ITI 1121. Introduction to Computing II[†]

Marcel Turcotte
(with contributions from R. Holte)

School of Electrical Engineering and Computer Science
University of Ottawa

Version of January 11, 2015

[†] Please don't print these lecture notes unless you really need to!

Great coders are today's rock stars
Will.I.Am
Watch the video at code.org!

Review

Objectives:

1. Discussing several concepts related to data types
2. Understanding the implications of the differences between primitive and reference types
3. Reviewing call-by-value
4. Understanding the concept of scope

Lectures:

- Pages 597–631 of E. Koffman and P. Wolfgang.

Plan

1. What are variables and data types
2. Primitive vs reference
3. Comparison operators (primitive vs reference)
4. Auto-boxing/auto-unboxing;
5. Passing parameters
6. Scope
7. Memory management

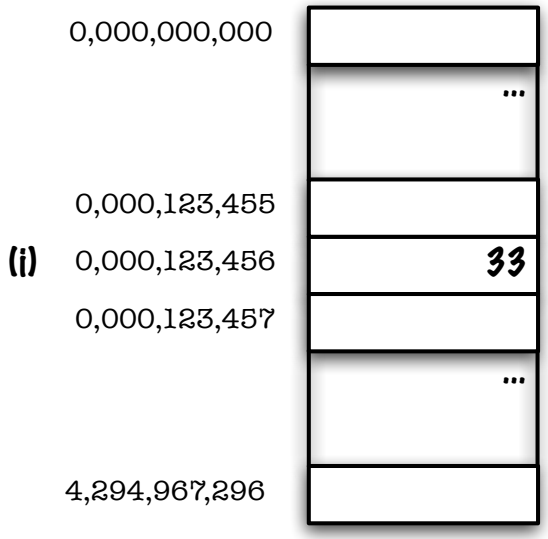
Variables

What is a variable?

Variables

What is a variable?

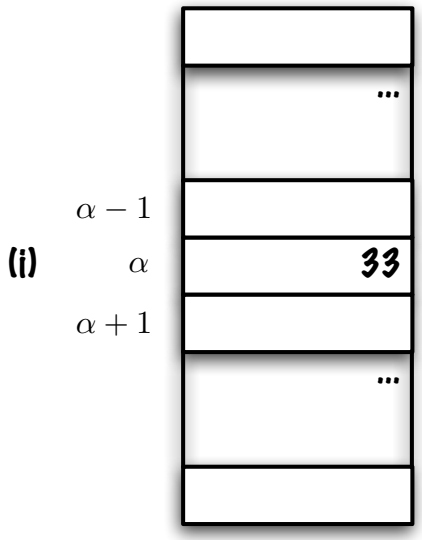
- A variable is a place in memory, to hold a **value**, which we refer to with help of a **label**




```
byte i = 33;
```

Variables

I will be using Greek letters to designate memory locations (addresses) since in Java we don't know the location of "objects" and should not care!




```
byte i = 33;
```

Data types

What are data types for?

Data types

What are data types for?

- Yes, it tells the compiler how much memory to allocate:

```
double formula;          // 8 bytes
char c;                  // 2 bytes
```

Data types

What are data types for?

- Yes, it tells the compiler how much memory to allocate:

```
double formula;          // 8 bytes
char c;                  // 2 bytes
```

- But it also?

Data types

What are data types for?

- Yes, it tells the compiler how much memory to allocate:

```
double formula;          // 8 bytes
char c;                  // 2 bytes
```

- But it also? It gives information about the meaning (semantic) of the data: **which operations are allowed**, which data are compatible.

Data types

What are data types for?

- Yes, it tells the compiler how much memory to allocate:

```
double formula;          // 8 bytes
char c;                  // 2 bytes
```

- But it also? It gives information about the meaning (semantic) of the data: **which operations are allowed**, which data are compatible. Hence the following statement,

```
c = flag * formula;
```

will produce an error at compile time; data types are therefore also useful to help detect errors in programs early on.

Data types (contd)

- To be more precise, there are **concrete data types** and **abstract data types**

Data types (contd)

- To be more precise, there are **concrete data types** and **abstract data types**
- Concrete data types specify both, the allowed operations **and** the representation of the data

Data types (contd)

- To be more precise, there are **concrete data types** and **abstract data types**
- Concrete data types specify both, the allowed operations **and** the representation of the data
- **Abstract Data Types (ADTs) specify only the allowed operations**

Data Types in Java

In Java, we have primitive and reference data types:

- Primitive are:

Data Types in Java

In Java, we have primitive and reference data types:

- Primitive are:
  - numbers, characters (but not Strings) and booleans

Data Types in Java

In Java, we have primitive and reference data types:

- Primitive are:
  - numbers, characters (but not Strings) and booleans
  - **the value is stored at the memory location designated by the label of the variable**

Data Types in Java

In Java, we have primitive and reference data types:

- Primitive are:
  - numbers, characters (but not Strings) and booleans
  - **the value is stored at the memory location designated by the label of the variable**
- References:
  - Predefined:

Data Types in Java

In Java, we have primitive and reference data types:

- Primitive are:
  - numbers, characters (but not Strings) and booleans
  - **the value is stored at the memory location designated by the label of the variable**
- References:
  - Predefined:
    - Arrays
    - Strings
  - User defined,

Data Types in Java

In Java, we have primitive and reference data types:

- Primitive are:
  - numbers, characters (but not Strings) and booleans
  - **the value is stored at the memory location designated by the label of the variable**
- References:
  - Predefined:
    - Arrays
    - Strings
  - User defined, reference to an instance of a class;

Data Types in Java

In Java, we have primitive and reference data types:

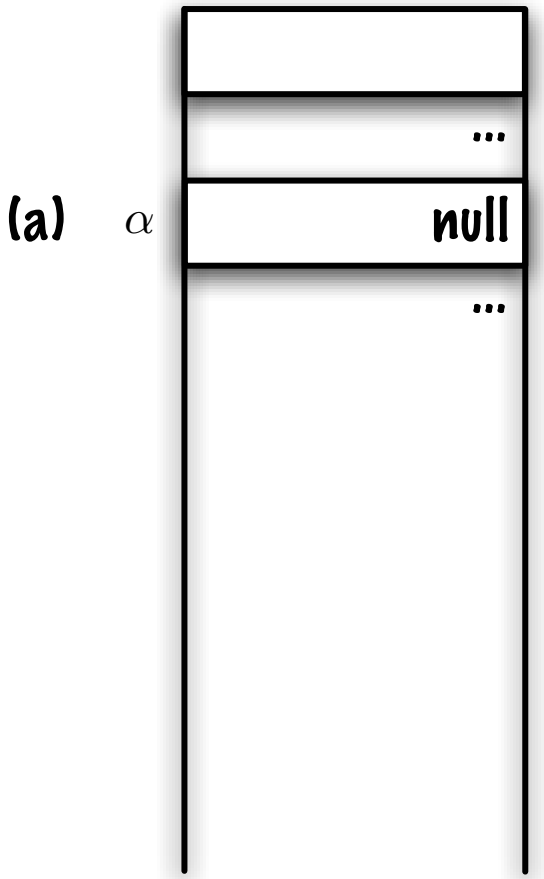
- Primitive are:
  - numbers, characters (but not Strings) and booleans
  - **the value is stored at the memory location designated by the label of the variable**
- References:
  - Predefined:
    - Arrays
    - Strings
  - User defined, reference to an instance of a class;
  - **The value of a reference variable is a memory location, which points/references to the location of an object; it is a pointer, a "link", it's a reference**

Data Types in Java

In Java, we have primitive and reference data types:

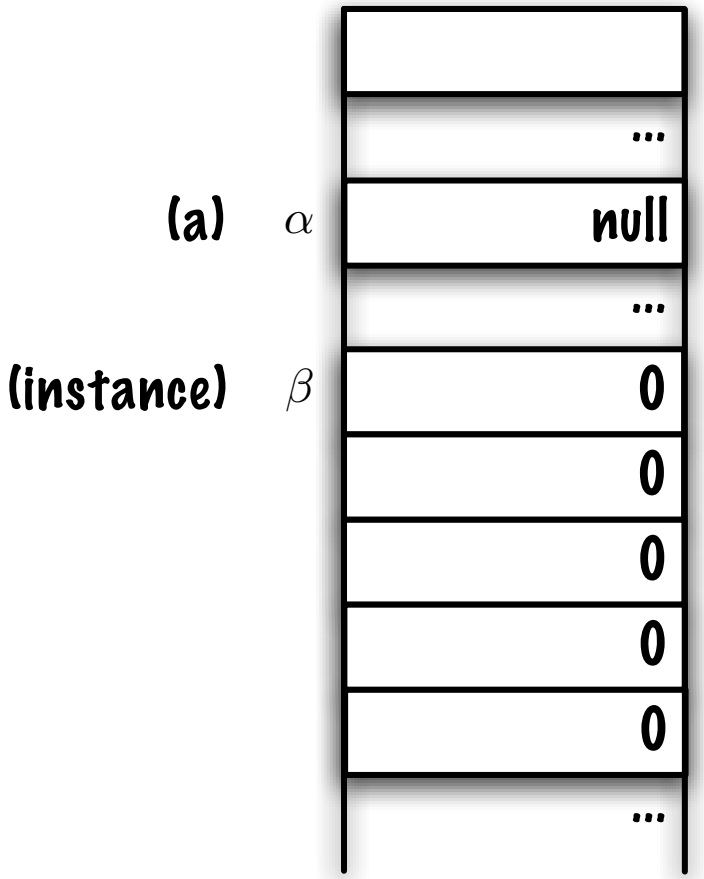
- Primitive are:
  - numbers, characters (but not Strings) and booleans
  - **the value is stored at the memory location designated by the label of the variable**
- References:
  - Predefined:
    - Arrays
    - Strings
  - User defined, reference to an instance of a class;
  - **The value of a reference variable is a memory location, which points/references to the location of an object; it is a pointer, a "link", it's a reference**
  - The declaration of a reference variable **does not create an object, does not allocate space for an object**, it only allocates memory to store the address of an object

```
> int[] a;
  a = new int[5];
```



