Reference Guide for Stack Tracing Java

February 6, 2023

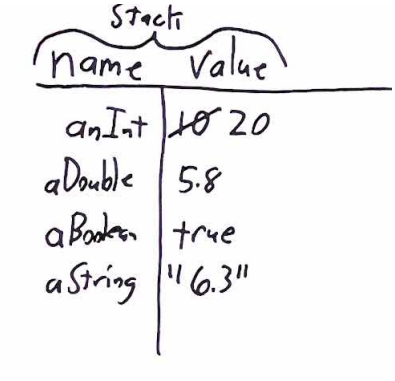
FYI

- Only a single color is required for memory diagrams, different colors are used (and order written into code) for greater clarity in this document

1 Basic variables

```
int anInt = 10;
double aDouble = 5.8;
boolean aBoolean = true;
String aString = "6.3";
anInt=20;
```

- variable changes result in previous values being crossed out and new ones written in so that progression can be seen



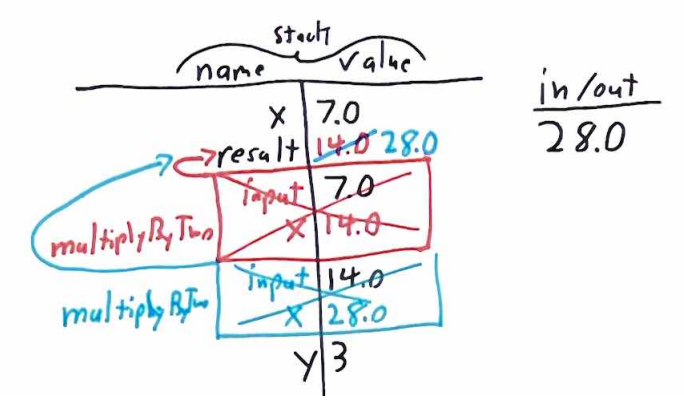
2 Function calls

2.1 Return value

```
public static double multiplyByTwo(double input){
    double x=input*2;
    return x;
}

public static void main(String[] args) {
    double x=7.0;
    double result=multiplyByTwo(x);
    result=multiplyByTwo(result);
    System.out.println(result);
    int y=3;
}
```

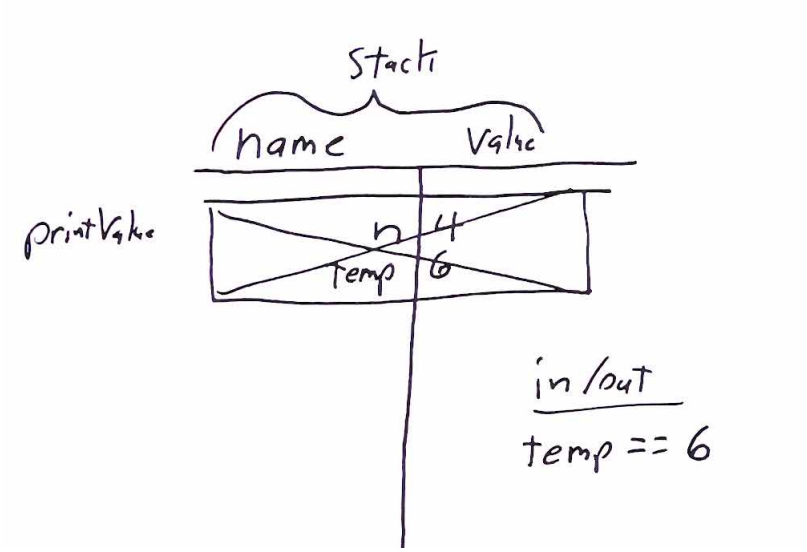
- each function call is put in its own stack frame
- variables created after the function call appear further down the stack
- in/out stands for input/output and is where any command line user input or outputs in terms of print statements appear



2.2 No return value

```
public static void printValue(int n): {
  int temp=n+2
  System.out.println("temp == " + temp)
}

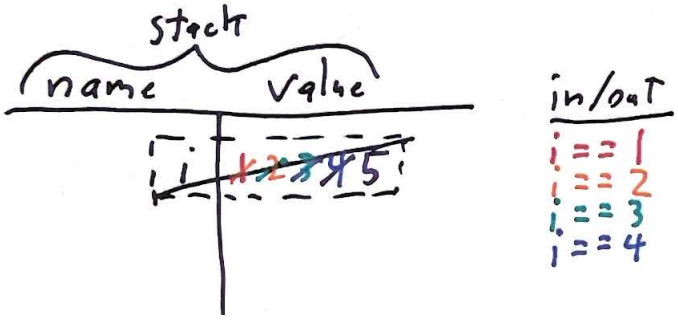
public static void main(String[] args) {
  printValue(4)
}
```



