Section 9.5: The Null literal

The Null literal (written as `null`) represents the one and only value of the null type. Here are some examples

```
MyClass object = null;

MyClass[] objects = new MyClass[]{new MyClass(), null, new MyClass()}; myMethod(null);
if (objects != null) {
// Do something
}
```

The null type is rather unusual. It has no name, so you cannot express it in Java source code. (And it has no runtime representation either.)

The sole purpose of the null type is to be the type of `null`. It is assignment compatible with all reference types, and can be type cast to any reference type. (In the latter case, the cast does not entail a runtime type check.)

Finally, `null` has the property that `null instanceof <SomeReferenceType>` will evaluate to `false`, no matter what the type is.

Section 10.5: Converting Primitives

In Java, we can convert between integer values and floating-point values. Also, since every character corresponds to a number in the Unicode encoding, `char` types can be converted to and from the integer and floating-point types. `boolean` is the only primitive datatype that cannot be converted to or from any other primitive datatype.

There are two types of conversions: *widening conversion* and *narrowing conversion*.

A *widening conversion* is when a value of one datatype is converted to a value of another datatype that occupies more bits than the former. There is no issue of data loss in this case.

Correspondingly, A *narrowing conversion* is when a value of one datatype is converted to a value of another datatype that occupies fewer bits than the former. Data loss can occur in this case.

Java performs *widening conversions* automatically. But if you want to perform a *narrowing conversion* (if you are sure that no data loss will occur), then you can force Java to perform the conversion using a language construct known as a `cast`.

Widening Conversion:

```
int a = 1;
double d = a;      // valid conversion to double, no cast needed (widening)
```

Narrowing Conversion:

```
double d = 18.96
int b = d;// invalid conversion to int, will throw a compile-time error
int b = (int) d; // valid conversion to int, but result is truncated (gets rounded down)
// This is type-casting
// Now, b = 18
```

Chapter 11: Strings

Strings (`java.lang.String`) are pieces of text stored in your program. Strings are **not** a primitive data type in Java, however, they are very common in Java programs.

In Java, Strings are immutable, meaning that they cannot be changed. (Click here for a more thorough explanation of immutability.)

Section 11.1: Comparing Strings

```
String firstString = "Test123"; String secondString = "Test" + 123;

if (firstString.equals(secondString)) {
// Both Strings have the same content.
}
```

This example will compare them, independent of their case:

```
String firstString = "Test123"; String secondString = "TEST123";

if (firstString.equalsIgnoreCase(secondString)) {
// Both Strings are equal, ignoring the case of the individual characters.
}
```

Do not use the == operator to compare Strings
String orderings

The `String` class implements `Comparable<String>` with the `String.compareTo` method (as described at the start of this example). This makes the natural ordering of `String` objects case-sensitive order. The `String` class provide a `Comparator<String>` constant called `CASE_INSENSITIVE_ORDER` suitable for case-insensitive sorting.

Section 11.2: Changing the case of characters within a String

The `String` type provides two methods for converting strings between upper case and lower case:

- `toUpperCase` to convert all characters to upper case
- `toLowerCase` to convert all characters to lower case

These methods both return the converted strings as new `String` instances: the original `String` objects are not modified because `String` is immutable in Java.

```
String string = "This is a Random String";
```