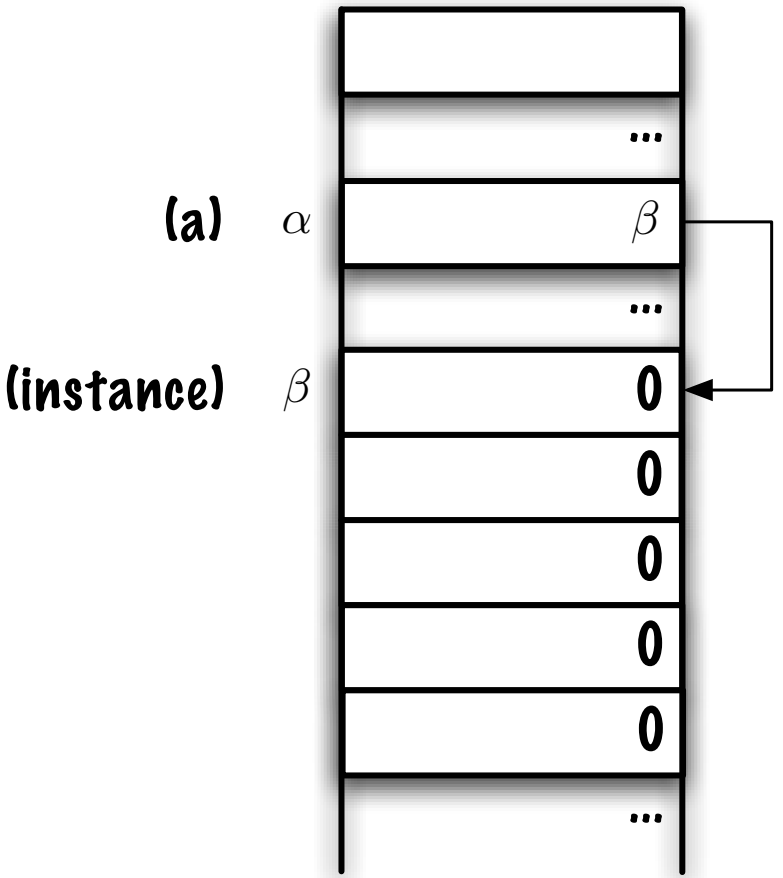
⇒ The declaration of a reference variable only allocates memory to hold a reference (sometimes called pointer or address), *null* is a special value (literal), which does not reference an object

```
   int[] a;
> a = new int[ 5 ];
```



⇒ The creation of a new instance, `new int[ 5 ]`, allocates memory to hold 5 integer values (and the housekeeping information). Each cell of the array is initialized with the default `int` value, which is 0.

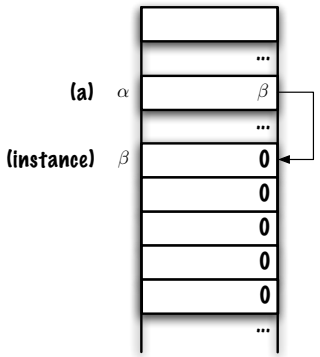
```
    int[] a;
>   a = new int[ 5 ];
```



⇒ Finally, the reference of the newly created object is assigned to the location designed by the label **a**.

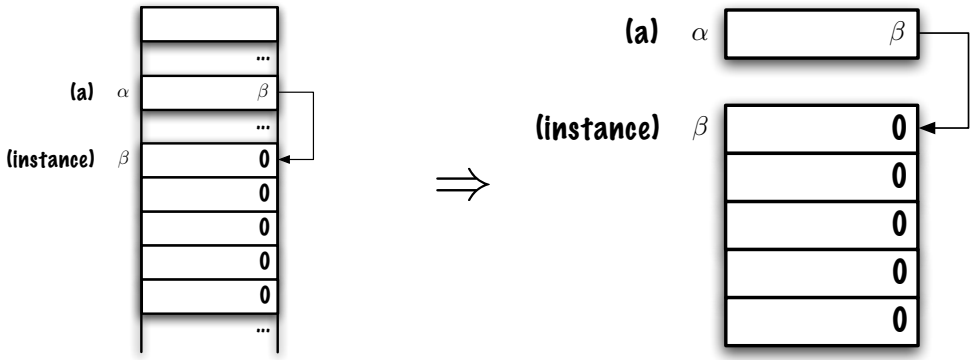
Memory diagram

Because we don't know (and shouldn't care) about the actual memory layout, we often use memory diagrams such as the following,



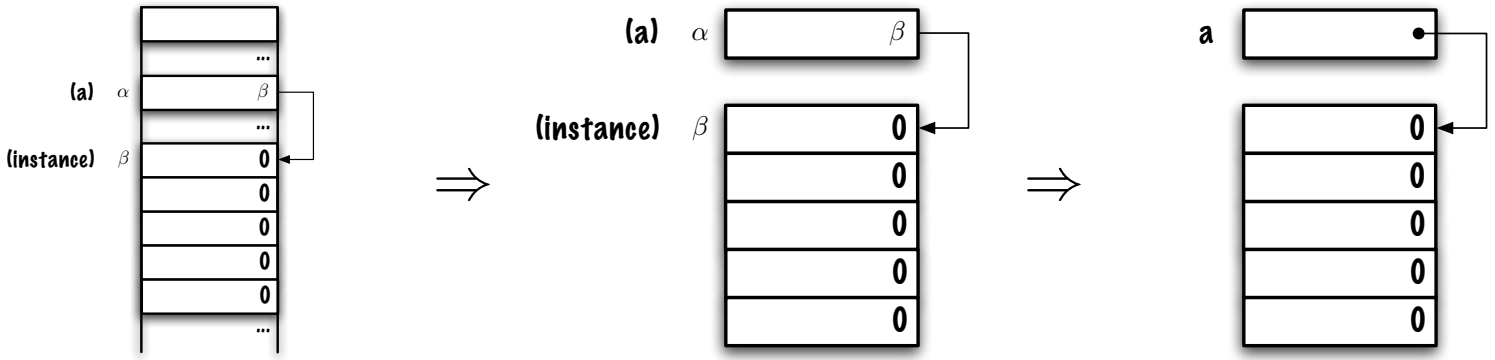
Memory diagram

Because we don't know (and shouldn't care) about the actual memory layout, we often use memory diagrams such as the following,



Memory diagram

Because we don't know (and shouldn't care) about the actual memory layout, we often use memory diagrams such as the following,



Memory diagram

In a memory diagram, I want to see:

- a box for every reference variable with an arrow pointing a the designated object
- a box for every primitive variable with the value inside the box
- a box for every object

```
int[] a;
a = new int[ 5 ];
```

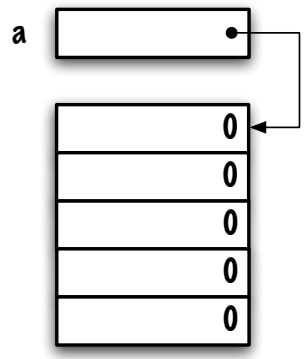
Memory diagram

In a memory diagram, I want to see:

- a box for every reference variable with an arrow pointing a the designated object
- a box for every primitive variable with the value inside the box
- a box for every object

```
int[] a;
a = new int[ 5 ];
```

$\Rightarrow$



Memory diagram

Given the following class declaration:

```
public class Constant {
    private final String name;
    private final double value;
    public Constant( String name, double value ) {
        this.name = name;
        this.value = value;
    }
}
```

Memory diagram

Given the following class declaration:

