COMP1409: Introduction to Software Development I

Mike Mulder (mmulder10@bcit.ca)

Week 7

Agenda

- Quiz
  - Quiz 6
  - Review Answers
- Assignment 2
- Lab 6B
- Week 6 Review
- Lesson 7
  - References
  - null
  - this
  - Debugging
- Lab 7 – In Class Only

Quiz 6

Closed book, laptop, phone, etc.

You have a maximum of 20 minutes to complete

Raise your hand when you are done, and I will retrieve your paper

We will review the answers afterwards

Short Term Logistics

Week	Date	Topics	Comments
5	Oct. 6	Thanksgiving – No Class	
6	Oct. 13	Arithmetic operators Overloading Switch/case	Quiz 4
7	Oct. 20	Composition (object interaction and external method calls)	Quiz 5, Assignment 1 Due
8	Oct. 27	References; identity versus equality null (again) this Debugging techniques	Quiz 6
NOTE: Course Withdrawal Deadline Please inform your instructor that you are dropping this course. You must also fill out and submit the **'REQUEST TO WITHDRAW FROM A PART-TIME STUDIES COURSE'** before session 8 or else you will receive a failing grade on your academic record**.**			
9	Nov. 3	Arrays while loops	Quiz 7
10	Nov. 10	**Remembrance Day – No Class**	
11	Nov. 17	More arrays for loops	Quiz 8, **Assignment 2 Due**
12	Nov. 24	ArrayList class Enhanced for (foreach) loop	Quiz 9

Assignments

Assignment 1

- Marked and results posted to D2L
- Good results (90+%) except for those that didn’t run the unit tests

Assignment 2

- Posted to D2L
- Due Nov. 16, 2018
- Similar to Assignment 1 but more thorough unit tests. Must run and pass the unit tests!

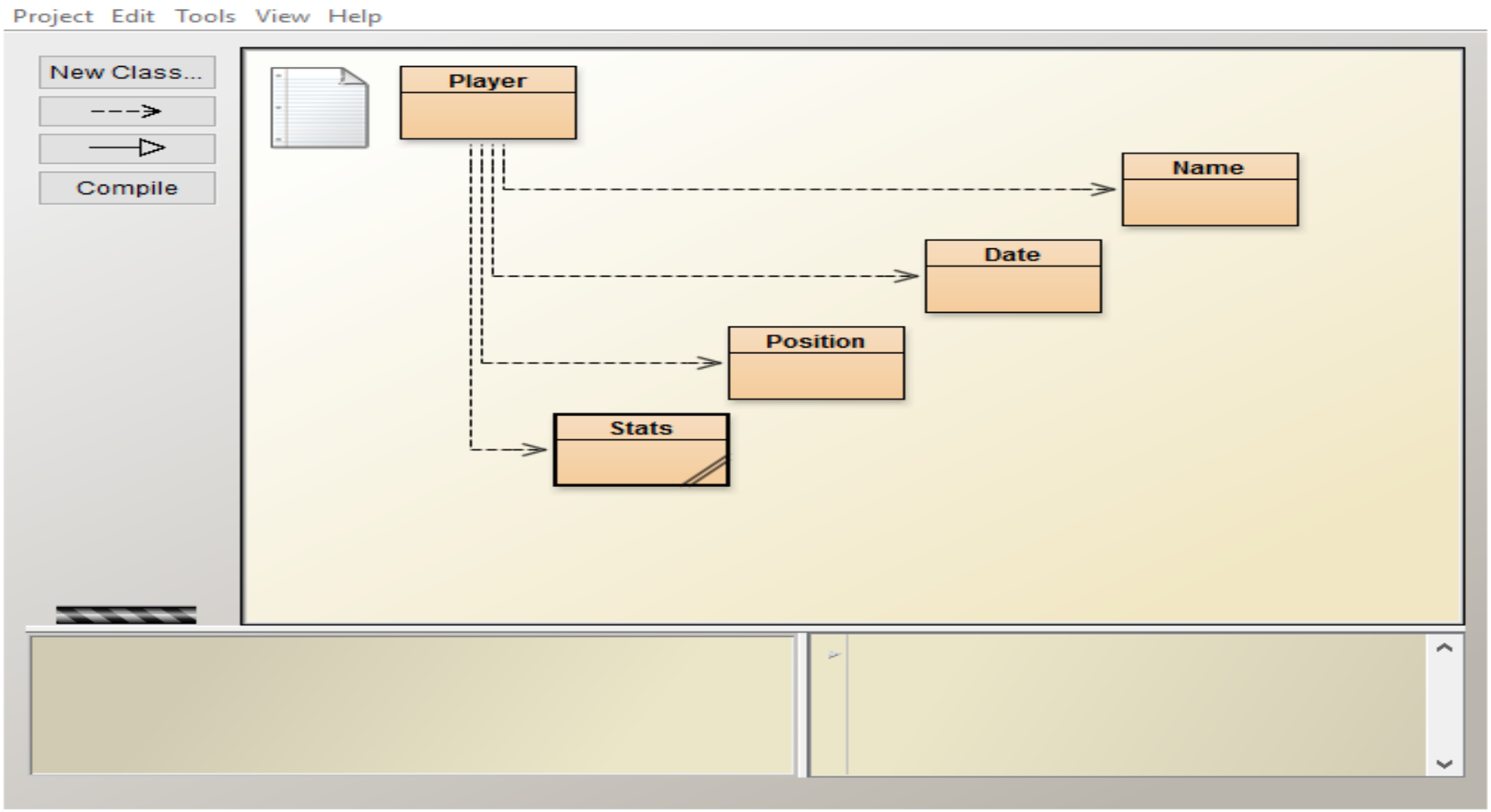
Note: Assignment 3 will be more complex – two classes and a composition relationship

