Variables

```
/** Print out the conversion formula (version 2) */
    public void printFormula () {
        String formula;                    "                    "
        formula = "Celsius = (Fahrenheit - 32) *5/9" ;
        UI.println( formula );
    }
```

Use a variable whenever you need the computer to remember something temporarily.

- A variable is a place in memory that can hold a value.
  - Must specify the **type** of value that can be put in the variable
    ⇒ "Declare" the variable.
  - Must put a value into a variable before you can use it
    ⇒ "assign" to the variable
  - Can *use* the value by specifying the variable's name
  - Can change the value in a variable
    (unlike mathematical variable)

Asking for a place

Assignment Statements

```
/** Print out the conversion formulas (version 2) */
    public void printFormula () {
        String formula;                    "                    "
        formula = "Celsius = (Fahrenheit - 32) *5/9" ;
        UI.println( formula );
    }
```

Putting a value into a variable

- Assignment Statement:

where = *what* ;

⟨variable⟩ → = → ⟨expression⟩ → ;

name-of-place = *specification-of-value* ;

formula = " Celsius = (Fahrenheit - 32) *5/9"

Meaning: Compute the value and put it in the place

Expressions

```
/** Convert from fahrenheit to Celsius */
public void doFahrenheitToCelsius(){
    double fahrenheit = UI.askDouble("Fahrenheit:" );
    double celsius = (fahrenheit – 32.0) * 5.0 / 9.0;
    UI.println(fahrenheit + "F is " + celsius + " C");
}
```

Version 2: combined into a single method that asks, computes and prints.

+ for Strings: "concatenates" the Strings

- Expressions describe how to compute a value.
- Expressions are constructed from
  - values
  - variables
  - operators (+, -, *, /, etc)
  - method calls that return a value
  - sub-expressions, using (…)
  - …

Method Calls and variables: a metaphor

Method Definition: Like a pad of worksheets

```
public void doFahrenheitToCelsius(){
    double fahrenheit = UI.askDouble("Fahrenheit:");
    double celsius = (fahrenheit – 32.0) * 5.0 / 9.0;
    UI.println(fahrenheit + " is " + celsius + " C");
}
```

Fahrenheit: 86

86.0F is 30.0C

Calling a Method:

86 0

30 0

tempCalc1.fahrenheitToCelsius();

⇒ get a "copy" of the method worksheet
⇒ perform each action in the body.
⇒ throw the worksheet away (losing all the information on it)

Method Definitions: Parameters

```
/** Convert from fahrenheit to centigrade */
    public void printCelsius(double temp){
        double celsius = (temp – 32.0) * 5.0 / 9.0;
        UI.println(temp +  " F -> " + celsius + " C");
    }
```

A parameter specifies
- the type of a value the method needs
- a name for the place where the value will be put when the method is called (a kind of variable, but special)

- Parameters are in defined in the headers of method definitions

Method Calls

Method Definition: Like a pad of worksheets

```
public void printCelsius(double temp){
    double celsius = (temp – 32.0) * 5.0 / 9.0;
    UI.println(temp + " F -> " + celsius + " C");
}
```

Calling a Method:

tempConv1.printCelsius(86);

⇒ get a "copy" of the method worksheet
⇒ copy the argument(s) into the parameters(s)
⇒ perform the actions in the body.

Summary of Java program structure

- A Class specifies a type of object
  - TemperatureCalculator.class describes TemperatureCalculator objects
- A Class contains a constructor
  - Constructor specifies what to do when objects are created
- A Class contains a collection of methods
  - Each method is an action the objects can perform.
  - TemperatureCalculator objects can do celsiusToFahrenheit, fahrenheitToCelsius, printFormula
  - If you have an object, you can call its methods on it.
- A constructor/method definitions contains statements
  - Each statement specifies one step of performing the action
  - Method call statements
  - Declaration and Assignment statements

What can the UI do?

- UI is a predefined object
- Has methods for
  - text input from the user
    eg UI.askString("What is your name?"); UI.askDouble ("How tall are you") ;
  - text output
    eg UI.println(" * " + name + " * ");
  - graphical output
    eg UI.drawRect(100, 100, 300, 150);
  - making buttons, sliders, etc
    - eg UI.addButton("Quit", UI::quit);
- How do you find out about all the methods?
- How do your find out what arguments you need to provide?

Read the Documentation!