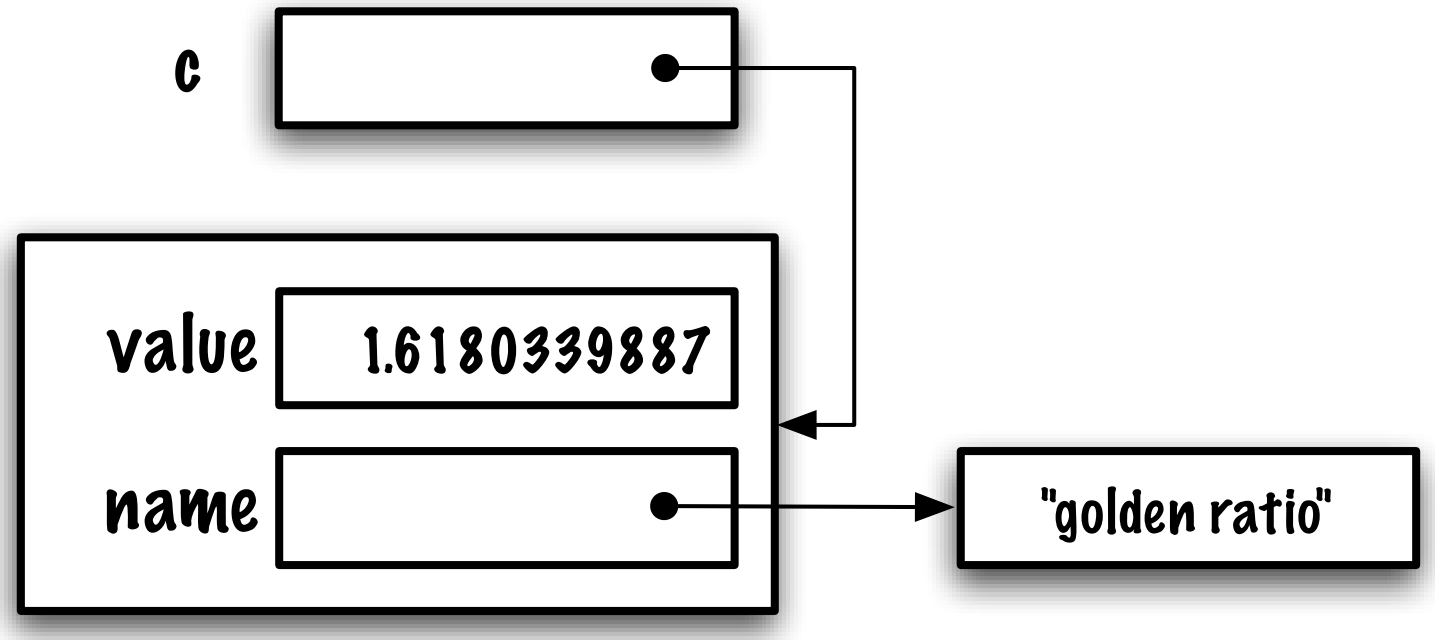
```
public class Constant {
    private final String name;
    private final double value;
    public Constant( String name, double value ) {
        this.name = name;
        this.value = value;
    }
}
```

Draw the memory diagram for the following statments:

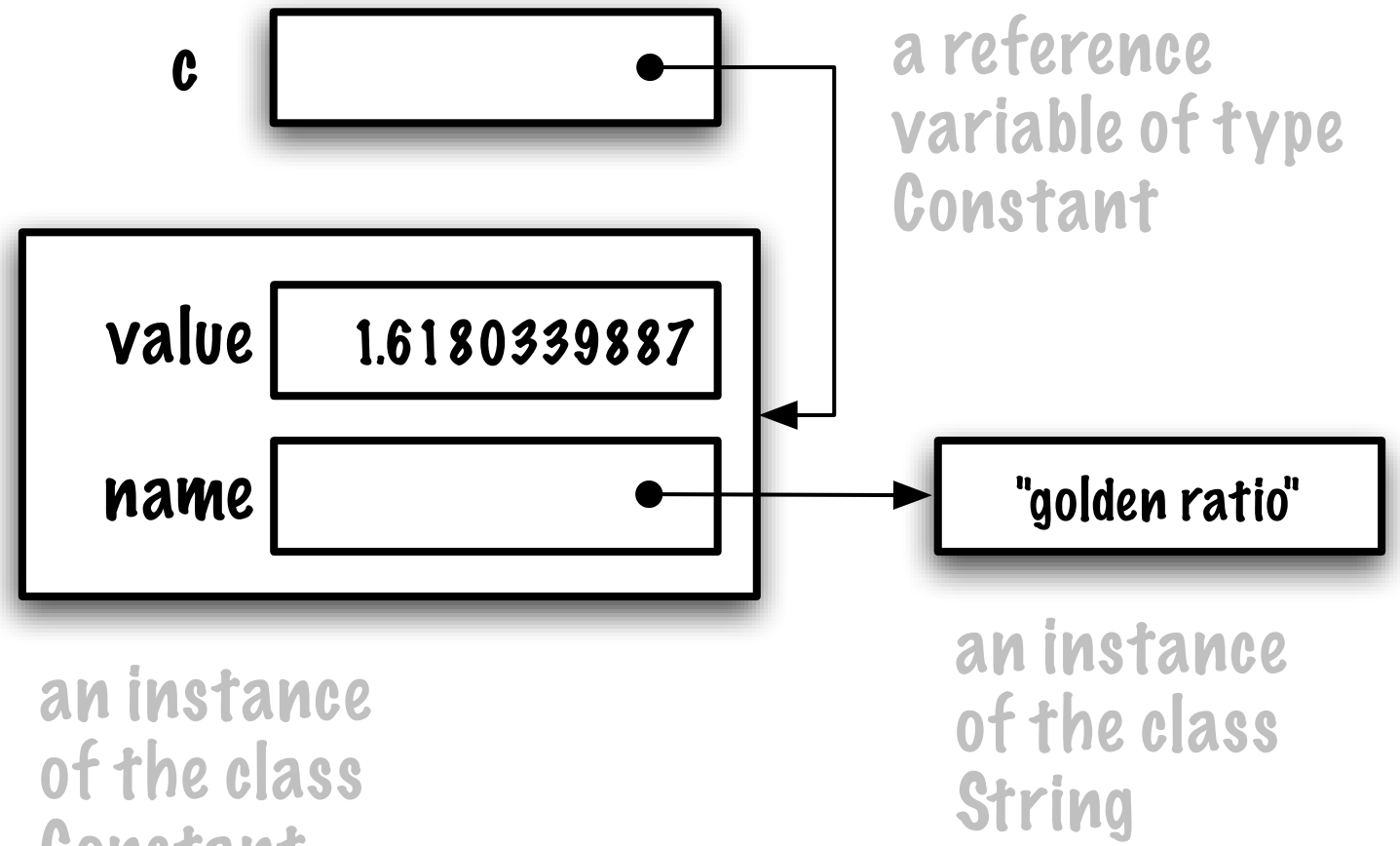
```
Constant c;
c = new Constant( "golden ratio", 1.61803399 );
```

Memory diagram

Memory diagram



Memory diagram



Remember

- **Variables have types**
- **Objects have classes**

Class Integer

In the following examples, we'll be using our own class **Integer**:

```
class Integer {
    int value;
}
```

Class Integer

In the following examples, we'll be using our own class **Integer**:

```
class Integer {
    int value;
}
```

Usage:

```
Integer a;
a = new Integer();
a.value = 33;
a.value++;
System.out.println( "a.value = " + a.value );
```

We use the dot notation to access the value of an instance variable.

Class Integer

In the following examples, we'll be using our own class **Integer**:

```
class Integer {
    int value;
}
```

Usage:

```
Integer a;
a = new Integer();
a.value = 33;
a.value++;
System.out.println( "a.value = " + a.value );
```

We use the dot notation to access the value of an instance variable.
Java has a pre-defined class named **Integer**. It is called a "wrapper" class; it wraps an **int** value into an object.

Class Integer

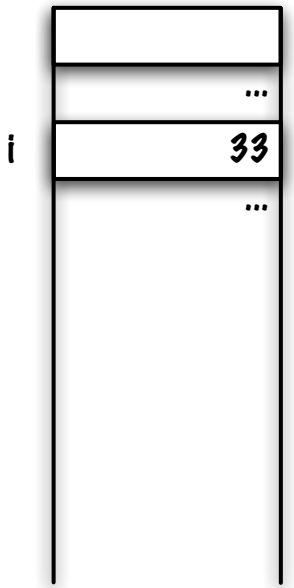
Adding a constructor

```
class Integer {
    int value;
    Integer( int v ) {
        value = v;
    }
}
```

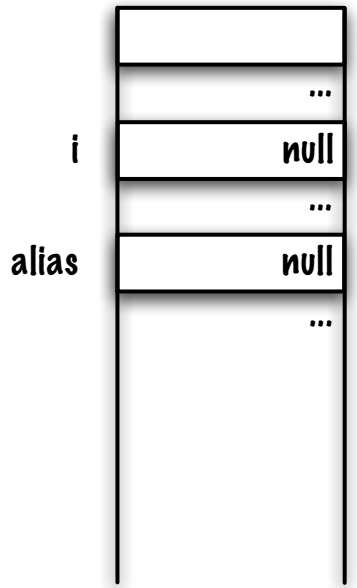
Usage:

```
Integer a;
a = new Integer( 33 );
```

Primitive vs reference variables



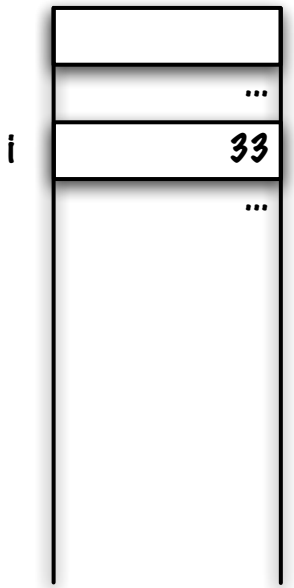

```
int i = 33;
```

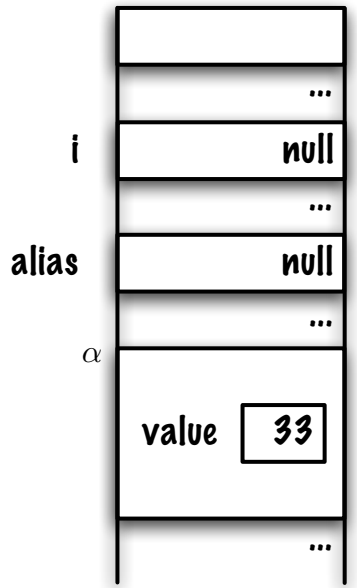



```
Integer i, alias;
i = new Integer( 33 );
alias = i;
```

