Problem: Button remembers the object!!

```
public  class  PuppetMaster{
private CartoonCharacter cc1 = new CartoonCharacter(200, 100, "blue");
private CartoonCharacter cc2 = new CartoonCharacter(500, 100, "green");
private CartoonCharacter selectedCC = cc1;

public  PuppetMaster(){
UI.addButton("Jan", this::doJan);
UI.addButton("Smile",  this.selectedCC::smile );
UI.addButton("Frown", this::doFrown);
.
}
public void doJan(){
this.selectedCC = this.cc2;
}
public void doSmile(){
this.selectedCC.smile();
}
public void doFrown(){
```

Doesn't work!!!

The button remembers the object in this.cc1 at the time the button was created!!!!

Shorthand: "Lambda expressions"

```
public class PuppetMaster{
    private CartoonCharacter cc1 = new CartoonCharacter(200, 100, "blue");
    private CartoonCharacter cc2 = new CartoonCharacter(500, 100, "green");
    private CartoonCharacter selectedCC = cc1;

    public PuppetMaster(){
        UI.addButton("Jan", this::doJan);
        UI.addButton("Smile",   () -> { this.selectedCC.smile(); } );
        UI.addButton("Frown", this::doFrown);
            :
    }
    public void doJan(){
        this.selectedCC = this.cc2;
    }
    public void doSmile(){
        this.selectedCC.smile();
    }
    public void doFrown(){
```

Lambda Expression:
Anonymous methods!!
- has parameters
- has body
- but no name

It is a value!!

Shorthand: "Lambda expressions"

```
public class PuppetMaster{
    private CartoonCharacter cc1 = new CartoonCharacter(200, 100, "green");
    private CartoonCharacter cc2 = new CartoonCharacter(500, 100, "blue");
    private CartoonCharacter selectedCC = cc1;
    private double walkDist = 20;

    public PuppetMaster(){
        UI.addButton("Jim",       () -> { this.selectedCC = this.cc1; } );
        UI.addButton("Jan",       () -> { this.selectedCC = this.cc2; } );
        UI.addButton("Smile",     () -> { this.selectedCC.smile(); } );
        UI.addButton("Frown",     () -> { this.selectedCC.frown(); } );
        UI.addButton( "Left",     () -> { this.selectedCC.lookLeft(); } );
        UI.addButton( "Right",    () -> { this.selectedCC.lookRight(); } );
        UI.addTextField("Say",    (String wds) -> { this.selectedCC.speak(wds); } );
        UI.addButton("Walk",      () -> { this.selectedCC.walk(this.walkDist); } );
        UI.addSlider( "Distance", 1, 100, this.walkDist, (double val) -> { this.walkDist = val; } );
    }
}
```

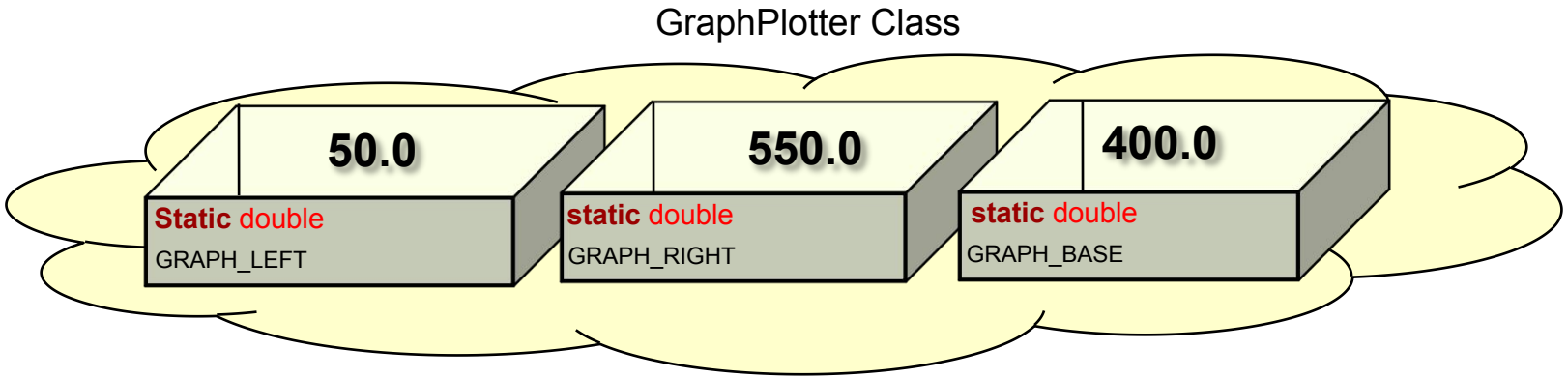
You do NOT HAVE TO USE THESE!! It is always safe to have an explicit, named method.

