CSCI 161
Introduction to Computer Science

Department of Mathematics
and Computer Science

Lecture 3
Data Types
Primitives vs. Classes

Outline

- Data Types
  - Primitives: ints, doubles, chars, booleans, etc.
    - Type Casting
  - Classes as data types
    - Object References and the null reference
    - Object Aliasing
    - Object Equality: Content Equality
- Conclusion

Data Types in Java

- **We know this:** Every unit of data (local variables, instance variables, input parameters) is associated with a *Data Type.*

- Java supports two categories of *Data Types*:
  - **1) Primitives:** int, double, boolean, char, long, byte, short, float

```
int secretNumber;     double y;     boolean gameOver;
```

Data Types in Java (2)

- **We know this:** Every unit of data (local variables, instance variables, input parameters) is associated with a *Data Type.*

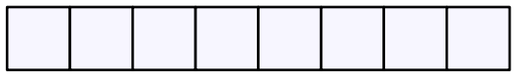
- Java supports two categories of *Data Types*:
  - **1) Primitives:** int, double, boolean, char, long, byte, short, float

```
int secretNumber;     double y;     boolean gameOver;
```

  - **2) Classes are also data types!** String, Random, Circle, Clock, GuessingGame...

```
Random rng;       Circle eyeBall;
```

Space Requirements of Primitive Types

- Basic unit of space in computers
  - One bit is just a 0 or a 1
  - One byte = sequence of 8 bits: 

Data Type	Space	Range	Fractions?	Op Speed
`double`	8 bytes	Highest	Precise up to **13th** digit after decimal	*Slowest*
`float`	4 bytes		Precise up to **6th** digit after decimal	*Slow*
`long`	8 bytes		does not support	Fast
`int`	4 bytes		does not support	Fast
`short`	2 bytes		does not support	Fast
`byte`	1 byte		does not support	Fast
`char`	2 bytes		n/a	Fast
`boolean`	1 byte	Lowest	n/a	Fast

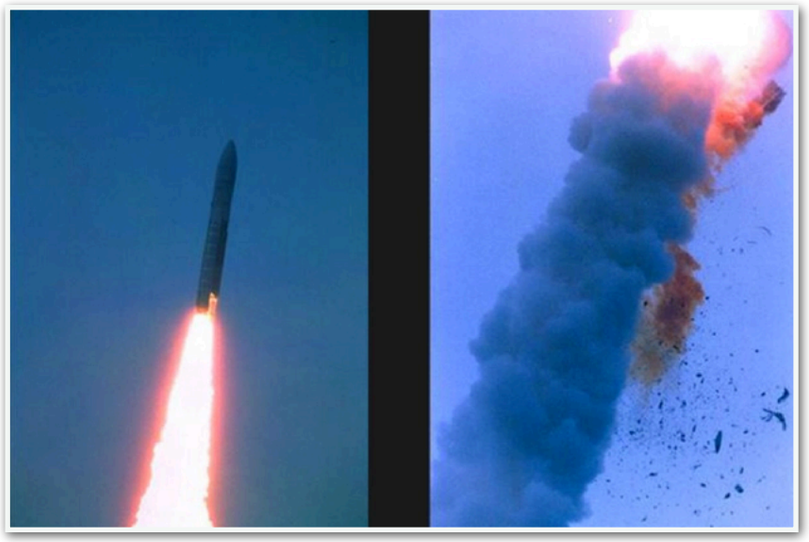
Range of Some Primitive Types

- Legal range of primitive types:
  - double $[-1.798 \times 10^{308}, 1.798 \times 10^{308}]$
  - float $[-3.4 \times 10^{38}, 3.4 \times 10^{38}]$
  - long $[-2^{63}, 2^{63} - 1]$
  - int $[-2147483648, 2147483647]$
  - short $[-32768, 32767]$
  - byte $[-128, 127]$
  - char `'a'`, `'A'`, `'b'`, `'B'`, `'#'`, `'\n'`, `'@'` ... and so on
  - boolean `true`, `false`

Integer Overflow

- Say we have an **`int`** variable...
  - What's the largest or smallest number it can store?
  - Is there a limit?

```
// Try this
int x = 2147483647;
x++;
System.out.println(x);
```



