Java Basics 2

Lecture 2

By Marina Barsky

Functions (static methods)
Variable scope
Reference type
Strings
Scanner

Extending primitive types

- We learned about default **data types** in Java: what are they?
- We can always extend existing data types by defining a new type (class) of objects

```
public class Dog {
   int size;
   void bark() {
     System.out.println("Ruff!");
   }
}
```

instance variable

method

```
public class DogTestDrive {
   public static void main (String[] args) {
   Dog d = new Dog();
     d.size = 40;
     d.bark();
   }
}
```

Declare a variable of type Dog

Set its size and call its method

dot operator

Static methods

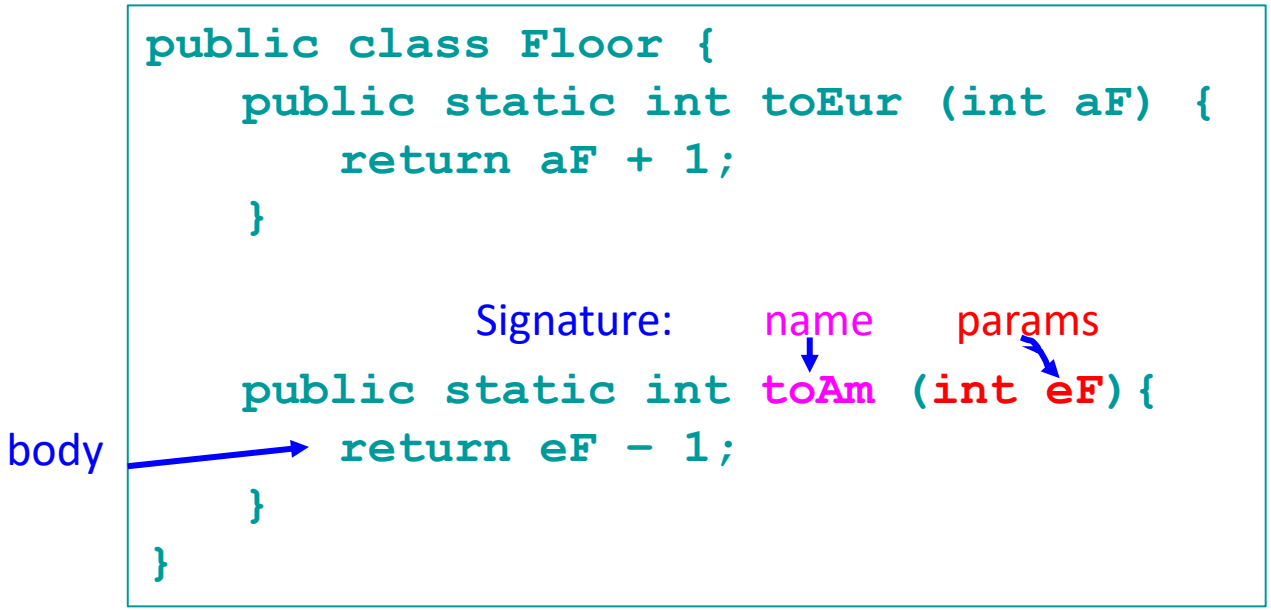
- If the method is declared as ***static***, we can use it **without creating an object**
- Static methods are associated with a given class name, and can be used similarly to functions in other languages

```
public class Floor {
    public static int toEur (int aF) {
        return aF + 1;
    }
    public static int toAm (int eF){
        return eF - 1;
    }
}
```

```
public class FloorTestDrive {
  public static void main (String[] args) {
     int eF = 5;
     int aF = Floor.toAm(eF);
    System.out.println(aF);
  }
}
```

Each method is composed of:

- Signature – defines the name and parameters
- Body – defines what the method does



Method Signature

```
[modifiers] returnType name ([params]) {
   // Method body
}
```

- Composed of method name and params
- A signature must be unique in a given class

```
public class Floor {
    public static int toEur (int aF) {
        return aF + 1;
    }


    public static double toEur (double aF){
        return aF + 1;
    }
}
```

The program will know which method to call based on the type of the parameter

Return Type

- Defines the type of the returned value
- The value returned from the method can be assigned to a variable of the same type

- If you do not need the method to return anything, declare it as *void*
- If the return type is not void, the method **must** have a return statement from any program path, and it must return an object of the corresponding type

Is this a valid Java method?

```
String static test(int x) {
   if (x == 2) {
      return "hello";
   } else {
      return 2;
   }
}
```

A. Yes
B. No
C. It depends on value of x

Which method will be called if I run test(5.5)?

```
String test(int x) {
   …
}
boolean test (double x){
   …
}
```

A. The first
B. The second
C. It depends
D. This will cause a compiler error

Which method will be called if I run test(5)?

```
String test(int x) {
   …
}
boolean test (int x){
   …
}
```

