Parameters

Readings: 3.1

Repetitive figures

- Consider the task of drawing the following figures:

```
************

*******

**********************************

**********
*        *
**********

*****
*   *
*   *
*****
```

- The lines and figures are similar, but not exactly the same.

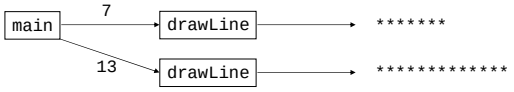
A solution?

```
public class Stars {
    public static void main(String[] args) {
        drawLineOf13Stars();
        drawLineOf7Stars();
        drawLineOf35Stars();
        draw10x3Box();
        draw5x4Box();
    }

    public static void drawLineOf13Stars() {
        for (int i = 1; i <= 13; i++) {
            System.out.print("*");
        }
        System.out.println();
    }

    public static void drawLineOf7Stars() {
        for (int i = 1; i <= 7; i++) {
            System.out.print("*");
        }
        System.out.println();
    }

    public static void drawLineOf35Stars() {
        for (int i = 1; i <= 35; i++) {
            System.out.print("*");
        }
        System.out.println();
    }
    ...
```

- *Observation*: Methods are redundant.
- Would constants help us solve this problem?

Generalizing methods

- What if we had the following?
  - `drawLine` - A method to draw a line of any number of stars.
  - `drawBox` - A method to draw a box of any size.



Parameterization

- **parameterized method**: A method that is given extra information (e.g. number of stars to draw) when it is called.
- **parameter**: A value *passed* to a method by its caller.
- To use a parameterized method:
  - *declare* it
    - Write a method that accepts parameters
  - *call* it
    - Pass the parameter values desired to the method

Declaring parameterized methods

- Parameterized method declaration syntax:

```
public static void <name> (<type> <name>) {
    <statement(s)>;
}
```

- The scope of the parameter is the entire method.
- Example:

```
public static void printSpaces(int count) {
    for (int i = 1; i <= count; i++) {
        System.out.print(" ");
    }
}
```

`count`'s scope

  - Whenever `printSpaces` is called, the caller must specify how many spaces to print.

Calling parameterized methods

- **passing a parameter**: Calling a parameterized method and specifying a value for its parameter(s).
- Parameterized method call syntax:
  <name>`(`***<expression>***`);`
- Example:

```
System.out.print("*");
printSpaces(7);
System.out.print("**");
int x = 3 * 5;
printSpaces(x + 2);
System.out.println("***");
```

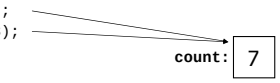
Output:

```
*       **                  ***
```


Passing parameters

- When the parameterized method call executes:
  - the value passed to the method is *copied* into the parameter variable
  - the method's code executes using that value

```
public static void main(String[] args) {
    printSpaces(7);
    printSpaces(13);
}

public static void printSpaces(int count) {
    for (int i = 1; i <= count; i++) {
        System.out.print(" ");
    }
}
```



Value semantics

- **value semantics**: When primitive variables (such as `int` or `double`) are passed as parameters, their values are *copied* into the method's parameter variable.
  - Modifying the method's parameter variable will **NOT** affect the the variable which was passed to the method.

```
public static void main(String[] args)
    int x = 23;
    strange(x);
    System.out.println("2. x = " + x);  // this x unchanged
    ...
}

public static void strange(int x) {
    x = x + 1;     // modifies my x
    System.out.println("1. x = " + x);
}
```

Output:

```
1. x = 24
2. x = 23
```


Errors in coding

- **ERROR**: Not passing a parameter to a method that accepts parameters.

```
printSpaces();   // ERROR: parameter value required
```

- **ERROR**: Passing a parameter of the wrong type.

```
printSpaces(3.7);   // ERROR: must be of type int
```

  - The parameter must satisfy the *domain* of the method.

Parameters: Exercise

- Change the `Stars` program to use parameterized methods.

```
public class Stars {
    public static void main(String[] args) {
        drawLineOf13Stars();
        drawLineOf7Stars();
        drawLineOf35Stars();
        draw10x3Box();
        draw5x4Box();
    }

    public static void drawLineOf13Stars() {
        for (int i = 1; i <= 13; i++) {
            System.out.print("*");
        }
        System.out.println();
    }

    public static void drawLineOf7Stars() {
        for (int i = 1; i <= 7; i++) {
            System.out.print("*");
        }
        System.out.println();
    }
    ...
```


Parameters: Solution

```
// Prints several lines of stars.
// Uses a parameterized method to remove redundancy.

public class Stars2 {
    public static void main(String[] args) {
        drawLine(13);
        drawLine(7);
        drawLine(35);
    }

    // Prints the given number of stars plus a line break.
    public static void drawLine(int count) {
        for (int i = 1; i <= count; i++) {
            System.out.print("*");
        }
        System.out.println();
    }
}
```