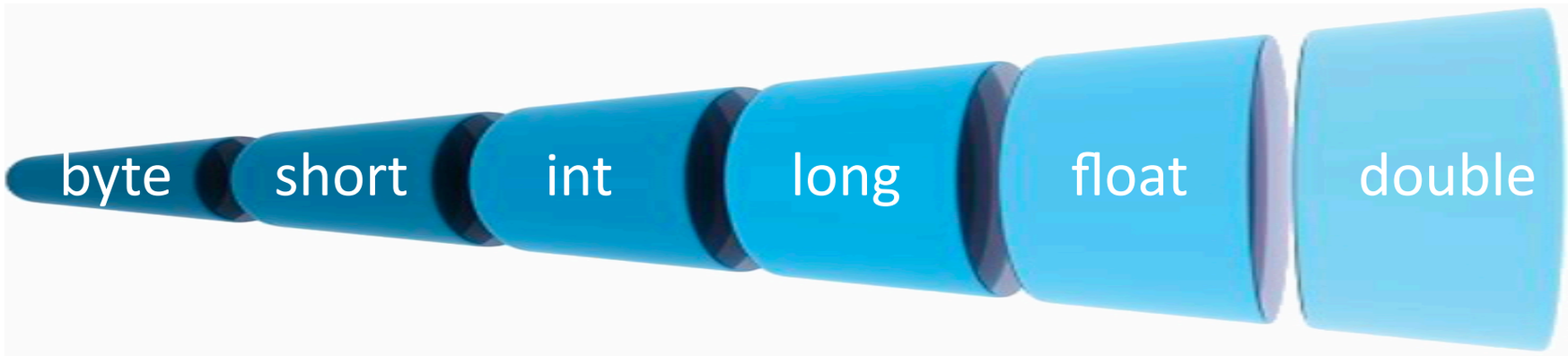
Ariane 5 Rocket

Therac 25

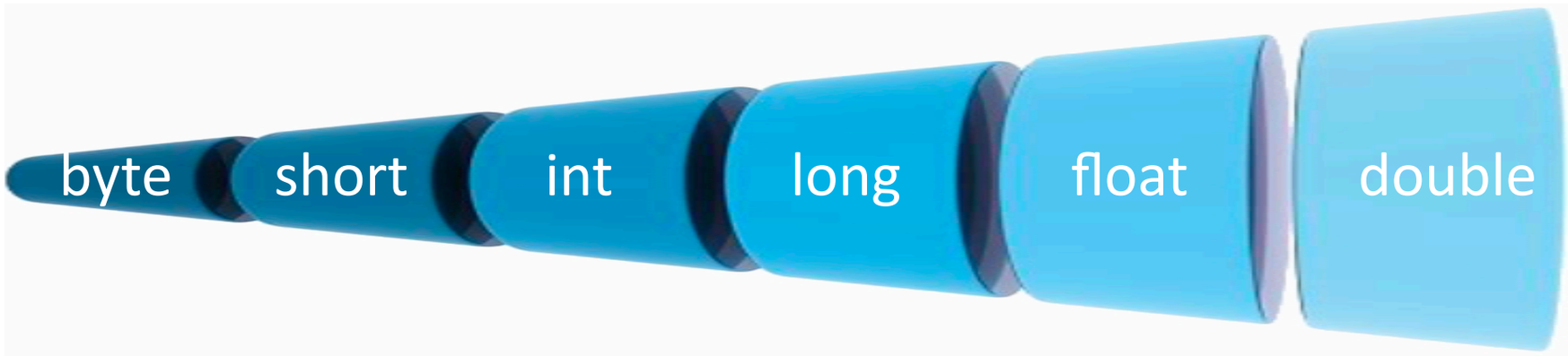
Outline

- Data Types
  - Primitives: ints, doubles, chars, booleans, etc.
    - Type Casting
  - Classes as data types
    - Object References and the null reference
    - Object Aliasing
    - Object Equality: Content Equality
- Conclusion

Relative Ranges of Numerical Types



Relative Ranges of Numerical Types

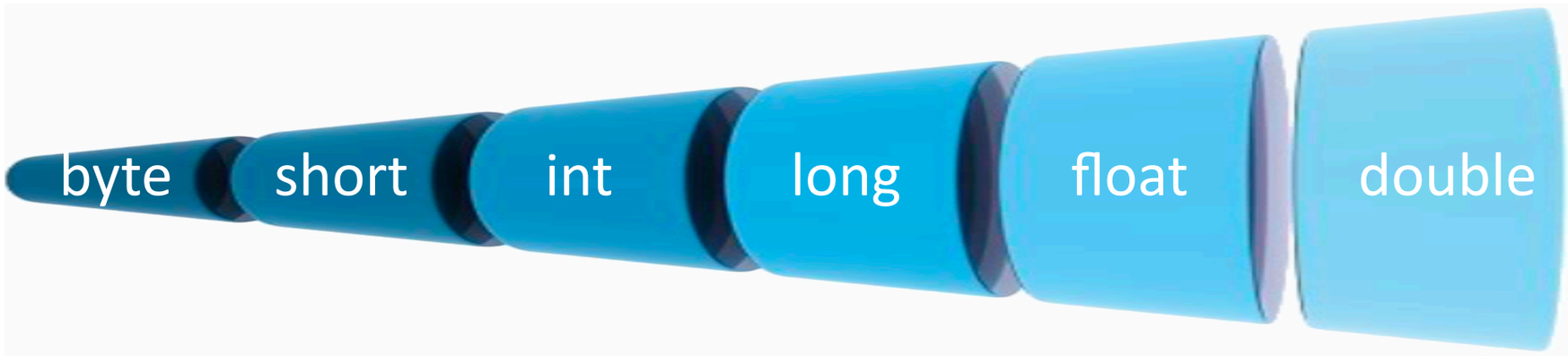


- An **int** variable can store a **short.** A **double** variable can store an **int**

  and so on...

```
int x = 64;

double y = x;    // This works! y is 64.0
float z = x;     // So is this! z is also 64.0
```

Relative Ranges of Numerical Types



- What about the in other direction? Can an **int** store a **double**?

```
double x = 64.825;
int smallerNumber = x;      // Nope: int is too "narrow" to store a double
```

```
double w = 80.0;
int smallerNumber = w;      // Nope: even though 80.0 doesn't have a fraction!
```

Important: Type Casting (Conversion)

- To convert one numeric data type to another, we need to *Type Cast*
- Syntax: `(target-data-type) expression`

expression is any Java expression (e.g. a method call or an algebraic expression) that results in a wider data type.

- What's stored in **x** after the following code segment runs?

```
int x;
double y = 90.25;
x = (int) y;            // x holds 90
```

```
int x;
double y = 90.25;
x = (int) y * 100;    // x holds 9000
```

Casting to "Narrow" a Value

- "Narrowing" rounding down to the nearest whole number

- Example: Edgar in your game "levels up" after obtaining 150 XP (experience points). What level is your character if they have 436.7 XP?


```
double xp = 436.7;
int level = (xp / 150);  // But this won't compile (level is too narrow)
System.out.println("Edgar is on level: " + level);
```

Casting to "Narrow" a Value (Correct)

- "Narrowing" rounding down to the nearest whole number

- Example: Edgar in your game "levels up" after obtaining 150 XP (experience points). What level is your character if they have 436.7 XP?


```
double xp = 436.7;
int level = (int) (xp / 150);  // Corrected!
System.out.println("Edgar is on level: " + level);
```

Casting to "Widen" a Value

- "Widening": Need to get an integer to support fractions.
  - (e.g., a need to convert an **int** to **double**.)

