Computer Science: Create Your Own RPG Day #3

OPPTAG Explorations 2014

Brian Nakayama[1]

[1] Department of Computer Science, Iowa State University, Ames, IA 50010, USA

July 9th, 2014

Review: Spot the Runtime Errors

```
int array = new array[-1];
for(int y = -1; y <= array.length; y <= 1){
    array[y] = array[y + 1];
    array = null;
}
```

Review: Spot the Logic Errors

- We want the old array to hold the contents of the array.
- We want the new array to contain $\{9, 7, 5, 3, 6\}$

```
int oldarray = array;
int[] array = {5, 4, 3, 2, 1};
array = oldarray;
for(int y = 0; y < array.length; y ++){
    array[y] += array[(y + 1) % 4];
}
```

Are the following equivalent? #1

```
for(int y = 0; y < array.length; y ++){
    array[y] += 1;
}
```

?

```
int y = 0;
do{
    array[y] +=1
    y ++;
} while (y < array.length);
```

New Logins!

- Last time Eclipse was running really, really slow.
- We now have new accounts that should fix this:

Username: !comsguest
Password: wG4eQZag

20 Minutes of Work Time.

I can help debug code from yesterday during this time.

What is a method?

- A method is like a function. For example:

$$f(x) = x + 7$$

- When $x = 2$, $f(x) =$

What is a method?

- A method is like a function. For example:

$$f(x) = x + 7$$

- When $x = 2$, $f(x) = 9$. In java this would look like:

```
public  static  int  f (int x ){
    return x + 7;
}
```

What is a method?

- A method is like a function. For example:

$$f(x) = x + 7$$

- When $x = 2$, $f(x) = 9$. In java this would look like:

```
public static int f (int x ){
    return x + 7;
}
```

Modifier: *public* means that this method can be seen by all classes that can see the class the method is in.

What is a method?

- A method is like a function. For example:

$$f(x) = x + 7$$

- When $x = 2$, $f(x) = 9$. In java this would look like:

```
public static int f (int x ){
    return x + 7;
}
```

Modifier: *static* means that this method does not require instantiation (it is not part of an object). All static methods can only access other static methods or variables.

What is a method?

- A method is like a function. For example:

$$f(x) = x + 7$$

- When $x = 2$, $f(x) = 9$. In java this would look like:

```
public  static  int  f (int x ){
    return x + 7;
}
```

int is the type of value the method returns. If a method does not return anything, this is *void*

What is a method?

- A method is like a function. For example:

$$f(x) = x + 7$$

- When $x = 2$, $f(x) = 9$. In java this would look like:

```
public  static  int  f (int x ){
    return x + 7;
}
```

f is the name of the method. Like variables, you can name methods whatever you want.

What is a method?

- A method is like a function. For example:

$$f(x) = x + 7$$

- When $x = 2$, $f(x) = 9$. In java this would look like:

```
public  static  int  f (int x ){
    return x + 7;
}
```

int x is an argument for *f*. Methods, don't have to have arguments, but they are often useful.

What is a method?

- A method is like a function. For example:

$$f(x) = x + 7$$

- When $x = 2$, $f(x) = 9$. In java this would look like:

```
public  static  int  f (int x ){
    return x + 7;
}
```

return: if a method says it will return a value, then it must use `return` followed by a value or variable of the type given in the method declaration.

Test out some methods!

You are making a Dungeons and Dragons game. You need two methods:

- You need one method that rolls n dice with x sides and then totals up the result.

  ```
  public static int rollDice(int n, int x){ ...
  ```

- You need another method that calculates the health for players with armor a and health h after an attack x. An attack hits if the attack is higher than the armor. If an attack hits, it deals $x - a$ damage.

Test out some methods!

You are making a Dungeons and Dragons game. You need two methods:

- You need one method that rolls n dice with x sides and then totals up the result.

  ```
  public static int rollDice(int n, int x){ ...
  ```

- You need another method that calculates the health for players with armor a and health h after an attack x. An attack hits if the attack is higher than the armor. If an attack hits, it deals $x - a$ damage.