A. The first
B. The second
C. It depends
D. This will cause a compiler error

Variable scope: instance (class) variables

- *Instance variables* declared at the class level are accessible throughout the class, following the variable declaration

```
public class Dog {
  int size;    ← instance variable
  void grow() {
    size++;
  }

  void report() {
    System.out.println("I am a dog of size "+size);
  }
}
```

Variable scope: local variables

- For *local variables* declared **inside the method:**
  - The scope begins right after the variable is declared
  - The scope ends with the first closing curly bracket following the declaration

```
public class Dog {
  int size;
  void grow(int cm) {
    int weight = 0;
    size++;
    weight++;
  }

  void report() {
    System.out.println("I am a dog");
    System.out.println("My size is "+ size);
    System.out.println("My weight is "+weight);
  }
}
```

local variable

This will not compile: *weight* variable is out of scope!

Method parameters: scope

- Method **parameters** are passed **by copy** in Java.
- That means that new variables of the corresponding type are created and the value of a caller is copied into them

```
public class Dog {
  int size;
  void bark(int num) {
    while (num > 0) {
      System.out.println("Ruff!");
      num--;
    }
  }

  public static void main (String[] args) {
    Dog d = new Dog();
    int numBarks = 5;
    d.bark(numBarks);
    System.out.println(numBarks);
  }
}
```

Here the scope of *num* ends

numBarks is copied into a new variable *num*

numBarks is still 5

Iteration variables: scope

- Variables declared in the header of a for loop, are only accessible inside the loop

```
public class Dog {
   int size;
   void bark(int num) {
      for(int i=0; i< num; i++)
         System.out.println("Ruff!");
      System.out.println(i);
   }
}
```

This will not compile: *variable i* is out of scope!

Initial values

- Uninitialized ***instance variables*** of primitive type are given **default** values

```
int age;    // Initialized to 0
double speed; // Initialized to 0.0
char grade; // Initialized to \u0000 (Unicode)
boolean loggedIn; // Initialized to false
```

- Uninitialized ***local variables*** declared in a method are ***not*** given default values
  - Rule of Thumb: Always initialize a local variable when you declare it!
  - Compiler will warn you if you don't

Storing new types in a variable

- With a new class of Objects – we create a new data type
- How do we declare a variable of a new type – what is the size of a cup?

- An object reference variable doesn't hold the object itself, but it holds something like a pointer (or an address)
- Except, in Java we don't really know the value of this address
- And the JVM knows how to use the reference to get to the actual object

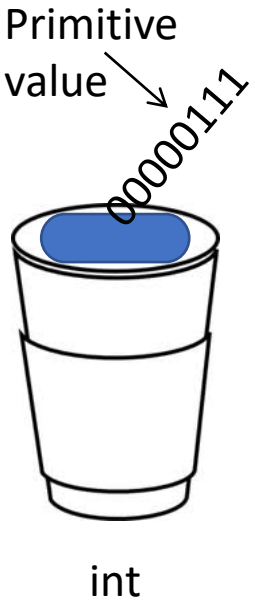
Reference and value

- An object reference is just another variable value.
- Something that goes into the cup.

Primitive variable:

```
int x=7;
```

The bits representing 7 go into the cup



Reference variable:

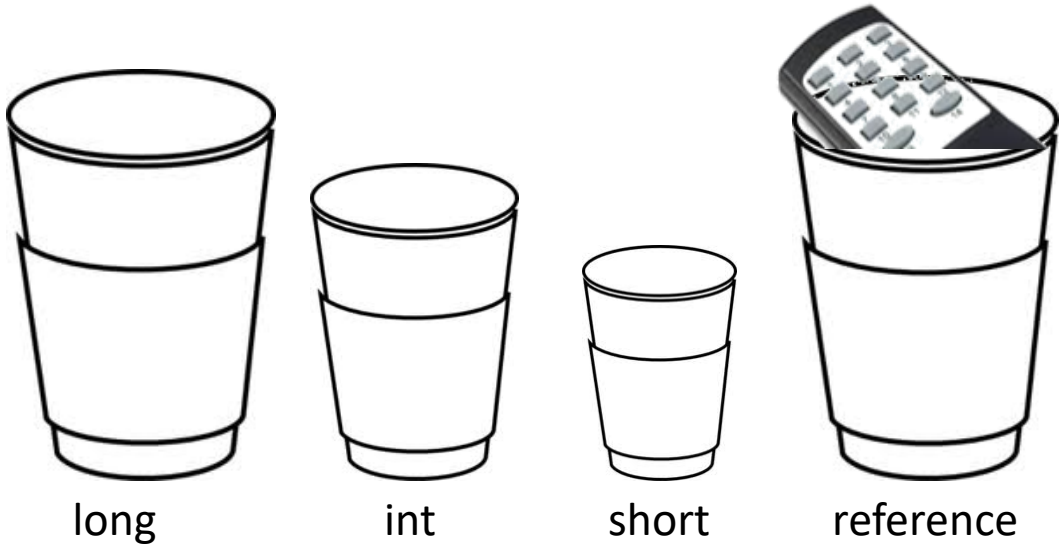
```
Dog d=new Dog();
```

The bits representing a way to get to the Dog object go into the cup

Reference variables

Dog myDog;