Multiple parameters

- Methods can accept as many parameters as you like.
  - When the method is called, it must be passed values for each of its parameters.
- Multiple parameters declaration syntax:

```
public static void <name> (<type> <name>,
        <type> <name>, ..., <type> <name>) {
    <statement(s)>;
}
```

- Multiple parameters call syntax:

```
<name>(<expression>, <expression>, ..., <expression>);
```


Multiple parameters: Example

```
public static void main(String[] args) {
    printNumber(4, 9);
    printNumber(17, 6);
    printNumber(8, 0);
    printNumber(0, 8);
}

public static void printNumber(int number, int count) {
    for (int i = 1; i <= count; i++) {
        System.out.print(number);
    }
    System.out.println();
}
```

Output:

```
444444444
171717171717

00000000
```


Multiple parameters: Exercise

- Write an improved version of the `Stars` program that draws its boxes of stars using parameterized methods.

Multiple parameters: Solution

```
// Prints several lines and boxes made of stars.
// Third version with multiple parameterized methods.

public class Stars3 {
    public static void main(String[] args) {
        drawLine(13);
        drawLine(7);
        drawLine(35);
        System.out.println();
        drawBox(10, 3);
        drawBox(5, 4);
        drawBox(20, 7);
    }

    // Prints the given number of stars plus a line break.
    public static void drawLine(int count) {
        for (int i = 1; i <= count; i++) {
            System.out.print("*");
        }
        System.out.println();
    }
```


Multiple parameters: Solution

```
    // Prints a box of stars of the given size.
    public static void drawBox(int width, int height) {
        drawLine(width);

        for (int i = 1; i <= height - 2; i++) {
            System.out.print("*");
            printSpaces(width - 2);
            System.out.println("*");
        }

        drawLine(width);
    }

    // Prints the given number of spaces.
    public static void printSpaces(int count) {
        for (int i = 1; i <= count; i++) {
            System.out.print(" ");
        }
    }
}
```


Parameter mystery

- What is the output of the following program?

```
public class Mystery {
    public static void main(String[] args) {
        int x = 5, y = 9, z = 2;
        mystery(z, y, x);
        System.out.println(x + " " + y + " " + z);
        mystery(y, x, z);
        System.out.println(x + " " + y + " " + z);
    }

    public static void mystery(int x, int z, int y) {
        x++;
        y = x - z * 2;
        x = z + 1;
        System.out.println(x + " " + y + " " + z);
    }
}
```

x: y: z:

x: z: y:

Exercise

- Rewrite the following program to use parameterized methods:

```
// Draws triangular figures of stars.
public class Loops {
    public static void main(String[] args) {
        for (int i = 1; i <= 5; i++) {
            for (int j = 1; j <= i - 1; j++) {
                System.out.print(" ");
            }
            for (int j = 1; j <= 10 - 2 * i + 1; j++) {
                System.out.print("*");
            }
            System.out.println();
        }

        for (int i = 1; i <= 12; i++) {
            for (int j = 1; j <= i - 1; j++) {
                System.out.print(" ");
            }
            for (int j = 1; j <= 25 - 2 * i; j++) {
                System.out.print("*");
            }
            System.out.println();
        }
    }
}
```


Solution

```
// Draws triangular figures using parameterized methods.
public class Loops {
    public static void main(String[] args) {
        triangle(5);
        triangle(12);
    }

    // Draws a triangle figure of the given size.
    public static void triangle(int height) {
        for (int i = 1; i <= height; i++) {
            printSpaces(i - 1);
            drawLine(2 * height + 1 - 2 * i);
        }
    }
}
```


Exercises

- Write a method named `printDiamond` that accepts a height as a parameter and prints a diamond figure.

```
  *
 ***
*****
 ***
  *
```

- Write a method named `multiplicationTable` that accepts a maximum integer as a parameter and prints a table of multiplication from 1 x 1 up to that integer times itself.
- Write a method named `bottlesOfBeer` that accepts an integer as a parameter and prints the "Bottles of Beer" song with that many verses.
  - http://99-bottles-of-beer.net/lyrics.html

Methods that return values

Readings: 3.2

Return values

- **return**: To send a value out as the result of a method, which can be used in an expression.
- A return value is like the opposite of a parameter.
  - Parameters pass information *in* from the caller to the method.
  - Return values pass information *out* from a method to its caller.
- How would this be useful?

Java's `Math` class

- Java has a class called `Math` that has several useful static methods to perform mathematical calculations.