- BONUS: Write a method called `goodfortune(int n, int x, int t)`! It works just like `rollDice(int n, int x)` except it rerolls t times and returns the highest result.

Are the following equivalent? #2

```
Scanner s = new Scanner(System.in);

String cont = s.nextLine();
while(cont.equals("yes")){
    System.out.println("Continue?");
    cont = s.nextLine();
}
```

?

```
String cont = s.nextLine();
if(cont.equals("yes")){
    do{
        System.out.println("Continue?");
        cont = s.nextLine();
    } while (cont.equals("yes"));
}
```

Mini-Review

- As of Java 7, Strings do work in Switch-Case statements!

Mini-Review

- As of Java 7, Strings do work in Switch-Case statements!
- Switch-case works for byte, char, short, int and enum.

Mini-Review

- As of Java 7, Strings do work in Switch-Case statements!
- Switch-case works for byte, char, short, int and enum.
- Switch-case can only be used to check for equality (or by using default inequality).

Mini-Review

- As of Java 7, Strings do work in Switch-Case statements!
- Switch-case works for byte, char, short, int and enum.
- Switch-case can only be used to check for equality (or by using default inequality).
- A switch-case works by jumping to a case and then *processing all code beneath the case* including code in other cases. To prevent this we use the keyword break.

Mini-Review

- As of Java 7, Strings do work in Switch-Case statements!
- Switch-case works for byte, char, short, int and enum.
- Switch-case can only be used to check for equality (or by using default inequality).
- A switch-case works by jumping to a case and then *processing all code beneath the case* including code in other cases. To prevent this we use the keyword break.
- Using a switch-case can save space, and works faster than if-else statements!

Switch Case Quiz!

```
char c = 'D';
switch(c){
case 'U':
    System.out.print("b");
case 'D':
    System.out.print("ear");
    break;
case 'L':
    System.out.print("left");
    break;
case 'R':
    System.out.print("ear");
default:
    System.out.print("ring");
}
```

Switch Case Quiz!

```
char c = 'G';
switch(c){
case 'U':
    System.out.print("b");
case 'D':
    System.out.print("ear");
    break;
case 'L':
    System.out.print("left");
    break;
case 'R':
    System.out.print("ear");
default:
    System.out.print("ring");
}
```

Switch Case Quiz!

```
char c = 'U';
switch(c){
case 'U':
    System.out.print("b");
case 'D':
    System.out.print("ear");
    break;
case 'L':
    System.out.print("left");
    break;
case 'R':
    System.out.print("ear");
default:
    System.out.print("ring");
}
```

Switch Case Quiz!

```
char c = 'L';
switch(c){
case 'U':
    System.out.print("b");
case 'D':
    System.out.print("ear");
    break;
case 'L':
    System.out.print("left");
    break;
case 'R':
    System.out.print("ear");
default:
    System.out.print("ring");
}
```

Switch Case Quiz!

```
char c = 'R';
switch(c){
case 'U':
    System.out.print("b");
case 'D':
    System.out.print("ear");
    break;
case 'L':
    System.out.print("left");
    break;
case 'R':
    System.out.print("ear");
default:
    System.out.print("ring");
}
```

Making Objects

- So far we've mostly used three objects: Strings, *Scanner*, and arrays.

Making Objects

- So far we've mostly used three objects: Strings, *Scanner*, and arrays.
- Objects are units composed of "behavior" (methods contained in the object), and "state" (variables contained in the object).

Making Objects

- So far we've mostly used three objects: Strings, *Scanner*, and arrays.
- Objects are units composed of "behavior" (methods contained in the object), and "state" (variables contained in the object).
- *We are objects*. We are each objects of the person class. We each have an age (int), a name (string), and a method called birthday (public void Birthday(){age ++;}).

