6.005 — Software Construction on MIT OpenCourseWare | OCW 6.005 Homepage
Spring 2016

# Reading 2: Basic Java

**Objectives**

- Learn basic Java syntax and semantics
- Transition from writing Python to writing Java

**Software in 6.005**

| Safe from bugs | Easy to understand | Ready for change |
|---|---|---|
| Correct today and correct in the unknown future. | Communicating clearly with future programmers, including future you. | Designed to accommodate change without rewriting. |

## Getting started with the Java Tutorials

The next few sections link to the **Java Tutorials** to get you up to speed with the basics.

You can also look at Getting Started: Learning Java as an alternative resource.

This reading and other resources will frequently refer you to the Java API documentation which describes all the classes built in to Java.

### Language basics

Read **Language Basics** .

You should be able to answer the questions on the *Questions and Exercises* pages for all four of the langage basics topics.

- Questions: Variables
- Questions: Operators
- Questions: Expressions, Statements, Blocks
- Questions: Control Flow

Note that each *Questions and Exercises* page has a link at the bottom to solutions.

Also check your understanding by answering some questions about how the basics of Java compare to the basics of Python:

### Numbers and strings

Read **Numbers and Strings** .

Don't worry if you find the `Number` wrapper classes confusing. They are.

You should be able to answer the questions on both *Questions and Exercises* pages.

- Questions: Numbers
- Questions: Characters, Strings

### Classes and objects

Read **Classes and Objects** .

You should be able to answer the questions on the first two *Questions and Exercises* pages.

- Questions: Classes
- Questions: Objects

Don't worry if you don't understand everything in *Nested Classes* and *Enum Types* right now. You can go back to those constructs later in the semester when we see them in class.

### Hello, world!

Read **Hello World!**

You should be able to create a new `HelloWorldApp.java` file, enter the code from that tutorial page, and compile and run the program to see `Hello World!` on the console.

## Snapshot diagrams

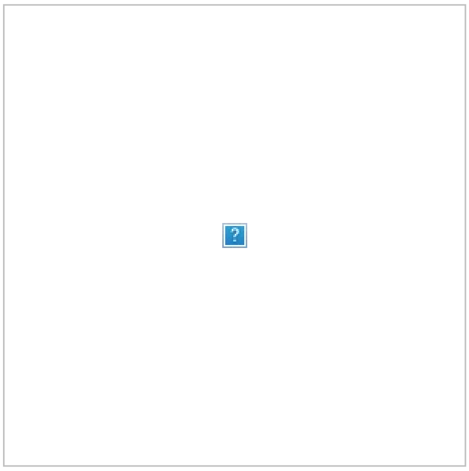
It will be useful for us to draw pictures of what’s happening at runtime, in order to understand subtle questions. **Snapshot diagrams** represent the internal state of a program at runtime – its stack (methods in progress and their local variables) and its heap (objects that currently exist).

Here’s why we use snapshot diagrams in 6.005:

- To talk to each other through pictures (in class and in team meetings)
- To illustrate concepts like primitive types vs. object types, immutable values vs. immutable references, pointer aliasing, stack vs. heap, abstractions vs. concrete representations.
- To help explain your design for your team project (with each other and with your TA).
- To pave the way for richer design notations in subsequent courses. For example, snapshot diagrams generalize into object models in 6.170.

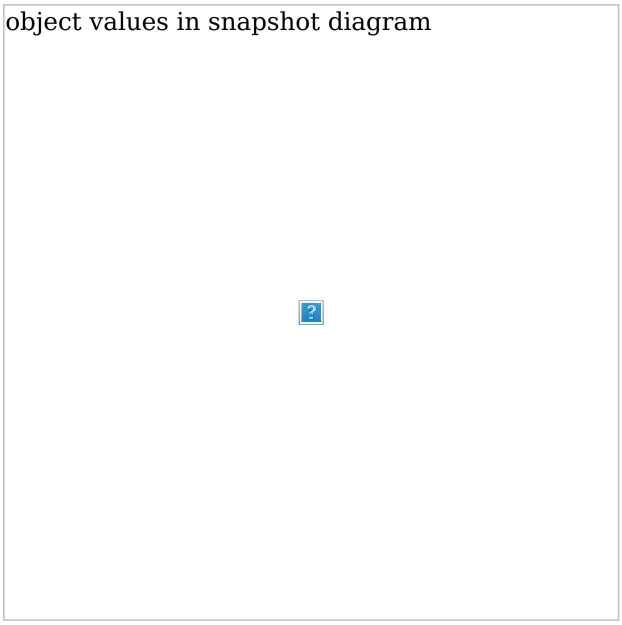
Although the diagrams in this course use examples from Java, the notation can be applied to any modern programming language, e.g., Python, Javascript, C++, Ruby.

