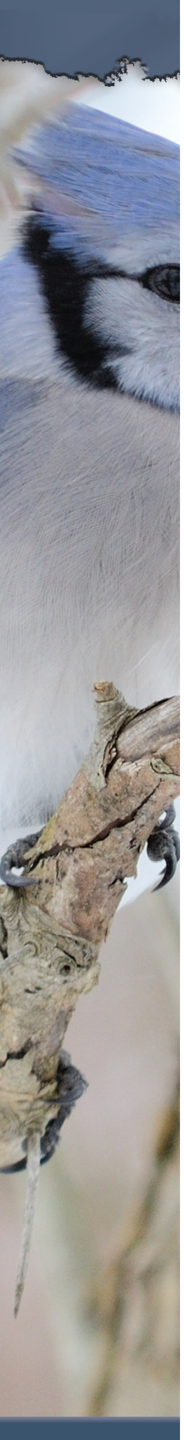
Object interaction

Creating cooperating objects

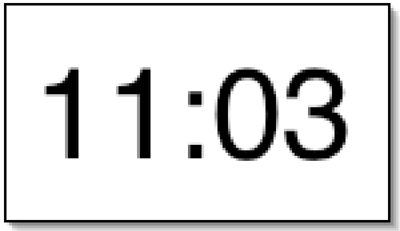
A digital clock



Abstraction and modularization

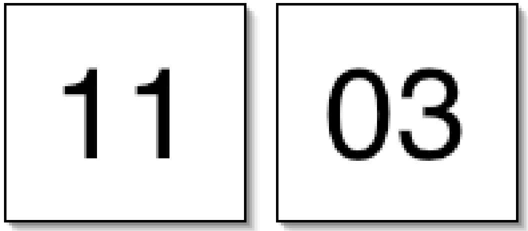
- **Abstraction** is the ability to ignore details of parts to focus attention on a higher level of a problem.
- **Modularization** is the process of dividing a whole into well-defined parts, which can be built and examined separately, and which interact in well-defined ways.

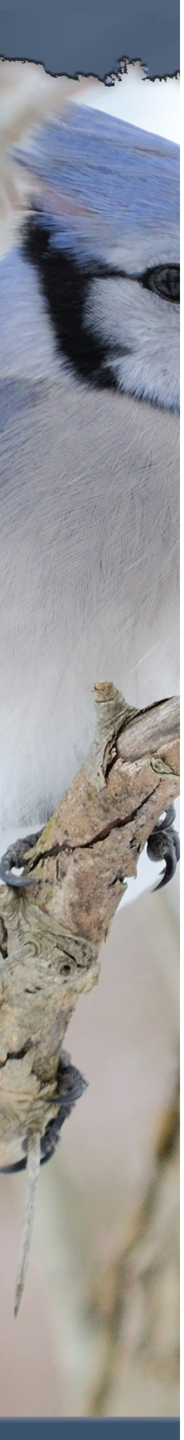
Modularizing the clock display



One four-digit display?

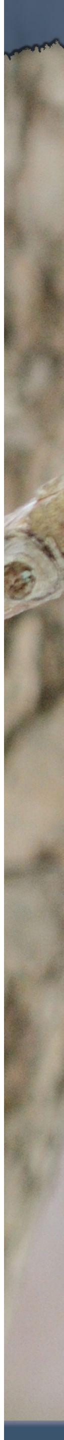
Or two two-digit displays?





Modeling a two-digit display

- We call the class **`NumberDisplay`**.
- Two integer fields:
  - The current value.
  - The limit for the value.
- The current value is incremented until it reaches its limit.
- It ‘rolls over’ to zero at this point.



Implementation - NumberDisplay

```
class NumberDisplay {
    private final int limit;
    private int value;

    NumberDisplay(int limit) {
        this.limit = limit;
        value = 0;
    }
    ...
}
```

Source code: NumberDisplay

```
String getDisplayValue() {
    if (value < 10) {
        return "0" + value;
    } else {
        return "" + value;
    }
}
```

increment method

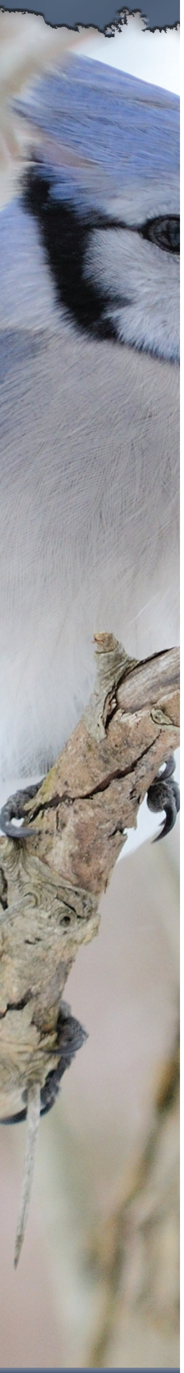
```
void increment() {
    value = value + 1;
    if (value == limit) {
        // keep the value within the limit
        value = 0;
    }
}
```

The modulo operator

- The 'division' operator (/), when applied to int operands, returns the result of an integer division.
- The 'modulo' operator (%) returns the remainder of an integer division.
- E.g., generally:

  17 / 5 gives result 3, remainder 2
- In Java:

  ```
  17 / 5 == 3
  17 % 5 == 2
  ```



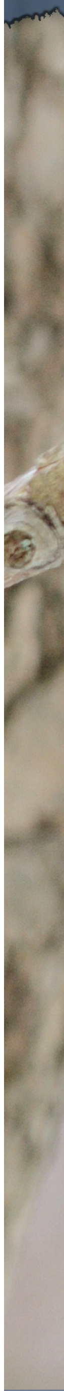
The modulo operator

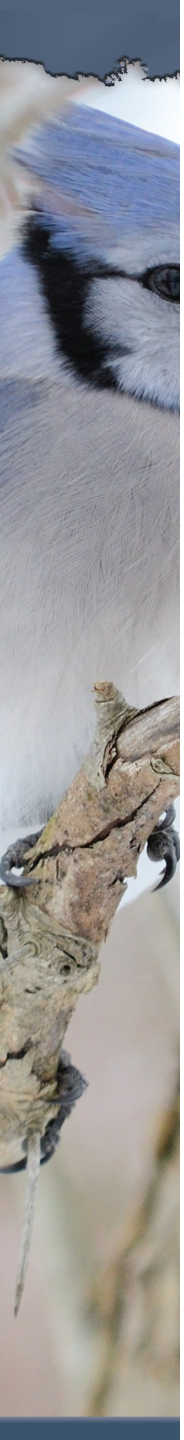
- The 'division' operator (/), when applied to int operands, returns the result of an integer division.
- The 'modulo' operator (%) returns the remainder of an integer division.
- E.g., generally:

  17 / 5 gives result 3, remainder 2
- In Java:

```
17 / 5 == 3
17 % 5 == 2
```

However, in Python





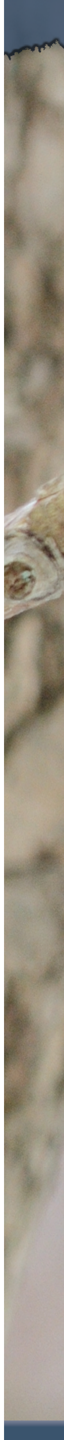
The modulo operator

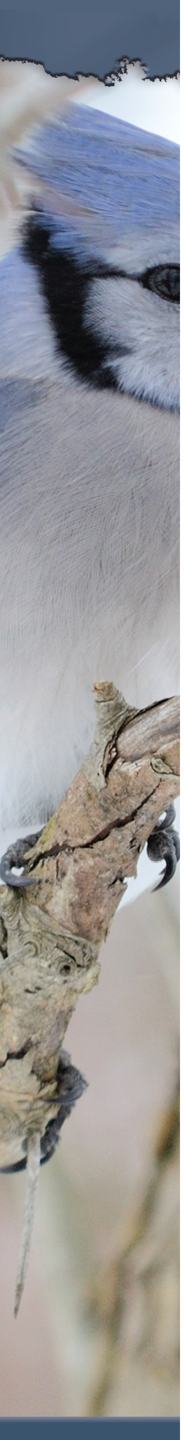
- The 'division' operator (/), when applied to int operands, returns the result of an integer division.
- The 'modulo' operator (%) returns the remainder of an integer division.
- E.g., generally:

  17 / 5 gives result 3, remainder 2
- In Java:

  17 / 5 == 3
  17 % 5 == 2

However, in Python
17 / 5 == 3.4





Quiz

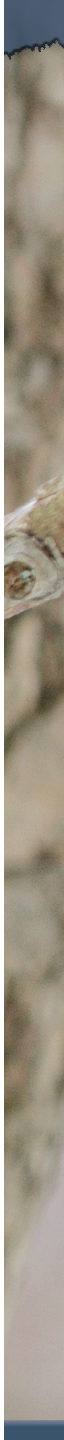
- What is the result of the expression

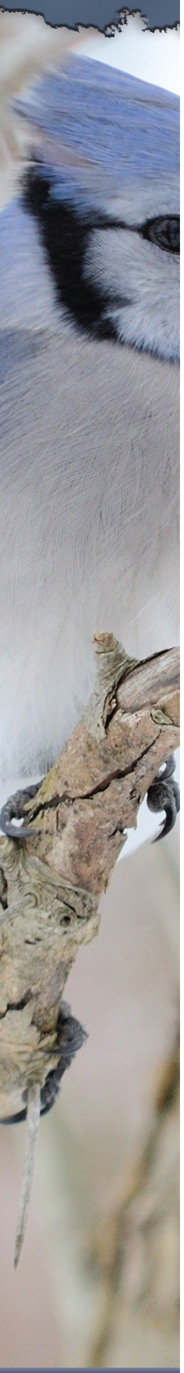
  8 % 3

- For integer `n >= 0`, what are all possible results of:

  n % 5

- Can `n` be negative?





Quiz

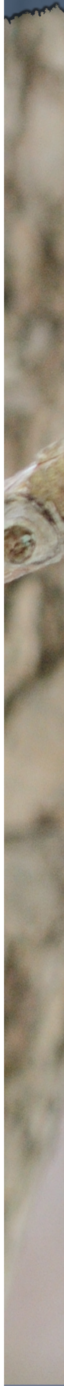
- What is the result of the expression

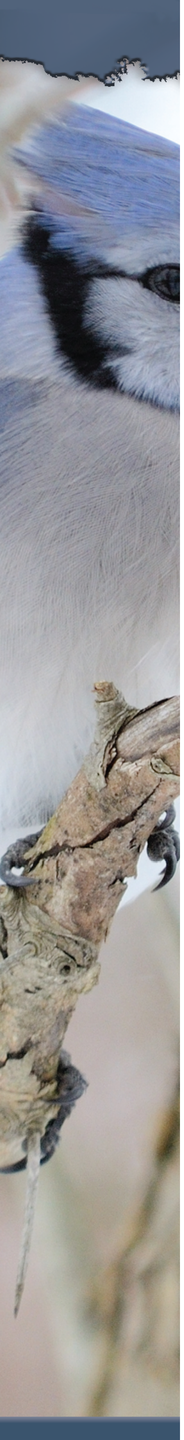
  8 % 3

- For integer `n >= 0`, what are all possible results of:

  n % 5

- Can `n` be negative? Java: -17 % 5 == -2





Quiz

- What is the result of the expression

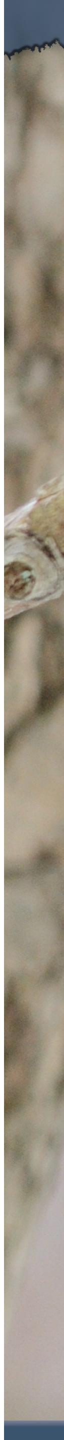
  8 % 3

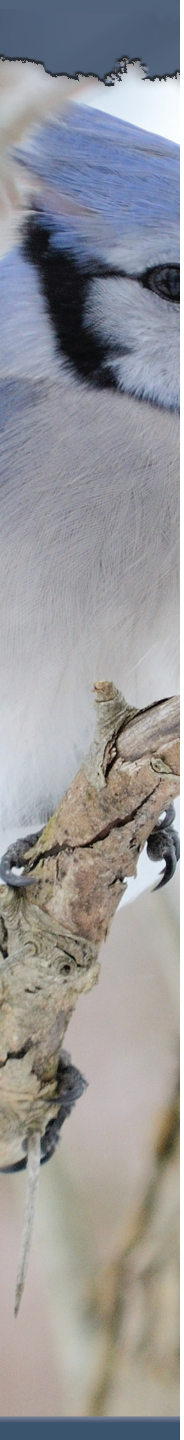
- For integer `n >= 0`, what are all possible results of:

  n % 5

- Can `n` be negative?

Java: -17 % 5 == -2
Python: -17 % 5 == 3

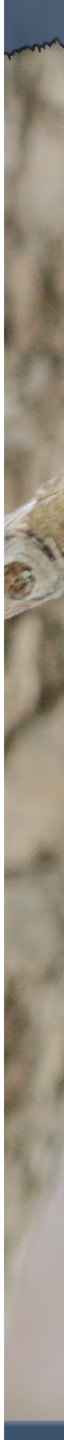




Alternative increment method

```
void increment() {
    value = (value + 1) % limit;
}
```

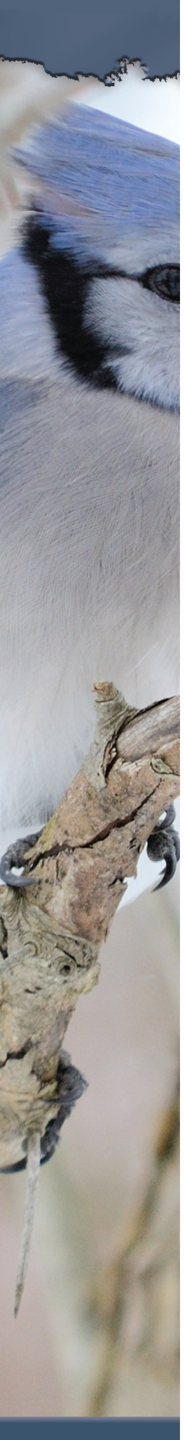
Check that you understand how the rollover works in this version.