Method name	Description
abs(*value*)	absolute value
cos(*value*)	cosine, in radians
log(*value*)	logarithm base *e*
log10(*value*)	logarithm base 10
max(*value1*, *value2*)	larger of two values
min(*value1*, *value2*)	smaller of two values
pow(*base*, *exponent*)	*base* to the *exponent* power
random()	random `double` between 0 and 1
round(*value*)	nearest whole number
sqrt(*value*)	square root

Using the `Math` class methods

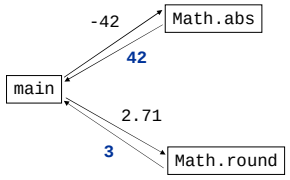
- `Math` method call syntax:
  `Math.`***<method name>***`(`***<parameter(s)>***`)`
- Examples:

```
double squareRoot = Math.sqrt(121.0);
System.out.println(squareRoot);          // 11.0

int absoluteValue = Math.abs(-50);
System.out.println(absoluteValue);       // 50

System.out.println(Math.min(3, 7) + 2);  // 5
```

  - Notice that the preceding calls are used in expressions; they can be printed, stored into a variable, etc…

Return values

- The Math methods do **NOT** print results to the console.
- Instead, each method evaluates to produce (or *return*) a numeric result, which can be used in an expression.



Exercises

- Evaluate the following expressions:
  - `Math.abs(-1.23)`
  - `Math.pow(3, 2)`
  - `Math.pow(10, -2)`
  - `Math.sqrt(121.0) - Math.sqrt(256.0)`
  - `Math.ceil(6.022) + Math.floor(15.9994)`
  - `Math.abs(Math.min(-3, -5))`
- `Math.max` and `Math.min` can be used to bound numbers. Consider an `int` variable named `age`.
  - What statement would replace negative ages with 0?
  - What statement would cap the maximum age to 40?

Writing methods that return values

- Declaring a method that returns a value:

```
public static <type> <name>(<parameters>) {
    <statement(s)>;
}
```

- Returning a value from a method:

```
return <expression>;
```

- Example:

```
// Returns the slope of the line between the given points.
public static double slope(int x1, int y1, int x2, int y2) {
    double dy = y2 - y1;
    double dx = x2 - x1;
    return dy / dx;
}
```

- Question: What return type have we used up until now?

Examples

```
// Converts Fahrenheit to Celsius.
public static double fToC(double degreesF) {
    return 5.0 / 9.0 * (degreesF - 32);
}


// Rounds the given number to the given number of decimal places.
// Example: round(3.14159265, 4) returns 3.1416.
public static double round(double value, int places) {
    double pow = Math.pow(10, places);
    value = value * pow;        // upscale the number
    value = Math.round(value);  // round to nearest whole number
    value = value / pow;        // downscale the number
    return value;
}
```


Errors in coding

- **ERROR**: Writing statements after a `return` statement.

```
public static int increment(int x) {
  return (x + 1);
  x = x + 1;              // ERROR: statement unreachable!
}
```

- **ERROR**: Confusing the return variable with a variable in the calling method, AKA ignoring the return value.

```
public class ReturnExample {
  public static void main(String[] args) {
    int x = 1;
    addOne(x);
    System.out.println("x = " + x);
  }

  public static int addOne(int x) {
    x = x + 1;
    return x;
  }
}
```

Important! Don't ignore the return value!

- Just because the return variable in the called method has the same name as the variable in the calling method, they are **NOT** the same. Think scope!

```
public class ReturnExample {
    public static void main(String[] args) {
        int x = 1;
        addOne(x);
        System.out.println("x = " + x);
    }

    public static int addOne(int x) {
        x = x + 1;
        return x;
    }
}
```

```
public class ReturnExample {
    public static void main(String[] args) {
        int x = 1;
        x = addOne(x);
        System.out.println("x = " + x);
    }

    public static int addOne(int x) {
        x = x + 1;
        return x;
    }
}
```


Exercises

- Write a method named `distanceFromOrigin` that accepts x and y coordinates as parameters and returns the distance between that (x, y) point and the origin.
- Write a method named `attendance` that accepts a number of sections attended by a student, and returns how many points a student receives for attendance. The student receives 4 points for each section up to a maximum of 20 points.

Using objects

Readings: 3.3

Recall: Data types

- **type**: A category of data values.
  - Example: integer, real number, string
- Data types are divided into two classes:
  - **primitive types**: Java's built-in *simple* data types for numbers, text characters, and logic.
    - Example: `int` `double`
  - **object types**: Coming soon!

Object types

- So far, we have seen:
  - **variables**, which represent data (categorized by **types**)
  - **methods**, which represent behavior
- **object:** An entity that contains data and behavior.
  - There are variables inside the object, representing its data.
  - There are methods inside the object, representing its behavior.
- **class:**
  - Basic building block of Java programs (what we have seen so far) or
  - Category or type of object

Class vs. object

- Theoretical examples:
  - A class `Person` could represent objects that store a name, height, weight, hair color, IQ, etc…
  - A class `Laptop` could represent objects that store speed, screen size, color, dimensions, brand, etc…
- Examples from Java:
  - The class `String` represents objects that store text characters.
  - The class `Point` represents objects that store (x, y) data.