### Primitive values



Primitive values are represented by bare constants. The incoming arrow is a reference to the value from a variable or an object field.

### Object values



An object value is a circle labeled by its type. When we want to show more detail, we write field names inside it, with arrows pointing out to their values. For still more detail, the fields can include their declared types. Some people prefer to write `x:int` instead of `int x` , but both are fine.

## Mutating values vs. reassigning variables

Snapshot diagrams give us a way to visualize the distinction between changing a variable and changing a value:

- When you assign to a variable or a field, you're changing where the variable's arrow points. You can point it to a different value.
- When you assign to the contents of a mutable value – such as an array or list – you're changing references inside that value.

### Reassignment and immutable values

reassigning a variable

For example, if we have a `String` variable `s` , we can reassign it from a value of `"a"` to `"ab"` .

```
String s = "a";
s = s + "b";
```

`String` is an example of an *immutable* type, a type whose values can never change once they have been created. Immutability (immunity from change) is a major design principle in this course, and we'll talk much more about it in future readings.

Immutable objects (intended by their designer to always represent the same value) are denoted in a snapshot diagram by a double border, like the `String` objects in our diagram.

### Mutable values

mutating an object

By contrast, `StringBuilder` (another built-in Java class) is a *mutable* object that represents a string of characters, and it has methods that change the value of the object:

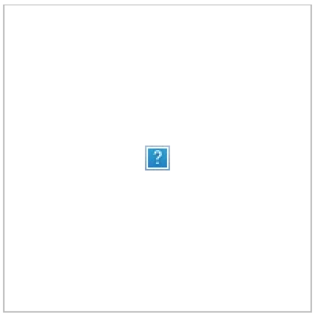
```
StringBuilder sb = new StringBuilder("a");
sb.append("b");
```

These two snapshot diagrams look very different, which is good: the difference between mutability and immutability will play an important role in making our code *safe from bugs* .

### Immutable references

Java also gives us immutable references: variables that are assigned once and never reassigned. To make a reference immutable, declare it with the keyword `final` :

```
final int n = 5;
```



If the Java compiler isn't convinced that your `final` variable will only be assigned once at runtime, then it will produce a compiler error. So `final` gives you static checking for immutable references.

In a snapshot diagram, an immutable reference ( `final` ) is denoted by a double arrow. Here's an object whose `id` never changes (it can't be reassigned to a different number), but whose `age` can change.

Notice that we can have an *immutable reference* to a *mutable value* (for example: `final StringBuilder sb` ) whose value can change even though we're pointing to the same object.

We can also have a *mutable reference* to an *immutable value* (like `String s` ), where the value of the variable can change because it can be re-pointed to a different object.

---

# Java Collections

The very first Language Basics tutorial discussed **arrays** , which are *fixed-length* containers for a sequence of objects or primitive values. Java provides a number of more powerful and flexible tools for managing *collections* of objects: the **Java Collections Framework** .

## Lists, Sets, and Maps

**A Java `List` is similar to a Python list .** A `List` contains an ordered collection of zero or more objects, where the same object might appear multiple times. We can add and remove items to and from the `List` , which will grow and shrink to accomodate its contents.

Example `List` operations:

| Java | description | Python |
|---|---|---|
| `int count = lst.size();` | count the number of elements | `count = len(lst)` |
| `lst.add(e);` | append an element to the end | `lst.append(e)` |
| `if (lst.isEmpty()) ...` | test if the list is empty | `if not lst: ...` |

In a snapshot diagram, we represent a `List` as an object with indices drawn as fields:

This list of `cities` might represent a trip from Boston to Bogotá to Barcelona.