Implementation - ClockDisplay

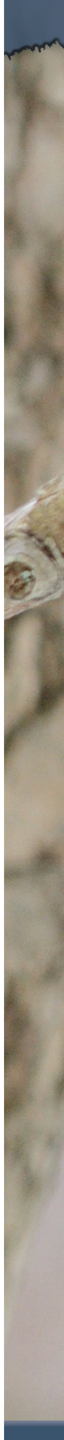
```
class ClockDisplay {
    private final NumberDisplay hours;
    private final NumberDisplay minutes;

    Constructor and
    methods omitted.
}
```

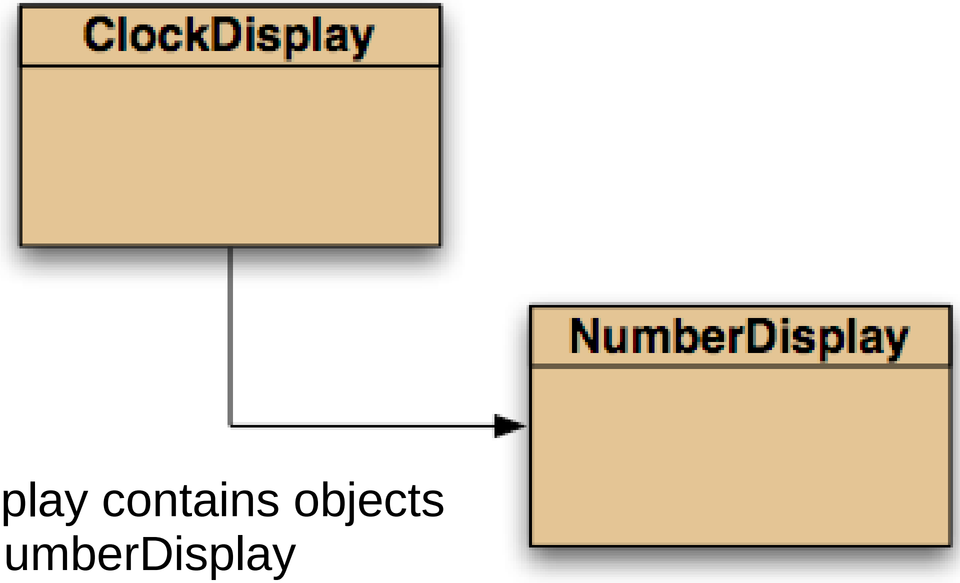


Classes as types

- Data can be classified under many different types; e.g. integer, boolean, floating-point.
- In addition, every class is a unique data type; e.g. **`String`**, **`TicketMachine`**, **`NumberDisplay`**.
- Data types, therefore, can be composites and not simply values.

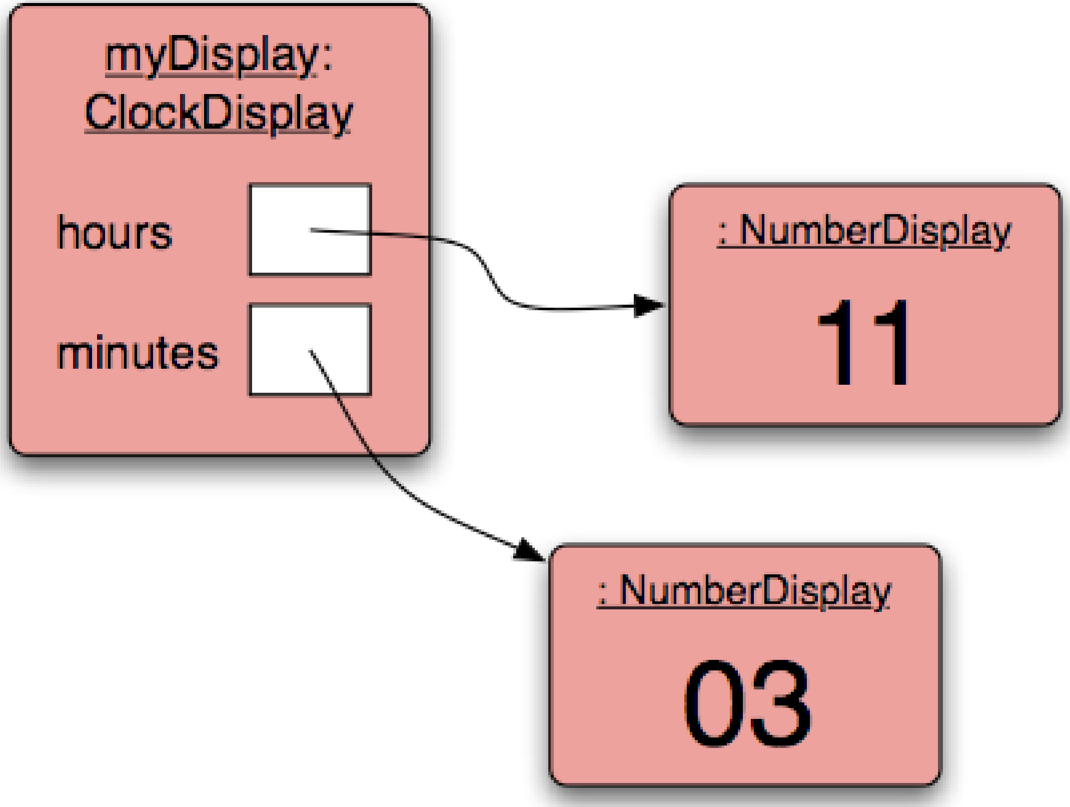


Class diagram



ClockDisplay contains objects of type NumberDisplay

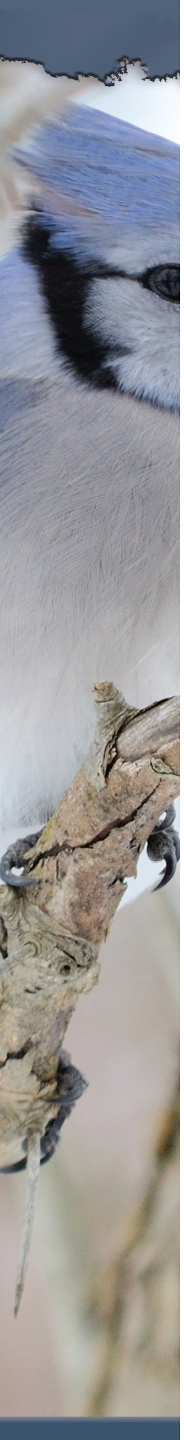
Object diagram



Objects creating objects

```
class ClockDisplay {
    private final NumberDisplay hours;
    private final NumberDisplay minutes;
    private String displayString;

    ClockDisplay() {
        hours = new NumberDisplay(24);
        minutes = new NumberDisplay(60);
        …
    }
}
```



Objects creating objects

in class ClockDisplay:

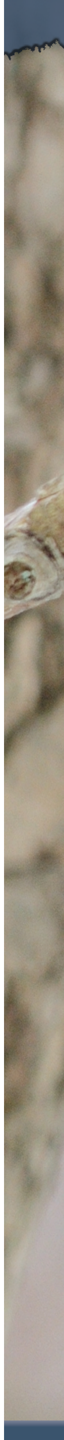
```
hours = new NumberDisplay(24);
```

actual parameter

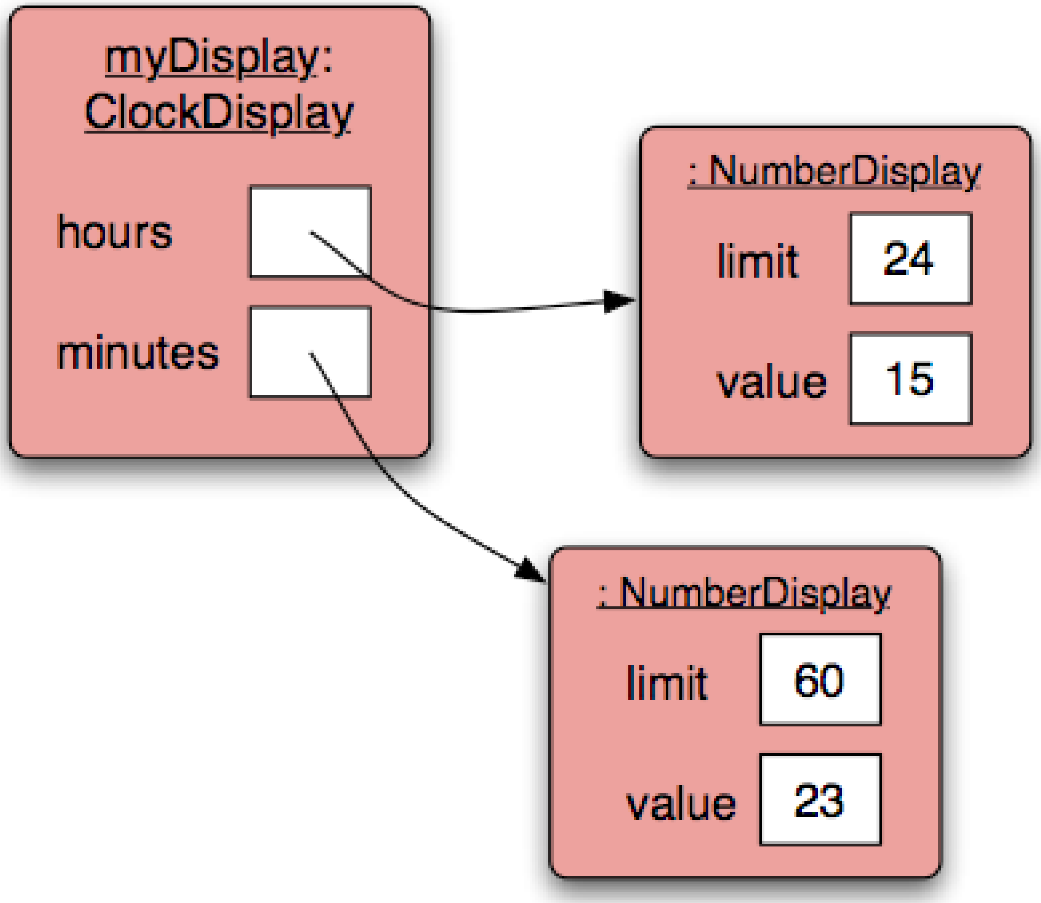
in class NumberDisplay:

```
NumberDisplay(int rollOverLimit){code}
```

formal parameter

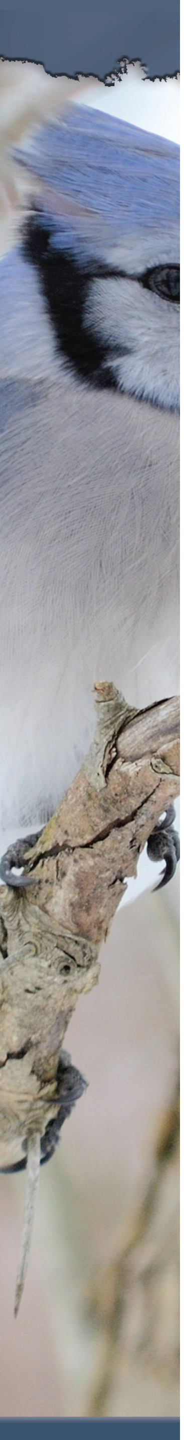


ClockDisplay object diagram



Quiz: What is the output?