- reference variable of type *Dog*
- does not reference any actual object yet
- has default value *null*
- cannot call any methods of Dog class



Size of reference variables is the same for a given operating system: for example it is long for 64-bit system

When the object is created

Dog myDog;

myDog=new Dog();

myDog.bark();

Now we can call the methods of class Dog

Where the object is created

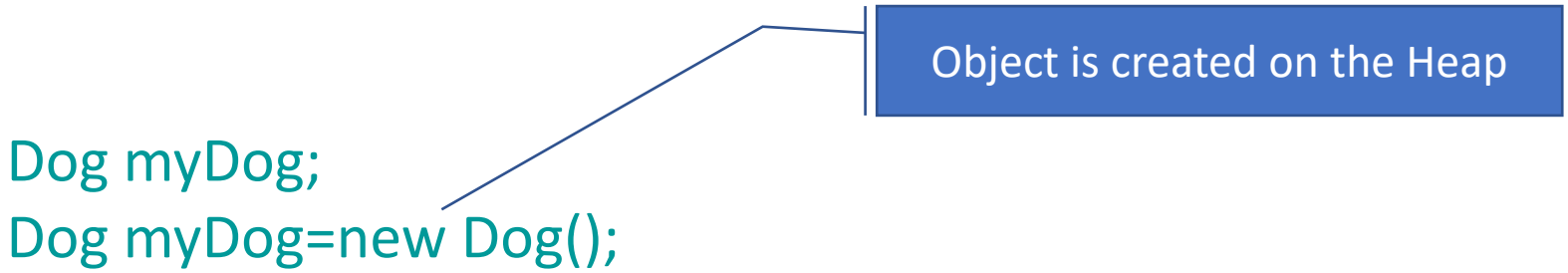
- There are several types of memory:

- Stack: very fast, limited amount

  All primitives and reference variables are allocated on the stack

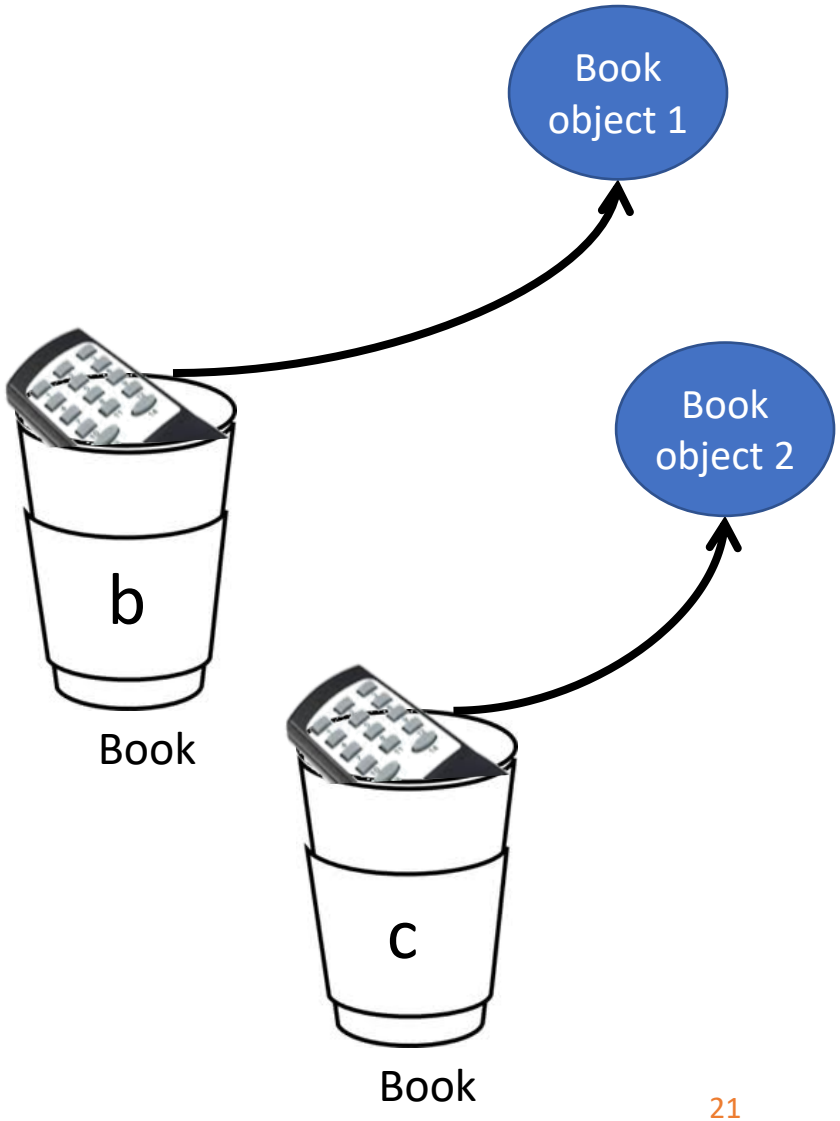
- Heap: slower, flexible, large as the RAM