Lab 6B

Will be marked this afternoon.

Sample Solution will also be posted to D2L this afternoon.

What do we call this type of relationship?



Composition

Classes that contain (or are composed of) instances of other classes.

This is one of the fundamental principles of Object Oriented Programming. It enables re-use and test-ablility.

Most of the software you build in real-life is composed of other classes built by your team or 3rd parties (i.e., open source)

Modularization and Abstraction

Player is **modular** – composed of name, position, stats, and date classes

```
public class Player
{
    private Name        name;
    private Position    position;
    private Stats       stats;
    private Date        dateDrafted
```

Player uses **abstraction** – we work with the components without worrying about their details

The getFullName method on the Player class is using a getFullName method on the (private) name object:

```
public String getFullName()
{
    return name.getFullName();
}
```

We don't need to know the details of how the Name class works, but we do need to know it's public interface. JavaDoc is a good starting point for this.

Types of Composition

- True Composition – Child classes that exist only inside the parent class

```
public Player(String firstName, String lastName) {
    name = new Name(firstName, lastName);
    position = new Position(FORWARD);
    …
}
```

- Aggregation – Child classes that exist outside the parent class

```
Name name = new Name(“Bill”, “Smith”);
Position position = new Position(DEFENSE);
…
Player myPlayer = new Player(name, position, stats, dateDrafted);
```

Classes as Data Types

- Anything that is not a primitive data type in Java, is an object type
- Note: We have worked with String objects already, which is an object type built into Java

```
private Name        name;
private Position    position;
private Stats       stats;
private Date        yearDrafted
```

Session 7 Learning Outcomes

- References (identity vs equality)
- null
- this
- Debugging techniques

Primitive Types vs Objects

Primitive

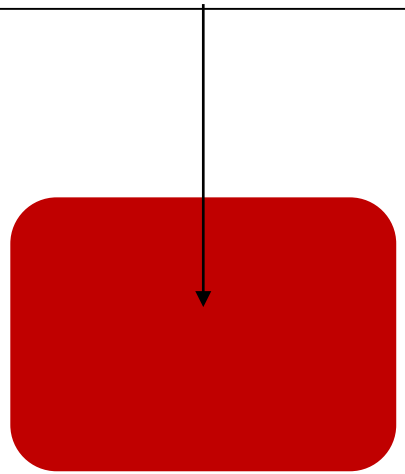
```
int i;
```

value (like 32)

Object Reference

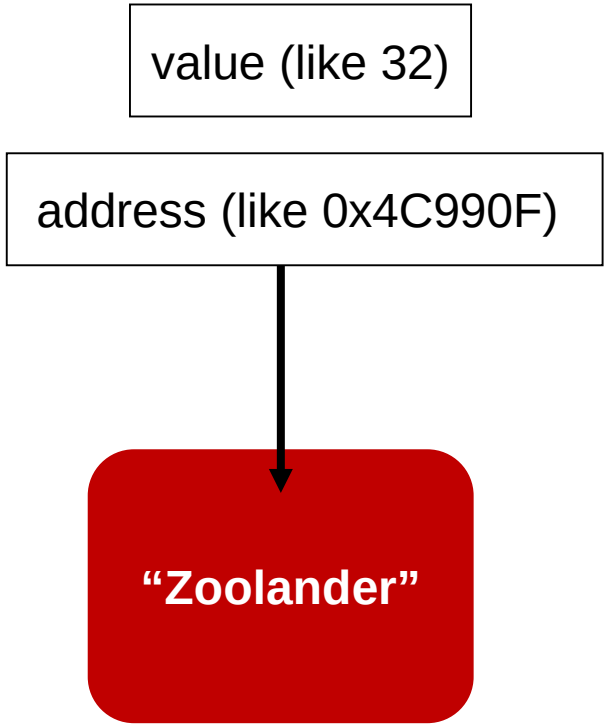
```
SomeObject obj;
```

address (or its hash) (like 0x4C990F)