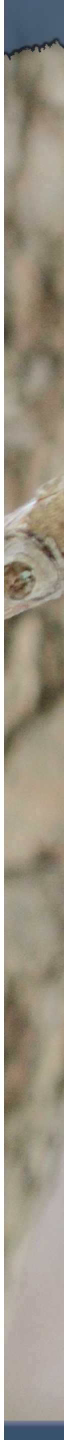
```
int a;
int b;
a = 32;
b = a;
a = a + 1;
System.out.println(b);
```

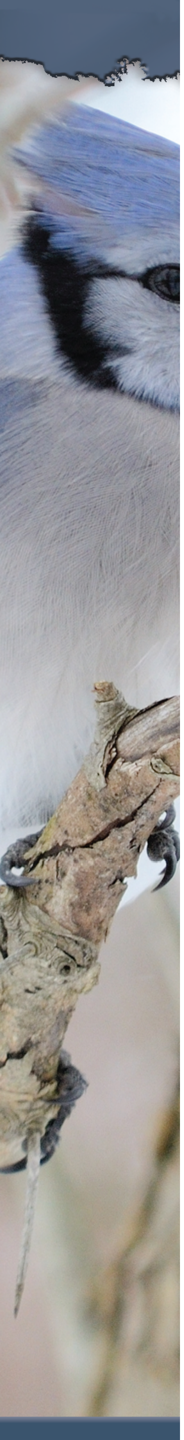


Quiz: What is the output?

```
int a;
int b;
a = 32;
b = a;
a = a + 1;
System.out.println(b);
```

32



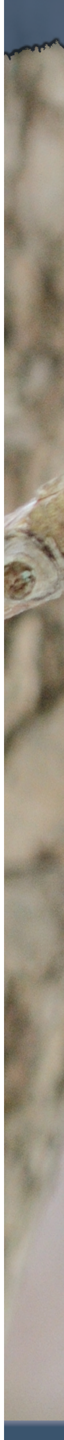


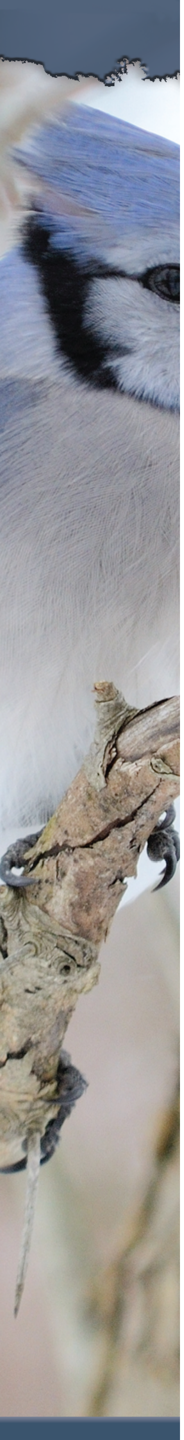
Quiz: What is the output?

-
  ```
  int a;
  int b;
  a = 32;
  b = a;
  a = a + 1;
  System.out.println(b);
  ```

32

-
  ```
  Person a;
  Person b;
  a = new Person("Everett");
  b = a;
  a.changeName("Delmar");
  System.out.println(b.getName());
  ```





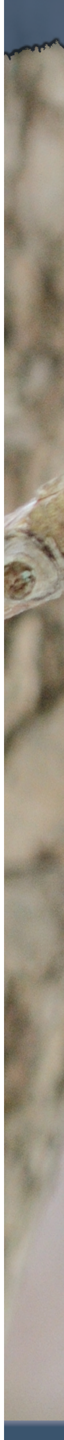
Quiz: What is the output?

-
  ```
  int a;
  int b;
  a = 32;
  b = a;
  a = a + 1;
  System.out.println(b);
  ```

  32

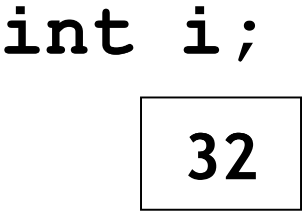
-
  ```
  Person a;
  Person b;
  a = new Person("Everett");
  b = a;
  a.changeName("Delmar");
  System.out.println(b.getName());
  ```

  Delmar



Primitive types vs. object types

Primitive types vs. object types



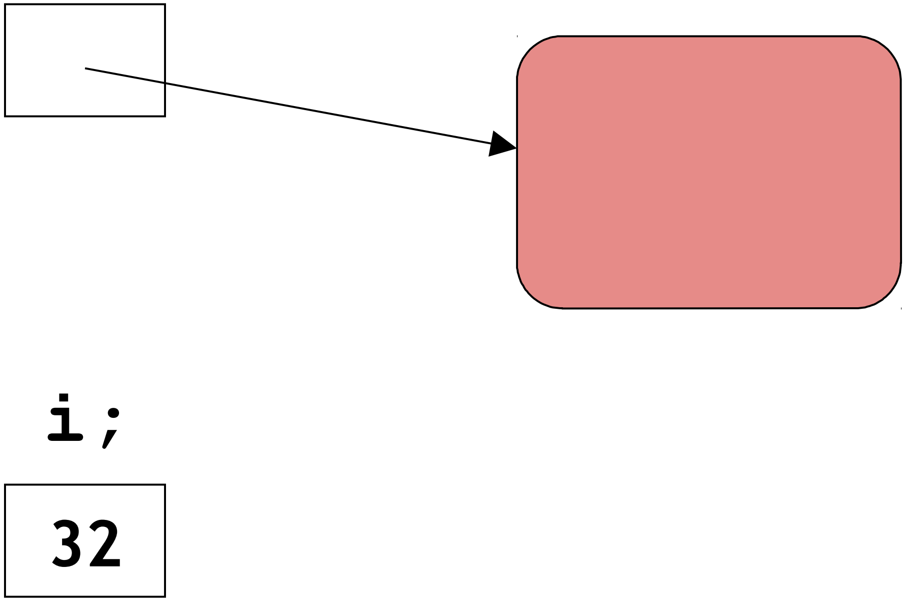
```
int i;
```

primitive type
(value type)

Primitive types vs. object types

```
SomeObject obj;
```

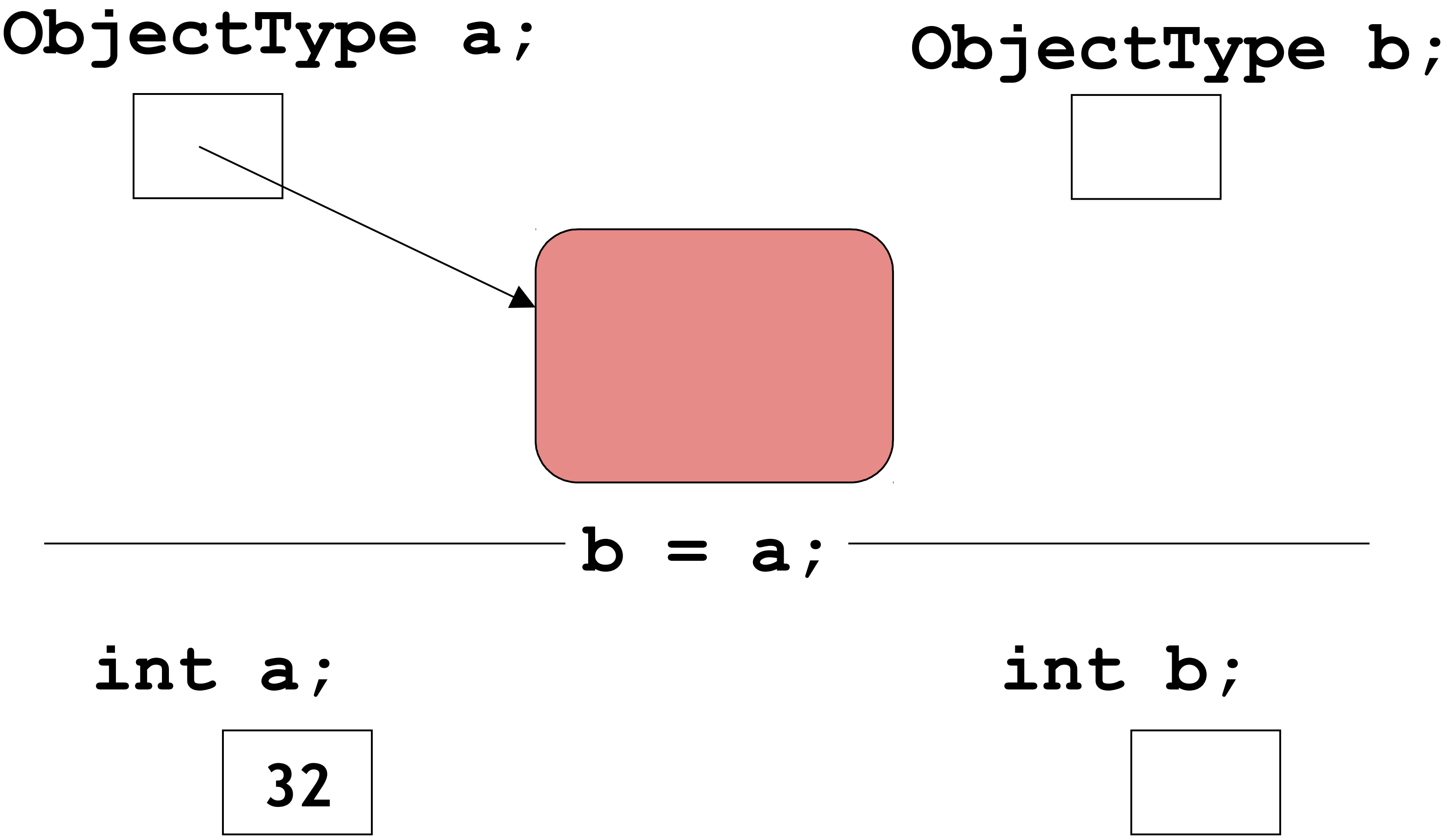
object type
(reference type)



```
int i;
```

primitive type
(value type)

Assignment
Primitive types vs. object types



ObjectType a;

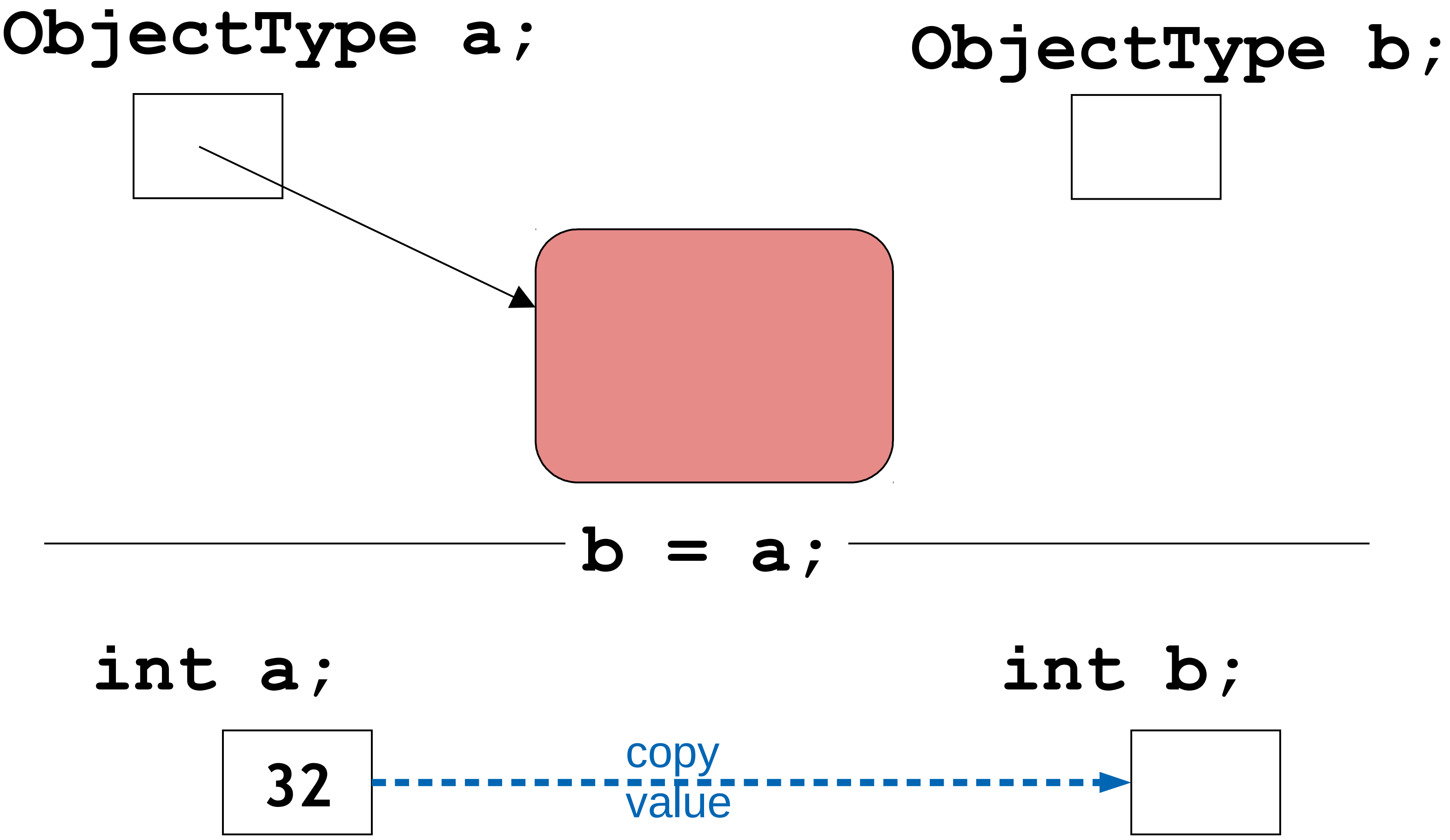
ObjectType b;

b = a;

int a;

int b;