At compile time the necessary memory to hold the reference is allocated

Primitive vs reference variables



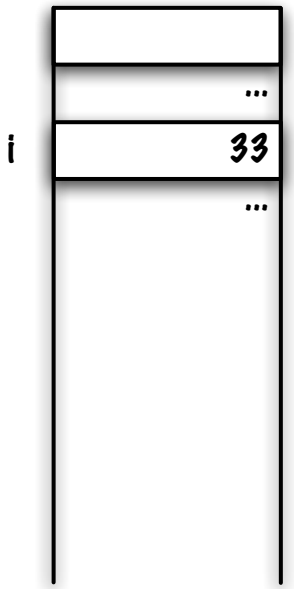


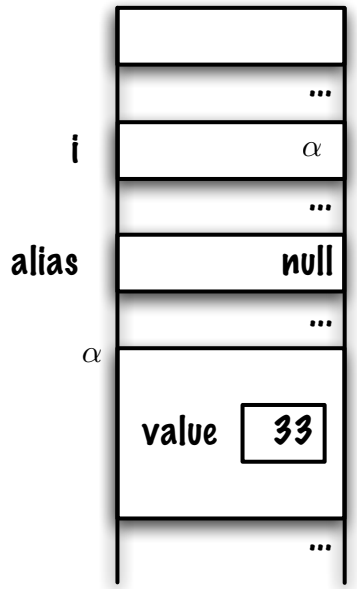

```
int i = 33;
```

```
Integer i, alias;
i = new Integer( 33 );
alias = i;
```

Creating an object (**new Integer(33)**)

Primitive vs reference variables



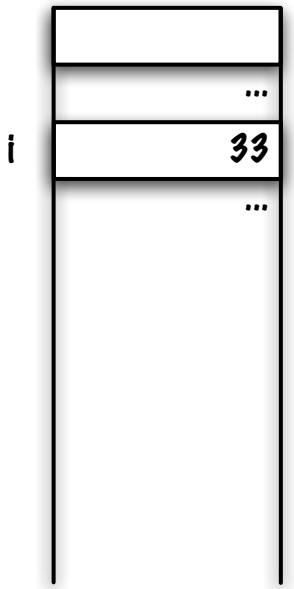



```
int i = 33;
```

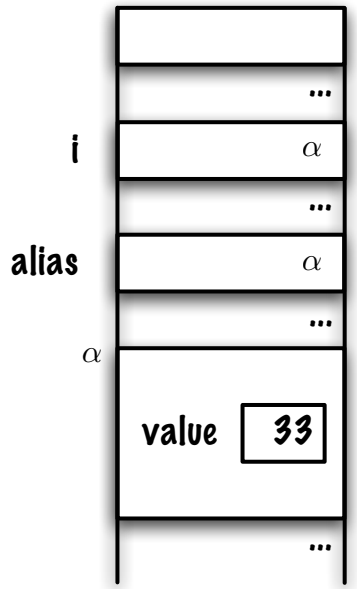
```
Integer i, alias;
i = new Integer( 33 );
alias = i;
```

Assigning the reference of that object to the reference variable **i**

Primitive vs reference variables



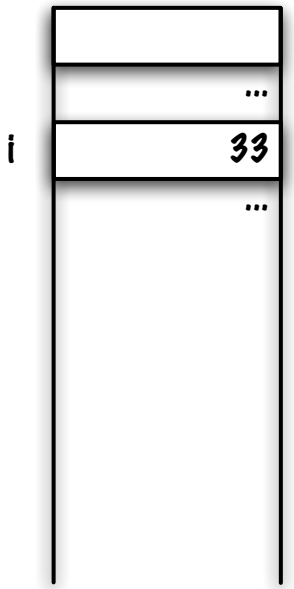

```
int i = 33;
```



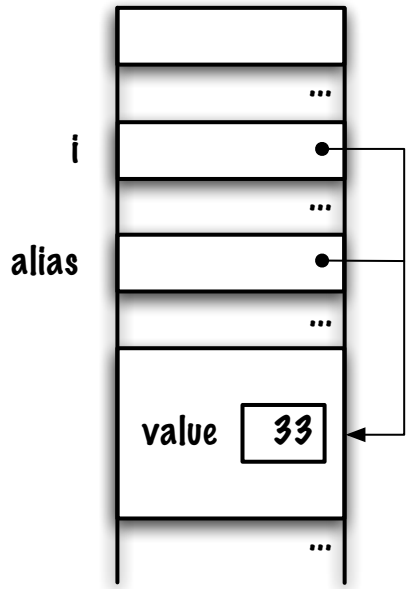

```
Integer i, alias;
i = new Integer( 33 );
alias = i;
```

Copying the value of the reference variable **i** into **alias**

Primitive vs reference variables




```
int i = 33;
```

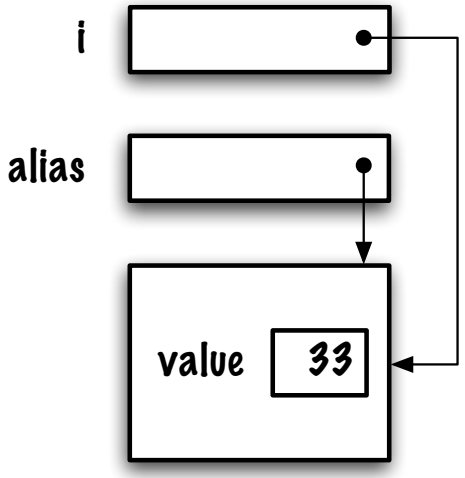



```
Integer i, alias;
i = new Integer( 33 );
alias = i;
```

i and **alias** are both designating the same object!

Primitive vs reference variables


```
int i = 33;
```




```
Integer i, alias;
i = new Integer( 33 );
alias = i;
```

Using the memory diagram representation

Wrappers

- For every **primitive type** there is an associated **wrapper class**

Wrappers

- For every **primitive type** there is an associated **wrapper class**
- For instance, **Integer** is the wrapper class for the primitive type **int**

Wrappers

- For every **primitive type** there is an associated **wrapper class**
- For instance, **Integer** is the wrapper class for the primitive type **int**
- A wrapper stores a value of a primitive type inside an object

Wrappers

- For every **primitive type** there is an associated **wrapper class**
- For instance, **Integer** is the wrapper class for the primitive type **int**
- A wrapper stores a value of a primitive type inside an object
- This will be paramount for stacks, queues, lists and trees

Wrappers

- For every **primitive type** there is an associated **wrapper class**
- For instance, **Integer** is the wrapper class for the primitive type **int**
- A wrapper stores a value of a primitive type inside an object
- This will be paramount for stacks, queues, lists and trees
- Besides holding a value, the wrapper classes possess several class methods, mainly to convert values from/to other types, e.g. **Integer.parseInt("33")**

Quiz

This is valid Java statement, **true** or **false**?

```
Integer i = 1;
```

Quiz

This is valid Java statement, **true** or **false**?

```
Integer i = 1;
```

- If this is a valid statement, what are the implications?

Quiz

This is valid Java statement, **true** or **false**?

```
Integer i = 1;
```

- If this is a valid statement, what are the implications?
- Okay, 1 is a value of a primitive type, **int**, but **i** is a reference variable, of type **Integer**

Quiz

This is valid Java statement, **true** or **false**?

```
Integer i = 1;
```

- If this is a valid statement, what are the implications?
- Okay, 1 is a value of a primitive type, **int**, but **i** is a reference variable, of type **Integer**
- In Java 1.4 or older, this would cause a **compile-time** error!

Quiz

This is valid Java statement, **true** or **false**?

```
Integer i = 1;
```

- If this is a valid statement, what are the implications?
- Okay, 1 is a value of a primitive type, **int**, but **i** is a reference variable, of type **Integer**
- In Java 1.4 or older, this would cause a **compile-time** error!
- However, in Java 5, 6 or 7, this is a valid statement!

Quiz

This is valid Java statement, **true** or **false**?

```
Integer i = 1;
```

- If this is a valid statement, what are the implications?
- Okay, 1 is a value of a primitive type, **int**, but **i** is a reference variable, of type **Integer**
- In Java 1.4 or older, this would cause a **compile-time** error!
- However, in Java 5, 6 or 7, this is a valid statement!

Quiz

This is valid Java statement, **true** or **false**?

```
Integer i = 1;
```

- If this is a valid statement, what are the implications?
- Okay, 1 is a value of a primitive type, **int**, but **i** is a reference variable, of type **Integer**
- In Java 1.4 or older, this would cause a **compile-time** error!
- However, in Java 5, 6 or 7, this is a valid statement! Why?

Auto-boxing

This is because Java 5, 6 and 7 **automagically** transform the following statement

```
Integer i = 1;
```

into

```
Integer i = new Integer( 1 );
```

Auto-boxing

This is because Java 5, 6 and 7 **automagically** transform the following statement

```
Integer i = 1;
```

into

```
Integer i = new Integer( 1 );
```

This is called **auto-boxing**.

Auto-unboxing

Similarly, the statement **i = i + 5**

```
Integer i = 1;
i = i + 5;
```

is transformed into

```
i = new Integer( i.intValue() + 5 );
```

where the value of the wrapper object designated by **i** is extracted, **unboxed**, with the method call, **i.intValue()**.

Boxing/unboxing

Primitive	Reference
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
bool	Boolean
char	Character

All 8 primitive types have a corresponding **wrapper** class.