- Example: Calculate the average between your first two exams.

```
int exam1 = 74;
int exam2 = 93;
double avg = (exam1 + exam2) / 2;  // This compiles, but you get the wrong answer!
                                   // 83.0 gets stored but it should've been 83.5
```

Casting to "Widen" a Value (Correct)

- Example: Calculate the average between your first two exams.

```
int exam1 = 74;
int exam2 = 93;
double avg = (exam1 + exam2) / 2;  // 83.0 is stored in avg! (Incorrect!)
```

```
int exam1 = 74;
int exam2 = 93;
double avg = (double) (exam1 + exam2) / 2;  // 83.5 is stored in avg! (Correct)
```

```
int exam1 = 74;
int exam2 = 93;
double avg = (exam1 + exam2) / (double) 2;  // 83.5 is stored in avg! (Correct)
```

Mixing Types (Implicit Widening)

- There's an even easier way to *widen* types.

- If there are terms with mixed types in the same expression,
  - The resulting type takes on the *widest type* appearing in the expression!

```java
int exam1 = 74;
int exam2 = 93;
double avg = (exam1 + exam2) / 2.0;  // 83.5 is stored in avg! (Correct)
```

Data Type Exercises

```
int    iResult, num1 = 25,   num2 = 40,   num3 = 17, num4 = 5;
double dResult, val1 = 17.0;

iResult = num1 / num4;  // #0
> 5
dResult = num1 / num4;  // #1
> 5.0
iResult = num3 / num4;  // #2
> 3
dResult = num3 / num4;  // #3
> 3.0                        <---- Tricky. Know why it's 3.0 and not 3.4)
iResult = num3 % num4;  // #4
> 2
dResult = val1 / num4;  // #5
> 3.4
dResult = (double) num1 / num2;   // #6
> 0.625                                <---- Tricky. Know why it's not 0.0)
dResult = (double) (num1 / num2);  // #7
> 0.0
iResult = (int) ((double) 80 / num2);  // #8
> 2
```

Outline

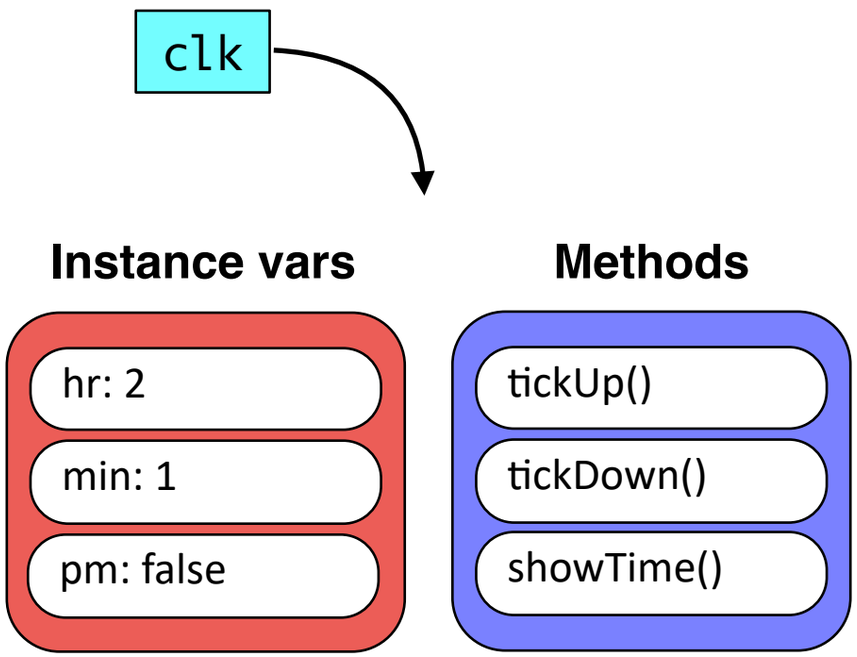
- Data Types
  - Primitives: ints, doubles, chars, booleans, etc.
    - Type Casting
  - Classes as data types
    - Object References and the null reference
    - Object Aliasing
    - Object Equality: Content Equality
- Conclusion

Primitive Type Variables vs. Object Variables

- Object variables store *"pointers" or "references" (→) to an existing object, like a Clock, TicketMachine, Triangle, ...*

- For example, this code declares a Clock type variable named **clk**.
  - Then it *instantiates* (using the **new** keyword) a concrete Clock object (2:00 am) and points to it.

```
Clock clk = new Clock(2,1,false);
```