`String` objects

Readings: 3.3

The Return of the `String`

- **string**: A sequence of text characters.
  - One of the most common types of objects.
  - Represented as objects of the class `String`.
- `String` variables can be declared and assigned, just like primitive values:

```
String <name> = "<text>";
String <name> = <expression that produces a String>;
```

- Example:

```
String hobbit = "Frodo B.";
String point = "(" + 3 + ", " + 4 + ")";
```


`String` index

- The characters in a `String` are each internally numbered with an *index*, starting with 0:
- Example:

```
String hobbit = "Frodo B.";
```

index	0	1	2	3	4	5	6	7
character	'F'	'r'	'o'	'd'	'o'	' '	'B'	'.'

`String` methods

- Recall that objects are data bundled with methods.

Method name	Description
`charAt(`*index*`)`	returns the character at the given index
`indexOf(`*str*`)`	returns the index where the start of the given string appears in this string (-1 if not found)
`length()`	returns the number of characters in this string
`substring(`*index1*`,`*index2*`)`	returns the characters in this string from *index1* up to, but not including, *index2*
`toLowerCase()`	returns a new string with all lowercase letters
`toUpperCase()`	returns a new string with all uppercase letters

Calling methods on objects

- Since the methods are bundled in the objects, calling these methods requires specifying which object we are talking to.
- Calling a method of an object, general syntax:
  ***<variable>*.*<method name>*(*<parameters>*)**
  - The results may vary from one object to another.
- Examples:

```
String hobbit = "Frodo B.";
System.out.println(hobbit.length());    // 8

String clown = "Homey da Clown";
System.out.println(clown.length());     // 14
```


Madness to the method

- The methods that appear to modify a string (`substring`, `toLowerCase`, `toUpperCase`, etc.) actually create and return a new string.

```
String s = "skee-lo";
s.toUpperCase();
System.out.println(s);   // output: skee-lo
```

vs.

```
String s = "skee-lo";
s = s.toUpperCase();
System.out.println(s);   // output: SKEE-LO
```

`Point` objects

Readings: 3.3

Constructing objects

- **construct**: To create a new object.
  - Objects are *constructed* with the `new` keyword.
- Constructing objects, general syntax:
  `<type> <name> = new <type>(<parameters>);`
- Examples:
```
Point p = new Point(7, -4);
Color orange = new Color(255, 128, 0);
```
- Q: Wait a minute! Why don't we construct strings with `new`?
- A: Strings are one of the most commonly used objects, so they have special syntax (quotation marks) to simplify their construction.

`Point` object: Construction

- Constructing a `Point` object, general syntax:
```
Point <name> = new Point(<x>, <y>);
Point <name> = new Point();  // the origin, (0, 0)
```
- Examples:
```
Point p1 = new Point(5, -2);
Point p2 = new Point();
```


`Point` object

- Data stored in each `Point` object:

Field name	Description
x	the point's x-coordinate
y	the point's y-coordinate

- Useful methods in each `Point` object:

Method name	Description
`distance(p)`	how far away the point is from point *p*
`setLocation(x, y)`	sets the point's x and y to the given values
`translate(dx, dy)`	adjusts the point's x and y by the given amounts

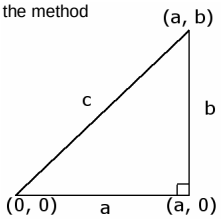
Using `Point` objects: Example

```
import java.awt.*;

public class PointMain {
    public static void main(String[] args) {
        // construct two Point objects
        Point p1 = new Point(7, 2);
        Point p2 = new Point(4, 3);

        // print each point and their distance apart
        System.out.println("p1 is " + p1);
        System.out.println("p2: (" + p2.x + ", " + p2.y + ")");
        System.out.println("distance = " + p1.distance(p2));

        // translate the point to a new location
        p2.translate(1, 7);
        System.out.println("p2: (" + p2.x + ", " + p2.y + ")");
        System.out.println("distance = " + p1.distance(p2));
    }
}
```

To use the `Point` class, you have to *import* it from the `java.awt` *package* in Java. Certain classes like `String` are automatically imported, and thus don't need an `import` statement.

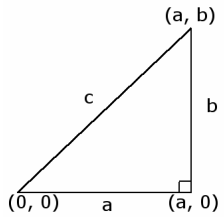
Using `Point` objects: Exercise

- Write a method `computePerimeter` that computes a right triangle's perimeter given two integer side lengths (`a` and `b`).
  - The perimeter is the sum of the triangle's side lengths `a+b+c`.
- Example: Given side lengths of `12` and `5`, the method should return `30.0`.