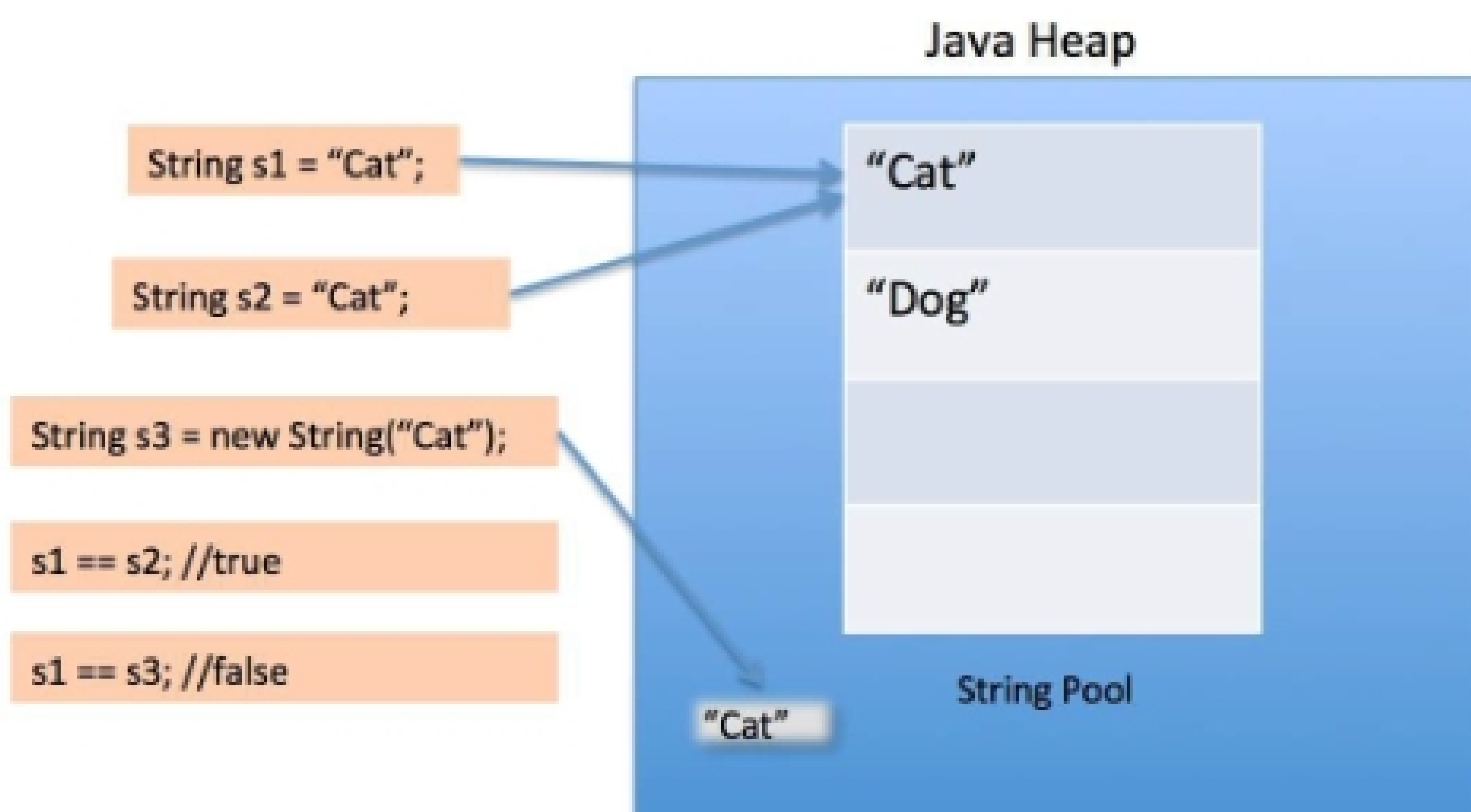
```
String upper = string.toUpperCase(); String lower = string.toLowerCase();

System.out.println(string); // prints "This is a Random String" System.out.println(lower); // prints "this is a random string" System.out.println(upper); // prints "THIS IS A RANDOM STRING"
```

```
b.contains(a); // Return true if a is contained in b, false otherwise
```

To find the exact position where a String starts within another String, use String.indexOf():

```
String s = "this is a long sentence";
int i = s.indexOf('i');// the first 'i' in String is at index 2
int j = s.indexOf("long"); // the index of the first occurrence of "long" in s is 10
int k = s.indexOf('z');// k is -1 because 'z' was not found in String s
int h = s.indexOf("LoNg"); // h is -1 because "LoNg" was not found in String s
```

Like many Java objects, **all** `String` instances are created on the heap, even literals. When the JVM finds a `String` literal that has no equivalent reference in the heap, the JVM creates a corresponding `String` instance on the heap **and** it also stores a reference to the newly created `String` instance in the String pool. Any other references to the same `String` literal are replaced with the previously created `String` instance in the heap.



However using new operator, we force String class to create a new String object in heap space. We can use intern() method to put it into the pool or refer to other String object from string pool having same value.

Section 11.5: Splitting Strings

You can split a `String` on a particular delimiting character or a Regular Expression, you can use the `String.split()` method that has the following signature:

```
public String[] split(String regex)
```

Note that delimiting character or regular expression gets removed from the resulting String Array.

Example using delimiting character:

```
String lineFromCsvFile = "Mickey;Bolton;12345;121216"; String[] dataCells =
lineFromCsvFile.split(";");
// Result is dataCells = { "Mickey", "Bolton", "12345", "121216"};
```

Example using regular expression:

```
String lineFromInput = "What    do you need    from me?";
String[] words = lineFromInput.split("\\s+"); // one or more space chars
// Result is words = {"What", "do", "you", "need", "from", "me?"};
```

You can even directly split a `String` literal:

```
String[] firstNames = "Mickey, Frank, Alicia, Tom".split(", ");
// Result is firstNames = {"Mickey", "Frank", "Alicia", "Tom"};
```