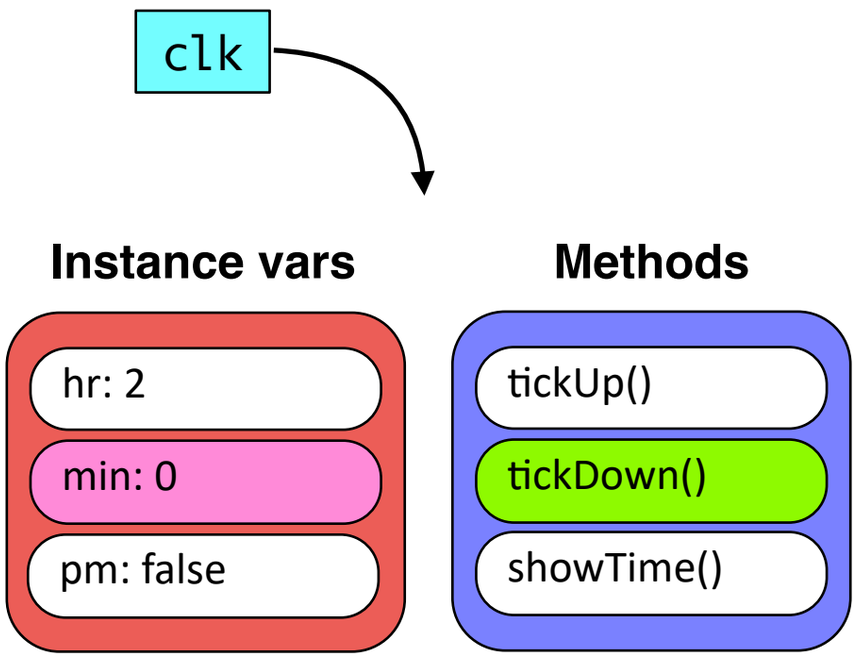


Primitive Type Variables vs. Object Variables

- Object variables store *"pointers" or "references" (→) to an existing object, like a Clock, TicketMachine, Triangle, ...*

- For example, this code declares a Clock type variable named **clk**.
  - Then it *instantiates* (using the **new** keyword) a concrete Clock object (2:00 am) and points to it.

```
Clock clk = new Clock(2,1,false);
clk.tickDown();
```

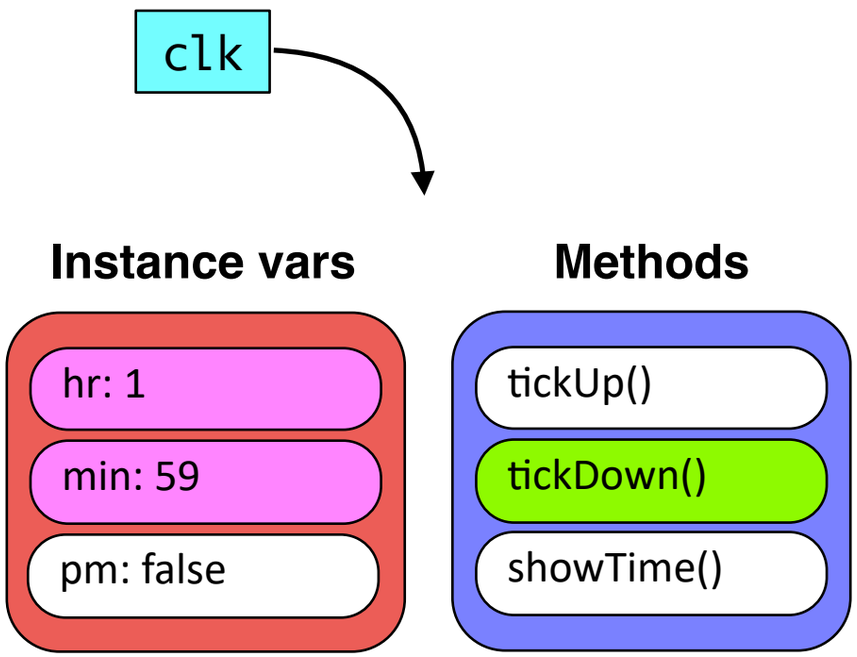


Primitive Type Variables vs. Object Variables

- Object variables store *"pointers" or "references" (→) to an existing object, like a Clock, TicketMachine, Triangle, ...*

- For example, this code declares a Clock type variable named **clk**.
  - Then it *instantiates* (using the **new** keyword) a concrete Clock object (2:00 am) and points to it.

```
Clock clk = new Clock(2,1,false);
clk.tickDown();
clk.tickDown();
```

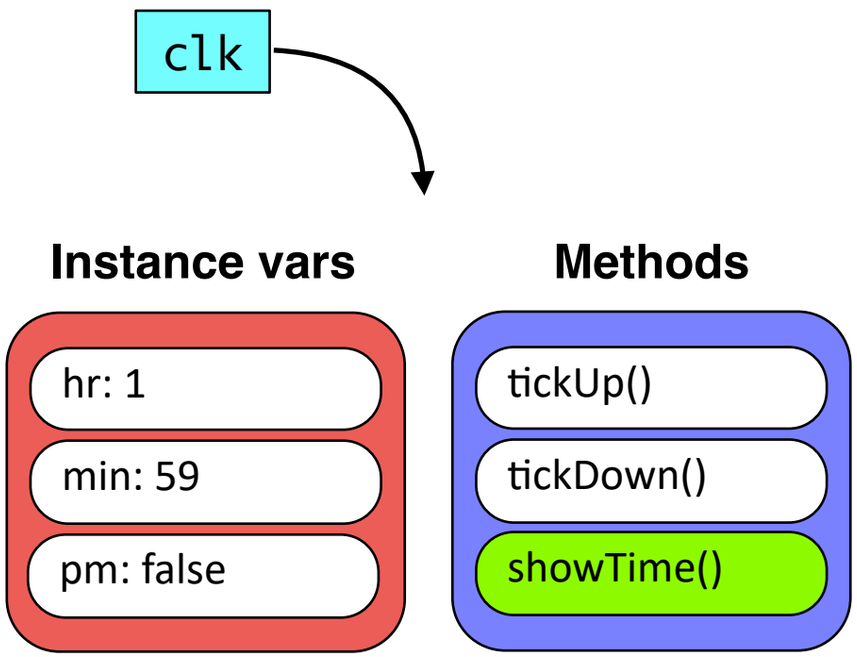


Primitive Type Variables vs. Object Variables

- Object variables store *"pointers" or "references" (→) to an existing object, like a Clock, TicketMachine, Triangle, ...*

- For example, this code declares a Clock type variable named **clk**.
  - Then it *instantiates* (using the **new** keyword) a concrete Clock object (2:00 am) and points to it.

```
Clock clk = new Clock(2,1,false);
clk.tickDown();
clk.tickDown();
clk.showTime();
> 01:59 AM
```



Food for Thought

- What do think would happen if we *never* instantiated **`clk`** prior to calling a method on it?

