Graphics, classes, objects and program design

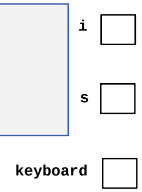
This lecture will

- Introduce the **`EasyGraphics`** class for simple 2D drawing
- Introduce the distinction between class **instance** methods and **static** methods
- Demonstrate the importance of **planning a solution** to a problem on paper before writing a program
- Introduce **top-down design** as a simple technique for developing program solutions
- Review a few simple some **problem-solving strategies**

Objects and Values

- If you declare a variable of a basic type Java creates space to store the variables value
- If you declare a variable which is an object Java creates space to store a pointer to an object but doesn't create space for the object

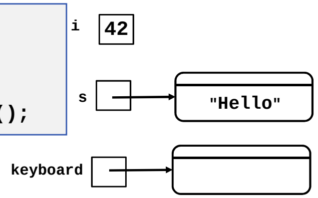
```
int i;
String s;
EasyReader keyboard;
```



Objects and Values

- You can't use variables until you assign values to them
- Generally if the declaration starts with a capital letter it is a reference to an object and anything you assign to it will need to have been created with the keyword **`new`**

```
int i = 42;
String s = "Hello";
EasyReader keyboard =
          new keyboard();
```



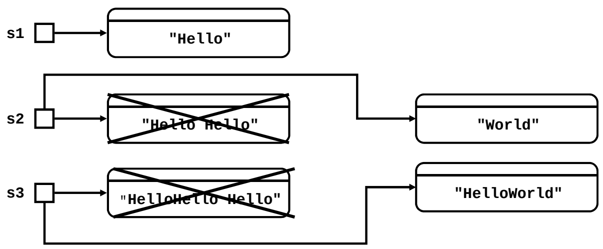
- But not **`String`**s

`String` objects

- Strings are unusual in that they are objects but can be created without **`new`**
- Once a **`String`** is created, its value cannot be changed – we say that a **`String`** object is **immutable**
- A new **`String`** object is created when an assignment is made to an existing string variable
- New memory space is allocated to store the new **`String`**s
- Old memory is reclaimed for future use in a process known as **garbage collection**
- This happens with any object, not just **`String`** objects

Example – garbage collection

```
String s1 = "Hello";
String s2 = "Hello Hello";
String s3 = s1+s2;
s2 = "World";
s3 = s1+s2;
```



The EasyGraphics class

- Part of the **sheffield** package. A simple alternative to classes of the Java API (AWT, Swing, Java2D etc.).
- A **graphics window** encloses a grid of pixels.
- The default (parameterless) constructor for **EasyGraphics** creates a window 200 pixels wide and 200 pixels high:

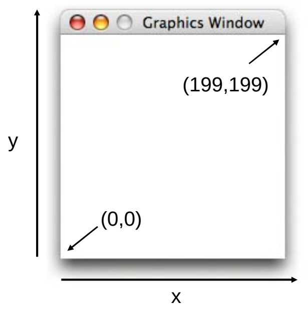
```
EasyGraphics g = new EasyGraphics();
```

- The constructor is **overloaded**; there are several versions with different parameters. We can also create a window with a specific width and height:

```
EasyGraphics g = new EasyGraphics(520,384);
```

An EasyGraphics window

For a 200 × 200 window the coordinate system runs from the origin (0,0) to (199,199)



Colour and plotting

- The colour for graphics operations is set using the **setColor** method, which requires a red, green and blue value between 0 and 255 and can be changed for each operation

  `public void setColor(int r, int g, int b)`

- For example, this sets the colour to bright red:

```
g.setColor(200,0,0);
```

- The **plot** method sets a pixel at coordinates **(x,y)** to the current colour:

  `public void plot (int x, int y)`

- Black (0, 0, 0) is the default colour

Drawing lines

- There is an (invisible) **graphics cursor** at the current pixel, initially set to the origin. We can move it to a new `(x,y)` position (without drawing anything) as follows:

```
g.moveTo(32,59);
```

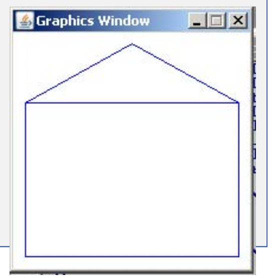
- The `lineTo` method draws a line between the current cursor position and a new point, and moves the cursor to the new position:

```
g.lineTo(45,96);
```

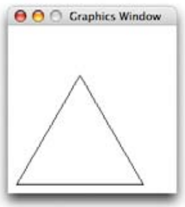
- After this call the graphics cursor is at pixel (45,96).