Making Objects

- So far we've mostly used three objects: Strings, *Scanner*, and arrays.
- Objects are units composed of "behavior" (methods contained in the object), and "state" (variables contained in the object).
- *We are objects*. We are each objects of the person class. We each have an age (int), a name (string), and a method called birthday (public void Birthday(){age ++;}).
- To create an object, you must use the *new* keyword. The *new* keyword allocates memory in the computer for our object, and it can also instantiate variables within the object.

Making Objects

- For example:

```
 Person Brian = new Person();
Brian.Birthday();
System.out.println(Brian.age);
```

Constructors

- To initialize an object we can create a *constructor*.

Constructors

- To initialize an object we can create a *constructor*.
- Constructors for a `class <name>` look like:

```
public <name>(<optionalArguments>){...}
```

Constructors

- To initialize an object we can create a *constructor*.
- Constructors for a `class <name>` look like:

```
public <name>(<optionalArguments>){...}
```

```
public class Person{
    int age;
    public Person(){this.age = 0;}
```

Constructors

- To initialize an object we can create a *constructor*.
- Constructors for a `class <name>` look like:

```
public <name>(<optionalArguments>){...}
```

```
public class Person{
    int age;
    public Person(){this.age = 0;}
```

- Constructors are not necessary. Without one, an object will be created without setting or changing any of its variables.

Constructors and Overloading

- *this* is a key word that refers to the object that a method belongs to. It can be used to find variables and methods:

```
this.age
```

Constructors and Overloading

- *this* is a key word that refers to the object that a method belongs to. It can be used to find variables and methods:

```
this.age
```

- Objects can have more than one constructor as long as each one has different arguments. For example:

```
public class Person{
    int age;
    String firstname=""; String lastname = "";
    public Person(){this.age = 0;}
    public Person(int age){this.age = age;}
    public Person(int age, String firstname,
          String lastname){
        this.age = 0; this.firstname = firstname;
        this.lastname = lastname;}
```

More Overloading

- Other methods can be overloaded as well. For example:

```
public class Person{
    int age;
    String firstname=""; String lastname = "";
    boolean married = false;

    public Person(){this.age = 0;}
    public Person(int age){this.age = age;}
    public Person(int age, String firstname, String lastname){
        this.age = 0; this.firstname = firstname;
        this.lastname = lastname;}
    marriage(String newlastname){
        married = true; this.lastname = newlastname;}
    marriage(String newlastname, boolean append){
        married = true;
        if(append){
            this.lastname += "-" + newlastname;
        } else {
            this.lastname = newlastname;
        }
```

More Overloading

- By definition, *overloading* is when we have more than one way to run a method based on what arguments are passed in.

Review: Static vs. Non-Static

- For the time being we are going to keep everything *public*.
- However, we can start making objects. This lets us use *non-static*, instance variables and methods.

Review: Static vs. Non-Static

- For the time being we are going to keep everything *public*.
- However, we can start making objects. This lets us use *non-static*, instance variables and methods.
- So far we've used a few classes that require *instantiation* (e.g. Scanner, Strings, arrays)

Review: Static vs. Non-Static

- For the time being we are going to keep everything *public*.
- However, we can start making objects. This lets us use *non-static*, instance variables and methods.
- So far we've used a few classes that require *instantiation* (e.g. Scanner, Strings, arrays)
- *Static* methods and variables do not require instantiating a class.
- Non static methods *do* require instantiating an object, and each instance gets its own *copy* of all the non-static variables.

Creating an Object

```
public class Person{
    static int population = 0;
    int counter = 0;
    int ssn;

    public Person(int ssn){
       this.ssn = ssn;
       population++;
       counter ++;
    }

    public int getSSN(){
       return ssn;
    }
```

Instantiating an Object

```
public class PersonTest{
    public static void main(String[] args){
        Person p = new person();
        p.getSSN();
    }
}
```

Case study: Objects with a Static Variable

Create a static method in the Person class that returns the population variable. Next in the PersonTest class, create more than one person using a loop. How does the population change? Finally, do the same for the counter variable. Does it change?