  All Java objects live on the heap



Assigning references I

```
Book b=new Book();
Book c=new Book();
```



References: 2

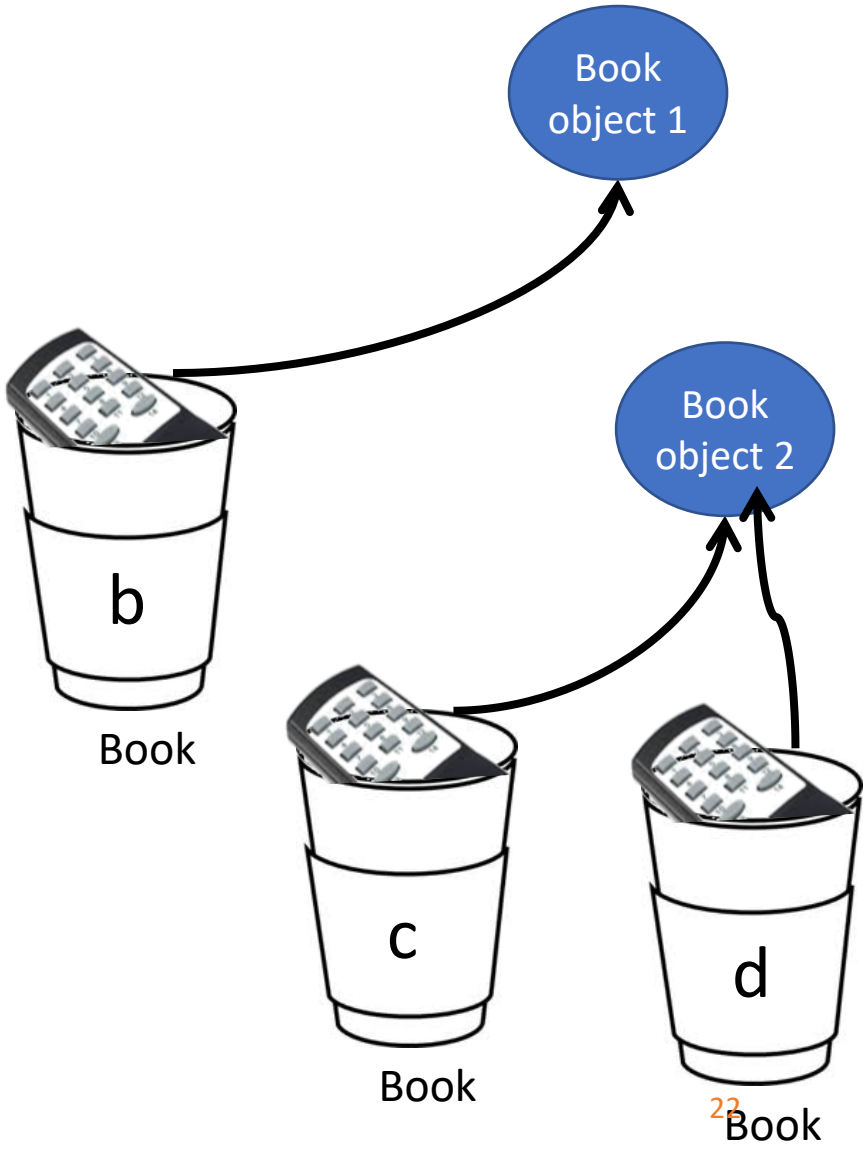
Objects: 2

Assigning references II

```
Book b=new Book();
Book c=new Book();
Book d=c;
```

References: 3

Objects: 2