Boxing/unboxing

Primitive	Reference
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
bool	Boolean
char	Character

All 8 primitive types have a corresponding **wrapper** class. The automatic conversion from primitive to reference type is called **boxing**,

Boxing/unboxing

Primitive	Reference
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
bool	Boolean
char	Character

All 8 primitive types have a corresponding **wrapper** class. The automatic conversion from primitive to reference type is called **boxing**, and the conversion from reference to primitive type is called **unboxing**.

Does it matter?

```
long s1 = (long) 0;
for ( j=0; j<10000000; j++ ) {
    s1 = s1 + (long) 1;
}
```

```
Long s2 = (long) 0;
for ( j=0; j<10000000; j++ ) {
    s2 = s2 + (long) 1;
}
```

Does it matter?

```
long s1 = (long) 0;
for ( j=0; j<10000000; j++ ) {
    s1 = s1 + (long) 1;
}
```

49 milliseconds

```
Long s2 = (long) 0;
for ( j=0; j<10000000; j++ ) {
    s2 = s2 + (long) 1;
}
```

340 milliseconds

Does it matter?

```
long s1 = (long) 0;
for ( j=0; j<10000000; j++ ) {
    s1 = s1 + (long) 1;
}
```

49 milliseconds

```
Long s2 = (long) 0;
for ( j=0; j<10000000; j++ ) {
    s2 = s2 + (long) 1;
}
```

340 milliseconds

- Why?

Does it matter?

```
long s1 = (long) 0;
for ( j=0; j<10000000; j++ ) {
    s1 = s1 + (long) 1;
}
```

49 milliseconds

```
Long s2 = (long) 0;
for ( j=0; j<10000000; j++ ) {
    s2 = s2 + (long) 1;
}
```

340 milliseconds

- Why?

On the right side, **s2** is declared as a **Long**, hence, the line,

```
s2 = s2 + (long) 1;
```

is rewritten as,

```
s2 = new Long( s2.longValue() + (long) 1 );
```

Programming tip: benchmarking your code

```
long start, stop, elapsed;

start = System.currentTimeMillis(); // start the clock

for ( j=0; j<10000000; j++ ) {
    s2 += (long) 1; // stands for 's2 = s2 + (long) 1'
}

stop = System.currentTimeMillis(); // stop the clock

elapsed = stop - start;
```

Programming tip: benchmarking your code

```
long start, stop, elapsed;

start = System.currentTimeMillis(); // start the clock

for ( j=0; j<10000000; j++ ) {
    s2 += (long) 1; // stands for 's2 = s2 + (long) 1'
}

stop = System.currentTimeMillis(); // stop the clock

elapsed = stop - start;
```

where **System.currentTimeMillis()** returns the number of milliseconds elapsed since midnight, January 1, 1970 UTC (Coordinated Universal Time).
System.nanoTime() also exists.

Billion-dollar mistake (Null reference)

"I call it my billion-dollar mistake. It was the invention of the null reference in 1965. At that time, I was designing the first comprehensive type system for references in an object oriented language (ALGOL W). My goal was to ensure that all use of references should be absolutely safe, with checking performed automatically by the compiler. **But I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement.** This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years."

Tony Hoare

Comparison operators – primitive data types

Variables of primitive data types can be compared directly

```
int a = 5;
int b = 10;

if ( a < b ) {
   System.out.println( "a < b" );
} else if ( a == b ) {
   System.out.println( "a == b" );
} else {
   System.out.println( "a > b" );
}
```

What will be printed out on the output?

Comparison operators – primitive data types

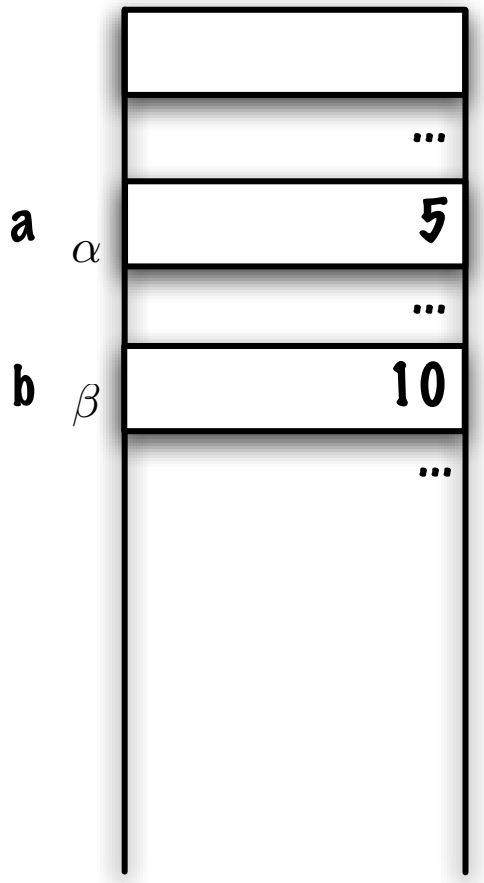
Variables of primitive data types can be compared directly

```
int a = 5;
int b = 10;

if ( a < b ) {
   System.out.println( "a < b" );
} else if ( a == b ) {
   System.out.println( "a == b" );
} else {
   System.out.println( "a > b" );
}
```

What will be printed out on the output?

$\Rightarrow$ Prints "$a < b$"

```
int a = 5;
int b = 10;

if ( a < b ) {
  System.out.println( "a < b" );
} else if ( a == b ) {
  System.out.println( "a == b" );
} else {
  System.out.println( "a > b" );
}
```



Comparison operators: primitive and reference types

What will happen and why?

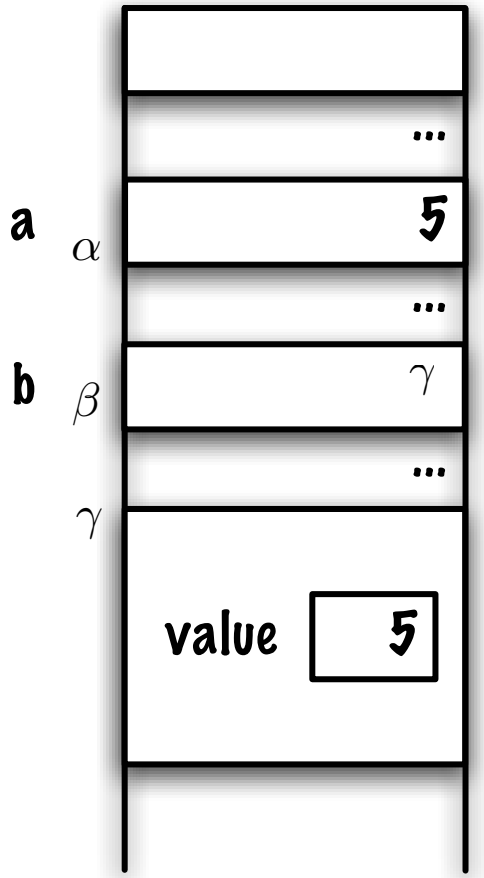
```
int a = 5;
Integer b = new Integer( 5 );
if ( a < b ) {
    System.out.println( "a < b" );
} else if (a == b) {
    System.out.println( "a == b" );
} else {
    System.out.println( "a > b" );
}
```

Comparison operators: primitive and reference types

What will happen and why?

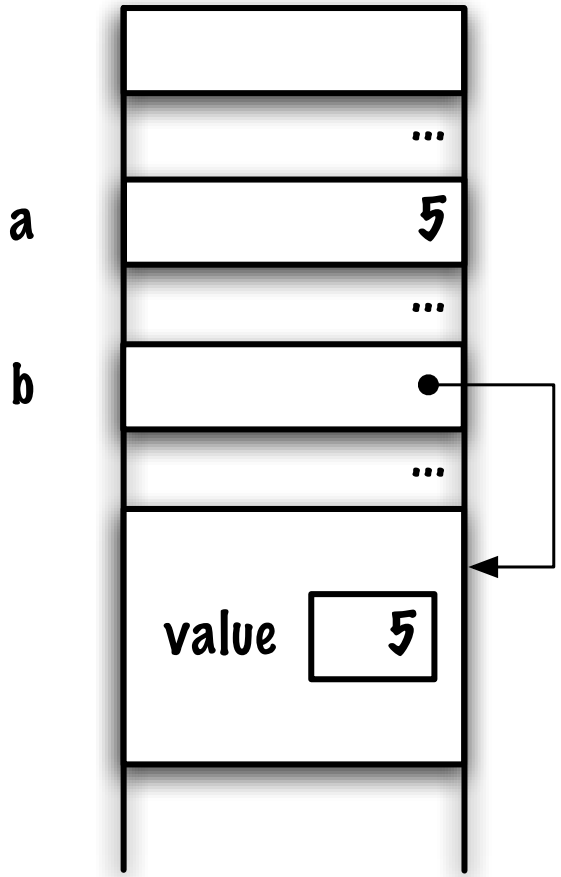
```
int a = 5;
Integer b = new Integer( 5 );
if ( a < b ) {
    System.out.println( "a < b" );
} else if (a == b) {
    System.out.println( "a == b" );
} else {
    System.out.println( "a > b" );
}
```

```
References.java:7: operator < cannot be applied to int,java.lang.Integer
        if (a < b)
              ^
References.java:9: operator == cannot be applied to int,java.lang.Integer
        else if (a == b)
                   ^
2 errors
```




```java
int a = 5;
Integer b = new Integer( 5 );

if ( a < b ) {
    System.out.println( "a < b" );
} else if ( a == b ) {
    System.out.println( "a == b" );
} else {
    System.out.println( "a > b" );
}
```