- Is this legal? Or would we get an error?

```
Clock clk;
clk.tickUp();
```

- Hint: What would the drawing look like?

The `null` Reference

- Warmup: But what do think would happen if we never instantiated ***`clk`*** prior to calling a method on it?

- Is this legal? Or would we get an error?

```
Clock clk;
clk.tickUp();   // This causes a NullPointerException error (crash)
```

- Since we never assigned ***`clk`*** a value, its pointer simply points to "nothing" or, null. (Asking null to tickUp is meaningless!)

Throw up an error called **`NullPointerException`**

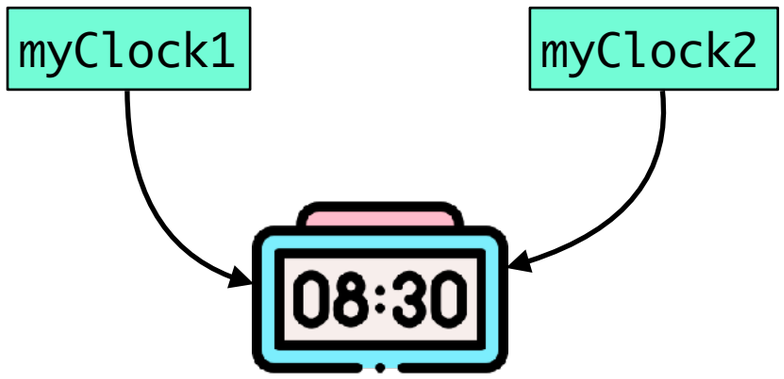
Outline

- Data Types
  - Primitives: ints, doubles, chars, booleans, etc.
    - Type Casting
  - Classes as data types
    - Object References and the null reference
    - Object Aliasing
    - Object Equality: Content Equality
- Conclusion

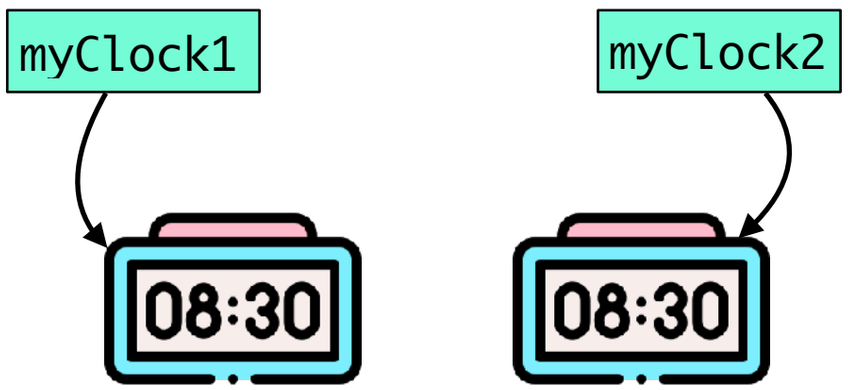
Object Aliasing

- An *"alias"* is a different name to mean the same thing.
  - In Java: Multiple variables can refer to the **same object**.
  - This is called *"Object Aliasing"*
- Interpretation of aliasing:

```
Clock myClock1 = new Clock(8,30,true);
Clock myClock2 = myClock1;
```



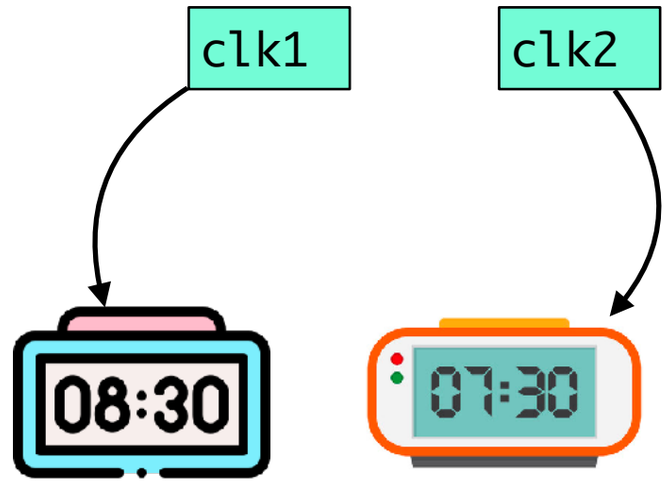
Correct interpretation!



Wrong interpretation!
myClock2 doesn't get a clone!

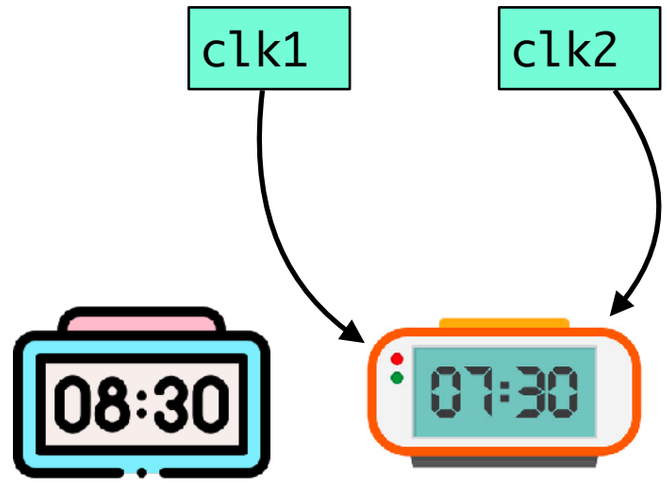
Another Aliasing Example

```
Clock clk1;
clk1 = new Clock(8, 30, false);

Clock clk2;
clk2 = new Clock(7, 30, false);
```