Using `Point` objects: Solution

```
public static double computePerimeter(int a, int b) {
   Point p1 = new Point();       // 0, 0
   Point p2 = new Point(a, b);
   double c = p1.distance(p2);
   return a + b + c;
}
```



Value vs. reference semantics

Readings: 3.3

Program mystery

- What does this code do?

```
public static void main(String[] args) {
    int a = 7;
    int b = 35;
    System.out.println(a + " " + b);

    int temp = a;
    a = b;
    b = temp;

    System.out.println(a + " " + b);
}
```


Swapping values

- Swapping is a common operation, so we might want to make it into a method.

```
public static void main(String[] args) {
    int a = 7;
    int b = 35;
    System.out.println(a + " " + b);

    // swap a with b
    swap(a, b);

    System.out.println(a + " " + b);
}

public static void swap(int a, int b) {
    int temp = a;
    a = b;
    b = temp;
}
```

- Does this work? Why or why not?

Recall: Value semantics

- **value semantics**: Behavior where variables are copied when assigned to each other or passed as parameters.
  - Primitive types in Java use value semantics.
  - Modifying the value of one variable does not affect other.

- Example:

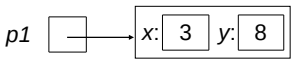
```
int x = 5;
int y = x;      // x = 5, y = 5
y = 17;         // x = 5, y = 17
x = 8;          // x = 8, y = 17
```


Reference semantics

- **reference semantics**: Behavior where variables refer to a common value when assigned to each other or passed as parameters.
  - Object types in Java use reference semantics.
  - Object variables do not store an object; they store the *address* of an object's location in the computer memory. We graphically represent addresses as arrows.

- Example:

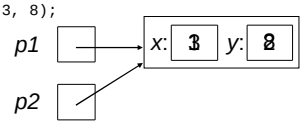
```
Point p1 = new Point(3, 8);
```



Reference semantics

- If two object variables are assigned the same object, the object is **NOT** copied; instead, the object's address is copied.
  - As a result, both variables will *point* to the same object.
  - Calling a method on either variable will modify the same object.
- Example:

```
Point p1 = new Point(3, 8);
Point p2 = p1;
p2.setLocation(1, 2);
```



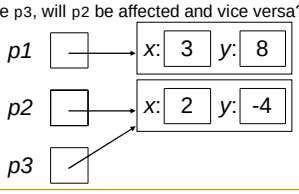
Reference semantics: Why?

- Objects have reference semantics for several reasons:
  - *efficiency*: Copying large objects would slow down the program.
  - *sharing*: Since objects hold important state, it's useful to share an object's data between methods.

Reference semantics: Example

```
Point p1 = new Point(3, 8);
Point p2 = new Point(2, -4);
Point p3 = p2;
```

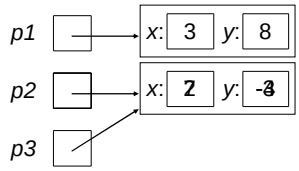
- How many unique objects are there? How do you know that?
  - Two, because (non-String) objects are only created with `new`.
- If we change `p3`, will `p2` be affected and vice versa?
  - Yes.



Reference semantics: Example

- If two variables refer to the same object, modifying one of them *will* also make a change in the other:

```
p3.translate(5, 1);
System.out.println("(" + p2.x + " " + p2.y + ")");
```



Output:

```
(7, -3)
```

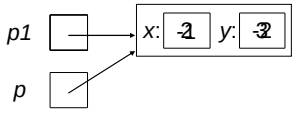

Objects as parameters

- When an object is passed as a parameter, the object is *not* copied. The same object is referred to by both the original variable and the method's parameter.
  - If a method is called on the parameter, it *will* affect the original object that was passed to the method.

Objects as parameters: Example

- Example:

```
public static void main(String[] args) {
    Point p1 = new Point(2, 3);
    move(p1);
}

public static void move(Point p) {
    p.setLocation(-1, -2);
}
```

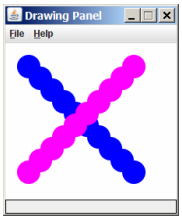


Introduction to Graphics

Readings: 3G.1 – 3G.3

Graphical objects

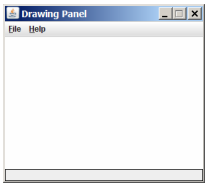
- To draw pictures, we will use three classes of objects:
  - `DrawingPanel`: A window on the screen.
  - `Graphics`: A "pen" that can draw shapes and lines onto a window.
  - `Color`: The colors the "pen" uses to draw shapes.