Review: Recursion Time!

```
public static int f(int n){
    if(n > 0){return 1 + f(n-1)} else {return 0;};
}
```

Review: Recursion Time!

```
public static int f(int n){
    if(n > 0){return 1 + f(n-1)} else {return 0;};
}
```

- $f_{n+1} = f_n + 1$ for $f_0 >= 0$

  What's a recursive function that returns all of the positive even numbers?

Review: Recursion Time!

```
public static int f(int n){
    if(n > 0){return 1 + f(n-1)} else {return 0;};
}
```

- $f_{n+1} = f_n + 1$ for $f_0 >= 0$

  What's a recursive function that returns all of the positive even numbers?

- $f_{n+1} = f_n + 2$ for $f_0 = 2$

```
public static int f(int n){
    if(n > 0){return f(n-1)*2} else {return 1;};
}
```

Review: Recursion Time!

```
public static int f(int n){
    if(n > 0){return 1 + f(n-1)} else {return 0;};
}
```

- $f_{n+1} = f_n + 1$ for $f_0 >= 0$

  What's a recursive function that returns all of the positive even numbers?

- $f_{n+1} = f_n + 2$ for $f_0 = 2$

```
public static int f(int n){
    if(n > 0){return f(n-1)*2} else {return 1;};
}
```

- $f_{n+1} = f_n * 2$ for $f_0 = 1$

Review: Recursion Time!

```
public static double f(double n, int count){
    if(count >= 2){
        return f(n-1, count-1)/f(n-2, count-2)
    }
    else if(count ==1){
        return 2;
    }
    else {
        return 1;
    }
}
```

Review: Recursion Time!

```
public static double f(double n, int count){
    if(count >= 2){
        return f(n-1, count-1)/f(n-2, count-2)
    }
    else if(count ==1){
        return 2;
    }
    else {
        return 1;
    }
}
```

- $f_{n+1} = f_n/f_{n-1}$ for $f_1 = 2, f_0 = 1$

What is I/O

- I/O stand for input/output. We already have used input and output several times!
- *System.out* is an output stream. We have used it to print to the terminal (Command Prompt).

What is I/O

- I/O stand for input/output. We already have used input and output several times!
- *System.out* is an output stream. We have used it to print to the terminal (Command Prompt).
- *System.in* is an input stream. *Scanner* intercepts the input and parses it into *tokens*, like a Line, Int, or Double.

What is I/O

- I/O stand for input/output. We already have used input and output several times!
- *System.out* is an output stream. We have used it to print to the terminal (Command Prompt).
- *System.in* is an input stream. *Scanner* intercepts the input and parses it into *tokens*, like a Line, Int, or Double.
- There are many forms of input and output, like I/O with servers and clients over the internet, or file streams.

What is I/O

- I/O stand for input/output. We already have used input and output several times!
- *System.out* is an output stream. We have used it to print to the terminal (Command Prompt).
- *System.in* is an input stream. *Scanner* intercepts the input and parses it into *tokens*, like a Line, Int, or Double.
- There are many forms of input and output, like I/O with servers and clients over the internet, or file streams.

Motivation

- Most games have a function that saves the state of the game, and another that loads it.

Motivation

- Most games have a function that saves the state of the game, and another that loads it.
- We will need a similar function for our games, and we will use the following classes:

```
 java.io.BufferedOutputStream;
java.io.File;
java.io.FileOutputStream;
java.io.PrintWriter;
java.util.Date;
```

What do these classes do?

- `BufferedOutputStream`: Accumulates information to be written to your computer's memory, to make your program more efficient.
- `File`: Holds the address(`G:\MyProjects\file.txt`) and other properties for a file.

What do these classes do?

- `BufferedOutputStream`: Accumulates information to be written to your computer's memory, to make your program more efficient.
- `File`: Holds the address(`G:\MyProjects\file.txt`) and other properties for a file.
- `FileOutputStream`: writes data to a file.