Primitive Types vs Objects

```
public class Person
{
    private int yearBorn;

    private String lastName;

    // …
}
```



Primitive Types

```
int a = 10;
```

10

```
int b = 0;
```

0

```
b = a;
```

```
a
```

10

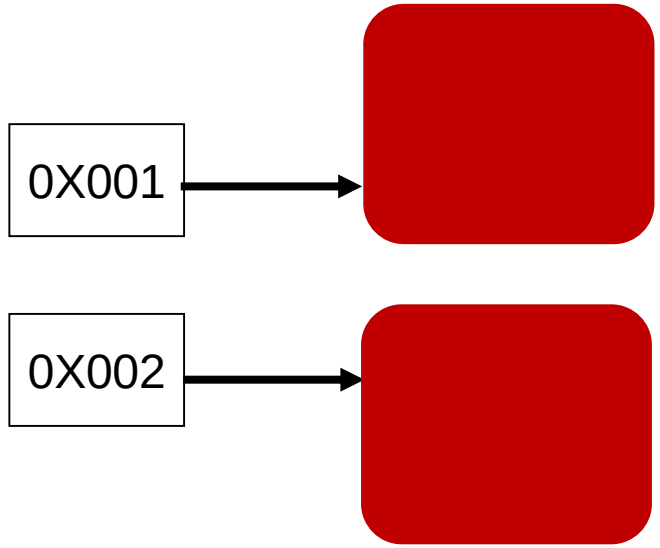
```
b
```

10

At the end of this, a and b are two unrelated ints

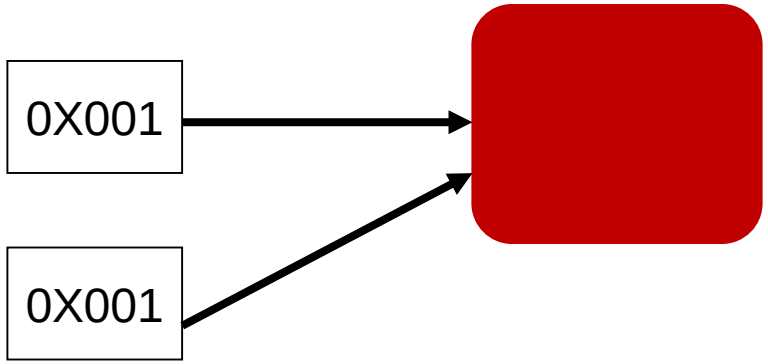
Object References

```
Guppy a = new Guppy();

Guppy b = new Guppy();

b = a;
```



a

b



Identity vs. Equality

```
if(input == "bye") {
    ...
}
```

Tests identity

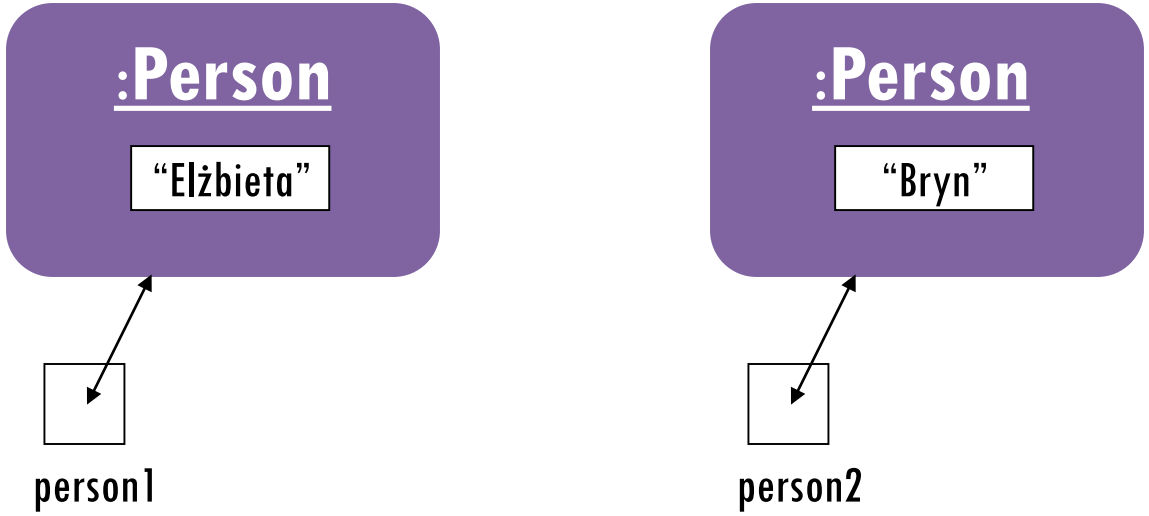
```
if(input.equals("bye")) {
    ...
}
```

