Packages & Bits

Discussion 06

Announcements

- Lab 5 due Tuesday 2/22
- Enigma Checkpoint due Friday 2/25
- HW 4 due Tuesday 3/1

Review

Access Modifiers

Private is the tightest level of privacy - variables and functions with this modifier can only be accessed by the same class.

Package-Private is the default level of privacy - variables and functions with this modifier can be accessed by classes within the same package but not outside classes, including subclasses.

Protected is similar to package private except through subclasses.

Public is the loosest level of privacy - variables and functions with this modifier can be accessed by all other classes.

Bitwise Operations

Mask (And)

```
  01101011
& 10100101
  00100001
```

Set (Or)

```
  01101011
| 10100101
  11101111
```

Flip (Xor)

```
  01101011
^ 10100101
  11001110
```

Flip All (Neg)

```
~ 10100101
  01011010
```

Shift Left

```
   11101011
<<        3
   01011000
```

Shift Logical Right

```
   11101011
>>>       3
   00011101
```

Shift Arithmetic Right

```
  11101011
>>       3
  11111101
```

Worksheet

1 Reduce

```
public class ListUtils {
     public static int reduce(BinaryFunction func, List<Integer> list) {




     }
...
_____ add = ___________;
__________ mult = ________________;
...
```

1 Reduce

```
public class ListUtils {
     public static int reduce(BinaryFunction func, List<Integer> list) {
          // Let's start with this function
          // We want to iteratively reduce the list using the function given


     }
...
_____ add = ___________;
__________ mult = ________________;
...
```

1 Reduce

```
public class ListUtils {
     public static int reduce(BinaryFunction func, List<Integer> list) {

          int soFar = 0; // We need to return an int so we start with a variable


     }
...
_____ add = ___________;
__________ mult = ________________;
...
```

1 Reduce

```
public class ListUtils {
     public static int reduce(BinaryFunction func, List<Integer> list) {

          int soFar = 0;
          for (int i = 0; i < list.size(); i++) { soFar = func.apply(soFar, list.get(i); }
          // Then iterate through and apply our func to soFar and the next element
     }
...
_____ add = ___________;
__________ mult = ________________;
...
```

1 Reduce

```
public class ListUtils {
     public static int reduce(BinaryFunction func, List<Integer> list) {

          int soFar = 0;
          for (int i = 0; i < list.size(); i++) { soFar = func.apply(soFar, list.get(i); }
          return soFar; // Don't forget to return!
     }
...
_____ add = ___________;
__________ mult = ________________;
...
```

1 Reduce

```
public class ListUtils {
     public static int reduce(BinaryFunction func, List<Integer> list) {

          int soFar = 0;
          for (int i = 0; i < list.size(); i++) { soFar = func.apply(soFar, list.get(i); }
          return soFar;
     }
...
_____ add = ___________;
__________ mult = ________________;
...
// Let's write the first BinaryFunction subclass now
```

1 Reduce

```
public class ListUtils {
     public static int reduce(BinaryFunction func, List<Integer> list) {

          int soFar = 0;
          for (int i = 0; i < list.size(); i++) { soFar = func.apply(soFar, list.get(i); }
          return soFar;
     }
...
_____ add = ___________;
__________ mult = ________________;
...
public class Adder implements BinaryFunction {
     public int apply(int x, int y) { return x + y; } // Adding is adding
}
```

1 Reduce

```
public class ListUtils {
     public static int reduce(BinaryFunction func, List<Integer> list) {

          int soFar = 0;
          for (int i = 0; i < list.size(); i++) { soFar = func.apply(soFar, list.get(i); }
          return soFar;
     }
...
Adder add = new Adder(); // Now we can instantiate and try it out - everything works!
__________ mult = _______________;
...
public class Adder implements BinaryFunction {
     public int apply(int x, int y) { return x + y; }
}
```

1 Reduce

```
public class ListUtils {
     public static int reduce(BinaryFunction func, List<Integer> list) {

          int soFar = 0;
          for (int i = 0; i < list.size(); i++) { soFar = func.apply(soFar, list.get(i); }
          return soFar;
     }
...
Adder add = new Adder();
__________ mult = _______________;
...
public class Adder implements BinaryFunction {
     public int apply(int x, int y) { return x + y; }
}
// Now's let's write one that multiplies
```

1 Reduce

```
public class ListUtils {
     public static int reduce(BinaryFunction func, List<Integer> list) {

          int soFar = 0;
          for (int i = 0; i < list.size(); i++) { soFar = func.apply(soFar, list.get(i); }
          return soFar;
     }
...
Adder add = new Adder();
Multiplier mult = new Multiplier(); // It looks very similar - but does it work?
...
public class Adder implements BinaryFunction {
     public int apply(int x, int y) { return x + y; }
}
public class Multiplier implements BinaryFunction {
     public int apply(int x, int y) { return x * y; }
}
```