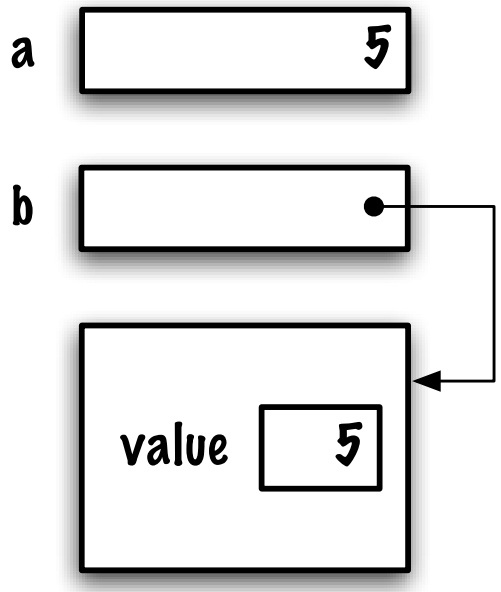
```
References.java:7: operator < cannot be applied to int,java.lang.Integer
        if (a < b)
              ^
References.java:9: operator == cannot be applied to int,java.lang.Integer
        else if (a == b)
                   ^
2 errors
```




```
int a = 5;
Integer b = new Integer( 5 );

if ( a < b ) {
    System.out.println( "a < b" );
} else if ( a == b ) {
    System.out.println( "a == b" );
} else {
    System.out.println( "a > b" );
}
```

```
References.java:7: operator < cannot be applied to int,java.lang.Integer
        if (a < b)
              ^
References.java:9: operator == cannot be applied to int,java.lang.Integer
        else if (a == b)
                   ^
2 errors
```

```
int a = 5;
Integer b = new Integer( 5 );

if ( a < b ) {
    System.out.println( "a < b" );
} else if ( a == b ) {
    System.out.println( "a == b" );
} else {
    System.out.println( "a > b" );
}
```




```
References.java:7: operator < cannot be applied to int,java.lang.Integer
        if (a < b)
              ^
References.java:9: operator == cannot be applied to int,java.lang.Integer
        else if (a == b)
                   ^
2 errors
```

Comparison operators and reference types

What will be the result?

```
MyInteger a = new MyInteger( 5 );
MyInteger b = new MyInteger( 10 );

if ( a < b ) {
  System.out.println( "a equals b" );
} else {
  System.out.println( "a does not equal b" );
}
```

Comparison operators and reference types

What will be the result?

```
MyInteger a = new MyInteger( 5 );
MyInteger b = new MyInteger( 10 );

if ( a < b ) {
  System.out.println( "a equals b" );
} else {
  System.out.println( "a does not equal b" );
}
```

```
Less.java:14: operator < cannot be applied to MyInteger,MyInteger
        if ( a < b ) {
               ^
1 error
```

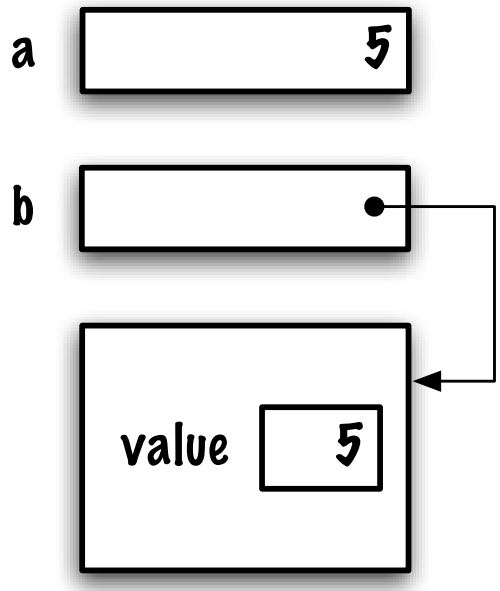
Remarks

- These error messsages are produced by pre-1.5 Java compilers
- Starting with Java 1.5, autoboxing masks the "problem"
- In order to get same behaviour with the two environments, let's use our wrapper, **MyInteger**

Class MyInteger

```
class MyInteger {
    int value;
    MyInteger( int v ) {
        value = v;
    }
}
```

```
int a = 5;
MyInteger b = new MyInteger( 5 );

if ( a < b ) {
    System.out.println( "a < b" );
} else if ( a == b ) {
    System.out.println( "a == b" );
} else {
    System.out.println( "a > b" );
}
```



- Fix this!

Solution

Solution

```
int a = 5;
MyInteger b = new MyInteger( 5 );

if ( a < b.value ) {
    System.out.println( "a is less than b" );
} else if ( a == b.value ) {
    System.out.println( "a equals b" );
} else {
    System.out.println( "a is greater than b" );
}
```

$\Rightarrow$ Prints "**a equals b**".

Comparison operators and reference types

What will happen and why?

```
MyInteger a = new MyInteger( 5 );
MyInteger b = new MyInteger( 5 );

if ( a == b ) {
    System.out.println( "a equals b" );
} else {
    System.out.println( "a does not equal b" );
}
```

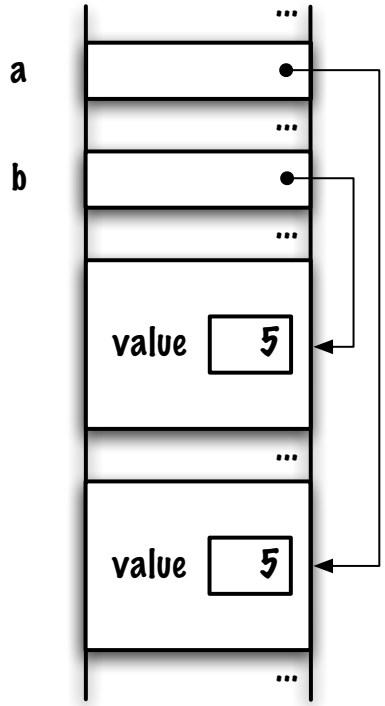
Comparison operators and reference types

What will happen and why?

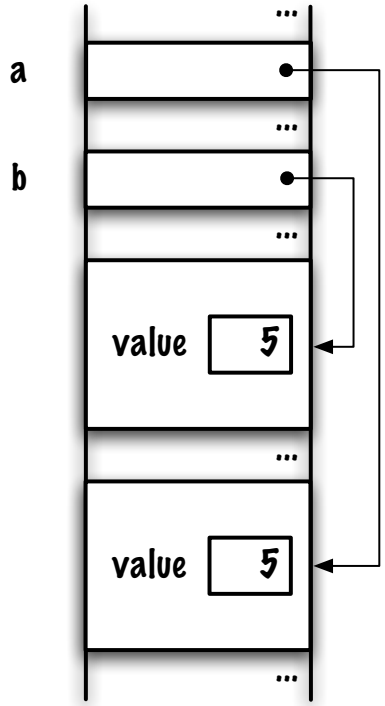
```
MyInteger a = new MyInteger( 5 );
MyInteger b = new MyInteger( 5 );

if ( a == b ) {
    System.out.println( "a equals b" );
} else {
    System.out.println( "a does not equal b" );
}
```

$\Rightarrow$ The result is "**a does not equal b**".

```
MyInteger a = new MyInteger( 5 );
MyInteger b = new MyInteger( 5 );

if ( a == b ) {
  System.out.println( "a equals b" );
} else {
  System.out.println( "a does not equals b" );
}
```

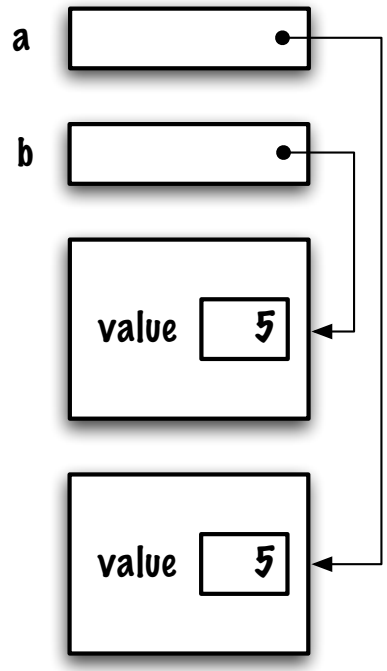


```
MyInteger a = new MyInteger( 5 );
MyInteger b = new MyInteger( 5 );

if ( a == b ) {
  System.out.println( "a equals b" );
} else {
  System.out.println( "a does not equals b" );
}
```



$\Rightarrow$ The result is "**a does not equal b**".

```
MyInteger a = new MyInteger( 5 );
MyInteger b = new MyInteger( 5 );

if ( a == b ) {
  System.out.println( "a equals b" );
} else {
  System.out.println( "a does not equals b" );
}
```



⇒ The result is "**a does not equal b**".

Solution

Solution

```
MyInteger a = new MyInteger( 5 );
MyInteger b = new MyInteger( 5 );

if ( a.equals( b ) ) {
  System.out.println( "a equals b" );
} else {
  System.out.println( "a does not b" );
}
```

where **equals** could have been defined as an instance method:

```
public boolean equals( MyInteger other ) {
  returns this.value == other.value;
}
```

$\Rightarrow$ Would print "**a equals b**".

What will happen?

```
MyInteger a = new MyInteger( 5 );
MyInteger b = a;

if ( a == b ) {
  System.out.println( "a == b" );
} else {
  System.out.println( "a != b" );
}
```

What will happen?

```
MyInteger a = new MyInteger( 5 );
MyInteger b = a;

if ( a == b ) {
  System.out.println( "a == b" );
} else {
  System.out.println( "a != b" );
}
```

$\Rightarrow$ Prints "**a == b**", why?