Example – a blue house

```
import sheffield.*;
public class House {
   public static void main(String[] args) {
      EasyGraphics g = new EasyGraphics(200,200);
      g.setColor(0,0,255);
      g.moveTo(10,140);
      g.lineTo(10,10);
      g.lineTo(190,10);
      g.lineTo(190,140);
      g.lineTo(10,140);
      g.lineTo(100,190);
      g.lineTo(190,140);
   }
}
```



Example – a triangle




```
import sheffield.*;
public class DrawTriangle {
   public static void main(String[] args)
     final double THETA= Math.PI/3.0;
     EasyReader keyboard = new EasyReader();
     int sideLen = keyboard.
          readInt("Enter the side length: ");
     EasyGraphics g = new EasyGraphics();
     g.moveTo(10,10);
     g.lineTo(
         10+(int)Math.round(sideLen*Math.cos(THETA)),
         10+(int)Math.round(sideLen*Math.sin(THETA)));
     g.lineTo(10+sideLen,10);
     g.lineTo(10,10);
   }
}
```

The `Math` class has trigonometric methods that work in radians

Other drawing methods

- The `moveTo` and `lineTo` methods can be combined using the method `drawLine`:

```
public void drawLine(int x0,int y0,int x1,int y1)
```

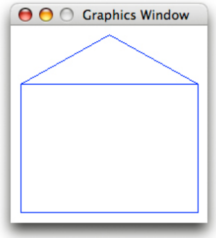
which draws a line from `(x0,y0)` to `(x1,y1)`

- Using `moveTo` and `lineTo` is easier if drawing a number of connected points.
- A method called `clear` is provided which erases the contents of the graphics window:

```
g.clear();
```

Example – house with `drawLine`

```
import sheffield.*;
public class HouseDrawLine {
   public static void main(String[] args) {
     EasyGraphics g = new EasyGraphics(200,200);
     g.setColor(0,0,255);
     g.drawLine(10,140,10,10);
     g.drawLine(10,10,190,10);
     g.drawLine(190,10,190,140)
     g.drawLine(190,140,10,140)
     g.drawLine(10,140,100,190)
     g.drawLine(100,190,190,140
  }
}
```



Drawing shapes

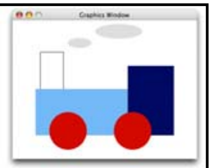
- These methods draw open rectangles or ellipses with the bottom left corner at **(x,y)**, width **w** and height **h**:

```
public void drawRectangle(int x, int y,
                          int w, int h)
public void drawEllipse(int x, int y,
                        int w, int h)
```

- These methods draw filled rectangles or ellipses:

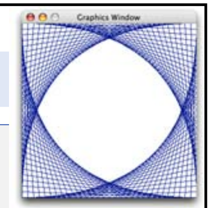
```
public void fillRectangle(int x, int y,
                          int w, int h)
public void fillEllipse(int x, int y,
                        int w, int h)
```

A lovely choo-choo train



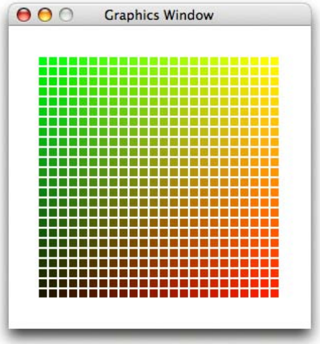
```
import sheffield.*;
public class Train {
   public static void main(String[] args) {
     EasyGraphics g = new EasyGraphics(400,300);
     g.drawRectangle(60,150,50,80); // funnel
     g.setColor(128,180,245);
     g.fillRectangle(50,50,200,100); // boiler
     g.setColor(0,0,100);
     g.fillRectangle(250,50,100,150); // cabin
     g.setColor(200,0,0);
     g.fillEllipse(80,20,80,80); // left wheel
     g.fillEllipse(220,20,80,80); // right wheel
     g.setColor(220,220,220);
     g.fillEllipse(120,240,50,20); // small puff
     g.fillEllipse(180,260,100,30); // big puff
   }
}
```