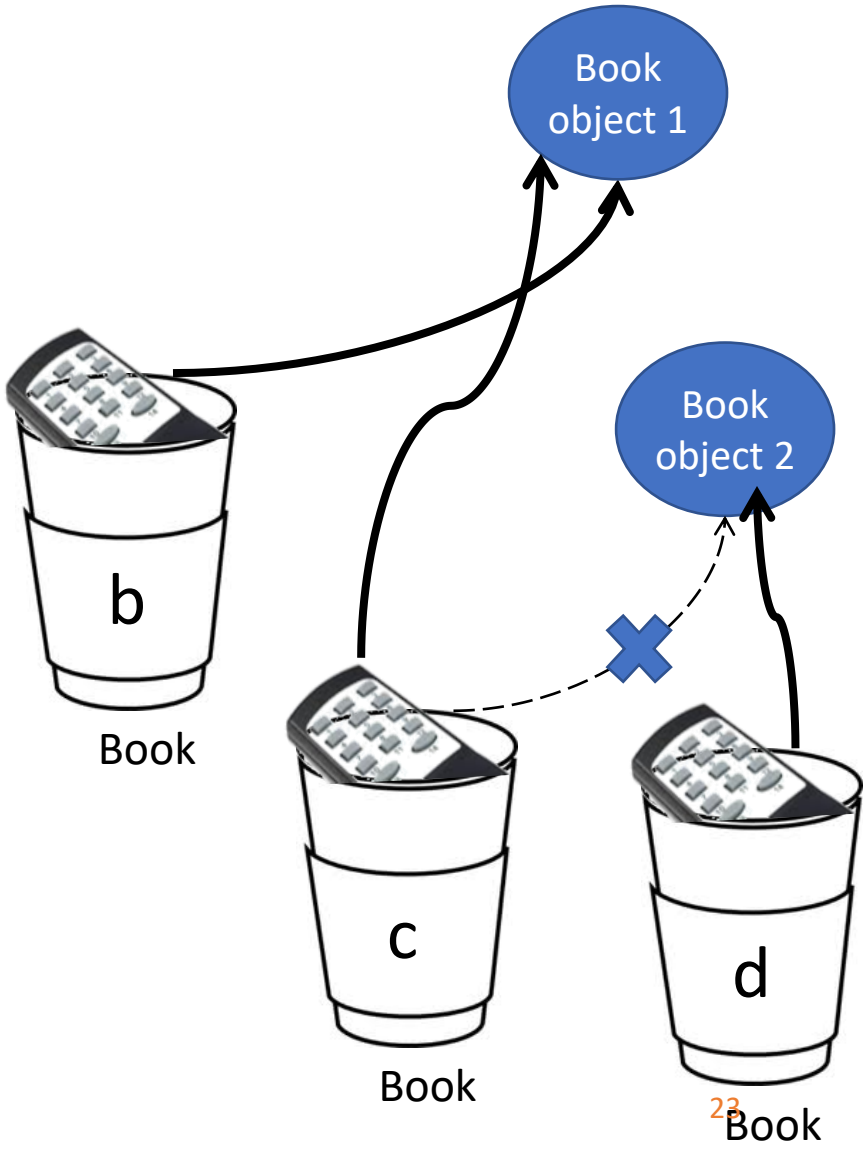


Assigning references III

```
Book b=new Book();
Book c=new Book();
Book d=c;
c=b;
```

References: 3

Objects: 2



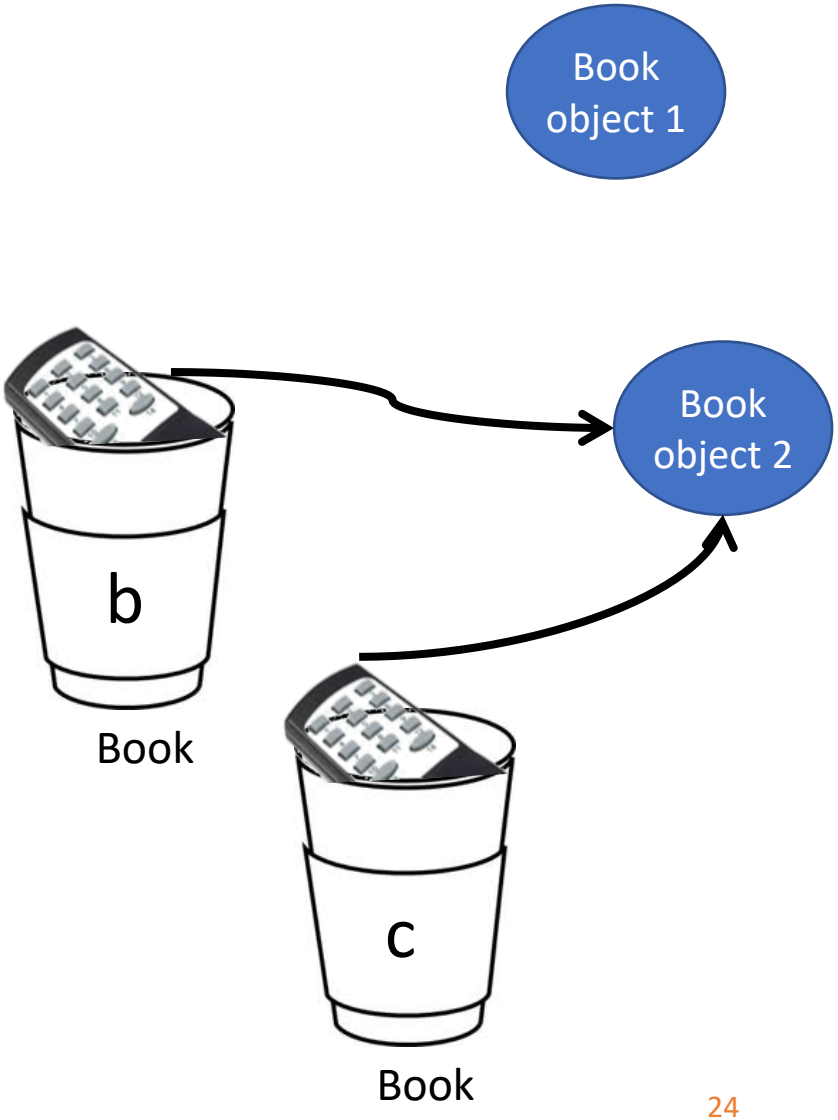
Assigning references IV

```
Book b=new Book();
Book c=new Book();
b=c;
```

