Extra Credit Assignment

• Posted on homework section of web site
• Due February 24
  – You must send me email by Friday, Feb 7 if you are going to work on this assignment
• Everything that you need to know should be on the web site
• Let me know if you have any questions

Last Time: Random ints

• Wanted to map doubles to integers so that each number from 1 to 10 is equally likely:
  – Multiply r by 10
  – Convert to an integer
  – Add 1

  static void printRandomNumbers(int count) {
    double r;
    int value;
    for ( int i = 0; i < count; i++ ) {
      r = Math.random();
      value = (int)( r * 10 ) + 1;
      System.out.println( r + "\t" + value);
    }
  }

Another suggestion

• Multiply r by 9
• Round to the nearest integer
• Add 1

  static void printRandomNumbers(int count) {
    double r;
    int value;
    for ( int i = 0; i < count; i++ ) {
      r = Math.random();
      value = (int) Math.round(r*9) + 1;
      System.out.println( r + "\t" + value + "\t" + value);
    }
  }

• Does this work?
Does this work?

- Unfortunately, no
  - It does convert $r$ to integer values from 1 to 10
  - But, it does not produce equally likely results
    - 1 and 10 only occur $\frac{1}{2}$ as frequently as the other values
- See RandomPrint3.java

```java
// This doesn't work correctly - don't use
r = Math.random();
value = (int) Math.round(r*9) + 1;

// This one is correct
r = Math.random();
value = (int) (r * 10) + 1;
```

More about pass-by-value

- We have looked at several examples of passing arguments to methods
  - As literals
    ```java
    printValues( 1, 2.0 );
    System.out.println( "Hello, world!" );
    ```
  - As variables
    ```java
    printValues( x, y );
    System.out.println( total );
    ```

More about pass-by-value

- We have also seen that expressions can be passed as method arguments
  ```java
  System.out.println( "The answer is " + result );
  return ( n * factorial( n - 1 ) );
  alphabet( (char) (firstLetter + 1) );
  ```
More about pass-by-value

- Remember, a method call is a type of expression
- So, a method argument can be the result of a method call
  ```java
  System.out.print( Math.sqrt(i) );
  System.out.print( Console.in.readInt() );
  ```
- Arguments get evaluated before being passed to the method

An Extreme Example

- Use the result of the method as an argument to itself
```java
class Square {
    public static void main( String[] args ) {
        int val;
        System.out.print("Please enter an integer: ");
        val = Console.in.readInt();
        val = Square( Square( val ) ); // What's this?
        System.out.println( val );
    }
    static int Square( int x ) {
        return ( x * x );
    }
}
```

An Extreme Example

- This code fragment:
```java
val = Console.in.readInt();
val = Square( Square( val ) ); // What's this?
System.out.println( val );
```
- Does the same thing as this one:
```java
val = Console.in.readInt();
val = Square( val );
val = Square( val );
System.out.println( val );
```
Arguments

• Java will widen argument values if necessary
  – For example, method definition of `Math.sqrt()` is
    `static double sqrt( double a )`
  – OK to call `Math.sqrt( 4 )` because Java
    widens the int 4 to a double

Arguments

• Java won’t narrow an argument
  – In our Alphabet example, we got a compiler
    error when we called `alphabet()` like this:
      `alphabet( firstLetter + 1 );`
    because the argument is supposed to be a
    char, but the expression is an int
  – The corrected call explicitly cast the
    expression
      `alphabet( (char) (firstLetter + 1) );`

Method Signature

• A method signature is made up of
  – the method’s name
  – its return type
  – its parameters
    • number of parameters
    • their type
    • their order
Method Signature

- The signature of
  static void printValues( int x, int y )
  is
  void printValues( int, int )

- The signature of
  public static double sqrt( double a )
  is
  double sqrt( double )

Method Signature

- The signature of
  static double foo( int x, double y, float z )
  is
  double foo( int, double, float )