- Full documentation for all the standard Java library code
  (the "API" : Application Programming Interface)
- Version of Java API documentation on course web site:
  - "Java Documentation" in side bar
  - http://ecs.victoria.ac.nz/foswiki/pub/Main/JavaResources/javaAPI-102.html
- Tailored for Comp 102
  - Includes documentation of the ecs100 library: (UI, Trace, etc,)
  - puts most useful classes at the top of the list.
- Use the documentation while you are programming!
  - Control-space in Bluej brings up the options plus documentation.

Some UI methods

Text:

UI.clearText()

UI.print(*anything*) UI.println(*anything*) UI.printf(*format-string*, *values*…)

UI.askString(*prompt-string*) UI.askToken(*prompt-string*)

UI.askDouble(*prompt-string*) UI.askInt(*prompt-string*)

UI.askBoolean(*prompt-string*)

Graphics:

Eg: Color.red

UI.clearGraphics() UI.setColor(*color*) UI.setLineWidth(*width*)

UI.drawRect(*left, top, wd, ht*) UI.fillRect(*left, top, wd, ht*)

UI.drawOval(*left, top, wd, ht*) UI.fillOval(*left, top, wd, ht*)

UI.drawLine(x_1, y_1, x_2, y_2)

UI.drawImage(*file, left, top*)

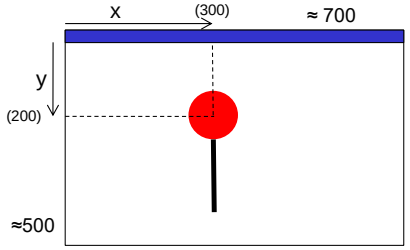
……

Lollipop program

Design:

Method drawLollipop():

- set line width to 10
- draw line
- set line width back to 1
- set color to red
- fill oval



- Must work out the coordinates:
  - line:
  - oval:

Programs with graphics output

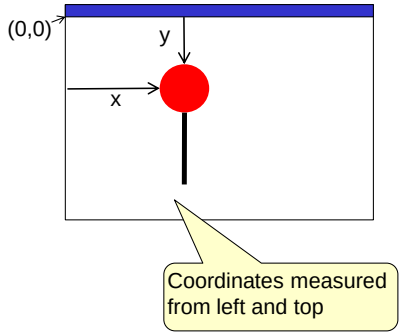
- Write a program to draw a lollipop:

Design

- What shapes can we draw?
  - UI has methods to draw rectangles, ovals, lines, arcs,…

⇒ Draw
one thick black line
one red oval,

Shapes are drawn on top of previous shapes



- How do we draw them?
  Need to set the color first (initially black)
  then call the draw/fill methods:
  - must specify the positions and size
    - rectangles/ovals: left, top, width, height
    - lines: x and y of each end.

Writing the program

- Need import statements
- Need a class (with a descriptive comment)
- Need a constructor
- Need a method (with a descriptive comment)

```
import ecs100.*;
import java.awt.Color;
/** Draws little shapes on the graphics pane */
public class Drawer {
    /** Constructor: Set up interface */
    public Drawer() {
        UI.addButton("Draw it", this::drawLollipop);
    }
    /** Draw an red lollipop with a stick */
    public void drawLollipop() {
        // actions
    }
```

Button that will call the drawLollipop method of this class

Write the method body as comments first

Writing the program: using comments

```
import ecs100.*;
import java.awt.Color ;
/** Draws little pictures on the graphics pane */
public class Drawer {
    public Drawer() {
        UI.addButton("Draw it", this::drawLollipop);
    }
    /** Draw an red lollipop on a stick */
    public void drawLollipop() {
        // set line width to 10
        // draw line        (300,200) to (300, 400)
        // set line width to 1
        // set color to red
        // fill oval        @(260,160)  80x80
    }
}
```

Do it in BlueJ!

Now express each comment in Java (look up documentation as necessary)

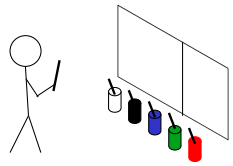
Writing the program

```
import ecs100.*;
import java.awt.Color ;
/** Draws little pictures on the graphics pane */
public class Drawer {
    public Drawer() {
        UI.addButton("Draw it", this::drawLollipop);
    }
    /** Draw a lollipop */
    public void drawLollipop() {
        UI.setLineWidth(10);            // set line width to 10
        UI.drawLine(300, 200, 300, 400);  // draw line
        UI.setLineWidth(1);             // set line width back to 1
        UI.setColor(Color.red);         // set color to red
        UI.fillOval(260, 160, 80, 80);  // draw blob
    }
}
```