Assignment
Primitive types vs. object types



`ObjectType a;`

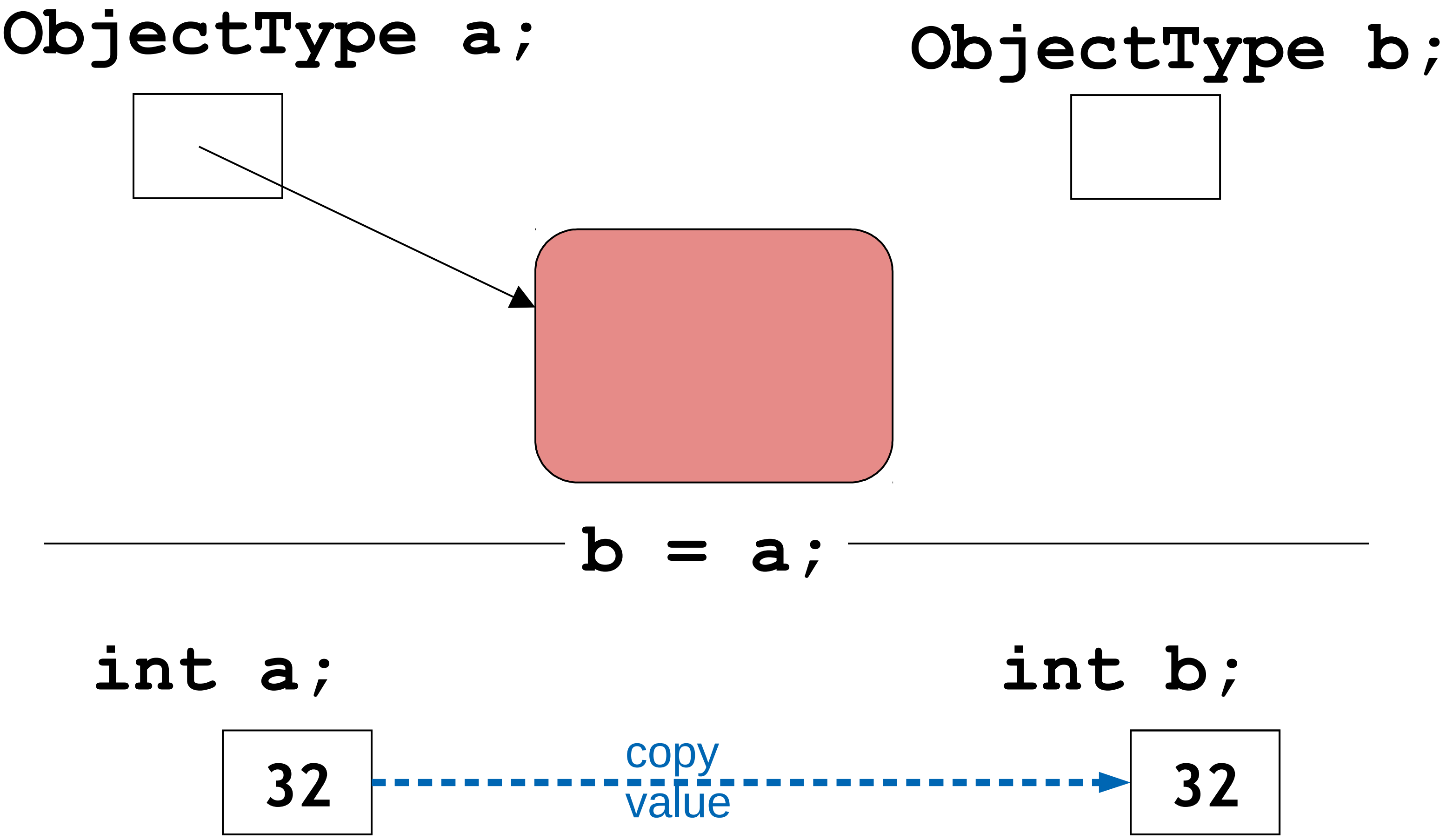
`ObjectType b;`

`b = a;`

`int a;`

`int b;`

Assignment
Primitive types vs. object types



ObjectType a;

ObjectType b;

b = a;

int a;

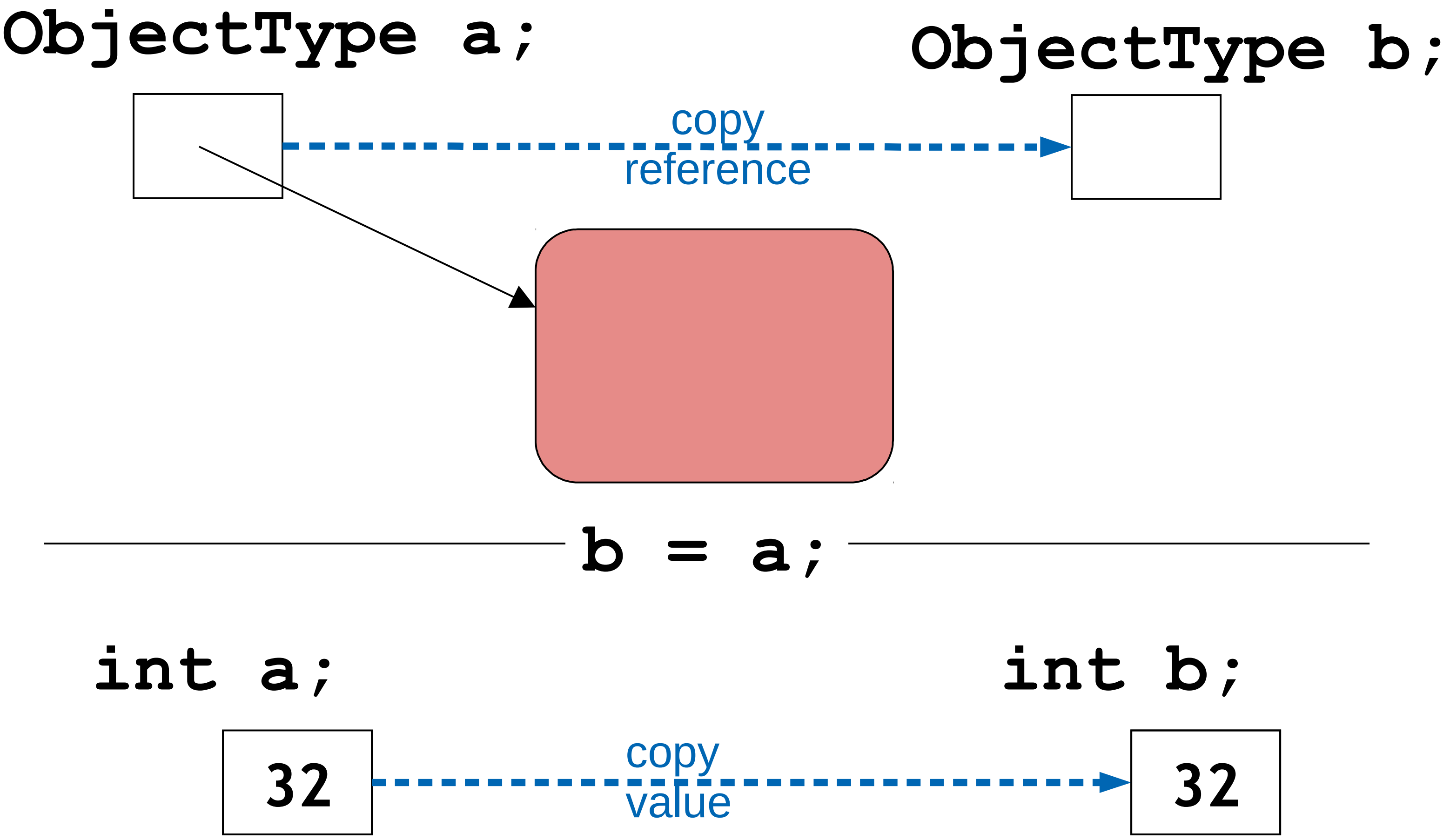
int b;

32

copy value

32

Assignment
Primitive types vs. object types



ObjectType a;

ObjectType b;

copy reference

b = a;

int a;

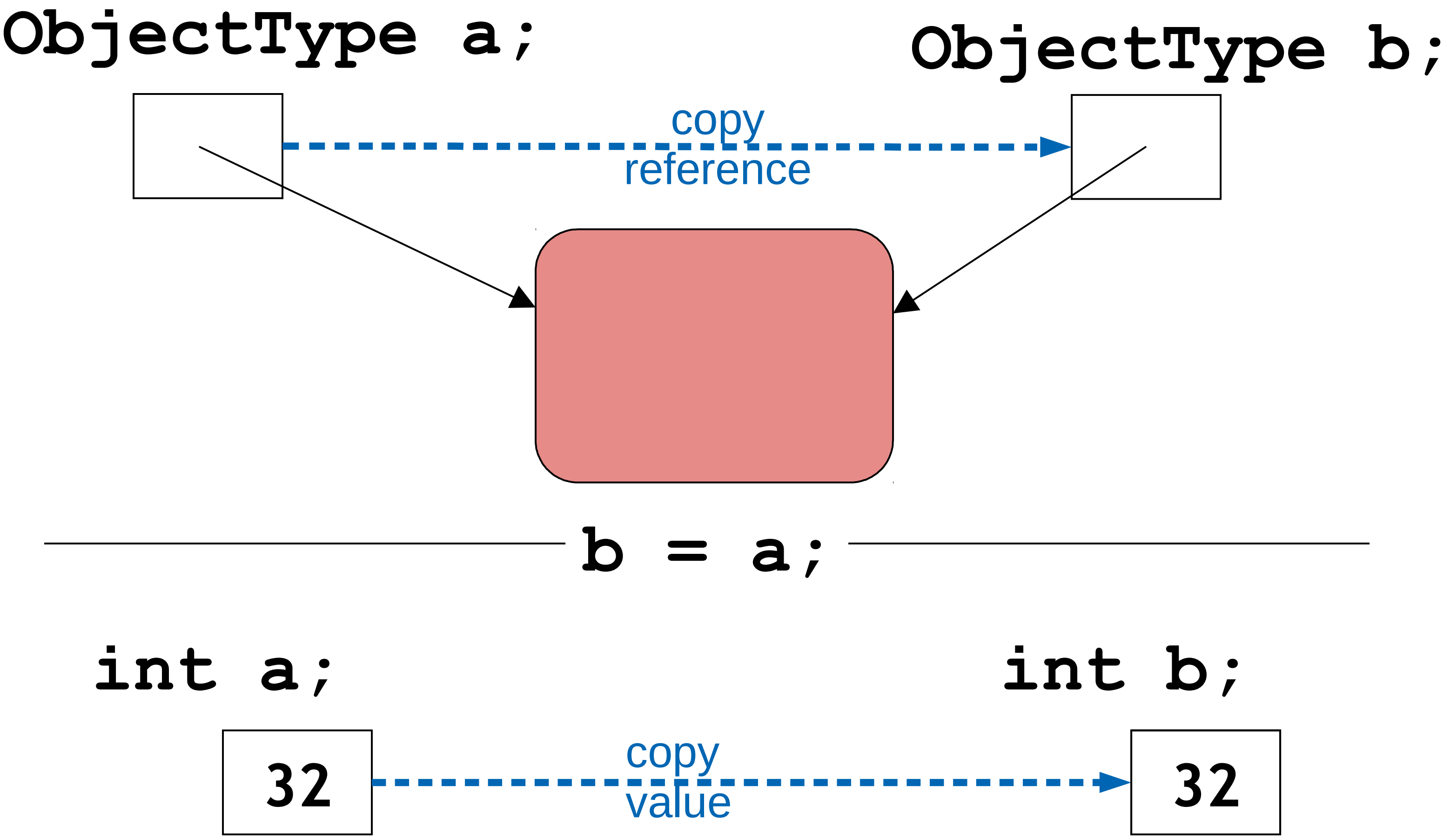
int b;

32

copy value

32

Assignment
Primitive types vs. object types



ObjectType a;

ObjectType b;

copy reference

b = a;

int a;

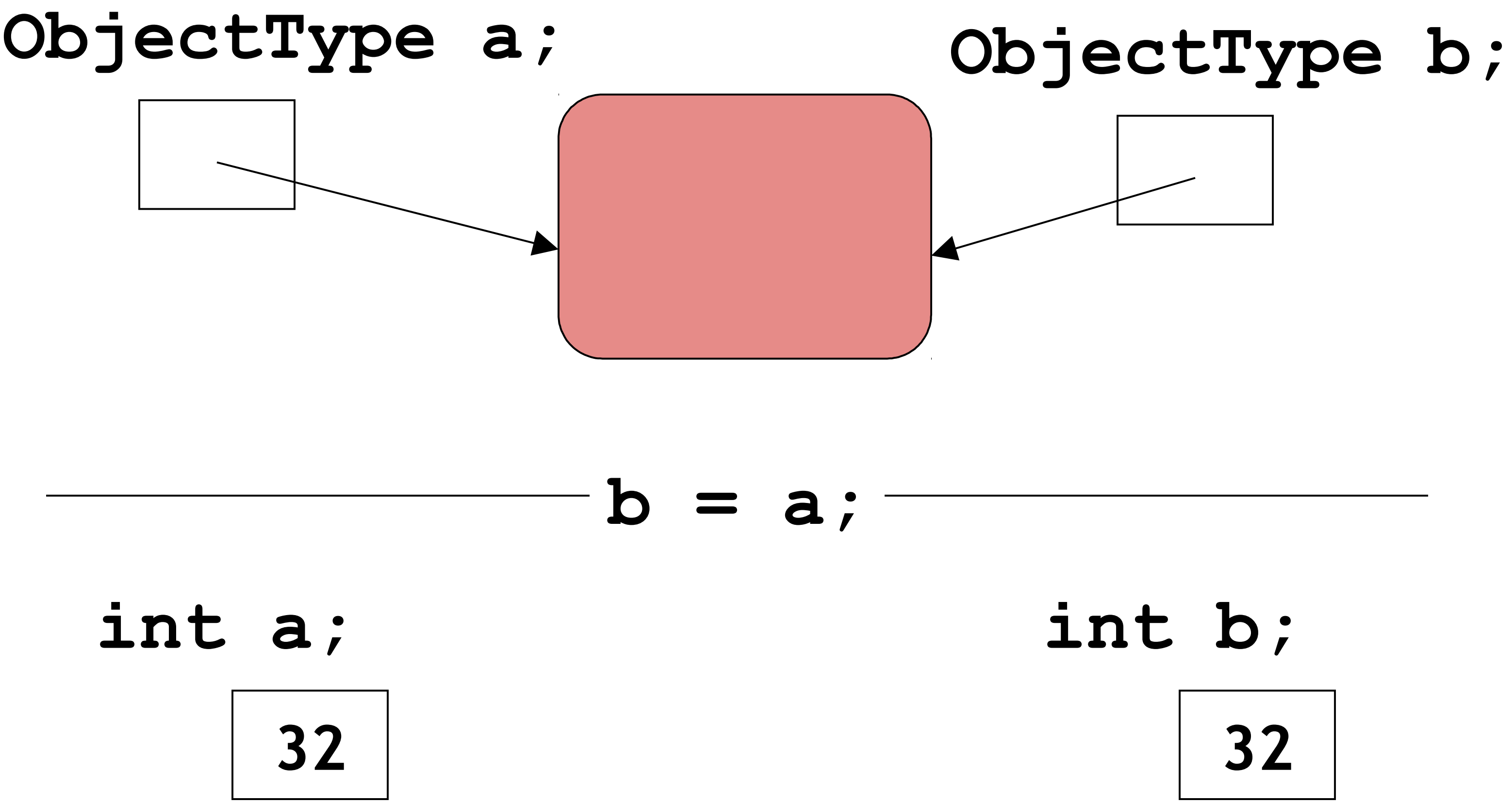
int b;

