Methods
Lecture 8 (7/13/2007)

Useless Fact of the Day
• MySpace is currently the most visited domain by U.S. internet users. Two years ago, only 0.1% of Internet visits were to MySpace sites, while today 4.46% of visits are. For comparison, 3.89% of visits are to Google.

Topics
• Usefulness
• Communication
• Parameters and Return

Anatomy of a Method
method name (identifier)
return type
parameter list
body
return statements

public boolean isAwesomeMovie(String script, int explosions)
{
    if (script.contains("pirate") || script.contains("ninja") ||
        script.contains("alligator wrestling") || explosions > 30)
    {
        return true;
    }
    else
    {
        return false;
    }
}
Usefulness - Organization

- Package a unit of code
  - Name and package a well-defined unit of work
  - Aids in higher-level design, where you don’t have to know the internal workings of the method; instead, the name becomes a proxy for the work
  - For example, FANG Engine methods (like `setScale()`, `setLocation()`, etc.)
  - Make your code cleaner and more organized (and easier to figure out what’s going on at a high level)

Usefulness - Organization

- These two blocks of code are exactly the same — which `advanceFrame` method is easier to understand at a glance?

Usefulness - Code Repetition

- Avoid repetitive code
  - If a section of code is repeated, or almost repeated, many times...
  - ...you can use often replace the section of code with a method
  - Write once, use many times
  - Methods can be flexible, to even handle non-identical (but very similar) repeated code

Usefulness - Code Repetition

- For example, this is the code for the `setScale` method in FANG’s Sprite class:

  ```java
  public void setScale(double scale) {
      if (scale == 0) {
          return;
      }
      double determinant = transform.getScaleX() * transform.getScaleY() - transform.getShearX() * transform.getShearY();
      scale = scale / Math.sqrt(determinant);
      if (scale == 1) {
          return;
      }
      transform.setTransform(scale * transform.getScaleX(), scale * transform.getShearY(), scale * transform.getShearX(), scale * transform.getScaleY(), transform.getTranslateX(), transform.getTranslateY());
      if (optimizeForBitmap) restoreImage();
  }
  ```

  - You wouldn’t want to have to repeat all that code every time you wanted to change the size of a Sprite — so instead, you can just call the `setScale` method for whatever Sprite you want, and give it whatever scale you want it to be as an argument
Communication

- Methods need to communicate/share info (data) with the rest of the program
- Three ways:
  - **Instance variables** - get info both in and out
    - These variables are “known” throughout a class, so any method in the class can see and modify them
  - **Parameters** - get info in (and sometimes out)
  - **Return statements** - get info out

Parameters

- Mainly get info into a method
  - *in-only for primitive-typed arguments* (int, double, float, boolean, char)
  - **Objects** (reference-typed data, like Sprites) passed in as parameters sometimes allow info to get **out**
  - This happens if the method modifies the passed-in object -- for example, a method could change the color of a Sprite that was given to it as an argument
  - You won’t see this extremely often (sometimes modifying an object is not even possible -- you cannot modify a String object, for example)

Return Statement

- Copies info **out of** a method
  - “Returns” some piece of data to the **invoking** statement or expression (the bit of code that called the method)
  - Often, the result is assigned to a variable, using =
  - The returned data basically replaces the method call when the code executes -- for example, `ball.getLocation()` would be replaced by a Coordinate object specifying the ball’s current location
  - Every method must specify a **return type**
  - If it doesn’t return anything, the return type is **void**

Return: Example

```java
public double square(double x)
{
    x = x * x;
    return x;
}
```

A common mistake:

```java
// (in another method)
double y = 3;
        
square(y);
System.out.println(y);
3.0
```

Corrected:

```java
// (in another method)
double y = 3;
        
square(y);
System.out.println(y);
```

Remember, primitive arguments can only send data in to a method -- not get it out! This means that whatever primitive variable you send to a method will remain (directly) unchanged by the method (see left). We must use the return value to get data out of the square method -- we use the return value that the method gives us to update y (see right).
Local Variables

- A **local variable** is a variable declared within a method.
- It can only be seen within the method, below where it was declared.
- A method’s **parameters** are also local variables.

```java
public void advanceFrame(double timePassed)
{
    Coordinate center = ball.getLocation();
    // check for it moving past the left OR right boundaries
    if (center.getX() < 0 || center.getX() > 1)
    {
        ballTracker.bounce(180);
    }
    // check for it moving past the top OR bottom boundaries
    if (center.getY() < 0 || center.getY() > 1)
    {
        ballTracker.bounce(0);
    }
}
```

Local Variables

- Remember, the **scope** of a variable is where it lives (where it can be seen/used).
- Variables can have the same names if they have different scopes -- in this case, they are different variables!
- Example: the following would be fine (no errors) -- there is one local variable called “x” in the first method, and another (different) local variable called “x” in the second method:

```java
public double square(double x)
{
    x = x * x;
    return x;
}
```

```java
// (in another method, like startGame)
double x = 3;
  x = square(x);
System.out.println(x);
```