Now compile and run the program

A model/metaphor for the computer

- If you are giving instructions to someone/something, it helps to understand how they think and what they can do!
- Your program is run by the "Java Virtual Machine"
- The JVM is like a clerk
  - with Alzheimer's – only remembers what he writes down
  - with a clipboard for the worksheets with the instructions he is currently working on
  - looking at the back of the UI window which he can write/paint on, but the writing/painting only appears on the outside – he can't see what he has written/drawn
  - can see a stream of characters that the user types on the keyboard



Improving the design

- This program is very inflexible:
- What if
  - We want the lollipop to be in a different position?
  - We want the lollipop to be bigger or smaller?
  - We want the stick to be longer?
  - ….
  - We want to draw two of them?
- Current design is filled with literal values

  ⇒ difficult to understand

  ⇒ difficult to change
  (have to find all the places and redo all the arithmetic

Move or resize the Lollipop.

```
import ecs100.*;
import java.awt.Color ;
/** Draws little pictures on the graphics pane */
public class  Drawer {
    /** Constructor:  Set up the interface with one button */
    public  Drawer() {
        UI.addButton("Draw it",  this::doDrawLollipop);
    }
    /** Draw a lollipop */
    public void doDrawLollipop() {
        UI.setColor(Color.black);                  // set color to black
        UI.setLineWidth(10);                       // set line width to 10
        UI.drawLine(300, 200, 300, 400);           // draw line
        UI.setLineWidth(1);                        // set line width back to 1
        UI.setColor(Color.red);                    // set color to red
        UI.fillOval(260, 160, 80, 80);             // draw blob
    }
}
```

Move it left

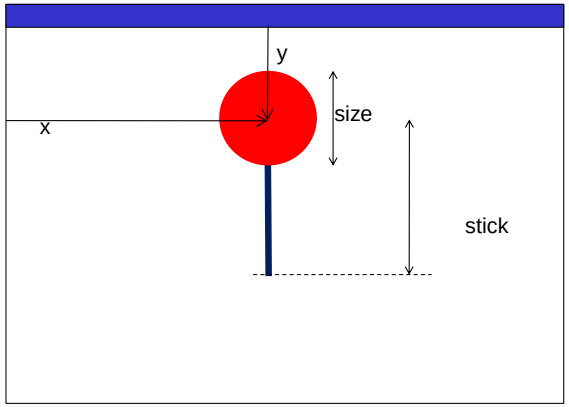
Move it down

Change blob size

Improving the design

- Better design: Use named constants and variables
  - ⇒ easier to write and easier to change
  - ⇒ get the computer to do the arithmetic
- Use named constants for values that won't change while the program is running.

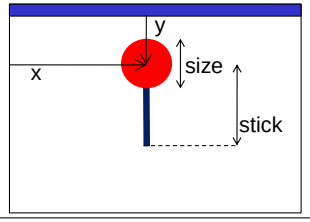
Values to specify lollipop & stick



Improving the program: constants

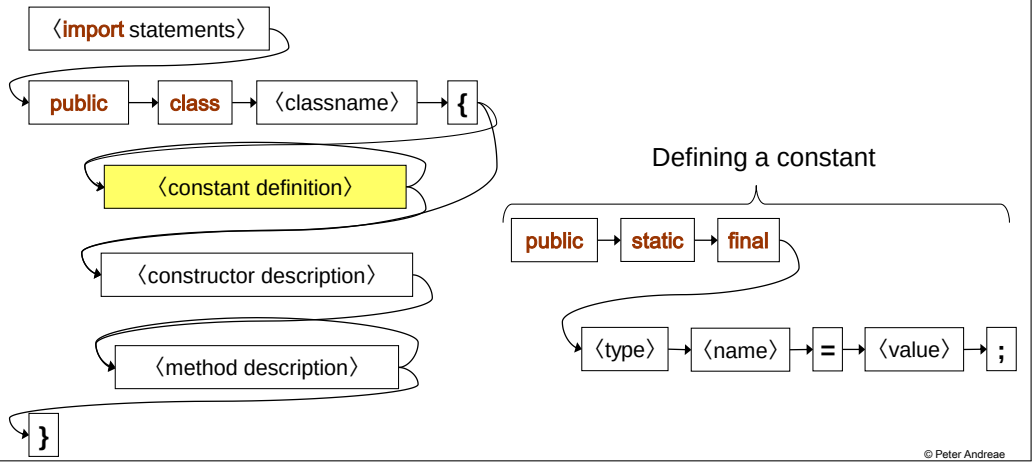
```
import ecs100.*;    import java.awt.Color;
/** Draw a lollipop with a stick */
public class Drawer {
    public static final double  x = 300.0;        // horizontal center of lollipop
    public static final double  y = 180.0;        // vertical center of lollipop
    public static final double  size = 80.0;      // diameter of lollipop
    public static final double  stick =  200.0;   // length of lollipop stick
    public Drawer() {
        UI.addButton("Draw it",  this::doDrawLollipop);
    }
    /** Draw a lollipop */
    public void doDrawLollipop() {
        UI.setLineWidth(size/8.0);
        UI.drawLine(x, y, x, y+stick);
        UI.setLineWidth(1);
        UI.setColor(Color.red);
        UI.fillOval(x-size/2.0,  y-size/2.0,  size, size);
    }
}
```