More about static

static means

"Belongs to class as a whole, Not to individual objects of this class."

```
/** Plot a graph of numbers from a file */
public class GraphPlotter {
    // Constants for plotting the graph
    public static final double GRAPH_LEFT = 50;
    public static final double GRAPH_RIGHT = 550;
    public static final double GRAPH_BASE = 400;
```



More about static

```
/** Plot a graph of numbers from a file */
public class GraphPlotter {
    // Constants for plotting the graph
    public static final double GRAPH_LEFT = 50;
    public static final double GRAPH_RIGHT = 550;
    public static final double GRAPH_BASE = 400;
```

static means

"Belongs to class as a whole, Not to individual objects of this class."

final means

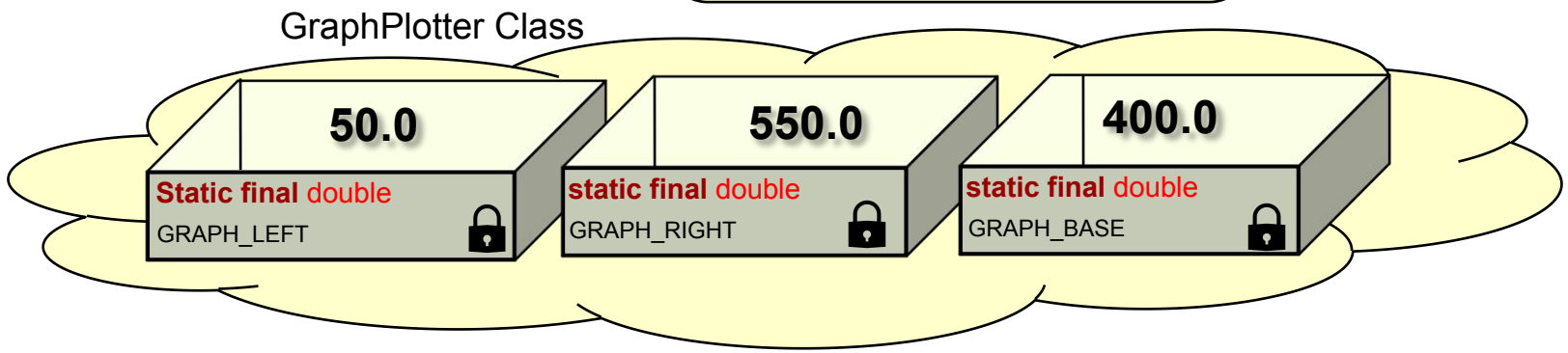
"Can't change the value once it has been set"

public means

"Can access this from code inside other classes"

private means

"Can only access this from code in **this** class"



Static methods:

- Static methods are methods that don't need an object:
  - Methods in the Math class are static methods:

    Math.min(…)

    Math.max(…)

    Math.random()

    Math.sqrt(…)

  - Methods in the UI class are static methods:

    UI.drawRect(…)

    UI.println(…)

    UI.askInt(…)

None of these methods need an object to be created first.

Methods are called on the class itself, not on an object of that class.

Static methods: main

```
import ecs100.*;
import java.util.*;
import java.io.*;

public class GraphPlotter {

    :
    :

    public static void main(String[ ] args){
        GraphPlotter gp = new GraphPlotter();
        gp.setupGUI();
    }
}
```

main method
- static, because belongs to the class, not an object of the class
- called when the program is run directly from Java
- used when running a jar file

Using main

- Normally, you need a main method to be able to run your program.
- Typically, it creates an object of the class, and calls a method on the object.
- It can do more than that.

- Note: BlueJ lets you create an object and call methods on it, using the mouse.
  - simpler methods
  - clearer understanding of objects and methods.
  - good for testing programs
  - ⇒ This course will use a minimal main(..) method.