References: 2

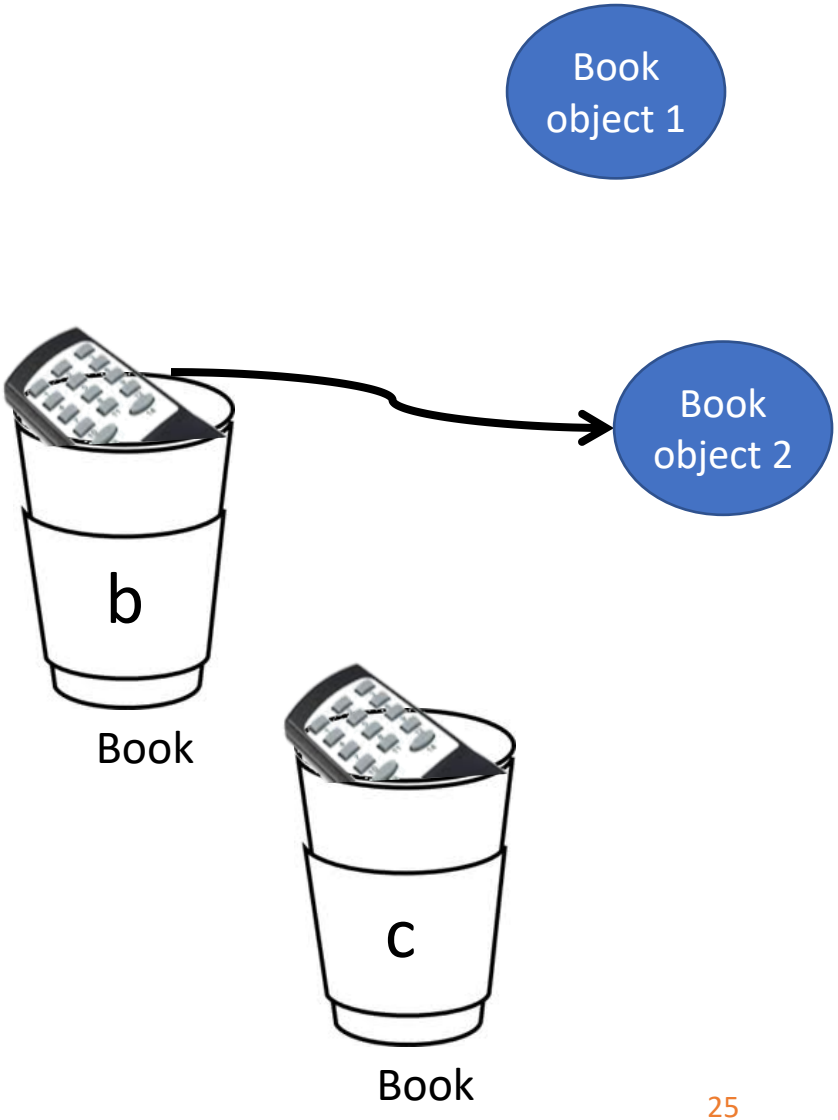
Reachable Objects: 1

Abandoned objects: 1



Assigning references V

```
Book b=new Book();
Book c=new Book();
b=c;
c=null;
```



Active References: 1
Null references: 1
Reachable Objects: 1
Abandoned objects: 1

Recycling abandoned objects

- Compiler manages all the memory used on the Stack during compilation and can automatically clean it
- However, if you create an object on the Heap, the compiler has no knowledge of its lifetime
- Java provides a feature called a **garbage collector** that automatically discovers when an object is no longer in use and destroys it
- The garbage collector provides a higher level of insurance against the insidious problem of *memory leaks*

Manipulating References

- Change reference to refer to another object

```
p1 = p2;
```

- Compare references and see if they *refer to the same object*

```
(p1 == p2)
```

- Cannot perform mathematical operations

```
p1 + p2
```

- Access internal fields or call methods using the dot operator

```
String s = "Hello World!";
System.out.println(s.length);
```

Reference variables gotchas

- If two objects are exactly the same but are located in different memory locations, comparing their references will yield *false*

```
(p1 == p2)
```
✖

- You need to implement a special method *.equals()* to compare objects themselves rather than their location addresses

```
(p1.equals(p2))
```

- Assigning references only copies a memory location and **does not copy** the object

```
p1 = p2;
```
✖

- You would need to implement the *.clone()* method to copy content of an object

```
p1 = p2.clone();
```