3 For loops

3.1 basic

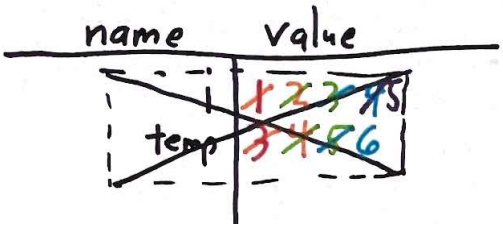
```
for (int i=1;i<5;i++){
  System.out.println("i == "+i);
}
```



3.2 scoped variable

```
for (int i=1;i<5;i++){
  int temp=i+2;
}
```

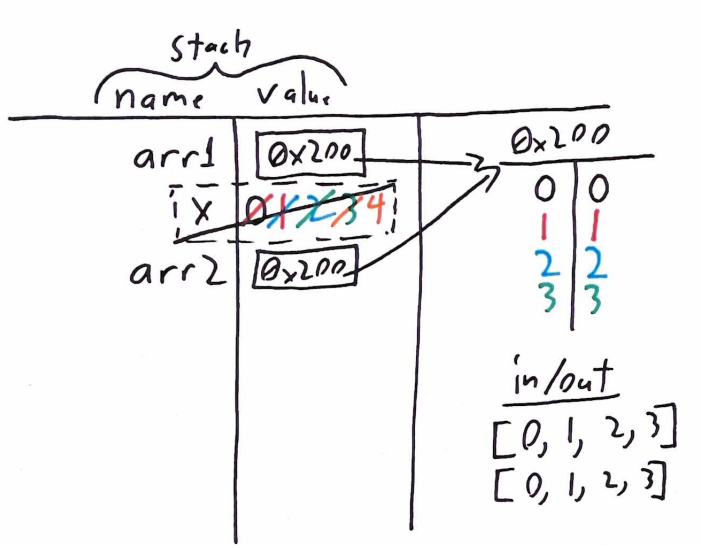
- scoped variables within the loop are crossed out after the loop completes



4 ArrayLists

```
public static void main(String[] args) {
    ArrayList<Integer> arr1=new ArrayList<>();
    for (int x=0;x<4;x++){
        arr1.add(x);
    }
    System.out.println(arr1);
    ArrayList<Integer> arr2=arr1;
    System.out.println(arr2);
}
```

- note that the assignment statement for arr2 assigns the memory address and does not do a deep copy
  - this point is emphasized for newer programmers
- memory addresses all start with 0x to indicate that they are hexadecimal numbers.
  - heap addresses are typically given 3 digit numbers
  - numbers are typically written in decimal as it is easier for students to grasp at first (as they are not familiar with hexadecimal, this detail will be corrected in later courses)
  - numbers are randomly generated and just must agree on the stack and heap