`DrawingPanel`

- To create a window, construct a `DrawingPanel` object:
  `DrawingPanel` ***<name>*** `= new DrawingPanel(`***<width>***`,` ***<height>***`);`
- Example:

```
DrawingPanel panel = new DrawingPanel(300, 200);
```



`Graphics`

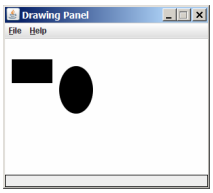
- Shapes are drawn using an object of class `Graphics`.
  - Just like the `Point` class, you must place an *import declaration* in your program: `import java.awt.*;`
  - Access it by calling the `getGraphics` method on your `DrawingPanel`.
- Example:

```
Graphics g = panel.getGraphics();
```


Using the `Graphics` object

- Once you have the `Graphics` object, draw shapes by calling its methods.
- Example:

```
g.fillRect(10, 30, 60, 35);
g.fillOval(80, 40, 50, 70);
```

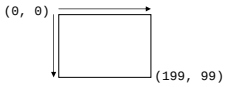


`Graphics` methods

Method name	Description
`drawLine(`*x1*`,` *y1*`,` *x2*`,` *y2*`)`	draws a line between points (*x1*, *y1*), (*x2*, *y2*)
`drawOval(`*x*`,` *y*`,` *width*`,` *height*`)`	draws outline of largest oval that fits in a box of size *width* * *height* with top-left corner at (*x*, *y*)
`drawRect(`*x*`,` *y*`,` *width*`,` *height*`)`	draws outline of rectangle of size *width* * *height* with top-left corner at (*x*, *y*)
`drawString(`*text*`,` *x*`,` *y*`)`	writes text with bottom-left corner at (x, y)
`fillOval(`*x*`,` *y*`,` *width*`,` *height*`)`	fills largest oval that fits in a box of size *width* * *height* with top-left corner at (*x*, *y*)
`fillRect(`*x*`,` *y*`,` *width*`,` *height*`)`	fills rectangle of size *width* * *height* with top-left corner at (*x*, *y*)
`setColor(`*Color*`)`	sets `Graphics` to paint subsequent shapes in the given color

The coordinate system

- Each (x, y) position on the `DrawingPanel` is represented by a *pixel* (picture element).
- The origin (0, 0) is at the window's top-left corner.
  - x increases rightward and y increases *downward*
- Example:

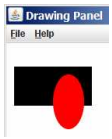


- The `DrawingPanel` shows the coordinates where your mouse pointer is located on the bottom of the panel.

`Color`

- Colors are specified by constants in the `Color` class named: `BLACK`, `BLUE`, `CYAN`, `DARK_GRAY`, `GRAY`, `GREEN`, `LIGHT_GRAY`, `MAGENTA`, `ORANGE`, `PINK`, `RED`, `WHITE`, and `YELLOW`
  - Pass these to the `Graphics` object's `setColor` method.
- Example:

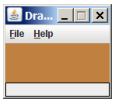
```
g.setColor(Color.BLACK);
g.fillRect(10, 30, 100, 50);
g.setColor(Color.RED);
g.fillOval(60, 40, 40, 70);
```



Making your own colors

- Colors are defined by three numbers (`ints` from 0 to 255) representing the amount of red, green, and blue (RGB).
  - More colors can be found here: http://www.pitt.edu/~nisg/cis/web/cgi/rgb.html
- Example:

```
DrawingPanel panel = new DrawingPanel(80, 50);
Color brown = new Color(192, 128, 64);
panel.setBackground(brown);
```



Background color

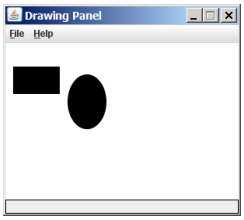
- The background color can be set by calling `setBackground` on the `DrawingPanel`.
- Example:

```
panel.setBackground(Color.YELLOW);
```

- NB: The `setBackground` method is called on the `DrawingPanel`, while `setColor` is called on the `Graphics` object.

A complete program

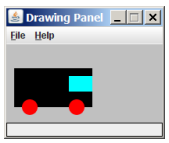
```
import java.awt.*;

public class DrawingExample1 {
    public static void main(String[] args) {
        DrawingPanel panel = new DrawingPanel(300, 200);
        Graphics g = panel.getGraphics();
        g.fillRect(10, 30, 60, 35);
        g.fillOval(80, 40, 50, 70);
    }
}
```



Order matters!

- Shapes drawn later will appear on top of previous ones.

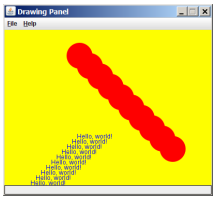
```
import java.awt.*;

public class DrawCar {
    public static void main(String[] args) {
        DrawingPanel panel = new DrawingPanel(200, 100);
        panel.setBackground(Color.LIGHT_GRAY);
        Graphics g = panel.getGraphics();

        // chassis
        g.setColor(Color.BLACK);
        g.fillRect(10, 30, 100, 50);

        // wheels
        g.setColor(Color.RED);
        g.fillOval(20, 70, 20, 20);
        g.fillOval(80, 70, 20, 20);

        // windshield
        g.setColor(Color.CYAN);
        g.fillRect(80, 40, 30, 20);
    }
}
```



Drawing with loops

- We can draw the same item at different x/y positions with `for` loops.
  - The x or y expression contains the loop counter, `i`, so that in each pass of the loop, when `i` changes, so does x or y.