**A `Set` is an unordered collection of zero or more unique objects.** Like a mathematical *set* or a Python set – and unlike a `List` – an object cannot appear in a set multiple times. Either it's in or it's out.

Example `Set` operations:

| Java | description | Python |
|---|---|---|
| `s1.contains(e)` | test if the set contains an element | `e in s1` |
| `s1.containsAll(s2)` | test whether *s1* ⊇ *s2* | `s1.issuperset(s2)`<br>`s1 >= s2` |
| `s1.removeAll(s2)` | remove *s2* from *s1* | `s1.difference_update(s2)`<br>`s1 -= s2` |

In a snapshot diagram, we represent a `Set` as an object with no-name fields:

Here we have a set of integers, in no particular order: 42, 1024, and -7.

**A `Map` is similar to a Python dictionary .** In Python, the **keys** of a map must be hashable . Java has a similar requirement that we'll discuss when we confront how equality works between Java objects.

Example `Map` operations:

| Java | description | Python |
|---|---|---|
| `map.put(key, val)` | add the mapping *key* → *val* | `map[key] = val` |
| `map.get(key)` | get the value for a key | `map[key]` |
| `map.containsKey(key)` | test whether the map has a key | `key in map` |

| | | |
|---|---|---|
| `map.remove(key)` | delete a mapping | `del map[key]` |

In a snapshot diagram, we represent a `Map` as an object that contains key/value pairs:

This `turtles` map contains `Turtle` objects assigned to `String` keys: Bob, Buckminster, and Buster.

## Literals

Python provides convenient syntax for creating lists:

```
lst = [ "a", "b", "c" ]
```

And maps:

```
map = { "apple": 5, "banana": 7 }
```

**Java does not.** It does provide a literal syntax for arrays:

```
String[] arr = { "a", "b", "c" };
```

But this creates an *array* , not a `List` . We can use a provided utility function to create a `List` from the array:

```
Arrays.asList(new String[] { "a", "b", "c" })
```

A `List` created with `Arrays.asList` does come with a restriction: its length is fixed.

## Generics: declaring List, Set, and Map variables

Unlike Python collection types, with Java collections we can restrict the type of objects contained in the collection. When we add an item, the compiler can perform *static checking* to ensure we only add items of the appropriate type. Then, when we pull out an item, we are guaranteed that its type will be what we expect.

Here’s the syntax for declaring some variables to hold collections:

```
List<String> cities;        // a List of Strings
Set<Integer> numbers;       // a Set of Integers
Map<String,Turtle> turtles; // a Map with String keys and Turtle values
```

Because of the way generics work, we cannot create a collection of primitive types. For example, `Set<int>` does *not* work. However, as we saw earlier, `int` s have an `Integer` wrapper we can use (e.g. `Set<Integer> numbers` ).

In order to make it easier to use collections of these wrapper types, Java does some automatic conversion. If we have declared `List<Integer> sequence` , this code works:

```
sequence.add(5);              // add 5 to the sequence
int second = sequence.get(1); // get the second element
```

## ArrayLists and LinkedLists: creating Lists

As we’ll see soon enough, Java helps us distinguish between the *specification* of a type – what does it do? – and the *implementation* – what is the code?

`List` , `Set` , and `Map` are all *interfaces* : they define how these respective types work, but they don’t provide implementation code. There are several advantages, but one potential advantage is that we, the users of these types, get to choose different implementations in different situations.

Here’s how to create some actual `List` s:

```
List<String> firstNames = new ArrayList<String>();
List<String> lastNames = new LinkedList<String>();
```

If the generic type parameters are the same on the left and right, Java can infer what’s going on and save us some typing:

```
List<String> firstNames = new ArrayList<>();
List<String> lastNames = new LinkedList<>();
```

`ArrayList` and `LinkedList` are two implementations of `List` . Both provide all the operations of `List` , and those operations must work as described in the documentation for `List` . In this example, `firstNames` and `lastNames` will behave the same way; if we swapped which one used `ArrayList` vs. `LinkedList` , our code will not break.