Numeric data types

We have seen three types of numeric values

- int:
  - integer, with no fractional part (size = 32 bits)
  - eg: 75 -14532
  - range: -2,147,483,648 to 2,147,483,647
    -2^{31} to $2^{31} - 1$ or
    Integer.MIN_VALUE to Integer.MAX_VALUE

- long:
  - integer, but allows a bigger range (size = 64 bits)
  - eg: 7111333555L -123456789123456789L (L to say it is a long, not an int)
  - range: -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
    -2^{63} to $2^{63} - 1$
    Long.MIN_VALUE to Long.MAX_VALUE

Numeric data types

We have seen three types of numeric values

- double:
  - number with a fractional part. (size = 64 bits)
  - eg: 3.4 -193.0 -0.0063 4.8769e23 (= 4.8769×10^{23})
  - range: -2^{1024} to 2^{1024} or roughly -1.8×10^{308} to 1.8×10^{308}
  - precision: (accuracy) 15 decimal digits (precisely, 52 bits)
  - Special values:
    - Double.MAX_VALUE: largest positive finite value 1.797693e+308
    - Double.MIN_VALUE: smallest positive finite value 4.900000e-324
    - Double.NEGATIVE_INFINITY: double value smaller than any other double.
    - Double.POSITIVE_INFINITY: double value larger than any other double.
    - Double.NaN: "not a number": the error value (eg 0.0/0.0).

More numeric data types

We have seen two "wrapper" types of numeric values

- Integer:
  - wrapping up an int as an object so that it can be put into a list (for example)
- Double:
  - wrapping up a double as an object so that it can be put into a list (for example)

There are wrapper types for all the other numeric types.

Java will (in most cases) convert automatically between primitive and wrapper types.

Other numeric types

Integer types:

- byte (8 bits) -128 to 127
- short (16 bits) -32,768 to 32,767
  - Seldom used – just use int normally

Floating point:

- float (32 bits) smaller than doubles, less precision
  - eg 1.0f -0.4f
  - Seldom used, but sometimes needed for colours, eg Color.getHSBColor(0.4f, 1.0f, 1.0f);

Types and Coercion

- Mismatching types:

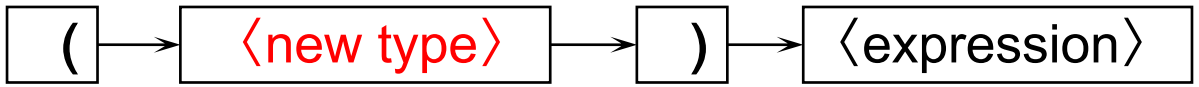
```
double num = scan.nextInt( );
int number = scan.nextDouble( );        ← Can't do this
double squareroot  =  Math.sqrt(25);    ← but sqrt wants double?
String name = "number-" + num;
```

- Java will "coerce" a value to the needed type if it can: eg
  - If a method needs a double and is given an int.
  - If a double variable is assigned an int value.
  - If "adding" any value to a String
  - converting between double and Double or int and Integer (or the other Wrapper Types)
- But only if it does not lose any information:
  - WON'T coerce a double to an int
  - WON'T coerce a String to a number, or vice versa (except when "adding" a number to a String)
  - WON'T coerce any object to a mismatching type (except when printing or "adding" to a String)

Casting

- Where it makes sense to convert a value into another type, but some information may be lost...
- You can *sometimes* "cast" the value to the other type:

```
int number = (int) Math.sqrt(49.5);
float red =  (float) Math.random();
```



- casting a double to an int will lose the fractional part and may mess up the value if the number is too big!
- Not everything can be cast to everything else!
  - ~~Scanner scan = (Scanner) (new PrintStream("data.txt"));~~

Dealing with lots of values

- We've used ArrayLists (and Lists)
  - Road Profiler,
  - WordSearcher, SalesVisualiser, FileEditor,
- ArrayLists of numbers, Strings, other objects.
- Created by methods
  - UI.askNumbers(…) and UI.askStrings(…)
  - Files.readAllLines(Path.of(filename)) (actually, gave us a List, not ArrayList)
- Used **for** each loops to step through items in an ArrayList
- What more can you do with an ArrayList?