Making a spiders web



```
import sheffield.*;
public class WebMaker {
      public static void main(String args[]) {
         final int WIN_SIZE = 300; // size of window
         final int STEP_SIZE = 10; // step size between lines
         EasyGraphics g=new EasyGraphics(WIN_SIZE,WIN_SIZE);
         g.setColor(0,0,140);
         for (int x=0; x<=WIN_SIZE; x+=STEP_SIZE) {
            g.drawLine(x,0,WIN_SIZE,x);
            g.drawLine(x,WIN_SIZE,0,x);
            g.drawLine(x,WIN_SIZE,WIN_SIZE,WIN_SIZE-x);
            g.drawLine(0,WIN_SIZE-x,x,0);
         }
      }
}
```

Output of the `Graticule` program



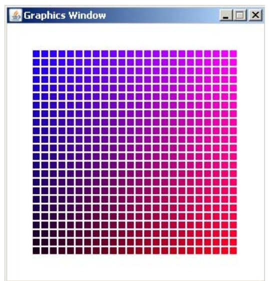
Nested loops – another example

```
// Graticule
final int WIN_SIZE = 300;  // window size
final int MARGIN = 30;     // margin around image
final int INCREMENT = 10;  // increment between squares
final int RECT_SIZE = 8;   // size of square

final int LAST_POSITION = WIN_SIZE-MARGIN-INCREMENT;
EasyGraphics g = new EasyGraphics(WIN_SIZE,WIN_SIZE);

for (int x=MARGIN; x<=LAST_POSITION; x+=INCREMENT)
  for (int y=MARGIN; y<=LAST_POSITION; y+=INCREMENT) {
    // choose some nice colours
    int red = (int)(x*255.0/LAST_POSITION);
    int green = (int)(y*255.0/LAST_POSITION);
    // draw a square at (x,y)
    g.setColor(red,green,0);
    g.fillRectangle(x,y,RECT_SIZE,RECT_SIZE);
  }
```

Output of variations on the Graticule program



Easy graphics methods

Method and Description
`clear()` Clear the EasyGraphics window
`drawEllipse(int x, int y, int w, int h)` Draw an empty ellipse
`drawLine(int x0, int y0, int x1, int y1)` Draw a line from (x0,y0) to (x1,y1)
`drawRectangle(int x, int y, int w, int h)` Draw an empty rectangle
`drawString(java.lang.String s, int x, int y, int fontSize)` Draw a string in the current foreground colour
`fillEllipse(int x, int y, int w, int h)` Draw a filled ellipse
`fillRectangle(int x, int y, int w, int h)` Draw a filled rectangle
`getBlue(int x, int y)` Get the blue RGB component of a given x,y coordinate
`getGreen(int x, int y)` Get the green RGB component of a given x,y coordinate
`getRed(int x, int y)` Get the red RGB component of a given x,y coordinate
`lineTo(int x, int y)` Draw a line from the current position to the given x,y coordinate
`moveTo(int x, int y)` Move the graphics cursor to the given x,y coordinate
`plot(int x, int y)` Plot a point at the given x,y coordinate
`setColor(int r, int g, int b)` Set the current drawing colour

- The list of methods effectively determines what we can do with the EasyGraphics class
- Or any class

Classes, objects and methods

- To use the EasyGraphics class we created an object, an instance of the class, using the word **`new`** and then called methods of the object
- In Java very little programming is done using the basic types **`byte, short, int, long, float, double, char`** and **`boolean`**
- Java is an object oriented language and most programming is done using classes, objects and their methods

Class methods and instance methods

- So far we have mainly used **instance methods**, i.e. methods that are invoked on an object, an instance of a class:

```
String s1 = "Sheffield";
String s2 = s1.replace('f','g');
```

- We can also have methods that belong to the class, rather than to an instance of the class

```
double d = 3.14;
String s = String.valueOf(d);  //s = "3.14"
```

- This is a **class method** or a **static method**

More about class (`static`) methods

- A class method is usually invoked via the class name

```
import javax.swing.JOptionPane;

public class TimesTwo {
 public static void main (String[] args) {

   String number =
      JOptionPane.showInputDialog("What number " +
           "would you like to multiply by two?");

    JOptionPane.showMessageDialog(null,
      "Two times "+number+
           " is "+Integer.valueOf(number)*2);
 }
}
```

How many class methods here?