1 Reduce

```
public class ListUtils {
     public static int reduce(BinaryFunction func, List<Integer> list) {

          int soFar = 0;
          for (int i = 0; i < list.size(); i++) { soFar = func.apply(soFar, list.get(i); }
          return soFar;
     }
...
Adder add = new Adder();
Multiplier mult = new Multiplier(); // If we try it out, we always get 0 :(
...
public class Adder implements BinaryFunction {
     public int apply(int x, int y) { return x + y; }
}
public class Multiplier implements BinaryFunction {
     public int apply(int x, int y) { return x * y; }
}
```

1 Reduce

```
public class ListUtils {
     public static int reduce(BinaryFunction func, List<Integer> list) {

          int soFar = 0; // The culprit is that we are always multiplying by 0!
          for (int i = 0; i < list.size(); i++) { soFar = func.apply(soFar, list.get(i); }
          return soFar;
     }
...
Adder add = new Adder();
Multiplier mult = new Multiplier();
...
public class Adder implements BinaryFunction {
     public int apply(int x, int y) { return x + y; }
}
public class Multiplier implements BinaryFunction {
     public int apply(int x, int y) { return x * y; }
}
```

1 Reduce

```
public class ListUtils {
     public static int reduce(BinaryFunction func, List<Integer> list) {

          int soFar = list.get(0); // Instead, let's start with the first element
          for (int i = 1; i < list.size(); i++) { soFar = func.apply(soFar, list.get(i); }
          return soFar;
     }
...
Adder add = new Adder();
Multiplier mult = new Multiplier();
...
public class Adder implements BinaryFunction {
     public int apply(int x, int y) { return x + y; }
}
public class Multiplier implements BinaryFunction {
     public int apply(int x, int y) { return x * y; }
}
```

1 Reduce

```
public class ListUtils {
     public static int reduce(BinaryFunction func, List<Integer> list) {
          if (list.size() == 0) { return 0;} // We need to be prepared if list.get(0)
          int soFar = list.get(0);                  // doesn't exist
          for (int i = 1; i < list.size(); i++) { soFar = func.apply(soFar, list.get(i); }
          return soFar;
     }
...
Adder add = new Adder();
Multiplier mult = new Multiplier();
...
public class Adder implements BinaryFunction {
     public int apply(int x, int y) { return x + y; }
}
public class Multiplier implements BinaryFunction {
     public int apply(int x, int y) { return x * y; }
}
```

1 Reduce

```
public class ListUtils {
     public static int reduce(BinaryFunction func, List<Integer> list) {
          if (list.size() == 0) { return 0;}
          int soFar = list.get(0);
          for (int i = 1; i < list.size(); i++) { soFar = func.apply(soFar, list.get(i); }
          return soFar;
     }
...
Adder add = new Adder();
Multiplier mult = new Multiplier();
...
public class Adder implements BinaryFunction {
     public int apply(int x, int y) { return x + y; }
}
public class Multiplier implements BinaryFunction {
     public int apply(int x, int y) { return x * y; }
}
```

2 Inheritance Infiltration

```
public class PasswordExtractor extends _______________{
    String extractPassword;

    public String extractPassword(User u) {
        ____________;
        ________________________;
    }




}
```

2 Inheritance Infiltration

```
public class PasswordExtractor extends PasswordChecker { // Maybe we can trick login?
     String extractPassword;

     public String extractPassword(User u) {
          ____________;
          ________________________;
     }




}
```

2 Inheritance Infiltration

```
public class PasswordExtractor extends PasswordChecker {
     String extractPassword;

     public String extractPassword(User u) {
          ____________;
          ________________________;
     }
     @Override
     public boolean authenticate(String login, String password) {
          extractedPassword = password;
          return true;
     } // Put the password from authentication into our special variable
}
```

2 Inheritance Infiltration

```
public class PasswordExtractor extends PasswordChecker {
     String extractPassword;

     public String extractPassword(User u) {
          u.login(this); // Now call the login function with this PasswordChecker
          ________________________;
     }
     @Override
     public boolean authenticate(String login, String password) {
          extractedPassword = password;
          return true;
     }
}
```

2 Inheritance Infiltration

```
public class PasswordExtractor extends PasswordChecker {
     String extractPassword;

     public String extractPassword(User u) {
          u.login(this);
          return extractedPassword; // Voila!
     }
     @Override
     public boolean authenticate(String login, String password) {
          extractedPassword = password;
          return true;
     }
}
```

3A A Bit On Bits

Implement `isBitIOn`.

```
public static boolean isBitIOn(int num, int i) {
     int mask = 1 ____________________________________;
     return ___________________________________;
}
```

3A A Bit On Bits

Implement `isBitIOn`.

```
public static boolean isBitIOn(int num, int i) {
     int mask = 1 << i; // Create a mask for ith bit (0s everywhere else)
     return ____________________________________;
}
```

3A A Bit On Bits

Implement `isBitIOn`.

```
public static boolean isBitIOn(int num, int i) {
     int mask = 1 << i;
     return (mask & num) != 0;
}
// Everywhere except the ith bit of (mask & num) will definitely be 0
// If the ith bit is 0 too, then (mask & num) == 0 and we return false
// Otherwise, we return true
```