32

copy value

32

Assignment
Primitive types vs. object types



ObjectType a;

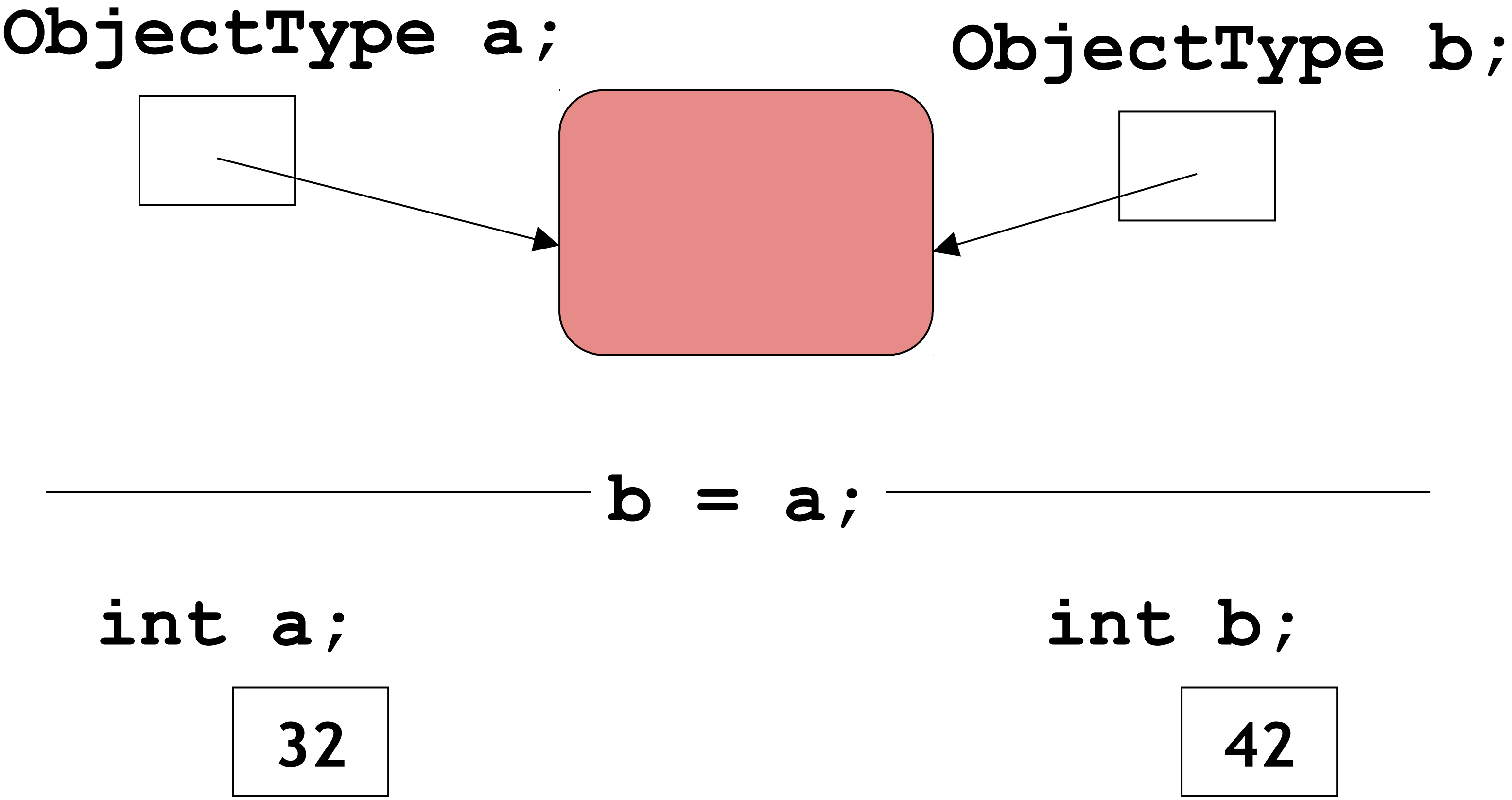
ObjectType b;

b = a;

int a;

int b;

Assignment
Primitive types vs. object types

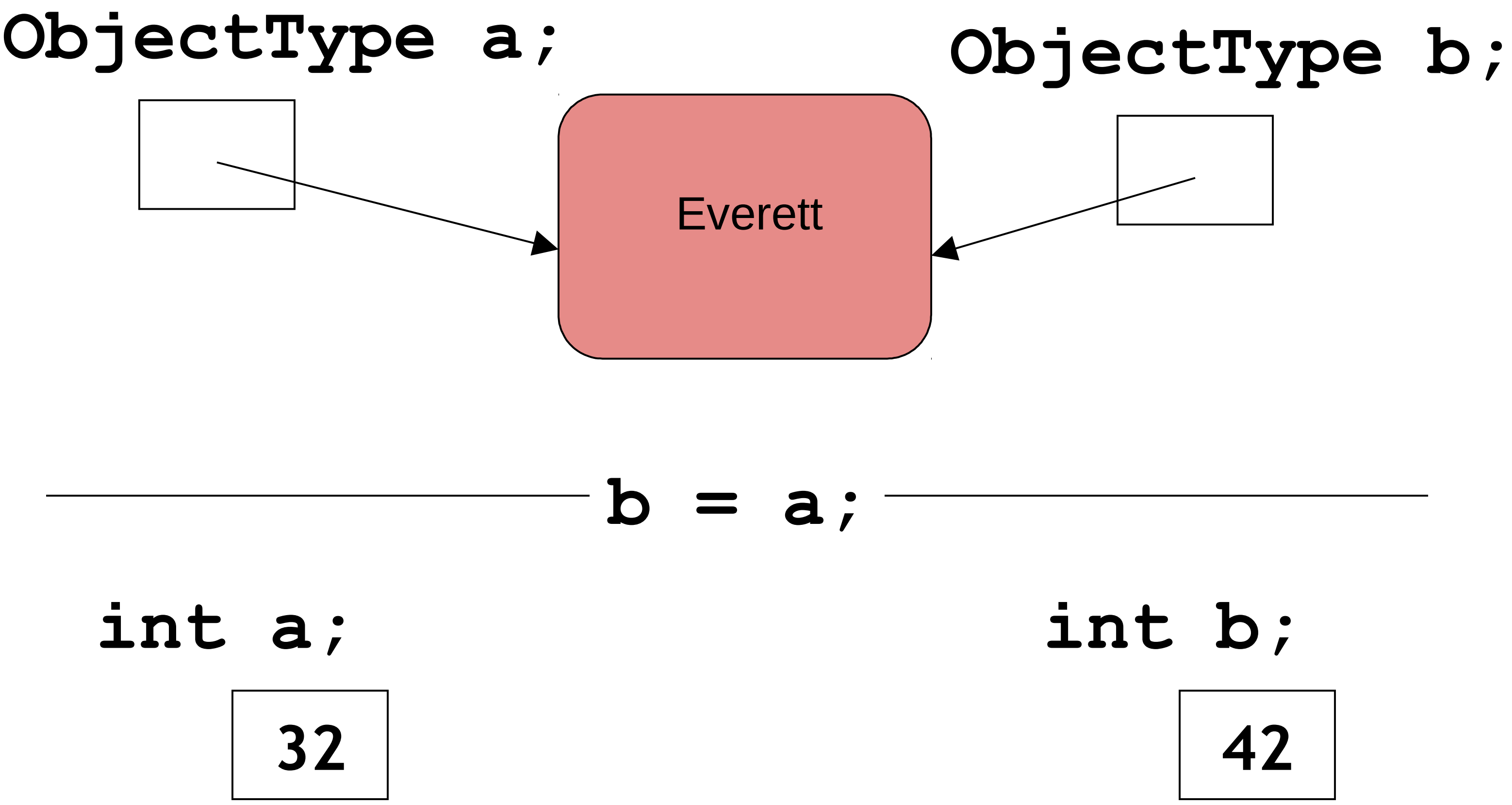


```
ObjectType a;                ObjectType b;
```

```
b = a;
```

```
int a;                       int b;
32                           42
```

Assignment
Primitive types vs. object types



```
ObjectType a;                ObjectType b;
```

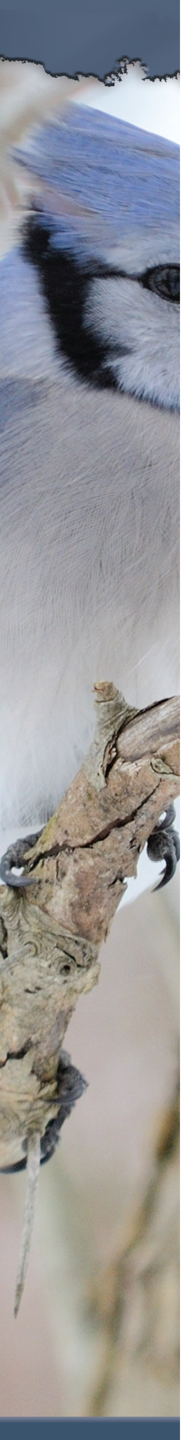
Everett

```
b = a;
```

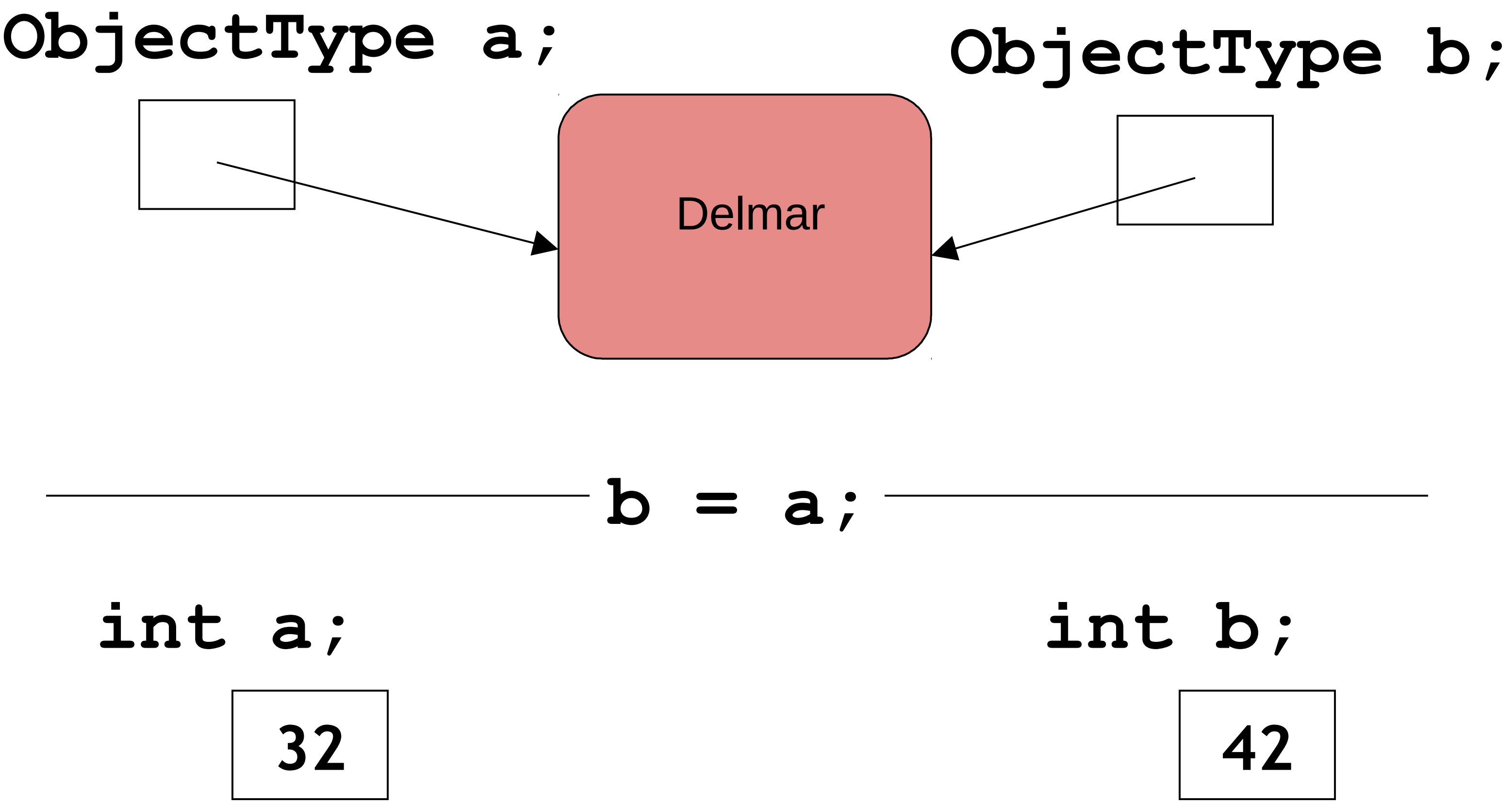
```
int a;                       int b;
```