More about class (`static`) methods

- A class method is usually invoked via the class name

```
String s = "3.14";
double d = Double.valueOf(s);
```

Float, Long, Byte etc. also work

- We don't need an instance of the class in order to use a static method (although static methods *could* be invoked on an instance)
- The **`Math`** class is entirely static and provides lots of useful class methods such as **`Math.abs()`**, **`Math.random()`**, **`Math.round()`**, **`Math.sin()`** and so on

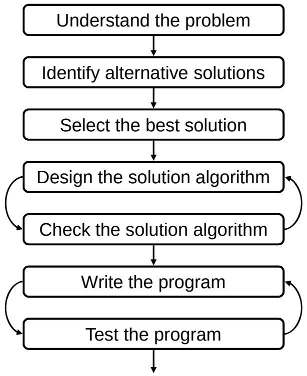
Writing programs

- Up to now we have been writing tiny little programs within the main method
- We are about to go on to writing more complicated programs creating our own classes with static and instance methods of their own and programming using objects of our own classes
- But we will start simply

Simple programs

- We need a disciplined approach to developing programs, in which we identify the problem and plan a solution
- A **software design methodology** is usually followed – guidelines or rules that dictate the steps that should be taken in the software design process
- Later we will look at **object-oriented design**
- Currently we will focus on **top-down design**, a simple approach that is suitable for the design of small software systems
- A key stage in top-down design is writing a description of the steps needed to solve the problem – an **algorithm**

Solving a programming problem

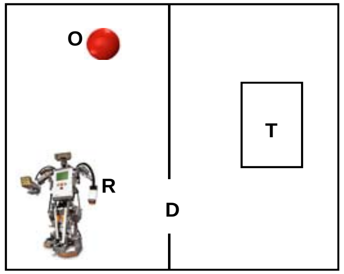


Top-down design

- The top-down approach to problem solving tries to decompose a large problem into subproblems
- This is also called **divide and conquer**
- We first list the steps needed to solve the problem
- Each step is then treated as a subproblem which is solved independently
- In turn, subproblems may give rise to sub-subproblems that must be solved, and so on
- This process of adding detail to a solution algorithm is called **stepwise refinement**

Example – moving a robot

- Starting from position **R**, a robot should collect an object **O** and then put it on the table **T**, moving via a doorway **D**.
- The robot can turn to face any direction, move straight ahead and grasp or release an object.



Developing the algorithm

- We assume that picking up and dropping an object are basic operations of the robot, so long as it is facing in the correct direction.
- The robot can achieve the goal with the following steps:
  1. Move from point **R** to point **O**
  2. Pick up the object at point **O**
  3. Move from point **O** to point **T**
  4. Put the object on the table at point **T**
- Steps 2 and 4 are basic operations. Steps 1 and 3 are subproblems that need to be solved independently.

Stepwise refinement

- Stepwise refinement of subproblem 1, move from point **R** to point **O**:
  - 1.1 Turn to face point **O**
  - 1.2 Move from point **R** to point **O**
- Stepwise refinement of subproblem 3, move from point **O** to point **T**:
  - 3.1 Turn to face the doorway (point **D**)
  - 3.2 Move from point **O** to point **D**
  - 3.3 Turn to face point **T**
  - 3.4 Move from point **D** to point **T**

Moving the robot – complete algorithm

1. Move from point **R** to point **O**
   - 1.1 Turn to face point **O**
   - 1.2 Move from point **R** to point **O**
2. Pick up the object at point **O**
3. Move from point **O** to point **T**
   - 3.1 Turn to face the doorway (point **D**)
   - 3.2 Move from point **O** to point **D**
   - 3.3 Turn to face point **T**
   - 3.4 Move from point **D** to point **T**
4. Put the object on the table at point **T**

When writing a real robot controller this algorithm would have to be refined even further (e.g., to describe how sensor data is used to detect the doorway).

More about writing algorithms

- Algorithm is written in stylized English called **pseudocode**.
- This can't be executed directly by the computer but uses the structural conventions of programming languages, whilst excluding language-specific details.
- There is no 'correct' way to write pseudocode – much variation between individual programmers.
- Use consistent numbering of pseudocode statements for sub-problems:

2 $\xrightarrow{\textit{refine}}$ 2.1, 2.2, 2.3

2.2 $\xrightarrow{\textit{refine}}$ 2.2.1, 2.2.2, 2.2.3

Selection and repetition in pseudocode

- We can also write simple decisions (selections) in pseudocode or repeat a step of the algorithm a number of times (looping), e.g.:

```
if the number is greater than one then
      add the number to the total
end

repeat
      ask the user for a letter
until the letter is 'y' or 'n'
```

Exercise

- Algorithms are not unique to computer programming or robotics – we use algorithms all the time to do specific tasks.

❷ **Write an algorithm for making a cup of tea.**

- You'll need to use selection (e.g., what if the kettle is empty?) and looping (e.g., how much sugar should you add?)