- The signature of
  static double foo( double x, float y, int z )
  is
  double foo( double, float, int )

- These methods have the same name, but different signatures

Method Overloading

- In Java, you can use the same name for more than one method
- This is called method overloading
- The methods must have
  – a different number of parameters, or
  – different types of parameters
- The return type is not used as part of signature matching for method overloading
- The parameter names are not used for matching
Method Overloading

- We've already seen an example of method overloading:
  - `System.out.println()` is overloaded to accept different types of parameters:
    ```java
    void System.out.println()
    void System.out.println( char )
    void System.out.println( int )
    void System.out.println( String )
    ```
- Better than `println()`, `printlnChar()`, `printlnInt()`, `printlnString()`, etc.

Method Overloading

- Java matches actual method calls to particular overloaded methods at compile time
- If the argument types in the call don't exactly match the formal parameter types, Java will try to widen the argument types

Method Overloading

- Overload the `min()` method:
  ```java
  static int min( int a, int b ) {
    if ( a < b )
      return a;
    else
      return b;
  }
  static double min( double a, double b ) {
    if ( a < b )
      return a;
    else
      return b;
  }
  ```
Method Overloading

- Which version of min() gets called here?

```java
int a = 3, b = 4;
double c = 5.3, d = 7.2;
float e = 1.4f;
byte f = 2;
long g = 5L;
System.out.println( min( a, b ) );
System.out.println( min( c, d ) );
System.out.println( min( c, e ) );
System.out.println( min( a, c ) );
System.out.println( min( a, f ) );
System.out.println( min( e, f ) );
System.out.println( min( a, g ) );
```

Ambiguous Overloaded Methods

- Sometimes, Java can’t figure out which overloaded method to call

```java
class AmbiguousOverload {  
    public static void main( String[] args ) {  
        int i = 1, j = 2;
        System.out.println( ambig( i, j ) );
    }
    static boolean ambig( float x, int y ) {  
        return x < y;
    }
    static boolean ambig( int x, float y ) {  
        return x > y;
    }
}
```

Removing ambiguity

- Sometimes, Java can’t figure out which overloaded method to call
  - You must explicitly cast arguments to remove the ambiguity

```java
class AmbiguousOverload {  
    public static void main( String[] args ) {  
        int i = 1, j = 2;
        System.out.println( ambig( (float) i, j ) );
    }
    static boolean ambig( float x, int y ) {  
        return x < y;
    }
    static boolean ambig( int x, float y ) {  
        return x > y;
    }
}
```
Overloading: return type not included

- You cannot have overloaded methods whose only difference is the return type

```java
class BadOverload {
    public static void main(String[] args) {
        System.out.println(foo(2));
    }

    static int foo(int i) {
        return i * 2;
    }

    static double foo(int j) {
        return j * 2.5;
    }
}
```

Recursion and Iteration

- Not all computer languages support recursion
- Recursive algorithms can be implemented using iteration
  - Not always easy
  - Sometimes the resulting code is hard to follow

Recursion Example: Factorial

- Factorial
  - \( n! = n \times (n-1) \times (n-2) \times \ldots \times 1 \)
  - \( n! = n \times (n-1)! \)
  - \( 1! = 1 \)

```java
static long factorial(int n) {
    if (n <= 1)
        return 1;
    else
        return n * factorial(n - 1);
}
```
Iteration Example: Factorial

- Factorial – recursive version
  ```
  static long factorial( int n ) {
    if ( n <= 1 )
      return 1;
    else
      return ( n * factorial( n - 1 ) );
  }
  ```

- Factorial – iterative version
  ```
  static long factorial( int n ) {
    long fact = 1;
    for (int i = 1; i <= n; i++)
      fact = fact * i;
    return fact;
  }
  ```

Iteration and Recursion

- This is much harder to do for complex recursive functions, such as
  - MergeSort
  - Searching