Tests equality

Always use .equals to test whether Strings (or other objects) are equal; do not use ==.

Identity vs. Equality

Object references

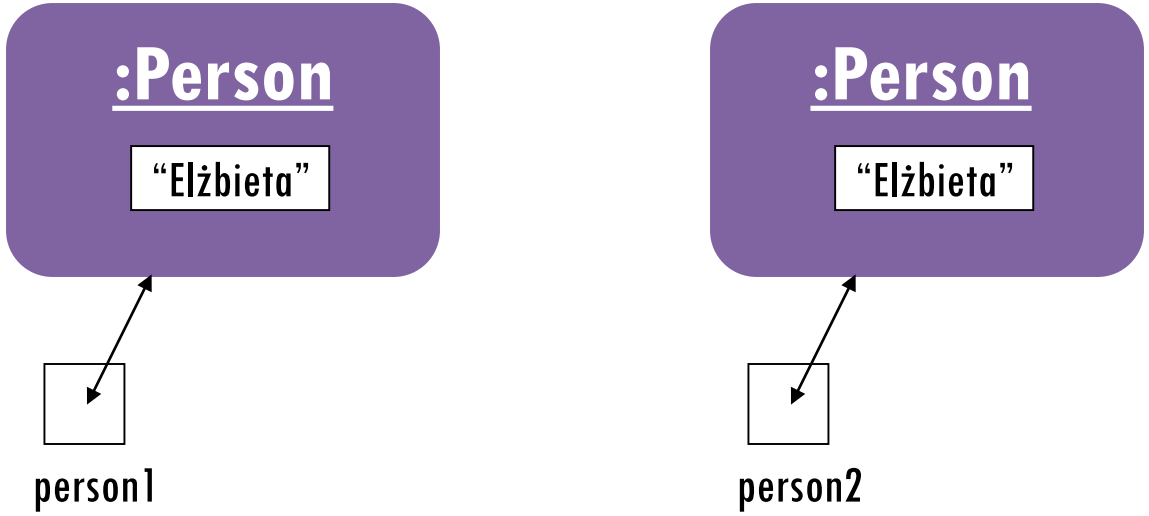


person1 == person2?

NO!

Identity vs. Equality

Object references

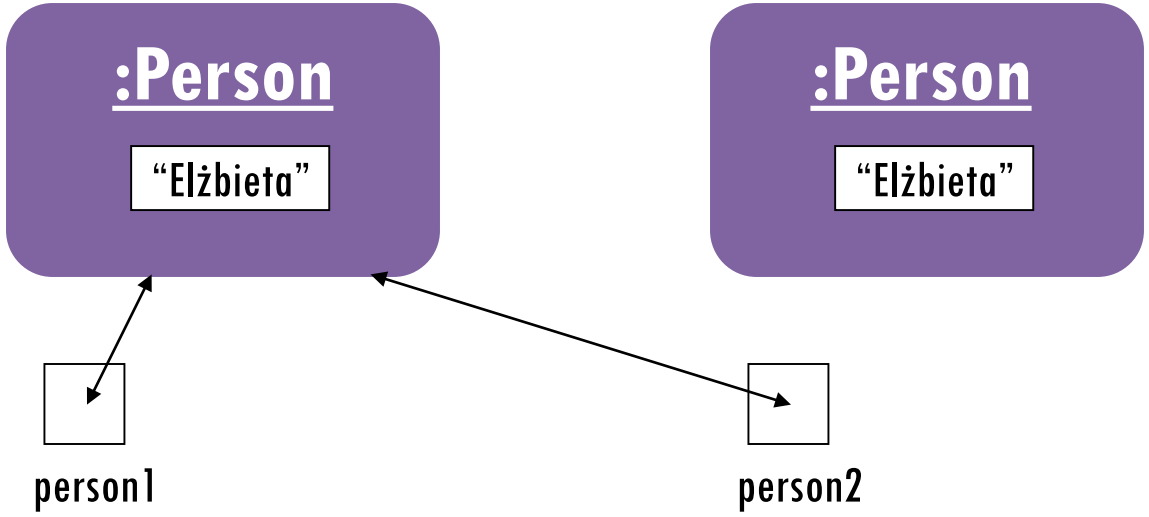


person1 == person2?