Aliasing Example

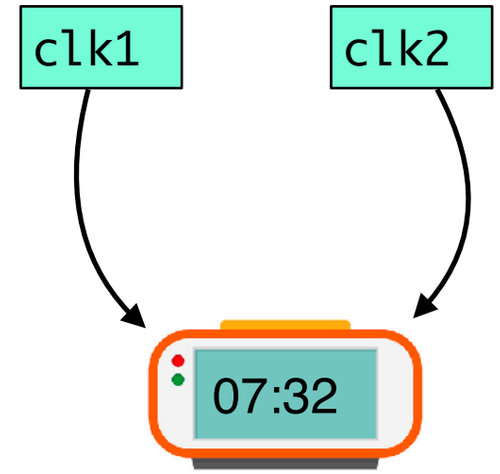
```
Clock clk1;
clk1 = new Clock(8, 30, false);

Clock clk2;
clk2 = new Clock(7, 30, false);

clk1 = clk2;    // Now clk1 and clk2 are now aliased!
```



But what happens to this clock? Nothing's pointing at it anymore.
(Garbage Collection)

Aliasing Example

```
Clock clk1;
clk1 = new Clock(8, 30, false);

Clock clk2;
clk2 = new Clock(7, 30, false);

clk1 = clk2;   // Now clk1 and clk2 are aliased!

clk1.tickUp();  // Same as clk2.tickUp()
clk2.tickUp();  // Same as clk1.tickUp()
```



Outline

- Data Types
  - Primitives: ints, doubles, chars, booleans, etc.
    - Type Casting
  - Classes as data types
    - Object References and the null reference
    - Object Aliasing
    - Object Equality: Reference Equality
- Conclusion

Object Equality

- There are two different interpretations of object equality.

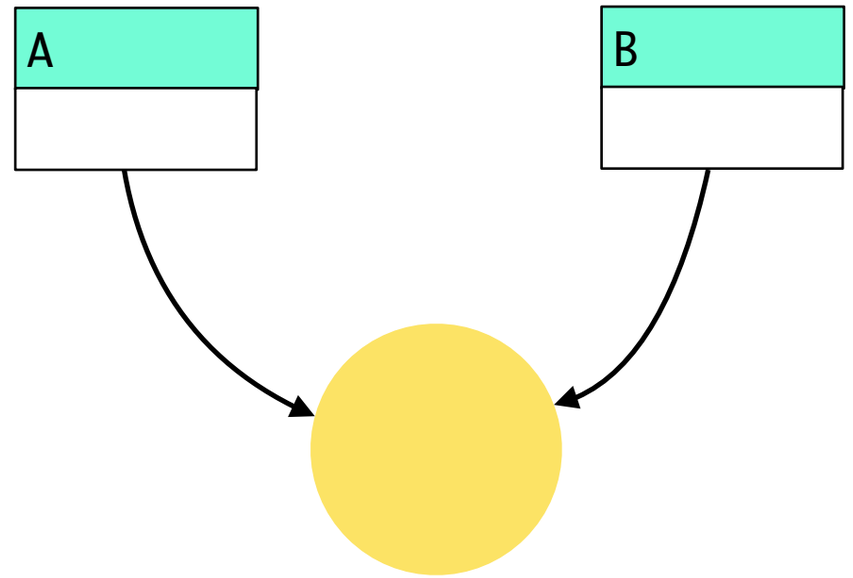
- Reference Equality (== and !=) asks,
  - "Are two object variables aliased?"
  - "Are they both *pointing* the same thing?"

- Content (Deep) Equality (`equals()` method) asks,
  - "Do two different objects share the same internal content?"

Reference Equality: Testing for Aliasing

- For example:

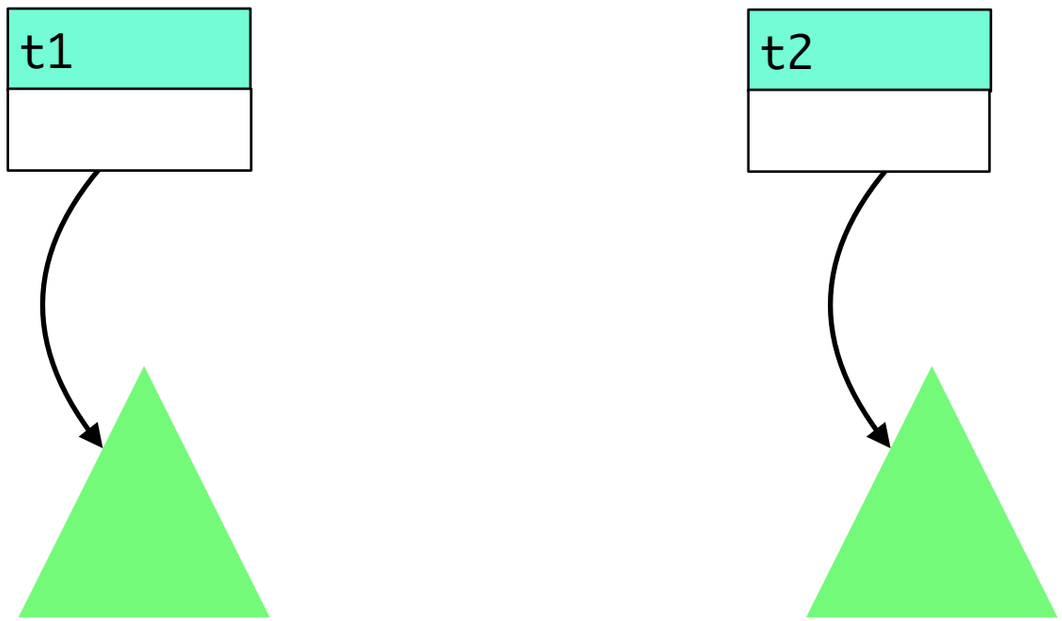
```
Circle A = new Circle();
Circle B = A;

if (A == B) {
   // Indeed this clause would trigger
}
```



