class type, such as an `ArrayList<Number>`, and then use it to store instances of any subclass of `Number`, like this:

```
ArrayList&lt;Number&gt; lst = new ArrayList();
lst.add(new Integer(5));
lst.add(new Float(3.14159));
lst.add(new BigDecimal(1.23456));
```

Even though these objects are all of different types, you can still store them in the same `ArrayList`. That's because the types of each object – `Integer`, `Float`, and `BigDecimal` – are all subclasses of `Number`.

Static variables and methods

You can also see how static variables work. Although each instance `Integer` has its own copy of `value`, which allows `i` and `j` to store different integers, static variables such as `Math.PI` are stored in class memory. Only one copy can exist, no matter how many instances of the class you have or what you do with those instances.

You might be wondering, if static *variables* are special because they're stored in class memory (and therefore shared between all instances of the class), and all *methods* are shared between instances of a class, what's so special about *static methods*? Well, it's a related concept – static methods belong to the class, rather than to an instance of a class. The normal kind of non-static methods (also known as **instance methods**) are called through objects:

```
1 Integer i = new Integer(5);
2 String s = i.toString();
```

In this example, the `toString()` method in `i` gets called and `s` will be assigned the string "5". You can think of this method call as asking `i` what its string representation is. `i` knows what value it's storing, and will reply with the string "5". On the other hand, a static method call looks more like this:

```
1 String s = Integer.toBinaryString(5);
```

In this case, there's no actual instance of `Integer` involved. Instead, we're asking the *class* `Integer` what the binary representation of the number 5 is. Of course, the class has no idea what value we want the binary representation for unless we tell it, so we have to pass it in as a parameter. (The type of 5 isn't `Integer`, by the way. If you look at the documentation for the `Integer` class, you'll see that this method expects an `int`.)

You might wonder if, given that Integer knows how to convert integers to binary, we could do this:

```
1 Integer i = new Integer(5);
2 String s = i.toBinaryString();
```

And the answer is that we can't. If we try, we get this error message on compilation:

```
error: method toBinaryString in class Integer cannot be applied to given types;
   s = i.toBinaryString();
                        ^
 required: int
 found: no arguments
 reason: actual and formal argument lists differ in length
1 error
```

This error message *doesn't* arise because `toBinaryString()` is static. It's because we've dropped the parameter. The compiler looks at the type of `i`, sees that it is an `Integer`, and goes looking in `Integer` for a method called `toBinaryString()` that takes no parameters. It doesn't find one, nor is there one in `Integer`'s parent class `Number`, or in `Number`'s parent `Object`. `Object` has no parent, so compilation gets no further.

If we put the 5 back between the brackets, the code will compile and run again – but it's going to ignore all `i`'s attributes, including `i.value`. It'll return the string "101" regardless of the value we store in `i`. Here's why.

Because static methods are designed to be called on classes rather than objects, they don't have access to any instance variables. They can't. Similarly, there's no `this` reference. When you're writing a static method, you can only access static variables, local variables, and parameters.