What is printed?

```
Dog a=new Dog();
Dog b=new Dog();
Dog c=a;

System.out.println(a==b);
System.out.println(a==c);
System.out.println(b==c);
```

- A
  ```
  true
  true
  true
  ```
- B
  ```
  false
  true
  false
  ```
- C
  ```
  false
  false
  false
  ```
- D
  ```
  true
  true
  false
  ```
- E
  ```
  None of
  the above
  ```

Reference variables as method parameters

- Parameters are still passed by copy: only this time we copy the memory location!
- Thus inside the method we can manipulate the same object through a copy of the reference

```
public class Dog {
   int size;
}
```

```
public class Dogs {
   static void grow(Dog d){
        d.size ++;
   }

   public static void main (String[] args){
        Dog myDog = new Dog();
        myDog.size = 5;
        grow(myDog);
        System.out.println(myDog.size);
   }
}
```

Manipulating the same object through a different reference

Copied myDog reference into a variable d

myDog has size 6

The *String* Class

- String is not a primitive type in Java, it is a *reference type*
- However, Java provides language-level support for Strings literals

```
s = "Bob was here!", t="-11.3", a=""
```

You do not have to use *new* with String

- A single character can be accessed using `charAt()`

  As with arrays, indexing starts at position 0

```
String s = "computer";
char c = s.charAt(5);     // c gets value 't'
c = "oops".charAt(4);     // run-time error!
```

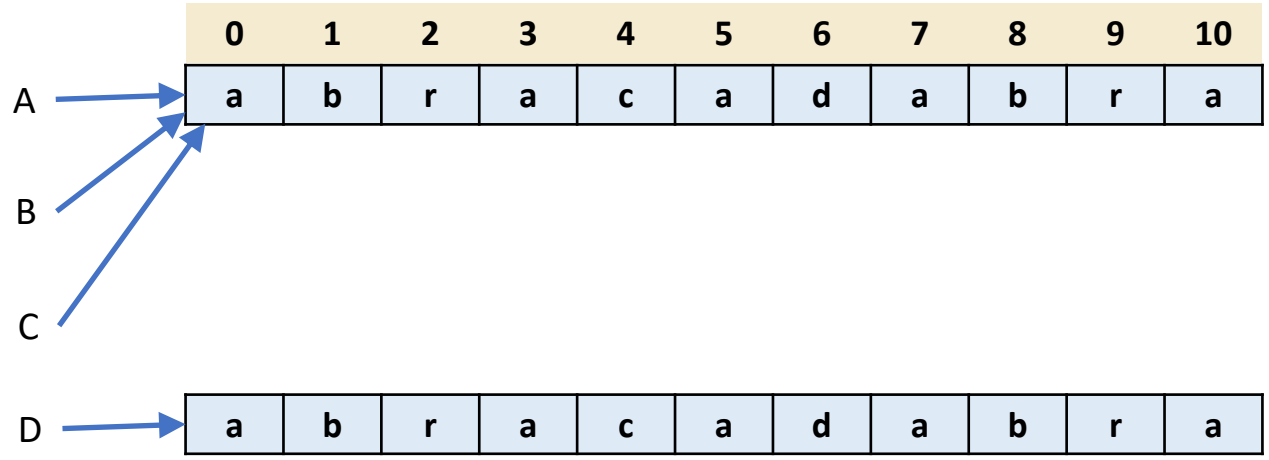
- String provides a length method

```
int len = s.length();     // len gets value 8
len = "".length();        // len gets value 0
```

- **`String` is immutable and the sequence of characters is read-only**

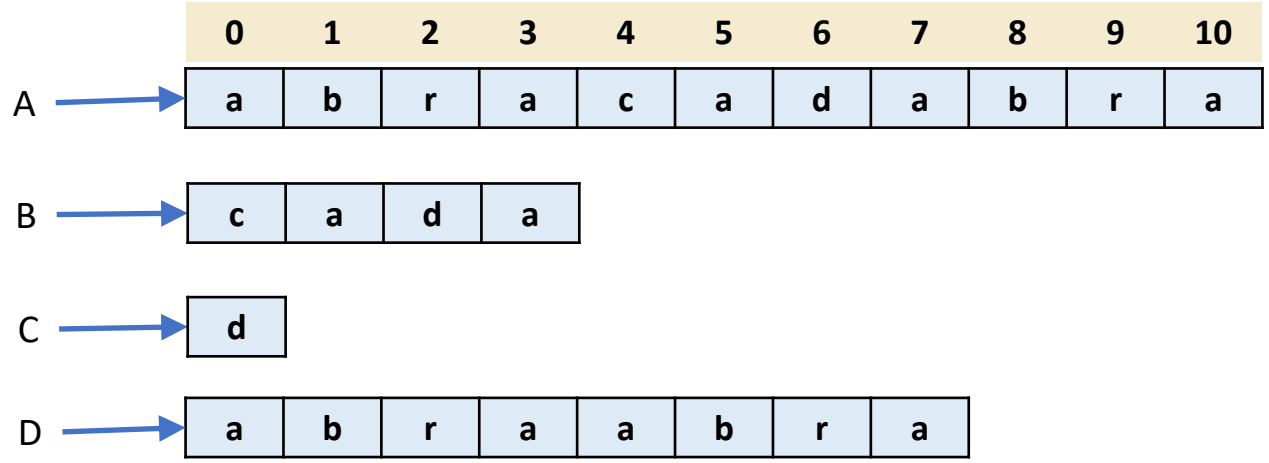
String is a reference type, not a primitive

```
String A = "abracadabra";
String B = A;
String C = "abracadabra";
String D = new String("abracadabra");
```