32

42



Assignment
Primitive types vs. object types

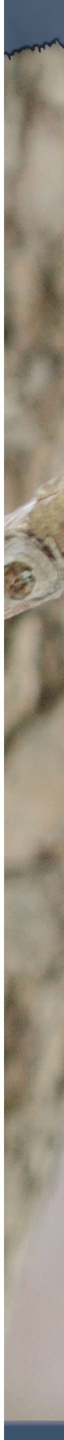


```
ObjectType a;
ObjectType b;
```

Delmar

```
b = a;
```

```
int a;
int b;
```



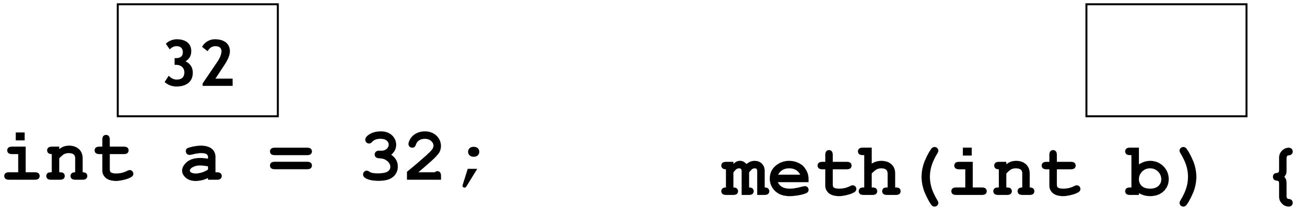
Argument passing
Primitive types vs. object types

32

```
int a = 32;
```

```
meth(int b) {
```

Argument passing
Primitive types vs. object types



```
int a = 32;
```

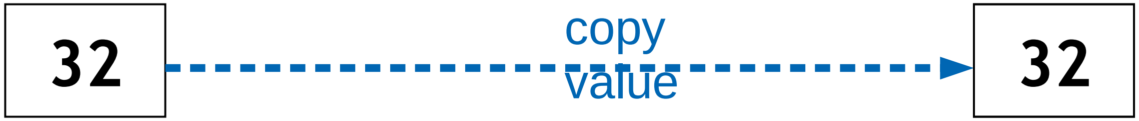
```
meth(int b) {
```

Argument passing
Primitive types vs. object types

```
int a = 32;
meth(a);
```

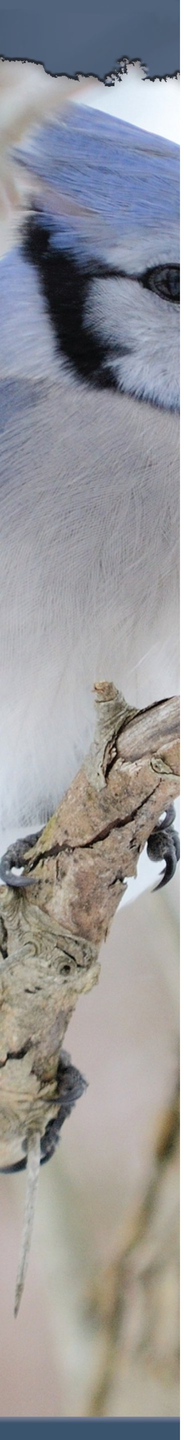
```
meth(int b) {
```

Argument passing
Primitive types vs. object types



```
int a = 32;
meth(a);
```

```
meth(int b) {
```



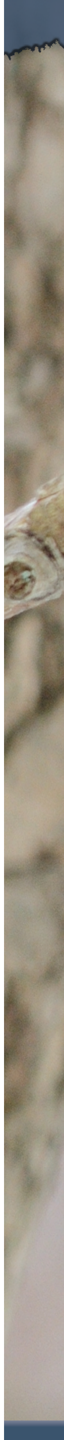
Argument passing
Primitive types vs. object types

32

```
int a = 32;
meth(a);
```

42

```
meth(int b) {
    b = 42;
}
```



Argument passing
Primitive types vs. object types

32

```
int a = 32;
meth(a);
print(a);
```

42

```
meth(int b) {
    b = 42;
}
```

Argument passing
Primitive types vs. object types

32

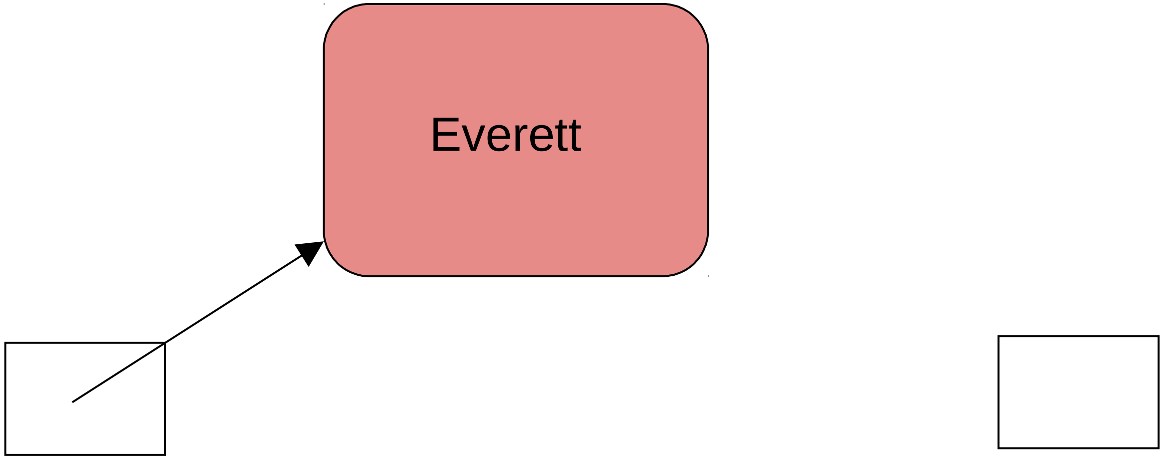
```
int a = 32;
meth(a);
print(a);
```

→ 32

42

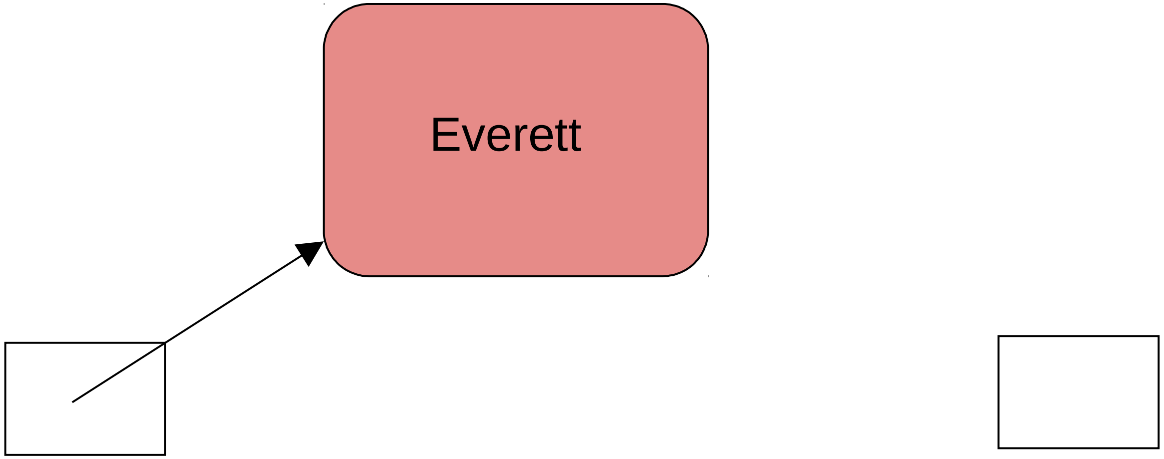
```
meth(int b) {
    b = 42;
}
```

Argument passing
Primitive types vs. object types



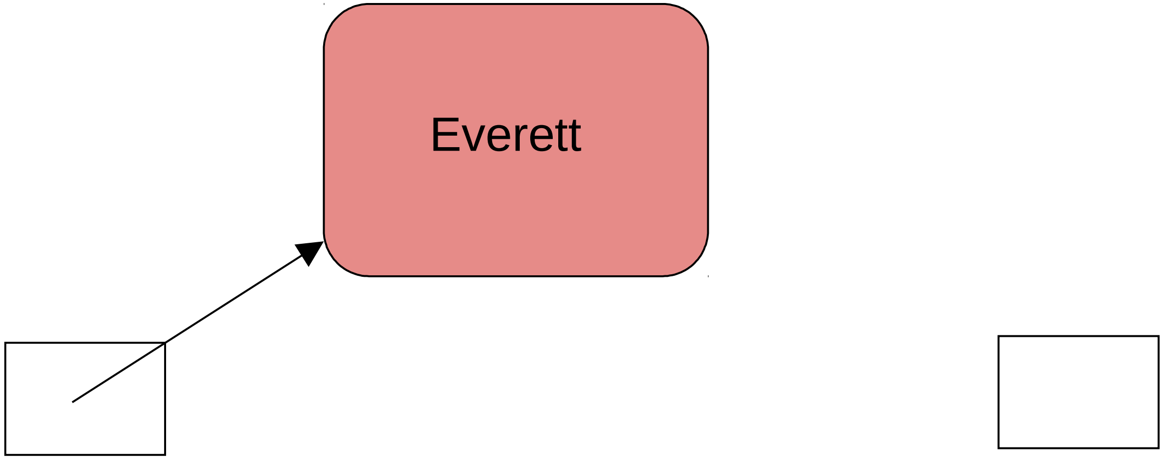
```
ObjectType a;          meth(ObjectType b) {
```

Argument passing
Primitive types vs. object types



```
ObjectType a;          meth(ObjectType b) {
```

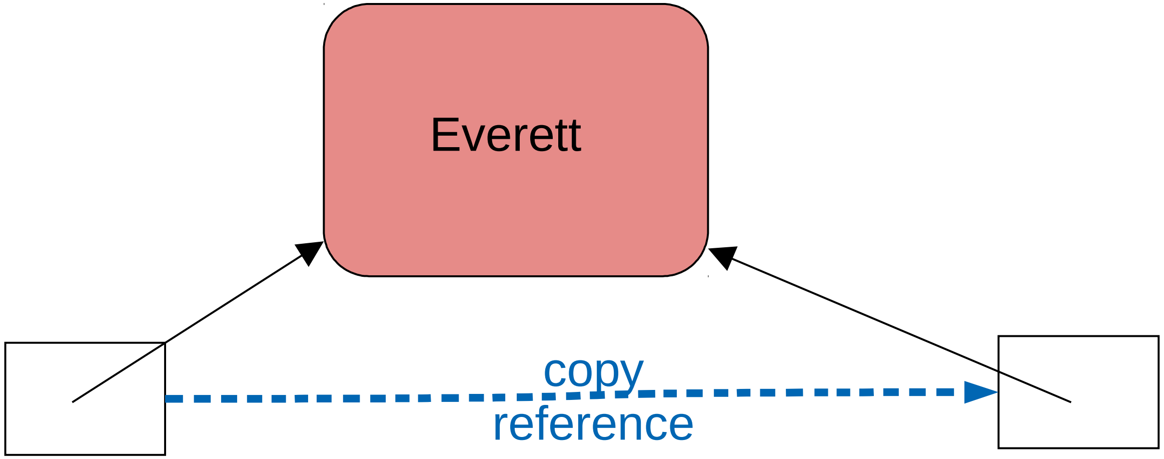
Argument passing
Primitive types vs. object types



```
ObjectType a;
meth(a);
```

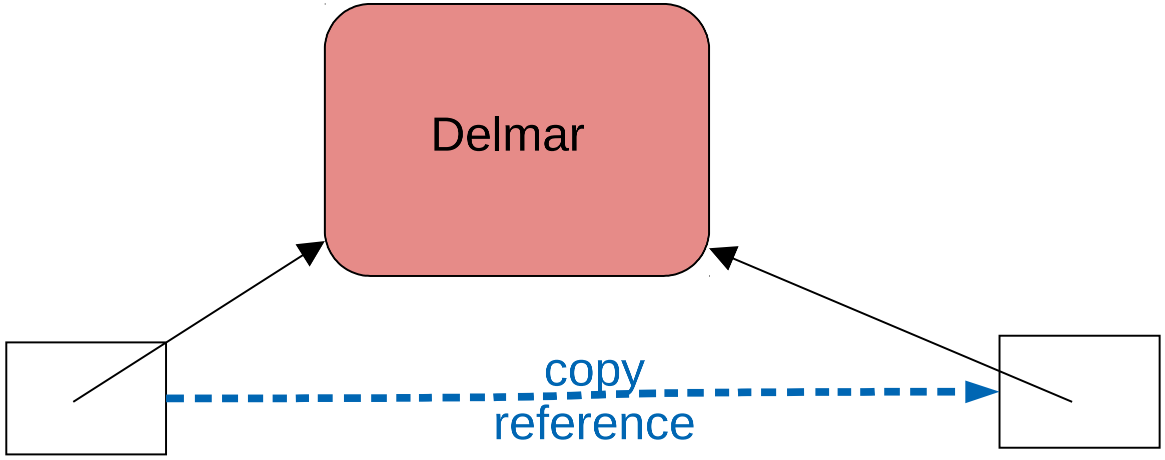
```
meth(ObjectType b) {
```

Argument passing
Primitive types vs. object types



```
ObjectType a;                meth(ObjectType b) {
meth(a);
```