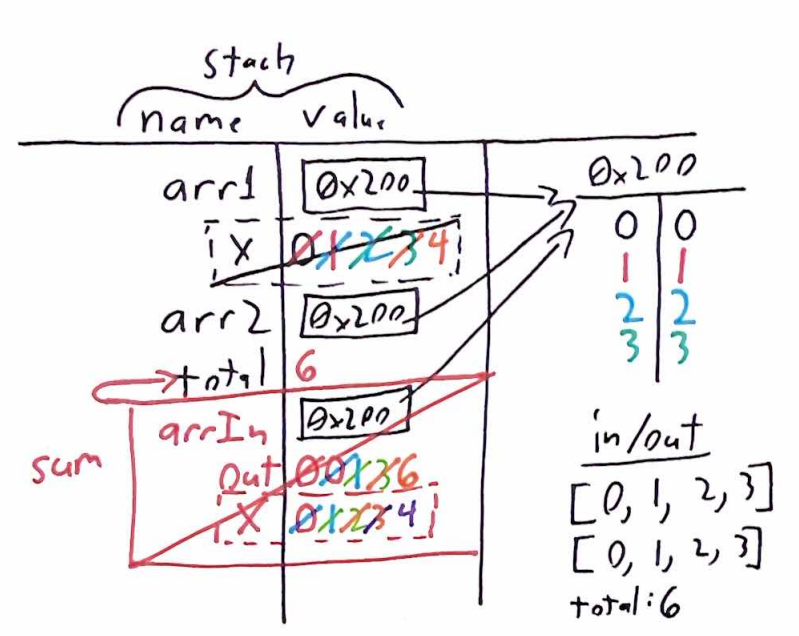


4.1 Passed to functions

```
public static int sum(ArrayList<Integer> arrIn){
    int out=0;
    for (int x=0;x<arrIn.size();x++){
        out+=arrIn.get(x);
    }
    return out;
}

public static void main(String[] args) {
    ArrayList<Integer> arr1=new ArrayList<>();
    for (int x=0;x<4;x++){
        arr1.add(x);
    }
    System.out.println(arr1);
    ArrayList<Integer> arr2=arr1;
    System.out.println(arr2);
    int total=sum(arr1);
    System.out.println("total: "+total);
}
```

- when passing variables to functions as arguments the value on the stack associated with the variable name is the argument to the function that sets the parameter
- the dashed lines around index indicate that it is a scoped variable that only exists while the loop exists

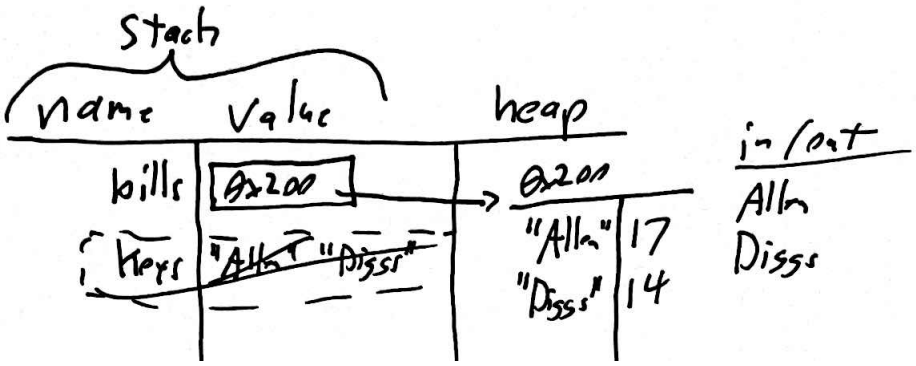


5 HashMaps

- HashMaps are like dictionaries in python
- we loop through them in a manner more similar to loops in python
- HashMap<String,Integer> bills=new HashMap<>();

```
bills.put("Allen",17);
bills.put("Diggs",14);

for (String keys : bills.keySet()){
   System.out.println(keys);
}
```



- note that the loop still has scoped variables like the previous for loop

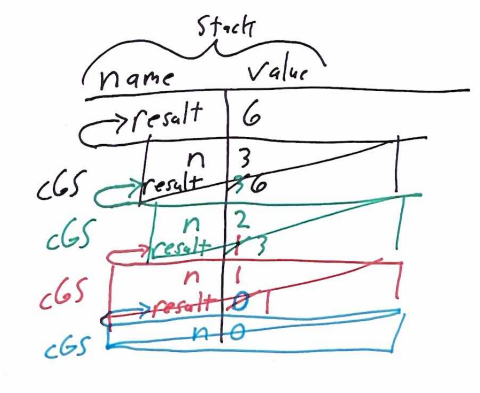
6 Recursion

6.1 standard recursion

```
public static int computeGeometricSum(int n){
    if (n>0){
        int result=computeGeometricSum(n-1);
        result+=n;
        return result;
    } else {
        return 0;
    }
}

public static void main(String[] args) {
    int result=computeGeometricSum(3);
    System.out.println(result);
}
```

- each new call of cGS is performed in a new color
  - returned values are kept in the color of the method that returns it