What will happen?

```
MyInteger a = new MyInteger( 5 );
MyInteger b = a;

if ( a == b ) {
  System.out.println( "a == b" );
} else {
  System.out.println( "a != b" );
}
```

$\Rightarrow$ Prints "**a == b**", why? because **a** and **b** reference the same object (instance), in other words, the two memory locations are the same; we say that **b** is an **alias** for **a**.

Comparison operators and reference types

What will happen?

```
MyInteger a = new MyInteger( 5 );
MyInteger b = a;

if ( a.equals( b ) ) {
  System.out.println( "a equals b" );
} else {
  System.out.println( "a does not equal b" );
}
```

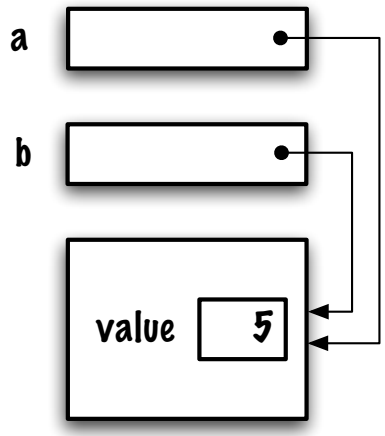
Comparison operators and reference types

What will happen?

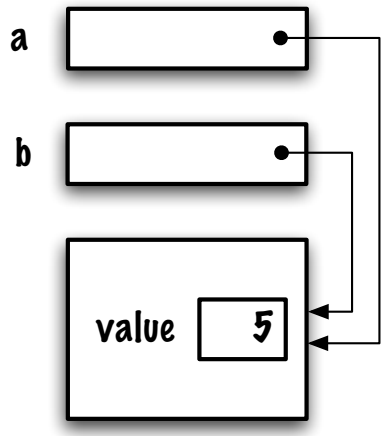
```
MyInteger a = new MyInteger( 5 );
MyInteger b = a;

if ( a.equals( b ) ) {
  System.out.println( "a equals b" );
} else {
  System.out.println( "a does not equal b" );
}
```

$\Rightarrow$ prints "**a equals b**" because the two values are equal.

```
MyInteger a = new MyInteger (5);
MyInteger b = a;

if (a == b) {
  System.out.println("a == b");
} else {
  System.out.println("a != b");
}
```



```
MyInteger a = new MyInteger (5);
MyInteger b = a;

if ( a.equals( b ) ) {
  System.out.println("a == b");
} else {
  System.out.println("a != b");
}
```



Remarks

- Two reference variables designate objects that are "logically equivalent" if these objects have the same "content", use the method **equals** to test for "content or logical equality"[‡]

```
if ( a.equals( b ) ) { ... }
```

vs

```
if ( a == b ) { ... }
```

[‡]We will continue the discussion when learning about inheritance!

Remarks

- Two reference variables designate objects that are "logically equivalent" if these objects have the same "content", use the method **equals** to test for "content or logical equality"[‡]
- In order to test if two reference variables **designate the same object**, use the comparison operators **'=='** and **'!='**

```
if ( a.equals( b ) ) { ... }
```

vs

```
if ( a == b ) { ... }
```

[‡]We will continue the discussion when learning about inheritance!

Exercises

Experiment comparing these objects using **equals** and ==, you might be surprised by the results.

```
String a = new String( "Hello" );
String b = new String( "Hello" );
int c[] = { 1, 2, 3 };
int d[] = { 1, 2, 3 };
String e = "Hello";
String f = "Hello";
String g = f + "";
```

In particular, try **a == b** and **e == f**.

Definition: arity

The **arity** of a method is simply the number of parameters; a method may have no parameter, one parameter or many.

```
MyInteger() {
   this.value = 0;
}
MyInteger( int v ) {
   this.value = v;
}
int sum( int a, int b ) {
   return a + b;
}
```

Definition: formal parameter

A **formal parameter** is a variable which is part of the definition of the method; it can be seen as a local variable of the body of the method.

```
int sum( int a, int b ) {
    return a + b;
}
```

$\Rightarrow$ **a** and **b** are formal parameters of **sum**.

Definition: actual parameter

An **actual parameter** is the variable which is used when the method is called to supply the value for the formal parameter.

```
int sum( int a, int b ) {
    return a + b;
}
...
int midTerm, finalExam, total;
total = sum( midTerm, finalExam );
```

midTerm and **finalExam** are actual parameters of **sum**, when the method is called the value of the actual parameters is copied to the location of the formal parameters.

Concept: call-by-value

Concept: call-by-value

In Java, when a method is called, the **values** of the actual parameters are **copied** to the location of the formal parameters.

Concept: call-by-value

In Java, when a method is called, the **values** of the actual parameters are **copied** to the location of the formal parameters.

When a method is called:

- the execution of the calling method is stopped

Concept: call-by-value

In Java, when a method is called, the **values** of the actual parameters are **copied** to the location of the formal parameters.
When a method is called:

- the execution of the calling method is stopped
- an activation frame (activation block or record) is created (it contains the formal parameters as well as the local variables)

Concept: call-by-value

In Java, when a method is called, the **values** of the actual parameters are **copied** to the location of the formal parameters.
When a method is called:

- the execution of the calling method is stopped
- an activation frame (activation block or record) is created (it contains the formal parameters as well as the local variables)
- the value of the actual parameters are copied to the location of the formal parameters

Concept: call-by-value

In Java, when a method is called, the **values** of the actual parameters are **copied** to the location of the formal parameters.
When a method is called:

- the execution of the calling method is stopped
- an activation frame (activation block or record) is created (it contains the formal parameters as well as the local variables)
- the value of the actual parameters are copied to the location of the formal parameters
- the body of the method is executed

Concept: call-by-value

In Java, when a method is called, the **values** of the actual parameters are **copied** to the location of the formal parameters.
When a method is called:

- the execution of the calling method is stopped
- an activation frame (activation block or record) is created (it contains the formal parameters as well as the local variables)
- the value of the actual parameters are copied to the location of the formal parameters
- the body of the method is executed
- a return value or (void) is saved

Concept: call-by-value

In Java, when a method is called, the **values** of the actual parameters are **copied** to the location of the formal parameters.
When a method is called:

- the execution of the calling method is stopped
- an activation frame (activation block or record) is created (it contains the formal parameters as well as the local variables)
- the value of the actual parameters are copied to the location of the formal parameters
- the body of the method is executed
- a return value or (void) is saved
- (the activation frame is destroyed)

Concept: call-by-value

In Java, when a method is called, the **values** of the actual parameters are **copied** to the location of the formal parameters.
When a method is called:

- the execution of the calling method is stopped
- an activation frame (activation block or record) is created (it contains the formal parameters as well as the local variables)
- the value of the actual parameters are copied to the location of the formal parameters
- the body of the method is executed
- a return value or (void) is saved
- (the activation frame is destroyed)
- the execution of the calling method restarts with the next instruction

```
public class Test {

  public static void increment( int a ) {
    a = a + 1;
  }

  public static void main( String[] args ) {
    int a = 5;
    System.out.println( "before: " + a );
    increment( a );
    System.out.println( "after: " + a );
  }
}
```

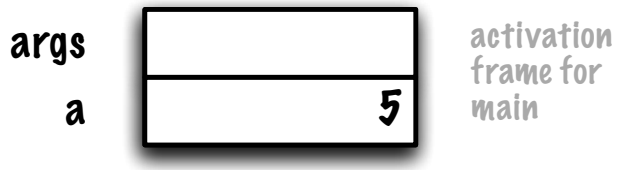
What will printed?

```
public class Test {

  public static void increment( int a ) {
    a = a + 1;
  }

  public static void main( String[] args ) {
    int a = 5;
    System.out.println( "before: " + a );
    increment( a );
    System.out.println( "after: " + a );
  }
}
```

What will printed?

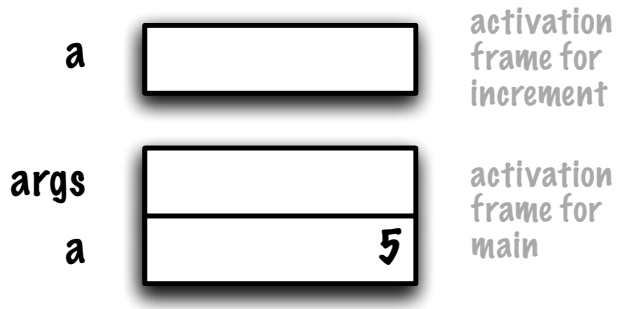
```
before: 5
after: 5
```

```
   public static void increment( int a ) {
     a = a + 1;
   }
   public static void main( String[] args ) {
>      int a = 5;
       increment( a );
   }
```



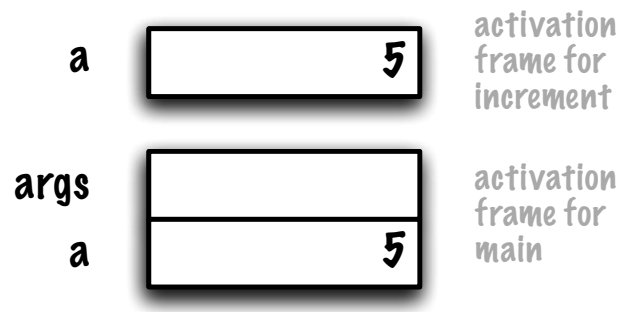
Each **method call** has its own **activation frame**, which holds the parameters and local variables (here, **args** and **a**)

```
   public static void increment( int a ) {
     a = a + 1;
   }
   public static void main( String[] args ) {
       int a = 5;
>      increment( a );
   }
```