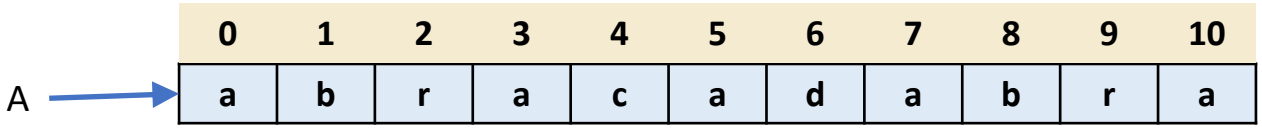
Substring Method

```
String A = "abracadabra";
String B = A.substring(4,8);
String C = A.substring(6,7);
String D = A.substring(0,4) + A.substring(7);
```



IndexOf Method

```
String A = "abracadabra";
int loc = A.indexOf("ra");
// loc = 2
loc = A.indexOf("ra",5);
// loc = 9
loc = A.indexOf("ra", A.indexOf("ra")+1);
// loc = 9
```



String methods in Java

- Useful methods (also check String javadoc page)
  - **`indexOf(string)`** : `int`
  - **`indexOf(string, startIndex)`** : `int`
  - **`substring(fromPos, toPos)`** : `String`
  - **`substring(fromPos)`** : `String`
  - **`charAt(int index)`** : `char`
  - **`equals(other)`** : `bool` ← *`Always use this!`*
  - **`toLowerCase()`** : `String`
  - **`toUpperCase()`** : `String`
  - **`compareTo(string)`** : `int`
  - **`length()`** : `int`
  - **`startsWith(string)`** : `boolean`
- Understand special cases!

Example: Delete substring

Strings are immutable

- No portion of a String can be altered
- To modify a String, copy portions of it

```
public class Slice{

    // method to remove first occurrence of sub from string s
    public static String delete(String s, String sub) {
        int upTo = s.indexOf(sub);  // End of left part of s
        if( upTo == -1) return s;   // s doesn't contain sub

        int thenFrom = upTo + sub.length(); // Start of right part
        return s.substring(0,upTo) + s.substring(thenFrom);
    }
}
```

Scanner Class

- We use `Scanner` class to get input from the console, from a String or from a file
- The `Scanner` class must be imported

```
import java.util.Scanner
```

- `System` class provides an object called `in` that allows low-level input:

  `in` is of type `InputStream`

- `Scanner` class provides higher-level input reading from an `InputStream`

```
Scanner s = new Scanner(System.in);
```

Consuming input with Scanner

- Intuition: `Scanner` provides methods to "consume" the data in an `InputStream`
- `Scanner` methods include
  - `hasNext()` → `boolean` : Is there more input remaining?
  - `nextLine()` → `String` : Consumes and returns the unread contents of current line
  - `next()` → `String` : Consumes and returns next "token" (String surrounded by white space)
  - `nextInt()` → `int` : Consumes and returns (as an int) next token, if token represents an int value
  - also `nextDouble()`, `nextFloat()`, `nextChar()`, …

Example: Scanner

```
import java.util.Scanner;

public class Sum5 {

    public static void main(String[] args) {

        // create a scanner for the terminal input
        Scanner in = new Scanner(System.in);

        int total = 0;   // running sum

        System.out.print("Give me a number (any non-int to end): ");
        while (in.hasNextInt()){
            int n = in.nextInt();
            total += n;
        }

        System.out.println("The total is " + total);
    }
}
```

Reference variables: summary

- Variables must have **name** and **type**
- There are 2 flavors of variables: **primitive** and **reference**
- **Primitive** variable stores the actual value: 5, ‘a’, 3.1415
- **Reference** variable stores an address of an object on the heap
- Through reference variable we can get to an object using dot operator
- Reference variable has value *null* when not referencing any actual object
- Objects that lost connection to the reference variable are disposed by *Garbage Collector*

To do list

- ❑Go over slides, ask for clarifications if needed (Piazza, emails to OWLs or instructors)
- ❑Read the demo code
- ❑Watch the second set of episodes “Python vs. Java”. Pay attention to creating new objects
- ❑Read chapter 8 of “Java for Python programmers”
- ❑Finish Home quiz 2
- ❑LAB 0: due Sunday Sept 18, 10 pm