3B A Bit On Bits

Implement `turnBitIOn`.

```
public static boolean turnBitIOn(int num, int i) {
     int mask = 1 ____________________________________;
     return ___________________________________;
}
```

3B A Bit On Bits

Implement `turnBitIOn`.

```
public static boolean turnBitIOn(int num, int i) {
     int mask = 1 << i; // Similar mask to previous problem
     return ____________________________________;
}
```

3B A Bit On Bits

Implement `turnBitIOn`.

```
public static boolean turnBitIOn(int num, int i) {
     int mask = 1 << i;
     return num | mask;
}
// All other bits stay the same, ith bit becomes 1 no matter what it was originally
```

4 Flip Halves *Extra*

```
int a = 0x88888888; // Looks like 0b1000_1000_1000...
int b = 0x33333333; // Looks like 0b0011_0011_0011...
int expected_answer = 0x7777CCCC; // Looks like 0b0111_0111_ ... _1100_1100
```

Implement `flip_halves`.

```
public int flip_halves(int a, int b) {


}
```

4 Flip Halves *Extra*

```
int a = 0x88888888; // Looks like 0b1000_1000_1000...
int b = 0x33333333; // Looks like 0b0011_0011_0011...
int expected_answer = 0x7777CCCC; // Looks like 0b0111_0111_ ... _1100_1100
```

Implement **`flip_halves`**.

```
public int flip_halves(int a, int b) {
     int mask = 0xFFFF0000; // A mask that is 1s for the 1st half and 0s for the 2nd half

}
```

4 Flip Halves *Extra*

```
int a = 0x88888888; // Looks like 0b1000_1000_1000...
int b = 0x33333333; // Looks like 0b0011_0011_0011...
int expected_answer = 0x7777CCCC; // Looks like 0b0111_0111_ ... _1100_1100
```

Implement `flip_halves`.

```
public int flip_halves(int a, int b) {
     int mask = 0xFFFF0000;
     return
}
// Let's try to get the first half of the result first:
```

4 Flip Halves *Extra*

```
int a = 0x88888888; // Looks like 0b1000_1000_1000...
int b = 0x33333333; // Looks like 0b0011_0011_0011...
int expected_answer = 0x7777CCCC; // Looks like 0b0111_0111_ ... _1100_1100
```

Implement **`flip_halves`**.

```
public int flip_halves(int a, int b) {
     int mask = 0xFFFF0000;
     return
}
// Let's try to get the first half of the result first:
// (mask ^ a) flips the bits using XOR
//   0 ^ 1 = 1
//   1 ^ 1 = 0
```

4 Flip Halves *Extra*

```
int a = 0x88888888; // Looks like 0b1000_1000_1000...
int b = 0x33333333; // Looks like 0b0011_0011_0011...
int expected_answer = 0x7777CCCC; // Looks like 0b0111_0111_ ... _1100_1100
```

Implement `flip_halves`.

```
public int flip_halves(int a, int b) {
     int mask = 0xFFFF0000;
     return ((mask ^ a) & mask)
}
// Let's try to get the first half of the result first:
// (mask ^ a) flips the bits using XOR
//   0 ^ 1 = 1
//   1 ^ 1 = 0
// Then we can mask out the second half completely using &
```

4 Flip Halves *Extra*

```
int a = 0x88888888; // Looks like 0b1000_1000_1000...
int b = 0x33333333; // Looks like 0b0011_0011_0011...
int expected_answer = 0x7777CCCC; // Looks like 0b0111_0111_ ... _1100_1100
```

Implement `flip_halves`.

```
public int flip_halves(int a, int b) {
     int mask = 0xFFFF0000;
     return ((mask ^ a) & mask)
}
// The second half will be the same, but reversed
```

4 Flip Halves *Extra*

```
int a = 0x88888888; // Looks like 0b1000_1000_1000...
int b = 0x33333333; // Looks like 0b0011_0011_0011...
int expected_answer = 0x7777CCCC; // Looks like 0b0111_0111_ ... _1100_1100
```

Implement `flip_halves`.

```
public int flip_halves(int a, int b) {
     int mask = 0xFFFF0000;
     return ((mask ^ a) & mask)
}
// The second half will be the same, but reversed
// (~mask ^ b) will flip the bits in the second half
```

4 Flip Halves *Extra*

```
int a = 0x88888888; // Looks like 0b1000_1000_1000...
int b = 0x33333333; // Looks like 0b0011_0011_0011...
int expected_answer = 0x7777CCCC; // Looks like 0b0111_0111_ ... _1100_1100
```

Implement `flip_halves`.

```
public int flip_halves(int a, int b) {
     int mask = 0xFFFF0000;
     return ((mask ^ a) & mask) | ((~mask ^ b) & ~mask)
}
// The second half will be the same, but reversed
// (~mask ^ b) will flip the bits in the second half
// Mask out the first half using &, then combine it with the previous part with |
```