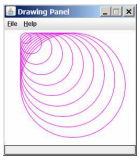
```
DrawingPanel panel = new DrawingPanel(400, 300);
panel.setBackground(Color.YELLOW);
Graphics g = panel.getGraphics();

g.setColor(Color.RED);
for (int i = 1; i <= 10; i++) {
    g.fillOval(100 + 20 * i,
               5 + 20 * i, 50, 50);
}

g.setColor(Color.BLUE);
for (int i = 1; i <= 10; i++) {
    g.drawString("Hello, world!",
         150 - 10 * i, 200 + 10 * i);
}
```



Drawing with loops

- A `for` loop can also vary a shape's size:

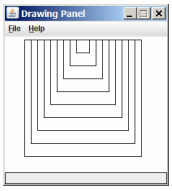
```
import java.awt.*;

public class DrawCircles {
    public static void main(String[] args) {
        DrawingPanel panel = new DrawingPanel(250, 220);
        Graphics g = panel.getGraphics();
        g.setColor(Color.MAGENTA);
        for (int i = 1; i <= 10; i++) {
            g.drawOval(30, 5, 20 * i, 20 * i);
        }
    }
}
```



Drawing with loops

- The loop in this program affects both the size and shape of the figures being drawn.

```
DrawingPanel panel = new DrawingPanel(250, 200);
Graphics g = panel.getGraphics();
for (int i = 1; i <= 10; i++) {
    g.drawRect(20 + 10 * i, 5,
       200 - 20 * i, 200 - 20 * i);
}
```

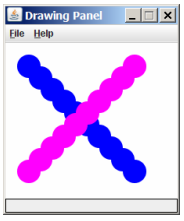


- Each pass of the loop, the square drawn becomes 20 pixels smaller in size, and shifts 10 pixels to the right.

Exercise

- What does the following code draw?

```
import java.awt.*;

public class DrawingExample2 {
   public static final int NUM_CIRCLES = 10;

   public static void main(String[] args) {
      DrawingPanel panel = new DrawingPanel(220, 200);
      Graphics g = panel.getGraphics();

      g.setColor(Color.BLUE);
      for (int i = 1; i <= NUM_CIRCLES; i++) {
         g.fillOval(15 * i, 15 * i, 30, 30);
      }

      g.setColor(Color.MAGENTA);
      for (int i = 1; i <= NUM_CIRCLES; i++) {
         g.fillOval(15 * (NUM_CIRCLES
              + 1 - i), 15 * i, 30, 30);
      }
   }
}
```



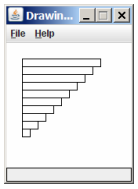
Counting from 0

- Often, it is useful to start counting from `0` instead of `1`.
  - The loop test must be changed to < from <=.
- A loop that repeats from `0` to < `10` still repeats 10 times, just like a loop that repeats from `1` to <= `10`.
- When the loop counter variable `i` is used to set the figure's coordinates, starting `i` at `0` will give us the coordinates we want.

Counting from 0: Example

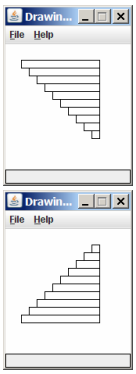
- Draw ten stacked rectangles starting at (20, 20), height 10, with widths that start at 100 and decrease by 10 each time:

```
DrawingPanel panel = new DrawingPanel(160, 160);
Graphics g = panel.getGraphics();

for (int i = 0; i < 10; i++) {
    g.drawRect(20, 20 + 10 * i,
               100 - 10 * i, 10);
}
```



Exercise

- Write variations of the preceding program that draw the following figures.



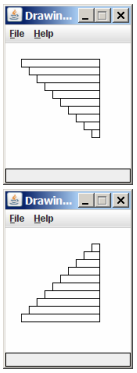
Solution

- Solution #1:

```
Graphics g = panel.getGraphics();

for (int i = 0; i < 10; i++) {
    g.drawRect(20 + 10 * i, 20 + 10 * i,
               100 - 10 * i, 10);
}
```

- Solution #2:

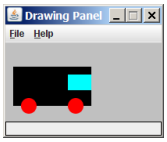
```
Graphics g = panel.getGraphics();

for (int i = 0; i < 10; i++) {
    g.drawRect(110 - 10 * i, 20 + 10 * i,
               10 + 10 * i, 10);
}
```



Structuring code

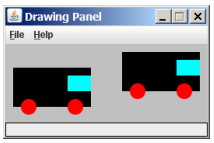
- Use static methods to structure the code.
  - Since you'll need to send commands to the "pen" to draw the figure, you should pass `Graphics g` as a parameter.

```
import java.awt.*;

public class DrawCar {
    public static void main(String[] args) {
        DrawingPanel panel = new DrawingPanel(200, 100);
        panel.setBackground(Color.LIGHT_GRAY);
        Graphics g = panel.getGraphics();
        drawCar(g);
    }

    public static void drawCar(Graphics g) {
        g.setColor(Color.BLACK);
        g.fillRect(10, 30, 100, 50);

        g.setColor(Color.RED);
        g.fillOval(20, 70, 20, 20);
        g.fillOval(80, 70, 20, 20);

        g.setColor(Color.CYAN);
        g.fillRect(80, 40, 30, 20);
    }
}
```



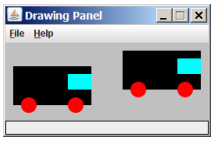
Parameterized figures: Translation

- If you want to draw the same figure many times, write a method to draw that figure and accept the x/y position as parameters.
  - Adjust the x/y coordinates of your drawing commands to take into account the parameters.