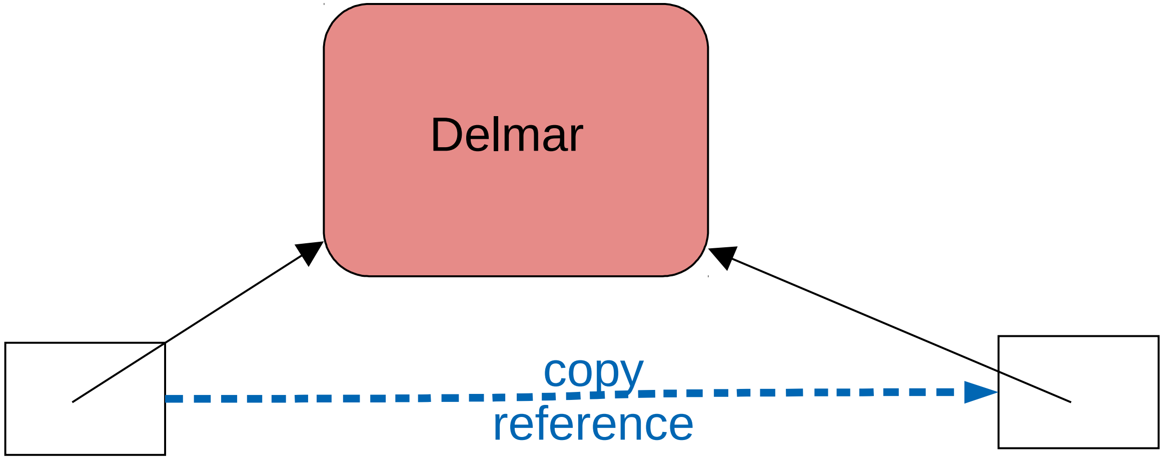
Argument passing
Primitive types vs. object types



```
ObjectType a;
meth(a);
```

```
meth(ObjectType b) {
    b.change(Delmar);
}
```

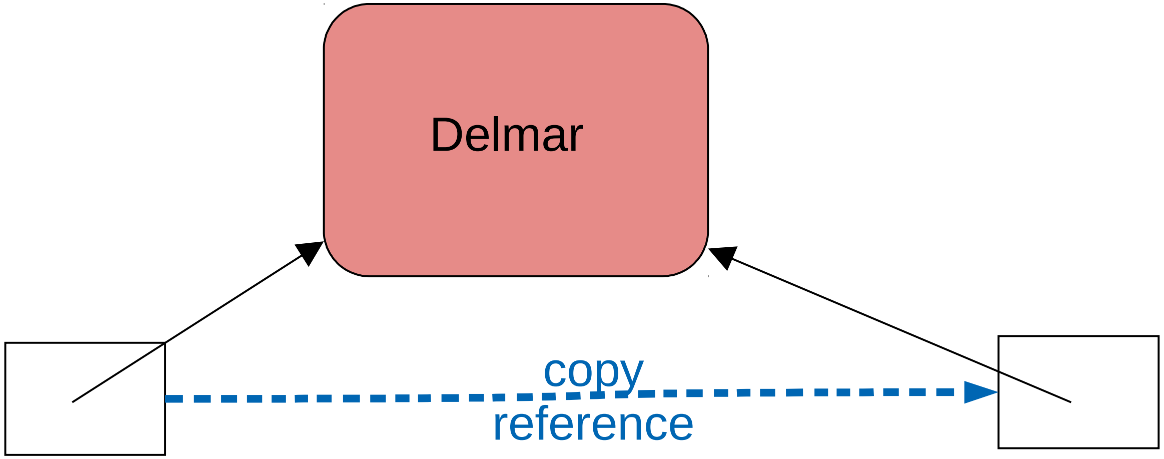
Argument passing
Primitive types vs. object types



```
ObjectType a;
meth(a);
print(a);
```

```
meth(ObjectType b) {
    b.change(Delmar);
}
```

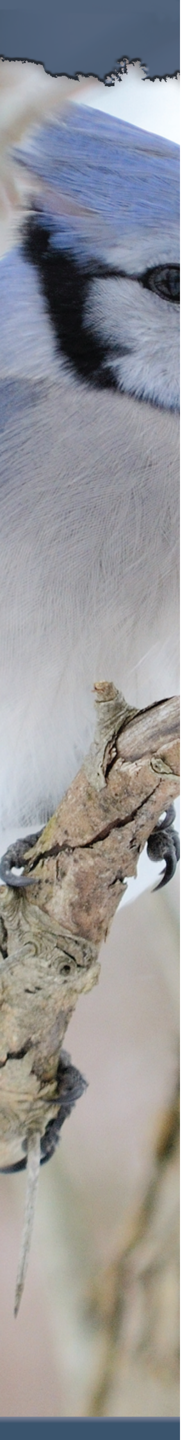
Argument passing
Primitive types vs. object types



```
ObjectType a;
meth(a);
print(a);
```

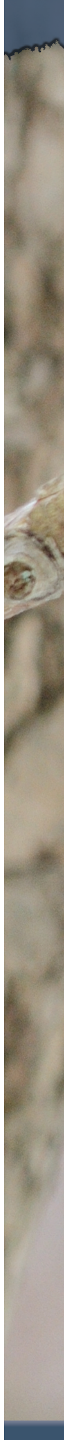
→ Delmar

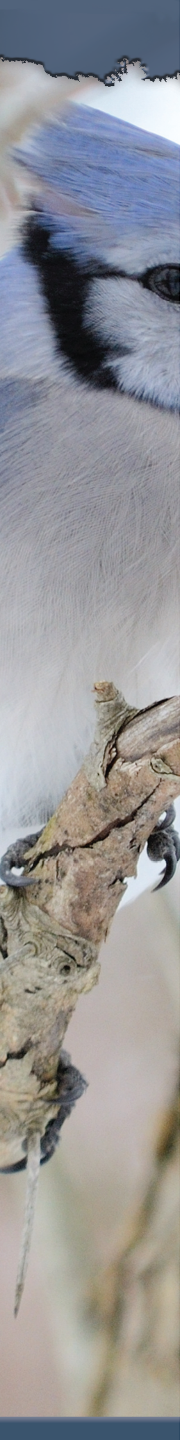
```
meth(ObjectType b) {
    b.change(Delmar);
}
```



Assignment & argument passing Primitive types vs. object types

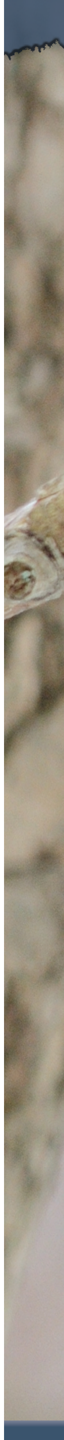
- Both in fact work exactly the same:
- Take what's here and **copy** it to there.
- The difference is in what's copied:
  - For primitive-types copy the **value**.
  - For reference-types copy the **reference**.
- However, the difference in behaviour is dramatic.

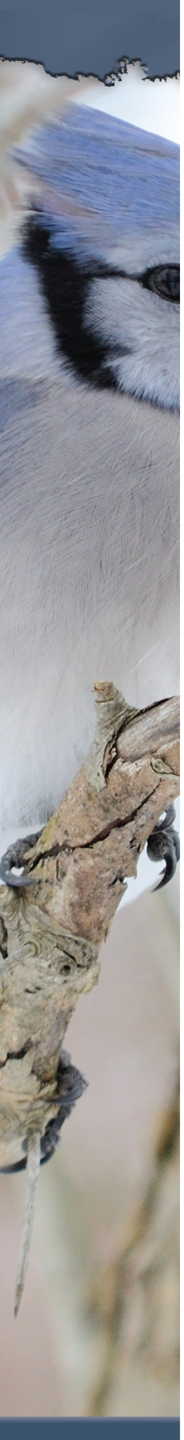




Assignment & argument passing Primitive types vs. object types

- Both in fact work exactly the same:
- Take what’s here and **copy** it to there.
- The difference is in what’s copied:
  - For primitive-types copy the **value**.
  - For reference-types copy the **reference**.
- However, the difference in behaviour is dramatic.

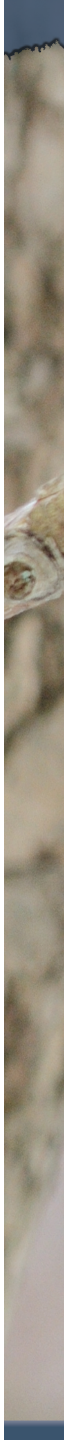




Assignment & argument passing Primitive types vs. object types

- Both in fact work exactly the same:
- Take what's here and **copy** it to there.
- The difference is in what's copied:
  - For primitive-types copy the **value**.
  - For reference-types copy the **reference**.
- However, the difference in behaviour is dramatic.

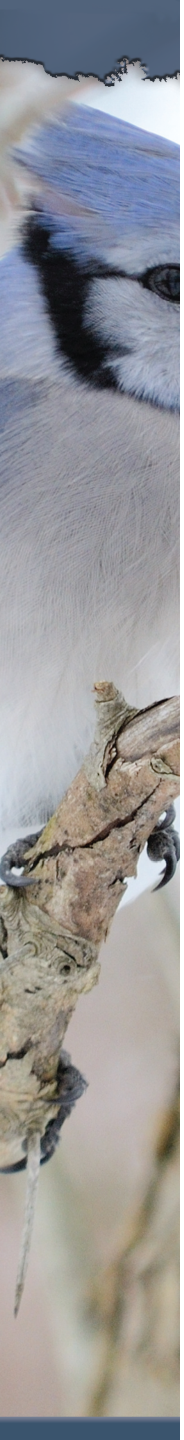
- Scoop! Coming soon to a Java near you! Value-types!
  - "Codes like a class, works like an int!"
  - Objects which behave like primitives…



Concepts

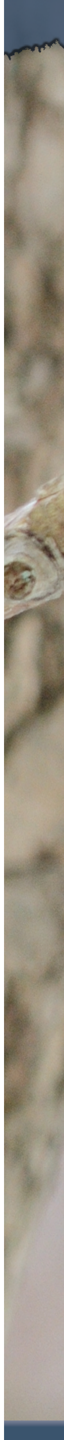
- abstraction
- modularization
- classes define types
- class diagram

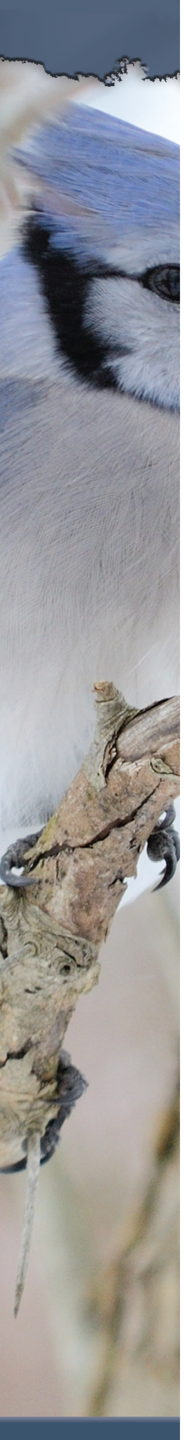
- object diagram
- object references
- object types
- primitive types



Object interaction

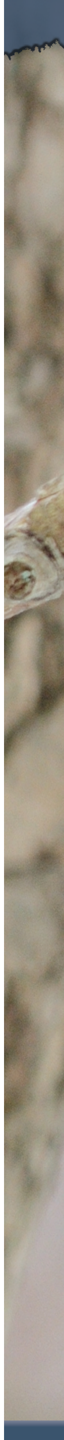
- Two objects interact when one object calls a method on another.
- The interaction is usually all in one direction (cf, ‘client’, ‘server’).
- The client object can ***tell*** the server object to do something.
- The client object can ***ask*** for data from the server object.
- A general principle is ***tell***, ***don't ask***