What do these classes do?

- `BufferedOutputStream`: Accumulates information to be written to your computer's memory, to make your program more efficient.
- `File`: Holds the address(`G:\MyProjects\file.txt`) and other properties for a file.
- `FileOutputStream`: writes data to a file.
- `PrintWriter`: used for printing text to an output stream.

What do these classes do?

- `BufferedOutputStream`: Accumulates information to be written to your computer's memory, to make your program more efficient.
- `File`: Holds the address(`G:\MyProjects\file.txt`) and other properties for a file.
- `FileOutputStream`: writes data to a file.
- `PrintWriter`: used for printing text to an output stream.
- `Date`: this gives us the date.

What do these classes do?

- `BufferedOutputStream`: Accumulates information to be written to your computer's memory, to make your program more efficient.
- `File`: Holds the address(`G:\MyProjects\file.txt`) and other properties for a file.
- `FileOutputStream`: writes data to a file.
- `PrintWriter`: used for printing text to an output stream.
- `Date`: this gives us the date.

Review: Importing

You can import them by using:

```
import java.io.*;
import java.util.*;
```

Review: Importing

You can import them by using:

```
 import java.io.*;
import java.util.*;
```

Pop Questions: What does the * (star) do?

Review: Importing

You can import them by using:

```
 import java.io.*;
import java.util.*;
```

Pop Questions: What does the * (star) do?
Do we have to import?

Review: Importing

You can import them by using:

```
 import java.io.*;
import java.util.*;
```

Pop Questions: What does the * (star) do?
Do we have to import?
Where do imports go?

Some File Code

```
public File f;

public boolean openDiary() {
    if(!f.exists()) {
        try{
            if (!f.createNewFile()){
                return false;
            }
        } catch(IOException e){
            e.printStackTrace();
            return false;
        }
    }
```

Example Output Stream

```
try {
    this.f = new File("G:/MyProject/text.txt")
FileOutputStream fos = new
          FileOutputStream(f, true);
BufferedOutputStream bos = new
BufferedOutputStream(fos);
pw = new PrintWriter(bos);
} catch (FileNotFoundException e) {
e.printStackTrace();
}
Date d= new Date();
pw.write("Today's date is: " + d.toString()+ ".\n");
//pw.flush();
pw.close();
```

Before Lunch Challenge

You have two options. Either *create a diary application that appends (concatenates) and saves text that you put in with the date into a file on your flash drive*, or *start creating a file saving system for your RPG or other project. We will go over how to open files after lunch. Confirm that it is saving correctly by opening the file using notepad.*

Review

- So far we've talked about *scope* as being where a variable exists.

Review

- So far we've talked about *scope* as being where a variable exists.
- A variable exists in the *block* of code where it was *declared*.

Review

- So far we've talked about *scope* as being where a variable exists.
- A variable exists in the *block* of code where it was *declared*.
- A declared variable with the same name as another variable in a different block of code is not the same.

Review

- So far we've talked about *scope* as being where a variable exists.
- A variable exists in the *block* of code where it was *declared*.
- A declared variable with the same name as another variable in a different block of code is not the same.

```
while (true){
    int a = 5;
    do{
        int a = 6;
        System.out.println(a); //Prints 6
    }while(false);
    System.out.println(a); //Prints 5
}
```

What is Printed?

```
int x = 5;
int y = 7;
public void doStuff(int x) { //A 3 is passed in.
   y = 0;
   int y = 8;
   y = 7;
   System.out.print(y);
   int z = x;
   for (int j = 0; j < 4; j++) {
      x = this.y;
      z += x + 1;
   }
}
```

What is Printed?

```
int x = 5;
int y = 7;
public void doStuff(int x) { //A 3 is passed in.
   y = 0;
   int y = 8;
   y = 7; //y = 7
   System.out.print(this.y);
   int z = x;
   for (int j = 0; j < 4; j++) {
      x = this.y;
      z += x + 1;
   }
}
```