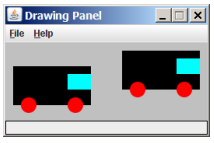
Translating figures: Exercise

- Modify the previous car-drawing method to work at any location, so that it can produce an image such as the following:
  - One car's top-left corner is at (10, 30).
  - The other car's top-left corner is at (150, 10).



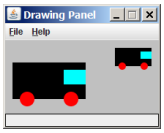
Translating figures: Solution

```
import java.awt.*;

public class DrawCar2 {
    public static void main(String[] args) {
        DrawingPanel panel = new DrawingPanel(260, 100);
        panel.setBackground(Color.LIGHT_GRAY);
        Graphics g = panel.getGraphics();
        drawCar(g, 10, 30);
        drawCar(g, 150, 10);
    }

    public static void drawCar(Graphics g, int x, int y) {
        g.setColor(Color.BLACK);
        g.fillRect(x, y, 100, 50);

        g.setColor(Color.RED);
        g.fillOval(x + 10, y + 40, 20, 20);
        g.fillOval(x + 70, y + 40, 20, 20);

        g.setColor(Color.CYAN);
        g.fillRect(x + 70, y + 10, 30, 20);
    }
}
```



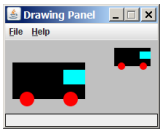
Parameterized figures: Scaling

- Methods can accept any number of parameters to adjust the figure's appearance.
- Exercise: Write a new version of the `drawCar` method that also allows the cars to be drawn at any size.



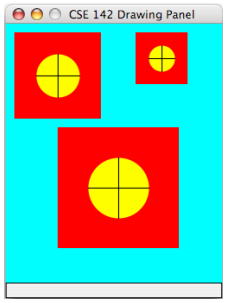
Scaling figures: Solution

```
import java.awt.*;

public class DrawCar3 {
    public static void main(String[] args) {
        DrawingPanel panel = new DrawingPanel(210, 100);
        panel.setBackground(Color.LIGHT_GRAY);

        Graphics g = panel.getGraphics();
        drawCar(g, 10, 30, 100);
        drawCar(g, 150, 10, 50);
    }

    public static void drawCar(Graphics g, int x, int y, int size) {
        g.setColor(Color.BLACK);
        g.fillRect(x, y, size, size / 2);

        g.setColor(Color.RED);
        g.fillOval(x + size / 10, y + 2 * size / 5,
                   size / 5, size / 5);
        g.fillOval(x + 7 * size / 10, y + 2 * size / 5,
                   size / 5, size / 5);

        g.setColor(Color.CYAN);
        g.fillRect(x + 7 * size / 10, y + size / 10,
                   3 * size / 10, size / 5);
    }
}
```



Parameterized figures: Exercise

- Display the following figures on a drawing panel of size 300x400:
  - top-left figure:
    - overall size = 100
    - top left corner = (10, 10)
    - inner rectangle and oval size = 50
    - inner top left corner = (35, 35)
  - top-right figure:
    - overall size = 60
    - top left corner = (150, 10)
    - inner rectangle and oval size = 30
    - inner top left corner = (165, 25)
  - bottom figure:
    - overall size = 140
    - top left corner = (60, 120)
    - inner rectangle and oval size = 70
    - inner top left corner = (95, 155)



Extra: Animating figures

- `DrawingPanel` has a method named `sleep` that pauses your program for a given number of milliseconds.
- You can use `sleep` to produce simple animations.

```
DrawingPanel panel = new DrawingPanel(250, 200);
Graphics g = panel.getGraphics();

g.setColor(Color.BLUE);
for (int i = 1; i <= NUM_CIRCLES; i++) {
    g.fillOval(15 * i, 15 * i, 30, 30);
    panel.sleep(500);
}
```

- Try adding `sleep` commands to loops in past exercises in these slides and watch the panel draw itself piece by piece!

Extra: Drawing polygons

1. Create a `Polygon` object and add points successively.

```
Polygon poly = new Polygon();
poly.addPoint(10, 20);
poly.addPoint(100, 40);
poly.addPoint(35, 100);
poly.addPoint(10, 80);
```

2. Tell the "pen" to draw (or fill) the polygon.

```
g.drawPolygon(poly);
```

or

```
g.fillPolygon(poly);
```

3. Now draw away!