Reference Equality: Testing for Aliasing (2)

- Another Example:

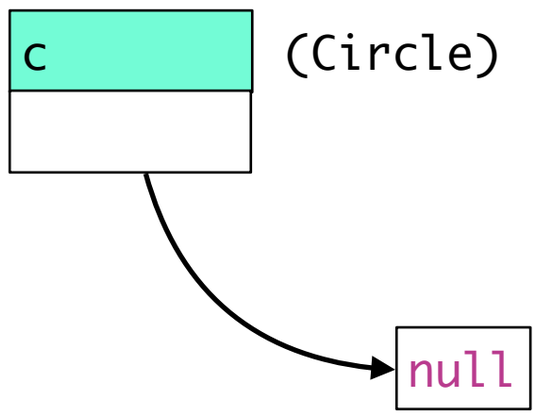
```
Triangle t1 = new Triangle();
Triangle t2 = new Triangle();

if (t1 != t2) {

     // This clause triggers!
}
```



Reference Equality: Testing for `null`

- Usually, reference equality is used to test for NULL references.

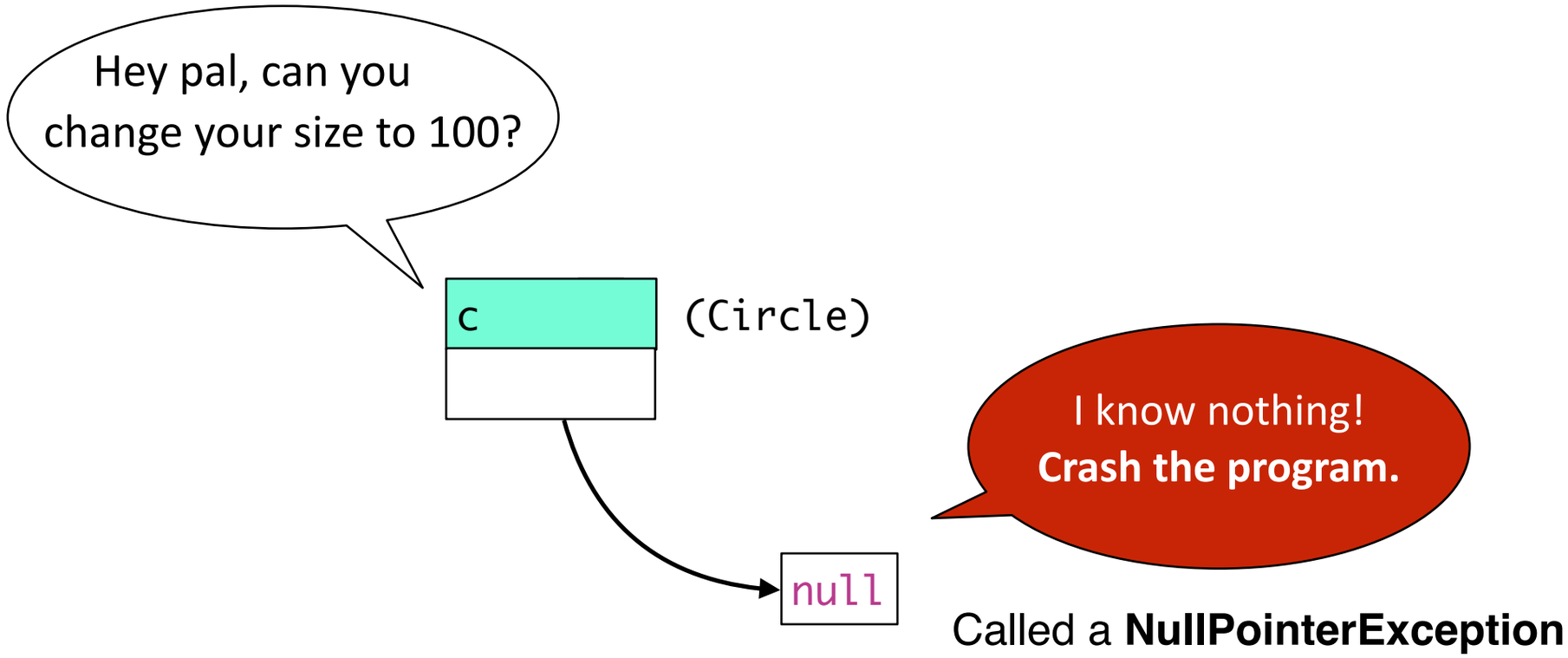
```
public void doStuff(Circle c) {   // what if user inputs a null for c?

    c.changeSize(100);
    c.changeColor("pink");

}
```



Reference Equality: Testing for `null` (2)

- The NULL test is often used for *defensive programming*

```
public void doStuff(Circle c) {   // what if user inputs a null for c?

    c.changeSize(100);
    c.changeColor("pink");

}
```



Reference Equality: Testing for `null` (3)

- The NULL test is often used for *defensive programming*

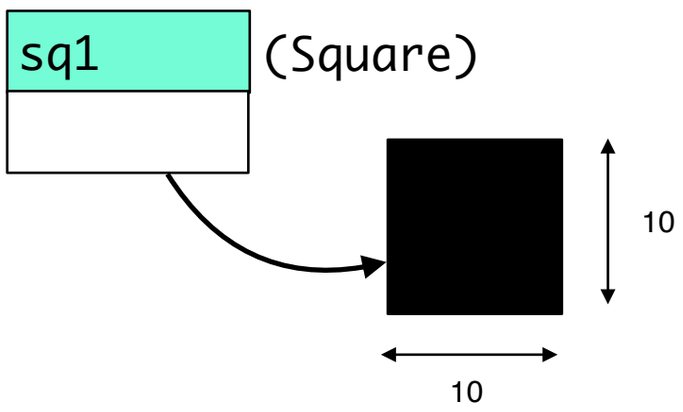
```
public void doStuff(Circle c) {

    // No more NullPointerException causing program to crash!
    if (c != null) {
        c.changeSize(100);
        c.changeColor("pink");
    }
}
```