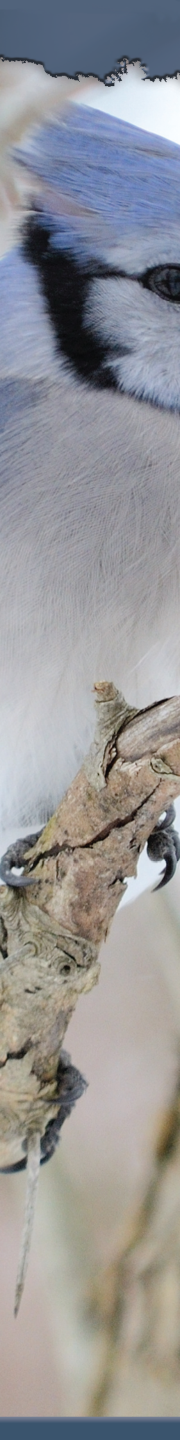




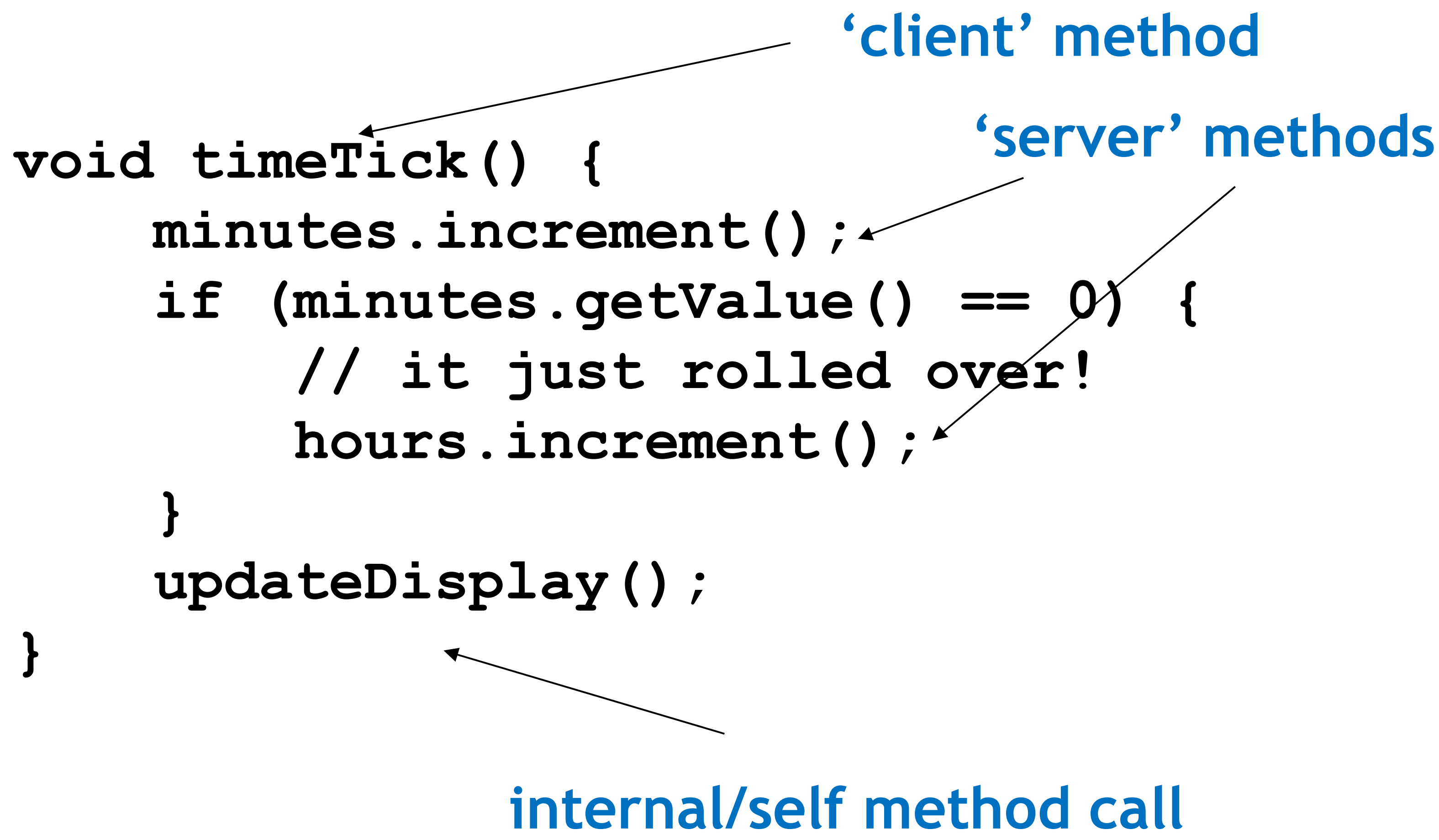
Object interaction

- Two NumberDisplay objects store data on behalf of a ClockDisplay object.
  - The ClockDisplay is the ‘client’ object.
  - The NumberDisplay objects are the ‘server’ objects.
  - The client calls methods in the server objects.





Method calling

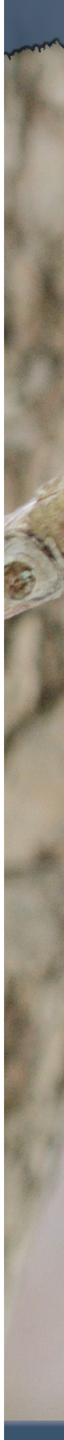


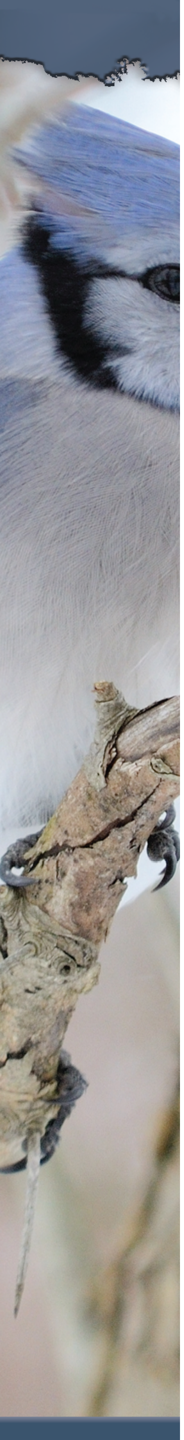
‘client’ method

‘server’ methods

```
void timeTick() {
    minutes.increment();
    if (minutes.getValue() == 0) {
        // it just rolled over!
        hours.increment();
    }
    updateDisplay();
}
```

internal/self method call





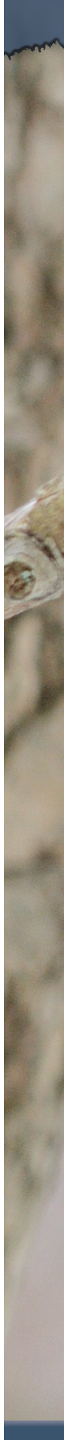
External method calls

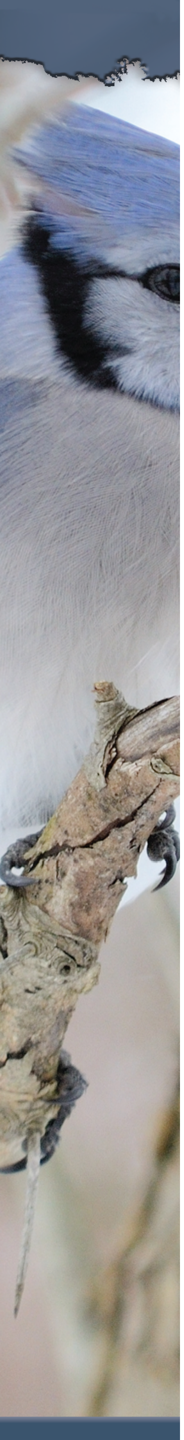
- General form:

  object.methodName(params)

- Examples:

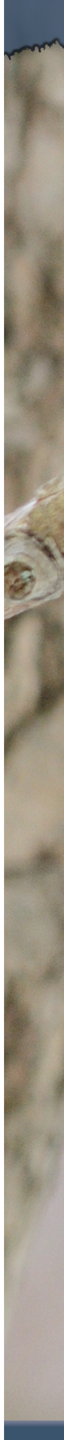
```
hours.increment()
minutes.getValue()
```





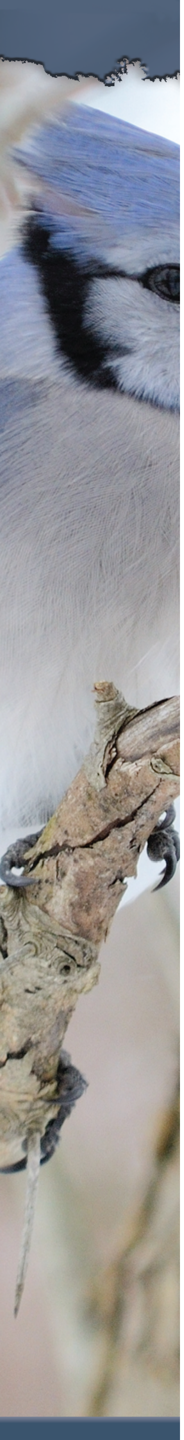
Internal method calls

- No variable name is required:
  `updateDisplay();`
- Internal methods often have **`private`** visibility.
  - Prevents them from being called from outside their defining class.



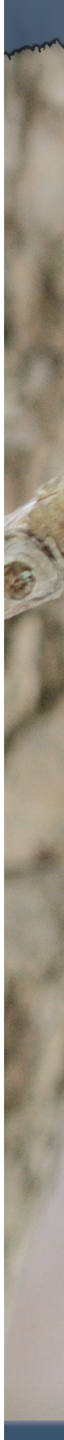
Internal method

```
/**
 * Update the internal string that
 * represents the display.
 */
private void updateDisplay() {
    displayString =
        hours.getDisplayValue() + ":"
        + minutes.getDisplayValue();
}
```



Method calls

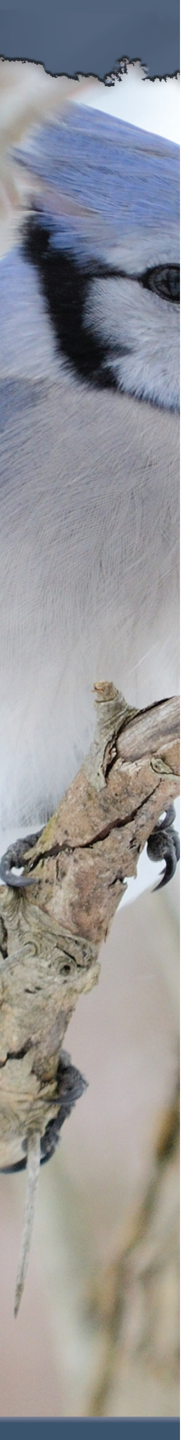
- NB: A method call on *another object of the same type* would also be an external call.
- ‘Internal’ means ‘this object’.
- ‘External’ means ‘any other object’, regardless of its type.



The **this** keyword

- Refers to *current object*.
- Used to distinguish parameters and fields of the same name. E.g.:

```
class ClockDisplay {
    private int limit;
    ClockDisplay(int limit) {
        this.limit = limit;
        value = 0;
    }
}
```

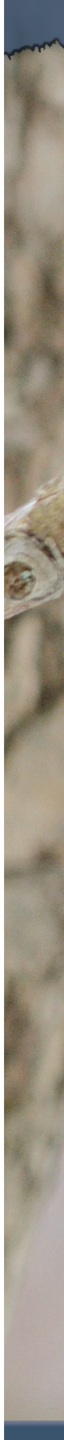


The `this` keyword

- Used to distinguish internal method calls from external method calls E.g.:

```
class ClockDisplay {
    private NumberDisplay hours;
    private void updateDisplay() {...}

    private void someMethod() {
        hours.getDisplayValue();
        updateDisplay();  // same as...
        this.updateDisplay();
}
```



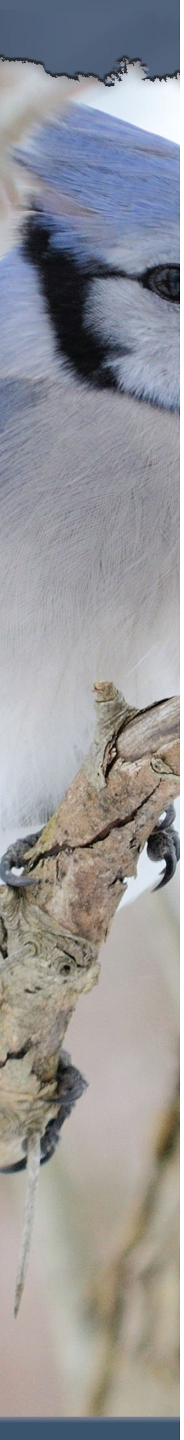
The this keyword

- Used to distinguish internal method calls from external method calls E.g.:

```
class ClockDisplay {
    private NumberDisplay hours;
    private void updateDisplay() {...}
```

external method call

```
    private void someMethod() {
        hours.getDisplayValue();
        updateDisplay();  // same as...
        this.updateDisplay();
}
```



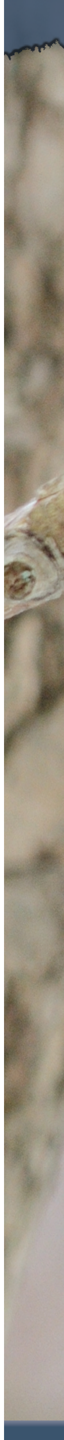
The `this` keyword

- Used to distinguish internal method calls from external method calls E.g.:

```
class ClockDisplay {
    private NumberDisplay hours;
    private void updateDisplay() {...}

    private void someMethod() {
        hours.getDisplayValue();
        updateDisplay();  // same as...
        this.updateDisplay();
}
```

internal method calls



So...what makes code OO?

- Consider (not real Java code)

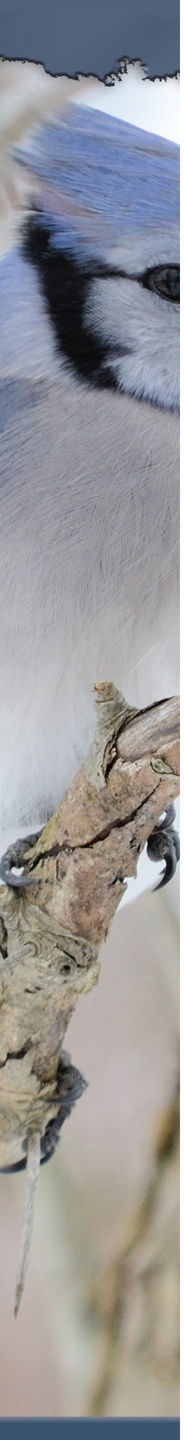
```
function dubble(objectWithData) {
    data = objectWithData.getData();
    if (data.typeof() == int) {
        return data.doubleInt();
    } else if (data.typeof() == String) {
        return data.doubleString();
    }
}
```

- Code works

```
dubble(5); → 10
dubble("To"); → "ToTo"
```

Procedural, not OO

- Code may work, but you'll get a lousy mark if you submit it 😭!
  - in this course
- Your code *asks* for data.
- Then, depending on data type, decides what objectWithData should to.



So…what makes code OO?

- Now consider

```
objectWithData.dubble();
```

- Your code ***tells*** object to double itself.
- Object knows what to do, whether int or String
  - We don't know / care how it does it.
  - Not our problem – details are hidden.
- Remember – "Tell, don’t ask".