What is Printed?

```
int x = 5;
int y = 7;
public void doStuff(int x) { //A 3 is passed in.
   y = 0; //this.y = 0
   int y = 8;
   y = 7;
   int z = x;
   System.out.print(z);
   for (int j = 0; j < 4; j++) {
      x = this.y;
      z += x + 1;
   }
}
```

What is Printed?

```
int x = 5;
int y = 7;
public void doStuff(int x) { //A 3 is passed in.
   y = 0;
   int y = 8;
   y = 7;
   int z = x; //z = 3
   System.out.print(this.x);
   for (int j = 0; j < 4; j++) {
      x = this.y;
      z += x + 1;
   }
}
```

What is Printed?

```
int x = 5;//this.x = 5
int y = 7;
public void doStuff(int x) { //A 3 is passed in.
   y = 0;
   int y = 8;
   y = 7;
   int z = x;
   for (int j = 0; j < 4; j++) {
      x = this.y;
      System.out.print(x);
      z += x + 1;
   }
}
```

What is Printed?

```
int x = 5;
int y = 7;
public void doStuff(int x) { //A 3 is passed in.
   y = 0;
   int y = 8;
   y = 7;
   int z = x;
   for (int j = 0; j < 4; j++) {
      x = this.y; //x = this.y = 0
      z += x + 1;
   }
   System.out.print(j);
}
```

What is Printed?

```
int x = 5;
int y = 7;
public void doStuff(int x) { //A 3 is passed in.
   y = 0;
   int y = 8;
   y = 7;
   int z = x;
   for (int j = 0; j < 4; j++) {
      x = this.y;
      z += x + 1;
   } //Error!
   System.out.print(z);
}
```

What is Printed?

```
int x = 5;
int y = 7;
public void doStuff(int x) { //A 3 is passed in.
    y = 0;
    int y = 8;
    y = 7;
    int z = x;
    for (int j = 0; j < 4; j++) {
        x = this.y;
        z += x + 1;
    }  //z = 7
}
```

Public vs Non-Public Classes

- Public: Visible to all classes in all packages. The name of the class must be the same as the name of the file.
- No Modifier: Visible only to classes in the same package. The name of the class does not have to be the same as the name of the file.

Public, Protected, and Private Variables and Methods

- Public: The variable or method can be seen by all classes in all packages.
- Private: The variable can only be seen by the class.
- Protected: The variable can be seen by all classes in the same package, and all *subclasses* (we will see this tomorrow).
- No Modifier: The variable can be seen by all classes in the same package, but not *subclasses* (we will see this tomorrow).

Easy Table for Memory

Modifier	Class	Package	Subclass	Other Packages
public	Y	Y	Y	Y
protected	Y	Y	Y	N
no modifier	Y	Y	N	N
private	Y	N	N	N

Reading from a file

```
public String read(){
   try {
      BufferedReader reader = new
            BufferedReader(new FileReader(f));
      String line;
      String s = "";
      while ((line = reader.readLine()) != null) {
         s += line + "\n";
      }
      reader.close();
      return s;
   } catch (Exception e) {
      return null;
   }
}
```

Extending the Diary Example

Online is a copy of the source code for the diary example. Download it, and try it out. Depending on what version you use there's also some GUI classes begin used! Update your game or project such that it loads information from a file.

Code for an NPC

```
import world.*
public class SimpleNPC extends SimpleSolid{
   private boolean right = false;

   public SimpleNPC() {
      //A red 16x16 square.
      this.setImage(0xFFFF0000, 16, 16);
   }

   public char id() {
      return 'P';
   }
```

Code for an NPC

```
    public void collision(SimpleObject s) {
        right = !right;
    }

    public void update() {
        if (right) {
            this.moveCell(1, 0, 10 ,true);
        } else {
            this.moveCell(-1, 0, 10 ,true);
        }
    }
}
```