Unfortunately, this ability to choose is also a burden: we didn’t care how Python lists worked, why should we care whether our Java lists are `ArrayLists` or `LinkedLists` ? Since the only difference is performance, for 6.005 *we don’t* .

When in doubt, use `ArrayList` .

## HashSets and HashMaps: creating Sets and Maps

`HashSet` is our default choice for `Set` s:

```
Set<Integer> numbers = new HashSet<>();
```

Java also provides sorted sets with the `TreeSet` implementation.

And for a `Map` the default choice is `HashMap`:

```
Map<String,Turtle> turtles = new HashMap<>();
```

### Iteration

So maybe we have:

```
List<String> cities        = new ArrayList<>();
Set<Integer> numbers       = new HashSet<>();
Map<String,Turtle> turtles = new HashMap<>();
```

A very common task is iterating through our cities/numbers/turtles/etc.

In Python:

```
for city in cities:
    print city

for num in numbers:
    print num

for key in turtles:
    print "%s: %s" % (key, turtles[key])
```

Java provides a similar syntax for iterating over the items in `List` s and `Set` s.

Here's the Java:

```
for (String city : cities) {
    System.out.println(city);
}

for (int num : numbers) {
    System.out.println(num);
}
```

We can't iterate over `Map` s themselves this way, but we can iterate over the keys as we did in Python:

```
for (String key : turtles.keySet()) {
    System.out.println(key + ": " + turtles.get(key));
}
```

Under the hood this kind of `for` loop uses an `Iterator`, a design pattern we'll see later in the class.

#### Iterating with indices

If you want to, Java provides different `for` loops that we can use to iterate through a list using its indices:

```
for (int ii = 0; ii < cities.size(); ii++) {
    System.out.println(cities.get(ii));
}
```

Unless we actually need the index value `ii` , this code is verbose and has more places for bugs to hide. Avoid.

## Java API documentation

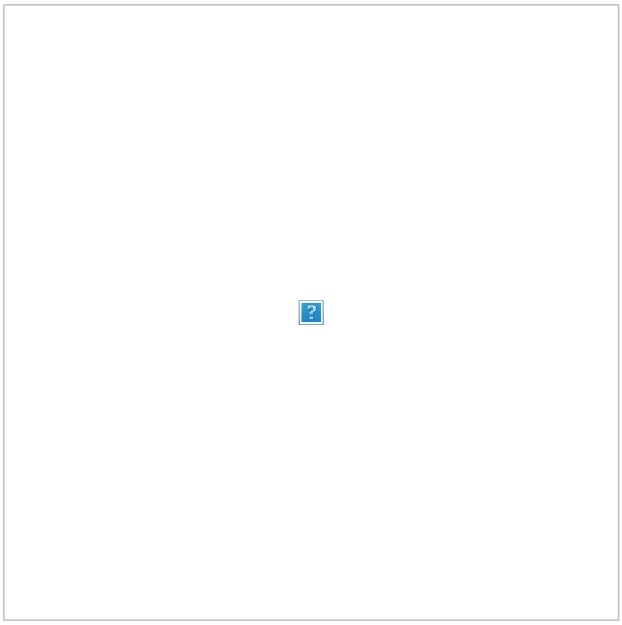
The previous section has a number of links to documentation for classes that are part of the Java platform API .

API stands for *application programming interface* . If you want to program an app that talks to Facebook, Facebook publishes an API (more than one, in fact, for different languages and frameworks) you can program against. The Java API is a large set of generally useful tools for programming pretty much anything.

- **`java.lang.String`** is the full name for `String` . We can create objects of type `String` just by using `"double quotes"` .
- **`java.lang.Integer`** and the other primitive wrapper classes. Java automagically converts between primitive and wrapped (or “boxed”) types in most situations.
- **`java.util.List`** is like a Python list, but in Python, lists are part of the language. In Java, `List` s are implemented in… Java!
- **`java.util.Map`** is like a Python dictionary.
- **`java.io.File`** represents a file on disk. Take a look at the methods provided by `File` : we can test whether the file is readable, delete the file, see when it was last modified…

- **`java.io.FileReader`** lets us read text files.
- **`java.io.BufferedReader`** lets us read in text efficiently, and it also provides a very useful feature: reading an entire line at a time.

