Modularity

- Also a structured programming topic:
  - Can replace a rectangle with a module
    - Modules *contain* stacked/nested structures
- Java modules:
  - methods (the most basic modular units)
  - classes (collections of related methods)
  - packages (collections of related classes)

Using methods – “invoking”

- Direct translation of algorithm – e.g.,

  ```
  getData();
  process();
  showResults();
  ```

- In turn, the method `process()` might do:

  ```
  result = calculate(x, y);
  ```

  where `calculate` is another method, one that returns a value based on `x` and `y`.

- And so on …

`static` methods and variables

- A.k.a. class methods and class variables
- Technically, same for all instances of a class
  - No particular instance (object) is involved
    - So instance variables have no meaning in a static context
  - Access by class name, not object reference
- Good for “self-contained” methods
  - i.e., all necessary info is local to the method
  - May not use non-static methods or variables of class
- Good for shared data and instance counts
  - e.g., `if (Martian.count > 10) retreat();`

`java.lang.Math` static methods

- `Math`'s public methods are all `static`
  - So no need to make an object first
  - Invoke by class name and the dot "`.`" operator

  `Math.max(x, y)` and `Math.min(x, y)`
    - `max` and `min` are overloaded – return type same as `x, y`
- Usually double parameters and return type

```
double r = Math.toRadians(57.);
System.out.println("Sine of 57 degrees is " +
                       Math.sin(r));
```

- Also two constant values: `Math.PI` and `Math.E`
- `Math` is in `java.lang` – so no need to `import`

About constants like PI and E

- `final` variables are “constants”
  - May only assign value once; usually when declared
  - More efficient code (and often programming)
- Should always avoid “magic numbers”
  - e.g., decipher this line of code:

    ```
    cost = price * 1.0775 + 4.5;
    ```
  - More typing, but worth it:

    ```
    final double TAX_RATE = 0.0775;
    final double SHIPPING = 4.5;
    cost = price * (1. + TAX_RATE) + SHIPPING;
    ```
- Class constants – `final static` variables
  - e.g., `Math.PI` is declared in `java.lang.Math` as follows:

```
public static final double PI = 3.14159265358979323846;
```

Some `String` methods

- Accessing sub-strings: (Note – positions start at `0`, not `1`)
  - `substring(int)` – returns end of string
  - `substring(int, int)` – returns string from first position to *just before* last position
  - `charAt(int)` – returns single `char`
- `length()` – the number of characters
- `toUpperCase()`, `toLowerCase(), trim(),` …
- `valueOf(…)` – converts *any* type to a String
  - But converting from a String more difficult – must use specialized methods to parse

Note: parameters are *copies*

- e.g., `void foo(int x)`

  `{ x = 5; }` // changes *copy* of the value passed
- So what does the following code print?

  ```
  int a = 1;
  foo(a);
  System.out.print(“a = “ + a);
  ```

  - Answer: `a = 1`
- Same applies to “immutable objects” like Strings

  ```
  String s = “APPLE”;
  anyMethod(s);
  System.out.print(s);
  ```
  // prints `APPLE`

But references *are* references

- A reference is used to send messages to an object
  - So the original object can change if not immutable
- e.g., `void foo(Rectangle x)`

  ```
  {   x.translate(5,5);  }
  ```

  // actually moves the rectangle
- Copy of reference is just as useful as the original
  - i.e., although methods cannot change a reference, they can change the original object
  - Moral: be careful about passing object references

Random simulations

- Can use Math.random() method
  - Pseudorandom double value – range 0 to almost 1

```
int diceValue = 1 + (int)(Math.random() * 6);
```

- Better to use a java.util.Random object

```
Random generator = new Random();
int diceValue = 1 + generator.nextInt(6);
```

  - e.g., RandomIntegers.java (Fig. 6.7, p. 221)
  - And more interesting Craps.java (Fig. 6.9, pp. 225-226)
- Not just for integers (and not just for dice)

```
double angle = 360 * generator.nextDouble();
boolean gotLucky = generator.nextBoolean();
```

Scope/duration of identifiers

- Depends on where declared
  - i.e., in which set of {}; in which “block”
- Instance variables:
  - Duration (“lifetime”): same as duration of object
  - Scope: available throughout the class
- Variables declared in method or other block (including formal parameters):
  - Duration: as long as block is being executed
  - Scope: available just within the block
- See Scope.java (Fig. 6.11, p. 230)

Overloading method names

- Method signature is: name (parameter list)
  - Can reuse a name with different parameter list
- List distinguished by (1) number of parameters, and (2) types and order of parameters
  - e.g., three greeting methods (for a robot?):

```
void hi() { System.out.print(“Hi”); }
void hi(String name)  // to greet a person by name
{ System.out.print(“Hi “ + name); }
void hi(int number)  // to greet a collection of people
{ System.out.print(“Hi you “ + number); }
```

  - Another example: MethodOverload.java (Fig. 6.13, p. 233)
- Cannot distinguish just by return type though (Fig. 6.15)

Another aside – Coloring and animating drawings

- e.g., DrawSmiley.java (Fig. 6.16, p. 236)
- Now let's spice up the Car drawing
  - First add a `Color` instance variable to `class Car`, and add ways to change a `Car`'s position
- Animation is `class CarComponent`'s responsibility
  - Change the two Car references to instance variables
  - Create Car objects the *first* time `paintComponent` is called – might as well make their colors random
  - Add `animate()` method – moves Cars, and uses a `Thread`:

    ```
    try { Thread.sleep(500); }
    catch(InterruptedException e) { }
    ```

    And includes repeated calls to `repaint()` after moves
  - Finally, must invoke `animate`() from class `CarViewer`