Making a cup of tea

```
1. Boil kettle
2. Make tea
3. Pour tea in cup
4. Add milk and sugar to taste
```

Step 1 refinement:

```
If kettle is empty then
      1.1 fill it with water
end
1.2 switch the kettle on
1.3 wait until the kettle has boiled
```

Step 1.2 refinement:

```
if the kettle is not plugged in then
      1.2.1 plug the kettle into the power socket
end
1.2.2 set the mains switch to on
1.2.3 set the switch on the kettle to on
```

Step 2 refinement

```
2.1 Put the tea bag in the teapot
2.2 Pour boiling water into the teapot
2.3 Wait until brewed
```

Step 3 refinement:

Basic operation, no need to refine

Step 4 refinement:

```
If milk is required then
      4.1 add milk
else
      4.2 add lemon
end
while tea is not sweet enough
      4.3 add a teaspoon of sugar
end
```

Understanding the problem

- An important part of understanding the problem is to identify the **inputs** to the problem, and the **outputs** that are produced.
- These can often be identified as **nouns** in the problem statement.

Example problem statement:

A program is required to prompt the computer user for the maximum and minimum temperature readings on a particular day, accept those readings as integers, and calculate and display the average temperature.

Identifying inputs and outputs

- The nouns are *program*, *computer user*, *maximum and minimum temperature*, *day*, *readings*, *integers*, *average temperature*.
- Some of these can be discounted, suggesting the following inputs and outputs:

  Inputs:
  maximum temperature
  minimum temperature

  Output:
  average temperature

- Also the maximum and minimum temperatures will be integers (the average is real).

Verbs indicate processing steps

- Verbs in the problem description give an indication of the **processing steps** required in the algorithm
- The verbs in our example are *prompt*, *accept*, *calculate*, *display*

Algorithm

1. Prompt for temperatures
2. Get the maximum and minimum temperatures
3. Compute the average temperature
4. Display the average temperature

Stepwise refinement

- Refinement of step 3:

  3.1 Add the maximum and minimum temperatures and divide by two

- Complete algorithm:

  1. Prompt for temperatures
  2. Get the maximum and minimum temperatures
  3. Compute the average temperature
     3.1 Add the maximum and minimum temperatures and divide by two
  4. Display the average temperature

The temperature program

```
import sheffield.*;
public class AverageTemp {
   public static void main(String[] args) {
      EasyReader keyboard = new EasyReader();
      // Get the maximum and minimum temperatures
      int maxTemp = keyboard.readInt("Enter the maximum temperature: ");
      int minTemp = keyboard.readInt("Enter the minimum temperature: ");
      // Compute the average temperature
      double average = (maxTemp+minTemp)/2.0;
      // Display the average temperature
      System.out.println("The average temperature is " + average);
   }
}
```

Exercise

- **If Tom has three times as many apples as Susan and Susan has a quarter as many as Joe, how many does Mary have if Mary has two more than Tom and Joe has 4?**

The apples program

```
public class Apples {
   public static void main(String[] args) {
      int susan, tom, mary, joe=4;
      // susan has a quarter as many apples as joe
      susan=joe/4;
      // tom has three times as many apples as susan
      tom=3*susan;
      // mary has two more apples than tom
      mary=tom+2;
      // display the result
      System.out.println("Mary has "+mary+" apples.");
   }
}
```

```
>java Apples
Mary has 5 apples.
```

Problem solving strategies - analogy

- **Solution by analogy** – easier to adapt an existing solution than to start from scratch.
- Example: find the minimum value in a list of numbers.

  set the minimum value to the first number in the list
  for each successive number
      If the number is less than the minimum then
          set the minimum to that number
      end
  end
  display the minimum value

- Now by analogy, write an algorithm to find the maximum value in a list of numbers.

Problem solving strategies - generalisation

- **Generalising a solution** – try to write general solutions to problems.
- What if we were asked the apples problem but Joe had 9 apples?
- We could – probably *should* – have written a more general version of the program that takes the number of apples as input from the keyboard.
- Many other approaches to problem solving, and it takes practice!

Summary of key points

- The **EasyGraphics** class provides simple commands for drawing
- Classes can have **static** methods that are not associated with any object of the class and **instance** methods that are associated with objects
- Before writing a program, make sure you understand the problem and plan a solution as an **algorithm**, a step-by-step description of what your program is to do
- The algorithm is written in structured English called **pseudocode**
- **Top-down design** (divide and conquer) is a good way to develop algorithms for small programs.