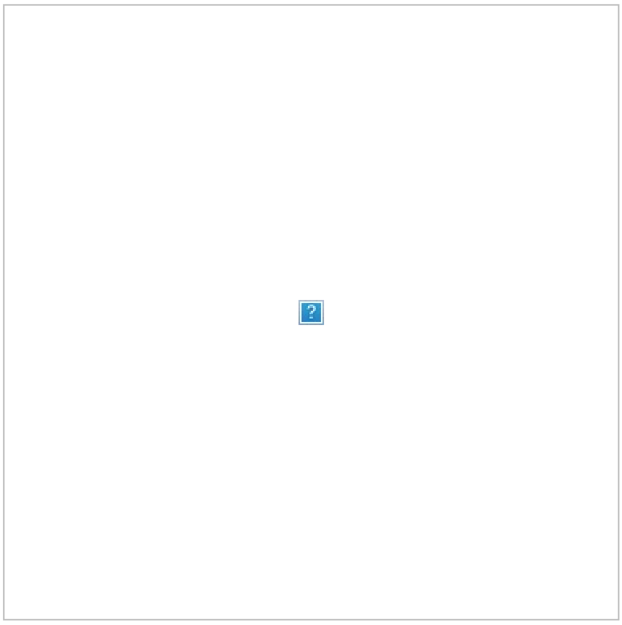
Let’s take a closer look at the documentation for `BufferedReader`. There are many things here that relate to features of Java we haven’t discussed! Keep your head and focus on the **things in bold** below.



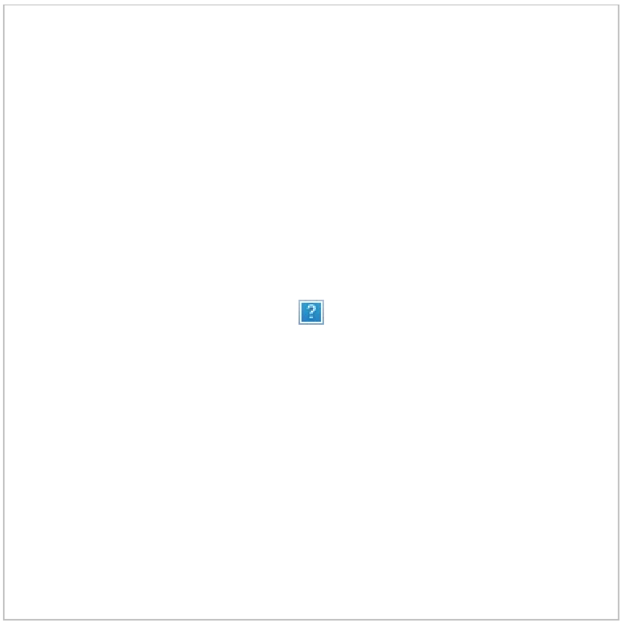
At the top of the page is the *class hierarchy* for `BufferedReader` and a list of *implemented interfaces* . A `BufferedReader` object has all of the methods of all those types (plus its own methods) available to use.

Next we see *direct subclasses* , and for an interface, *implementing classes* . This can help us find, for example, that `HashMap` is an implementation of `Map`.

Next up: **a description of the class** . Sometimes these descriptions are a little obtuse, but **this is the first place you should go** to understand a class.



If you want to make a new `BufferedReader` the **constructor summary** is the first place to look. Constructors aren’t the only way to get a new object in Java, but they are the most common.



Next: **the method summary lists all the methods we can call** on a `BufferedReader` object.

Below the summary are detailed descriptions of each method and constructor. **Click a constructor or method to see the detailed description.** This is the first place you should go to understand what a method does.

Each detailed description includes:

- The **method signature** : we see the return type, the method name, and the parameters. We also see *exceptions* . For now, those usually mean errors the method can run into.
- The full **description** .
- **Parameters** : descriptions of the method arguments.
- And a description of what the method **returns** .

## Specifications

These detailed descriptions are **specifications** . They allow us to use tools like `String` , `Map` , or `BufferedReader` *without* having to read or understand the code that implements them.

Reading, writing, understanding, and analyzing specifications will be one of our first major undertakings in 6.005, starting in a few classes.