NO!

Identity vs. Equality

Object references

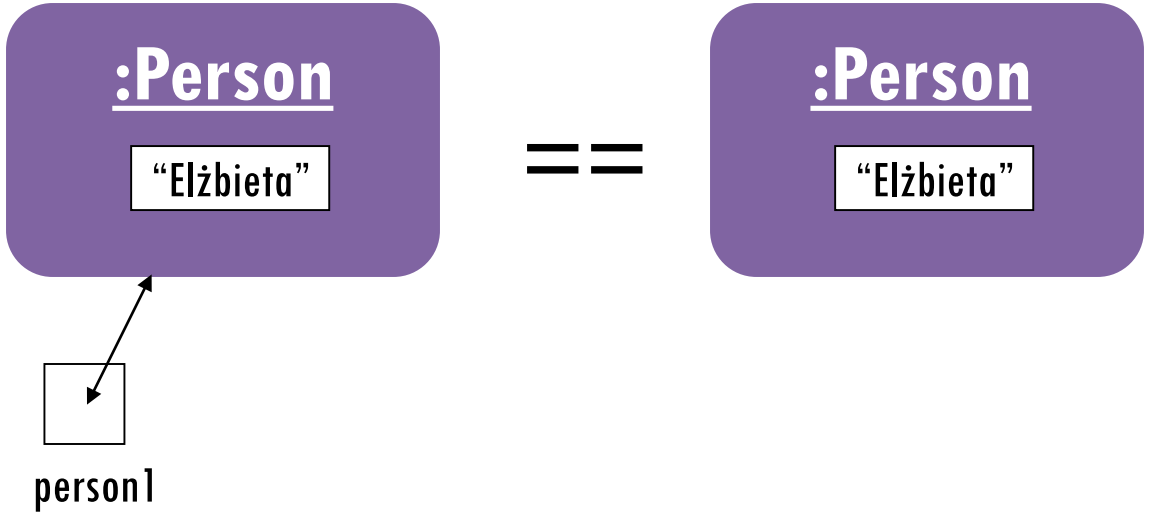


person1 == person2?

YES

Identity vs. Equality

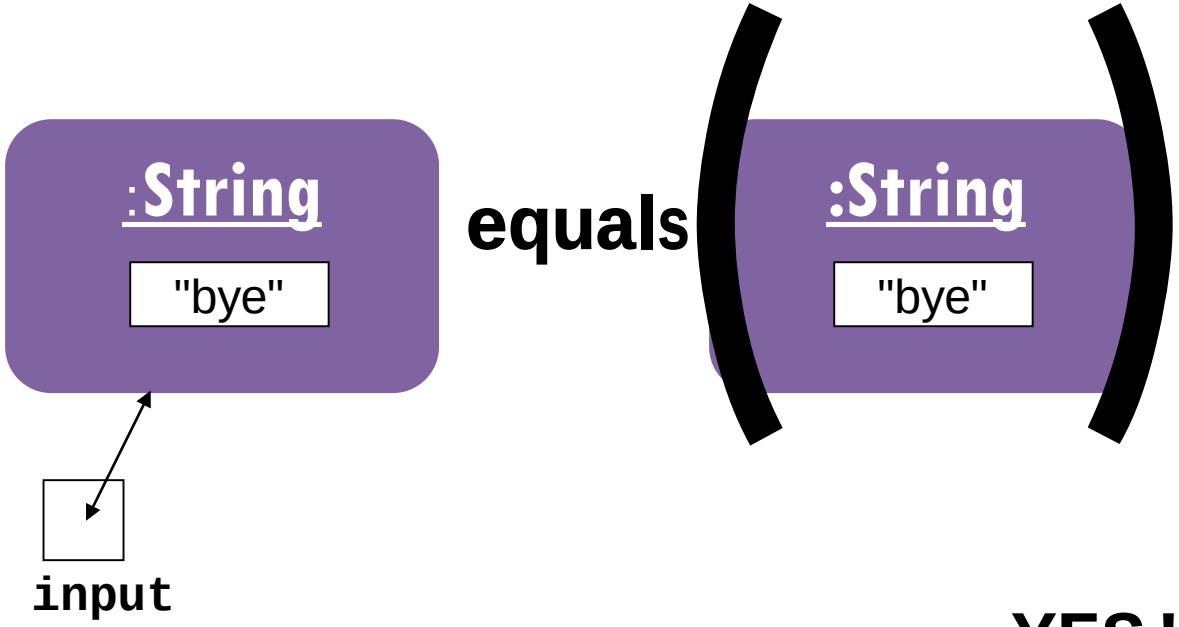
```
if(name == "Elżbieta") {
    ...
}
```



- Unpredictable. The compiler may merge identical String literals in the program code
- The result is reference equality for apparently distinct String objects
- But this cannot be done for identical strings that arise outside the program's code, e.g., from user input

Identity vs. Equality

```
if(input.equals("bye”)) {
    ...
}
```



YES!