Easy to change: one place!



Syntax rules: Program structure

- 2nd version

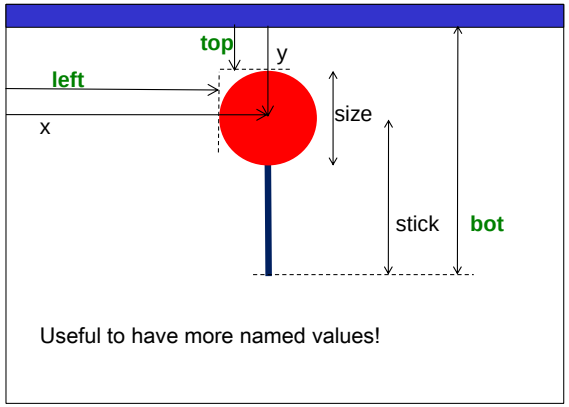


Improving the program: more names

```
    public static final double x = 300.0;        // horizontal center of lollipop
    public static final double y = 180.0;        // vertical center of lollipop
    public static final double size = 80.0;      // diameter of lollipop
    public static final double stick =  200.0;   // length of lollipop stick
    /** Constructor:  Set up the interface with one button */
    public  Drawer() {
        UI.addButton("Draw it",  this::doDrawLollipop);
    }
    /** Draw a lollipop */
    public void doDrawLollipop() {
        UI.setLineWidth(10);
        UI.drawLine(x, y, x, y+stick);
        UI.setLineWidth(1);
        UI.setColor(Color.red);
        UI.fillOval(x-size/2.0, y-size/2.0, size, size);
    }
```

Still have a problem:
What do these expressions mean?

Values to specify lollipop & stick



Improving the program: variables

```
public static final double x = 300.0;          // horizontal center of lollipop
public static final double y = 180.0;          // vertical center of lollipop
public static final double size = 80.0;        // diameter of lollipop
public static final double stick =  200.0;     // length of lollipop stick

public  Drawer() {
    UI.addButton("Draw it",  this::doDrawLollipop);
}
/** Draw a lollipop */
public void doDrawLollipop() {
    double left = x – size/2.0;          // left of lollipop
    double top = y – size/2.0;           // top of lollipop
    double bot =  y + stick;             // bottom of stick
    UI.setLineWidth(10);
    UI.drawLine(x, y, x, bot);
    UI.setLineWidth(1);
    UI.setColor(Color.red);
    UI.fillOval(left, top, size, size);
}
```


Principle of good design

- Use well named constants or variables wherever possible, rather than literal values
  - ⇒ easier to understand
  - ⇒ easier to get right
  - ⇒ much easier to modify
- Choosing the *right* constants or variables is an art!!
  - why did I choose "x" instead of "left" ?
  - why did I choose "y" instead of stick bottom?
- We have effectively *parameterised* the drawing
  - Four values (x, y, size, stick) control the whole thing.

Even better design: parameters

- Every time we want a lollipop of a different size or in a different position, we have to modify the code.
- How come we don't have to do that with drawRect?
- drawRect has four parameters:

Definition of drawRect:

```
public void drawRect(double left, double top, double wd, double ht) {……}
```

Parameters

In the library files

Calling drawRect:

```
UI.drawRect(200, 150, 50, 80);
UI.drawRect(200, 150, 50, 80);
```

Arguments

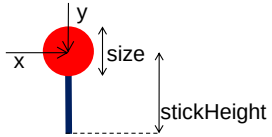
In our program

⇒ drawRect can make many different rectangles.