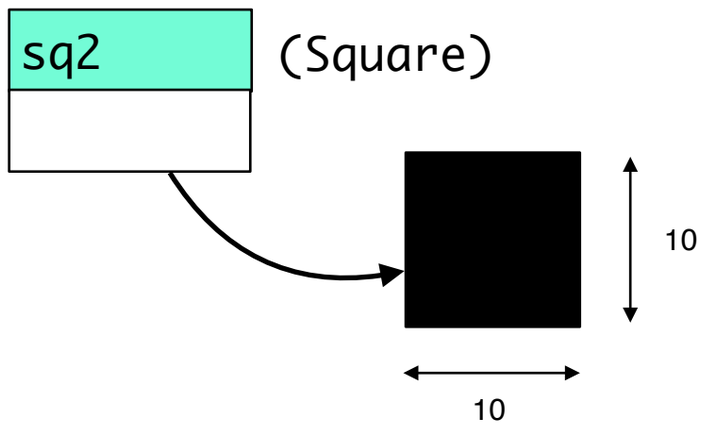
Outline

- Data Types
  - Primitives: ints, doubles, chars, booleans, etc.
    - Type Casting
  - Classes as data types
    - Object References and the null reference
    - Object Aliasing
    - Object Equality: Content Equality
- Conclusion

When Does *Reference Equality* Fall Short?

```
Square sq1 = new Square();
Square sq2 = new Square();
sq1.changeSize(10);
sq2.changeSize(10);
sq1.changeColor("black");
sq2.changeColor("black");

if (sq1 == sq2) {
   // This statement won't trigger, because sq1 and sq2 are not aliased!
   // But.... aren't they the same on the inside?
}
```





Content (Deep) Equality of Objects

- **New problem:** Reference equality doesn't check the *internals* of objects

- *An Object's Content (Deep) Equality:*
  - If we need to determine if two objects' contents are equal, we need to:

Step 1: *YOU decide* what it means for two objects to be equal internally. (Usually, some subset of their instance variables must be the same).

Step 2: Provide a `public boolean equals(`*`Class-Name other`*`)` method that returns the outcome.

Step 3: Test using:

```
if (obj1.equals(obj2)) {
    // yay they have the same internals
}
```

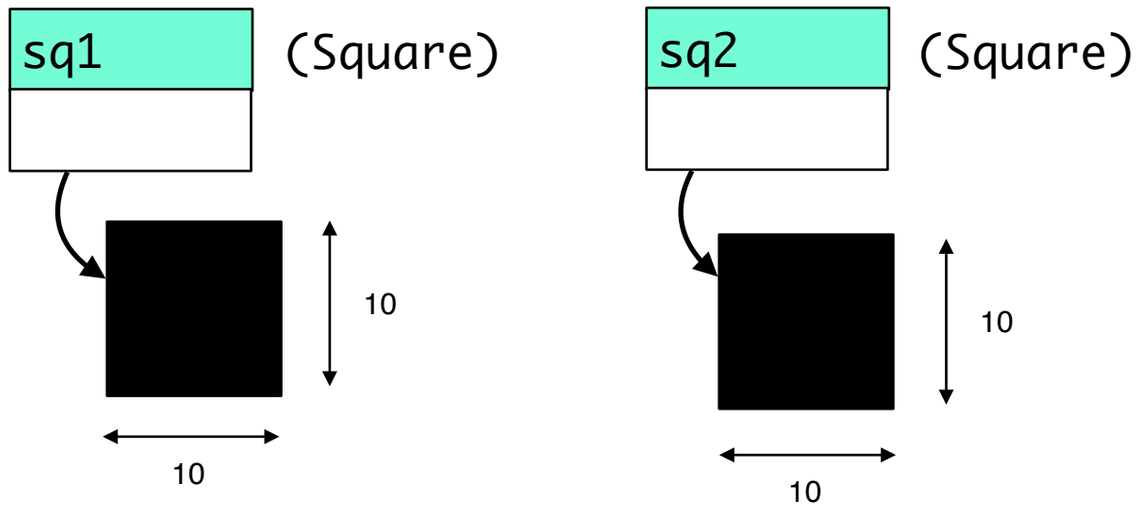
Programming the equals() Method

```java
public class Square {
   private int size;
   private int xCoord;
   private int yCoord;
   private String color;
   private boolean isVisible;

   // constructors and other methods omitted for space

   /**
    * Tests for content equality with another square. We say that
    * two squares are content-equal if they agree on size.
    *
    * @param other Pointer to another square object
    * @return true if equal in content, false otherwise
    */
   public boolean equals(Square other) {
      return (size == other.size);
   }
```

Comparing Objects (Cont.)

- And now, externally, in another area of code, this works!

```
Square sq1 = new Square();
Square sq2 = new Square();
sq1.changeSize(10);
sq2.changeSize(10);
sq1.changeColor("black");
sq2.changeColor("black");

//now this statement is true, because they are equal in size
if (sq1.equals(sq2)) {
    // Now this will trigger!
}
```



Your Turn: Content Equality of Clocks

- Write an `equals()` Method for the `Clock` class.

- Let say that two distinct Clocks are equal if their hours, minutes, and am/pm all agree.

```
public class Clock {
    private int hour;
    private int min;
    private boolean pm;
    private int alarmHr;
    private int alarmMin;
    private boolean alarmPm;

    public boolean equals(Clock other) {
        // TODO
    }
}
```