Equality for Objects

```
if (object1 == object2) // Compares object references

if (object1.equals(object2)) // Compares object equality
```

But the default equals method for a class is the same as ==

Your Classes

If you need to be able to compare instances of your class, you need to implement an equals() method. Note: This is an advanced topic!

Example ClockDisplay:

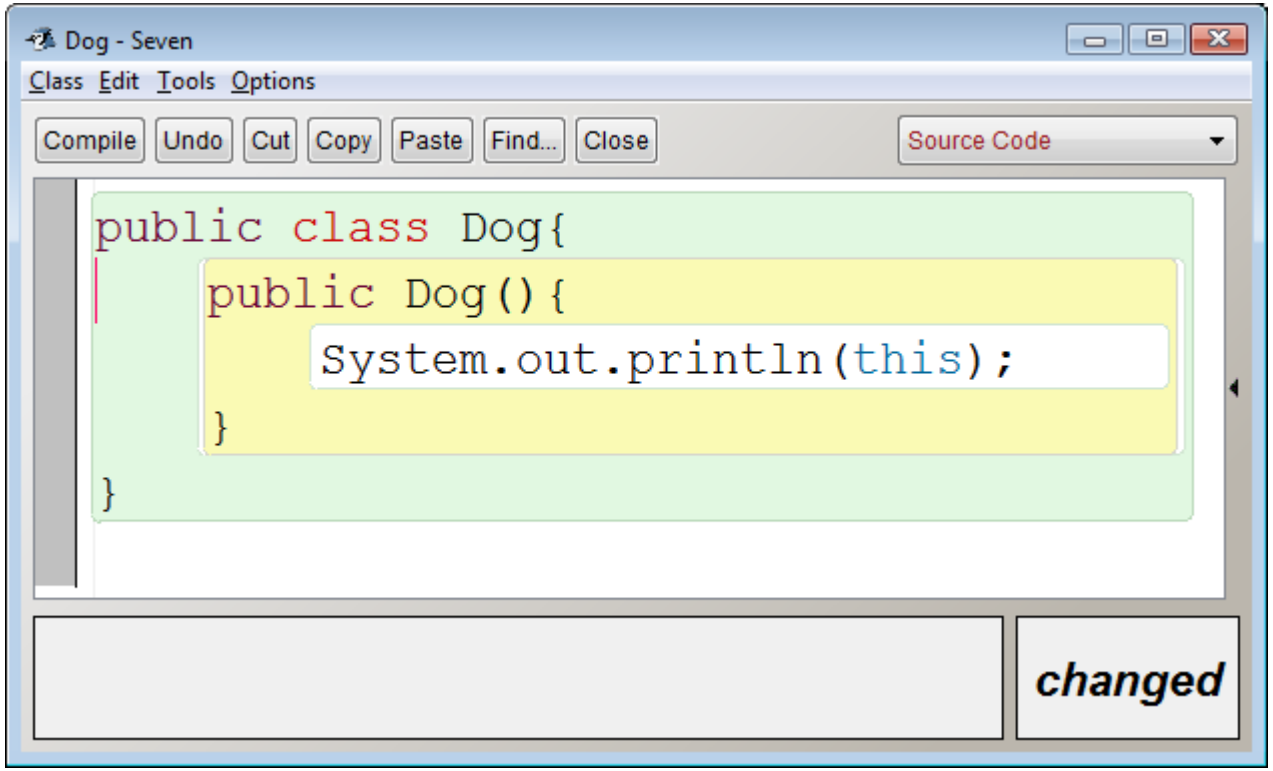
```
   public boolean equals(Object o) {

      if (o instanceof ClockDisplay) {
         ClockDisplay c = (ClockDisplay) o;
         if (this.hours == c.hours &&
             this.minute == c.minutes) return true;
       }
       return false;
   }
```