Why can't we do that with lollipop?

Improving the program: using parameters

```
/** Draw a lollipop at  (300, 180), asking the user for its size */
public void doDrawLollipop() {
    double size = UI.askDouble("Diameter:");
    double stickHeight =  UI.askDouble("Stick height");
    this.drawLollipop(300, 180, size, stickHeight);
}
public void drawLollipop(double x, double y, double size, double stick) {
    double left = x – size/2.0;          // left of lollipop
    double top = y – size/2.0;           // top of lollipop
    double bot =  y + stick;             // bottom of stick
    UI.setLineWidth(10);
    UI.drawLine(x, y, x, bot);
    UI.setLineWidth(1);
    UI.setColor(Color.red);
    UI.fillOval(left, top, size, size);
}
```



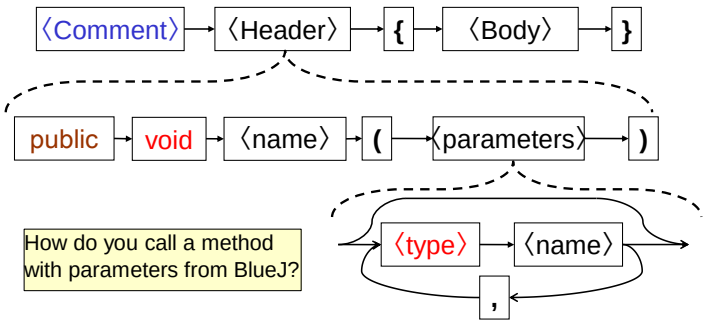
Parameters

Special variables which are given values each time the method is called.

Body of method can use the values in the parameters

Syntax: Method Definitions (v2)

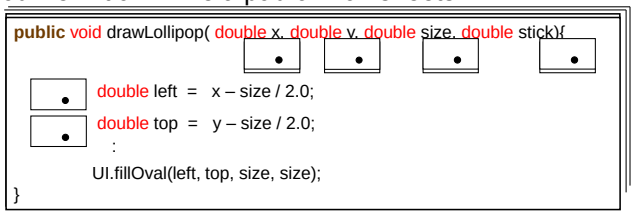
```
/** Draw a lollipop on a stick */
public void drawLollipop(double x, double y, double size, double stick ){
    double left = x – size/ 2.0;
        :
```



How do you call a method with parameters from BlueJ?

Method Calls with parameters

Method Definition: Like a pad of worksheets



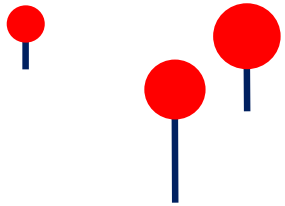
Calling a Method:

this.drawLollipop(300, 100, 75, 95);

⇒ get a "copy" of the method worksheet
⇒ copy the arguments to the parameter places
⇒ perform each action in the body
⇒ throw the worksheet away (losing all the information on it)

Calling drawLollipop

```
public class  Drawer {
    public void doDrawLollipops() {
        double diam = UI.askDouble("diameter:");
        this.drawLollipop(300, 180, diam, 200);
        this.drawLollipop(50, 60, diam/2.0, 90);
        this.drawLollipop(400, 100, diam, 70);
    }
    /** Draw a lollipop */
    public void drawLollipop(double x, double y, double size, double stick) {
        double left = x – size/2.0;          // left of lollipop
        double top = y – size/2.0;           // top of lollipop
        double bot =  y + stick;             // bottom of stick
        UI.setLineWidth(10);
        UI.drawLine(x, y, x, bot);
        UI.setLineWidth(1);
        UI.setColor(Color.red);
        UI.fillOval(left, top, size, size);
    }
```



Principle of good design

- Parameterising a method makes it more flexible and general
  - Allows us to call the same method with different arguments to do the same thing in different ways
  - Allows us to reuse the same bit of code

Arithmetic in Java

- normal arithmetic operators: + – * and /
  - on **doubles**, operators work as expected
    length / 2.5 =>

    length: 10.0

  - on **Strings**, (or a String and another value):
    - turns other value to a String,
    - Concatenates them together
      "Size is " + length + "cm" =>

  - on **integers**,
    - does "primary school" arithmetic
    - / (division) gives a whole number
    - % gives the remainder

    numDoors: 3

    numWalls: 8

    15 / 4 => 15 % 4 => 15 / 30 =>

    numDoors / numWalls => 1.0 * numDoors / numWalls =>