SFU

CMPT 225

Lab 02
Spring 2023

Primitive vs Non-Primitive Types

Earlier we saw the 8 primitive types in Java.

byte , short, int, long, char, boolean,double, float

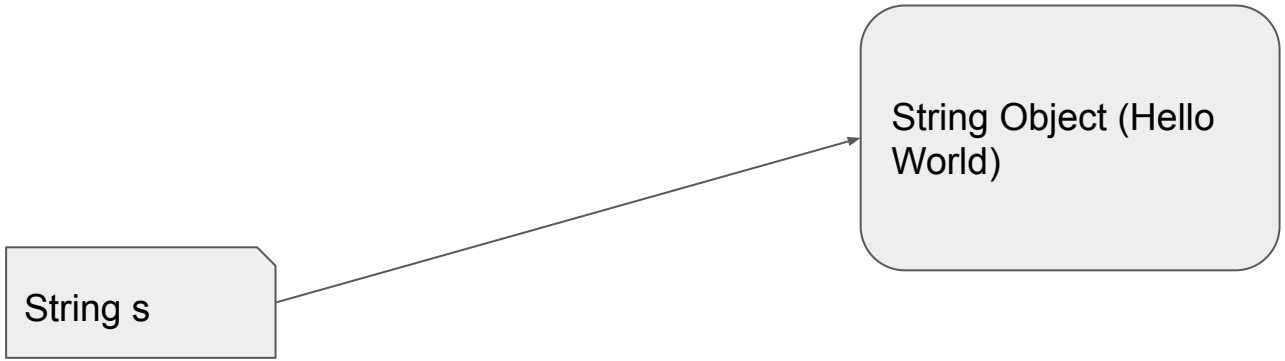
Every other type is non-primitive. Non-primitive type are also called reference types because they refers to objects. e.g.

String s = new String(“Hello World”);

s is a reference variable of type String

an instance(object) of class String is created with new

...



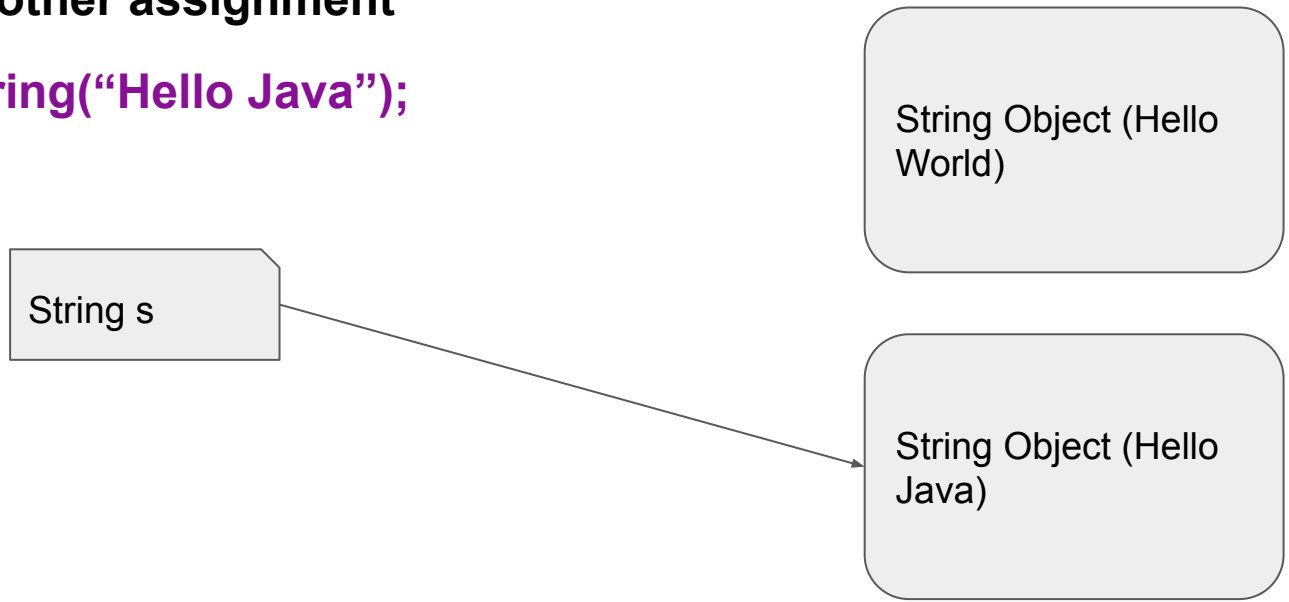
String s = new String(“Hello World”);

...

SFU

// we do another assignment

s = new String("Hello Java");



Size

SFU

Size of primitive types is typically in range of 1 byte to 8 bytes

Size of references is typically 4 bytes or 8 bytes(dependending on hardware)

Size of Objects can be huge(depending on heap memory available to JVM)

== vs equals

SFU

== is an **operator** that checks if both operands have same values.

i.e. in case both operands are references, it checks if they point to the same object.

equals is a **method** defined in Object class of Java(which we can override to

to check for equality of instances of our class)

```
public boolean equals(Object obj){                    //from Object class
```

At times we want to compare objects based on contents(or some attributes), we override equals method.(It is generally necessary to override the hashCode() method also, when overriding

equals() method)

== vs equals

SFU

```
public static void main(String[] args) {
    Student a = new Student( name: "Alice", id: "1");
    Student b = new Student( name: "Bob", id: "1");
    if( a==b ){
        System.out.println("a and b point to same object");
    }else if(a.equals(b)){
        System.out.println("a and b equals according to the equals method definition");
    }else{
        System.out.println("a and b are not equal");
    }
}
```

== vs equals …

SFU

```
@Override
public boolean equals(Object obj){
    if (obj instanceof Student) {
        return this.id.equals(((Student)obj).id);
    }
    return false;
}
```

compareTo Example

SFU

We want to compare objects of a class and assign them an order.

Implement interface `Comparable (defined in java.lang)` and implement compareTo() method

e.g. in String class

```
int compareTo(String anotherString)
```

returns -1 or 1 if they are not equal, 0 if they are equal

(We should implement compareTo such that the case 0 is compatible with equals method)

Generics

SFU

Generics enables us to pass ***types*** as parameter to classes/interfaces/methods

It provides us a way to use the same code with different types of input.

The type is passed in <...>

e.g.

```
List<Integer> list1= new ArrayList<>();
List<String> list2 = new ArrayList<>();
```

We used the same class List to define List of different types(Integer and String)

...

Helps us avoid code repetition.

Helps us to detect compile time bugs

e.g.

```
List<Integer> list1= new ArrayList<>();
list1.add("Hello");          //compile time error
```

Against

```
List list = new ArrayList();
list.add(1);list.add("Hello");   //no compile time error – leads to bugs
```

Implementing Comparable Interface

SFU

A Sorting Example

Both classes Student and Lake implements the Comparable interface

In Sorting class, method sort expects an array of Comparables.

We can use the same method to sort array of Students and array of Lakes, even though they compare different things.

Can you implement another sorting algorithm which is more efficient?

Comparator interface

Part of java.util package

- Can be passed to a sort method(Collections.sort or Arrays.sort)
- The actual classes which are being compared are not modified.

```
int compare(T o1,T o2)
```

Assignment 1 - Hints

SFU

We need to implement the given IntegerIterator interface.

In ArrayIterator.java, in the constructor, an array of integers is passed. We want to store a reference to this array in a variable, so that we can access it later.

We also want to keep a current index variable which we keep updating inside different methods.