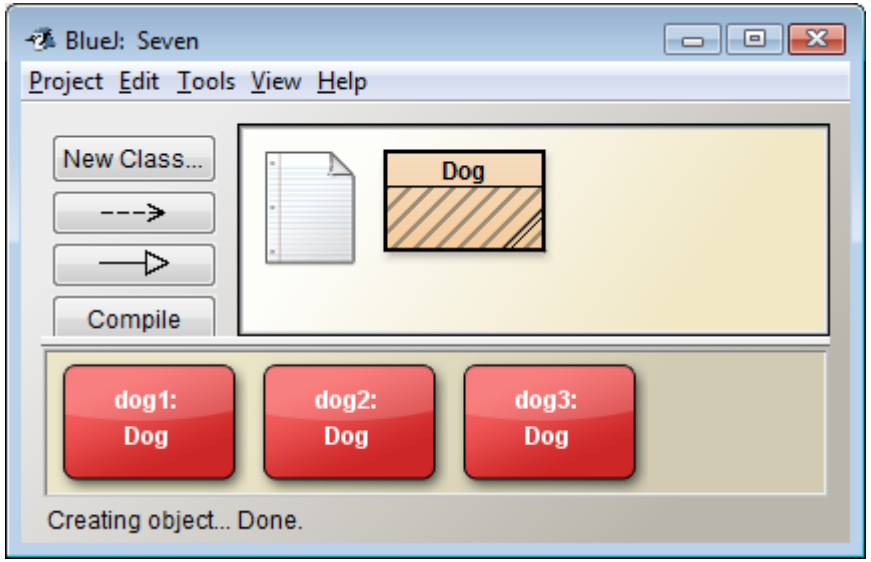
null

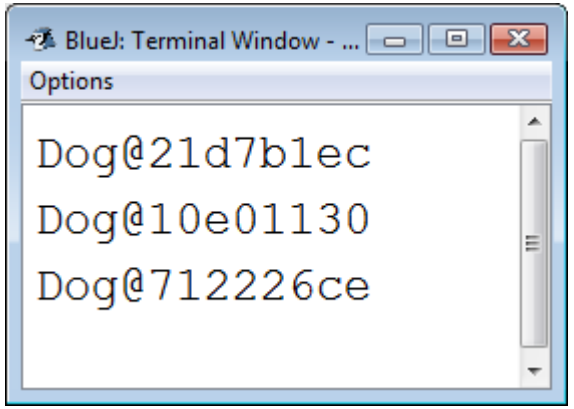
- null is a case sensitive keyword in Java: `null`
- Object fields are initialized to `null` by default
- A `null` value indicates an unset reference (a reference to nothing)
- We cannot assign `null` to primitive types
- We can always pass `null` as parameters to methods that expect object references
- We can always return `null` from methods that have object-reference return types.

```
if(firstName == null)          // checking the address; correct way to do this

int number = null; // WILL NOT COMPILE!

myLittleGuppy.setName(null); // COMPILES
```

Keyword **this**







Keyword **this**

- **this** refers to the same object in which we are executing code
- Definition: “this” is a reference to the current object
- Internal method call: A method invoked on ‘this’ object
- Common use: when we use the same name for a method parameter and an instance variable.

Error

```
public class Dog{
    private String          name;
    private int             yearOfBirth;

    public Dog(String name, int yearOfBirth){
        name        = name;                     // assigns parameter value to itself
        yearOfBirth = yearOfBirth;              // assigns parameter value to itself

        System.out.println(getDetails());
        System.out.println(this.getDetails());          // people usually don’t do this
    }

    public String getDetails(){
        return "I am a Dog named " + name + ". Born in " + yearOfBirth;
    }
}
```