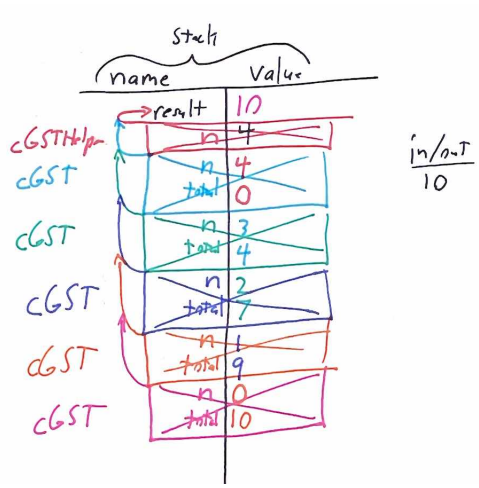


6.2 Tail recursion

```
public static int computeGeometricSumTail(int n,int total){
    if (n>0){
        return computeGeometricSumTail(n-1,total+n);
    } else {
        return total;
    }
}
public static int cGSTHelper(int n){
    return computeGeometricSumTail(n,0);
}

public static void main(String[] args) {
    int result=cGSTHelper(4);
    System.out.println(result);
}
```

- each new call of cGST is performed in a new color
  - returned values are kept in the color of the method that returns it
- Note that the returns go to the previous functions return and not a variable
  - this is why the memory of a stack frame can be released before the following recursive function call finishes

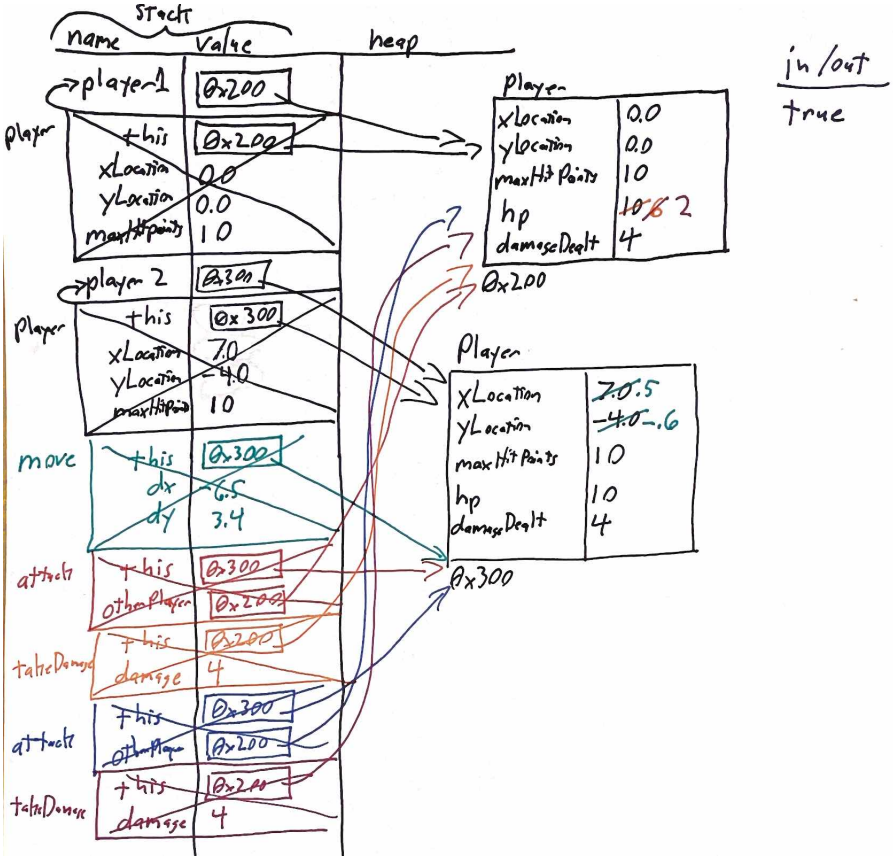


7 Classes

```
public class Player {
    private double xLoc;
    private double yLoc;
    private int maxHP;
    private int HP;
    private int damageDealt;

    public Player(double xLoc,double yLoc,int maxHP){
        this.xLoc=xLoc;
        this.yLoc=yLoc;
        this.maxHP=maxHP;
        this.HP=maxHP;
        this.damageDealt=4;
    }
    public int getHP(){
        return this.HP;
    }
    public void takeDamage(int damage){
        this.HP-=damage;
    }
    public void attack(Player otherPlayer){
        otherPlayer.takeDamage(this.damageDealt);
    }
    public void move(double dx,double dy){
        this.xLoc+=dx;
        this.yLoc+=dy;
    }

    public static void main(String[] args) {
        Player p1=new Player(0.0, 0.0, 10);
        Player p2=new Player(7.0, -4.0, 10);
        p1.move(-6.5, 3.4);
        p2.attack(p1);
        p2.attack(p1);
        System.out.println(p2.getHP==2);
    }
}
```