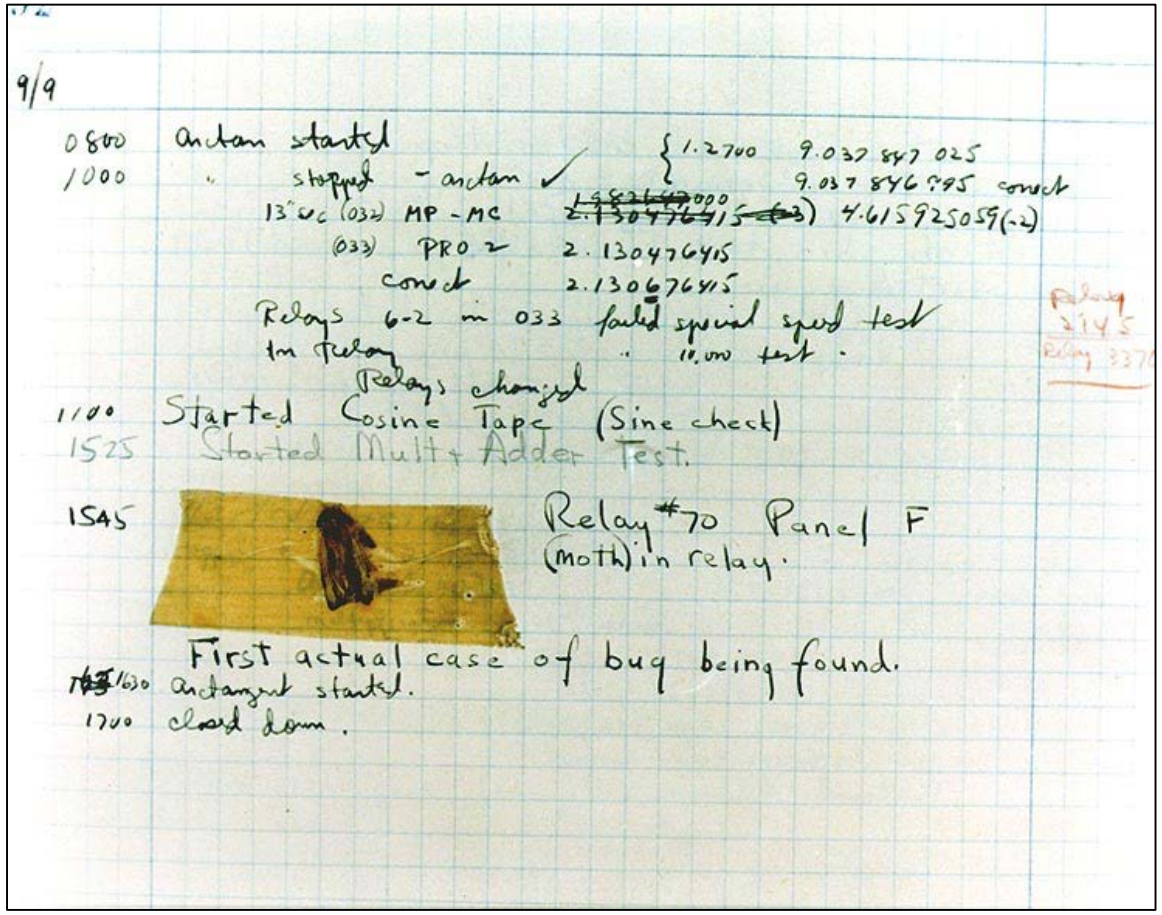
Keyword **this**

```
public class Dog{
    private String          name;
    private int             yearOfBirth;

    public Dog(String name, int yearOfBirth){
        this.name       = name;
        this.yearOfBirth = yearOfBirth;

        System.out.println(getDetails());
        System.out.println(this.getDetails());      // people usually don't do this
    }

    public String getDetails(){
        return "I am a Dog named " + name + ". Born in " + yearOfBirth;
    }
}
```

Debugging

- Debugging involves tracing through your code to locate and fix an error
- Before you can debug a piece of code you must know exactly what it should do when invoked
- The best ‘debugger’ is your brain!

The First Bug



Computer log entry from the Mark II with a moth taped to the page
Courtesy of the Naval Surface Warfare Center, Dahlgren, VA., 1988. - U.S. Naval Historical Center Online Library Photograph

Debugging Techniques

Some techniques for debugging:

1. **Hand Tracing**: use pencil and paper to ‘trace your way’ through the code and keep track of the contents of the variables
2. **Print statements**: Use `System.out.println` statements inserted in code tell you the current state of an object
3. **Inspect**: use BlueJ’s *Inspect* to examine the object
4. **Debugger**: use the BlueJ debugger (**Demo!**)

Print Statements

- Use system.out.println
- Review the output of the print statements in the terminal after the program runs
- Typical strategies:
  - Put print statements all over your program if you don’t know where the problem is. Isolate the problem between consecutive print statements
  - Put print statements in a method, if you know the problem is in that method. Print out the values of specific variables to see if they are as expected as the program runs.

What is a Debugger

- A tool that lets you set “breakpoints” at a line of code.
- When you run your code with the debugger, it will stop at the breakpoint(s)
- At the breakpoint, you can inspect the current state of the program (i.e., instance variables, local variables)

Debugger Breakpoints

When the debugger reaches a breakpoint, typically you can:

- Terminate the program
- Continue running the program
- Step over individual lines of code (in sequential order)
- Step into a line of code (if it has a method call).

Debugger Strategies

- Put a breakpoint at the top level of your code and step through it until you find the problem
- If you know or suspect the problem is in a specific method, put your breakpoint in that method and step through it when your program calls the method. Careful if you method gets called a lot.

BlueJ Debugger

- Stack trace
- Step
- Watch windows (static, instance, local variables)