stack
name
value
heap
in/out
true
player1
0x200
player
this
0x200
xLocation
0.0
yLocation
0.0
maxHitPoints
10
Player
xLocation
0.0
yLocation
0.0
maxHit Points
10
hp
10 6 2
damageDealt
4
0x200
player 2
0x300
player
this
0x300
xLocation
7.0
yLocation
-4.0
maxHitPoints
10
Player
xLocation
7.0 .5
yLocation
-4.0 -.6
max Hit Points
10
hp
10
damageDealt
4
0x300
move
this
0x300
dx
6.5
dy
3.4
attack
this
0x300
otherPlayer
0x200
takeDamage
this
0x200
damage
4
attack
this
0x300
otherPlayer
0x200
takeDamage
this
0x200
damage
4

8 Inheritance

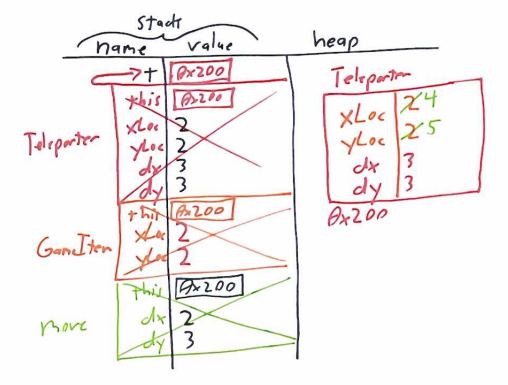
```
public class GameItem {
    private double xLoc;
    private double yLoc;

    public GameItem(double xLoc,double yLoc){
        this.xLoc=xLoc;
        this.yLoc=yLoc;
    }
    public void move(double dx, double dy){
        this.xLoc+=dx;
        this.yLoc+=dy;
    }
}
```

- ```
  public class Teleporter extends GameItem{
      private double dx;
      private double dy;

      public Teleporter(double xloc,double yLoc,double dx,double dy){
          super(xloc,yLoc);
          this.dx=dx;
          this.dy=dy;
      }

      public static void main(String[] args) {
          Teleporter t=new Teleporter(2,2,3,3);
          t.move(2,3);
      }
  }
  ```


  ```

9 Polymorphism

```
public class A {
    protected int a;

    public A(int a){
        this.a=a;
    }
}

public class B extends A{
    private int b;

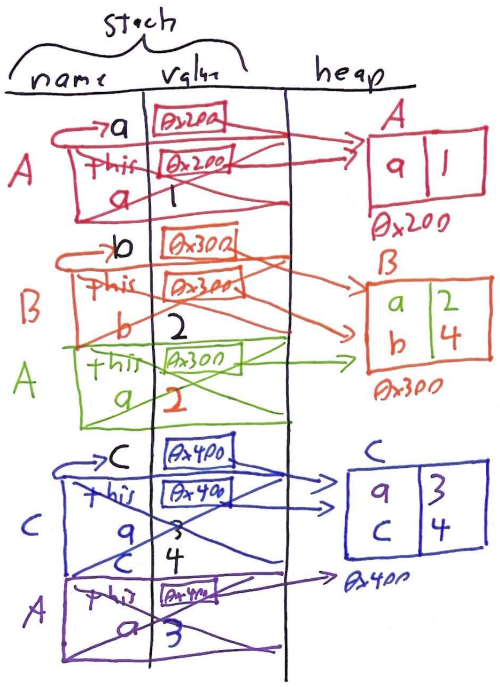
    public B (int b){
        super(b);
        this.b=b*2;
    }
}

public class C extends A{
    private int c;

    public C(int a,int c){
        super(a);
        this.c=c;
    }
}

public class RunABC {
    public static void main(String[] args) {
        A a=new A(1);
        A b=new B(2);
        A c=new C(3,4);
    }
}
```

- Note that polymorphism loooks no different than regular inheritance tracing because we do not include data type in our memory diagram


Stack
name
value
heap
A
a
this
a
1
0x200
B
b
this
b
2
this
a
1
0x300
C
c
this
a
3
c
4
this
a
3
0x400