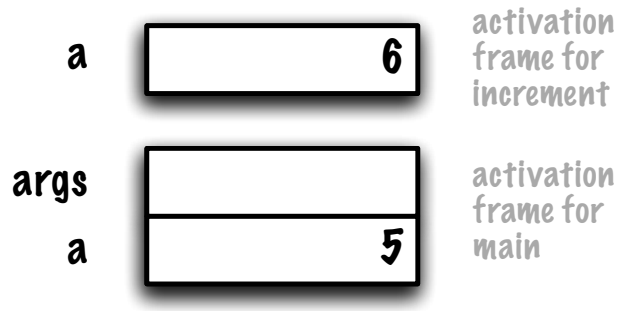
When **increment** is called a new activation frame is created

```
   public static void increment( int a ) {
     a = a + 1;
   }
   public static void main( String[] args ) {
       int a = 5;
>      increment( a );
   }
```



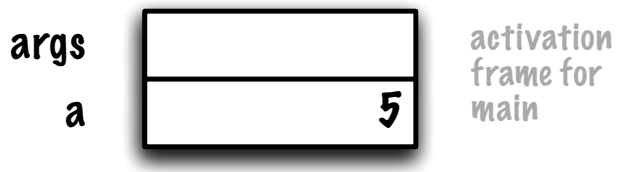
The value of the **actual parameter** is copied to the location of the **formal parameter**

```
   public static void increment( int a ) {
>      a = a + 1;
   }
   public static void main( String[] args ) {
      int a = 5;
      increment( a );
   }
```



The execution of **a = a + 1** changes the content of the formal parameter **a**, which is a distinct memory location the local variable **a** of the **main** method

```
   public static void increment( int a ) {
       a = a + 1;
   }
   public static void main( String[] args ) {
       int a = 5;
       increment( a );
> }
```



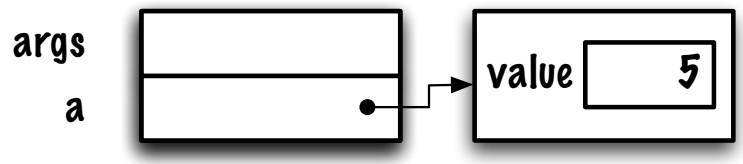
Control returns to the main method, the activation frame for **increment** is destroyed

References and method calls

```
class MyInteger {
    int value;
    MyInteger( int v ) {
        value = v;
    }
}
class Test {
    public static void increment( MyInteger a ) {
        a.value++;
    }
    public static void main( String[] args ) {
        MyInteger a = new MyInteger (5);
        System.out.println("before: " + a.value);
        increment(a);
        System.out.println("after: " + a.value);
    }
}
```

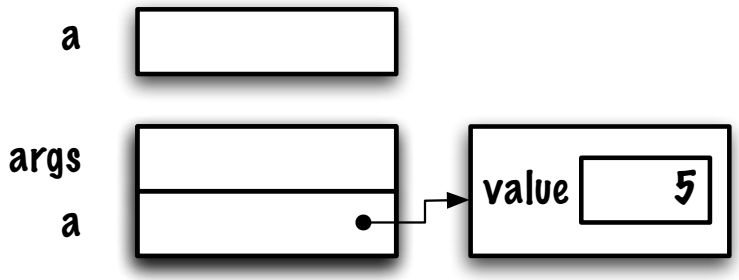
What will be printed out?

```
 static void increment( MyInteger a ) {
     a.value++;
 }
 public static void main( String[] args ) {
>    MyInteger a = new MyInteger( 5 );
     increment( a );
 }
```



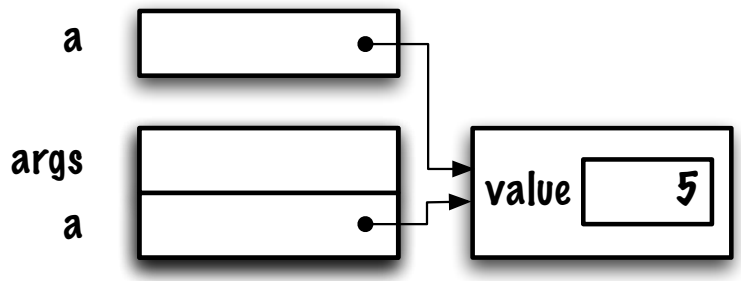
The local variable **a** of the **main** method is a reference to an instance of the class **MyInteger**

```
  static void increment( MyInteger a ) {
      a.value++;
  }
  public static void main( String[] args ) {
      MyInteger a = new MyInteger( 5 );
>     increment( a );
  }
```



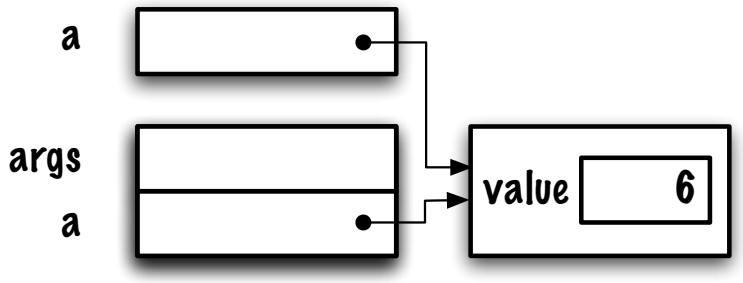
Calling **increment**, creating a new activation frame

```
  static void increment( MyInteger a ) {
      a.value++;
  }
  public static void main( String[] args ) {
      MyInteger a = new MyInteger( 5 );
>     increment( a );
  }
```



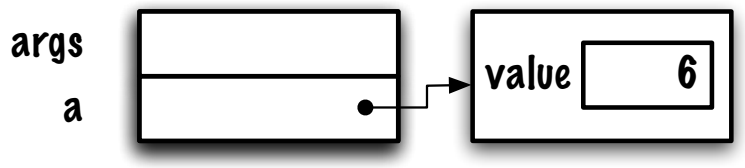
Copying the **value** of the **actual parameter** into the **formal parameter** of **increment**

```
  static void increment( MyInteger a ) {
>     a.value++;
  }
  public static void main( String[] args ) {
      MyInteger a = new MyInteger( 5 );
      increment( a );
  }
```



Executing **a.value++**

```
 static void increment( MyInteger a ) {
     a.value++;
 }
 public static void main( String[] args ) {
     MyInteger a = new MyInteger( 5 );
     increment( a );
>}
```



Returning the control to the **main** method

Definition: scope

Definition: scope

The **scope** *of a declaration is the region of the program within which the entity declared by the declaration can be referred to using a simple name*

The Java Language Specification,
Third Edition, Addison Wesley, p. 117.

Definition: scope of a local variable in Java

Definition: scope of a local variable in Java

The **scope of a local variable declaration** *in a block is the rest of the block in which the declaration appears, starting with its own initializer and including any further declarators to the right in the local variable declaration statement*

The Java Language Specification,
Third Edition, Addison Wesley, p. 118.

$\Rightarrow$ A.K.A. static or lexical scope

Definition: scope of a parameter Java

Definition: scope of a parameter Java

The **scope of a parameter** *of a method or constructor is the entire body of the method or constructor*

The Java Language Specification,
Third Edition, Addison Wesley, p. 118.

$\Rightarrow$ A.K.A. static or lexical scope

```
public class Test {
    public static void display() {
        System.out.println( "a = " + a );
    }
    public static void main( String[] args ) {
        int a;
        a = 9; // valid access, within the same block
        if (a < 10) {
            a = a + 1; // another valid access
        }
        display();
    }
}
```

Is this a valid program?

```
public class Test {
    public static void main( String[] args ) {
        System.out.println( sum );
        for ( int i=1; i<10; i++ ) {
            System.out.println( i );
        }
        int sum = 0;
        for ( int i=1; i<10; i++ ) {
            sum += i;
        }
    }
}
```

Is this a valid program?

```
public class Test {
    public static void main( String[] args ) {

        for ( int i=1; i<10; i++ ) {
            System.out.println( i );
        }
        int sum = 0;
        for ( int i=1; i<10; i++ ) {
            sum += i;
        }
    }
}
```

Is this a valid program?

Memory management

What happens to objects when they are not referenced? Here what happens to the object that contains the value 99?

```
MyInteger a = new MyInteger( 7 );
MyInteger b = new MyInteger( 99 );
b = a;
```

Memory management

What happens to objects when they are not referenced? Here what happens to the object that contains the value 99?

```
MyInteger a = new MyInteger( 7 );
MyInteger b = new MyInteger( 99 );
b = a;
```

- The JVM recuperates the memory space
- This process is called **garbage collection**
- Not all programming languages manage memory automatically

Memory management

What happens to objects when they are not referenced? Here what happens to the object that contains the value 99?

```
MyInteger a = new MyInteger( 7 );
MyInteger b = new MyInteger( 99 );
b = a;
```

- The JVM recuperates the memory space
- This process is called **garbage collection**
- Not all programming languages manage memory automatically

Java is not immune to memory leaks as will see in a few weeks...

Summary

- Strong typing help detecting certain kinds of errors early

Summary

- Strong typing help detecting certain kinds of errors early
- Comparison operators always compare the value of the expressions

Summary

- Strong typing help detecting certain kinds of errors early
- Comparison operators always compare the value of the expressions
- When comparing references we check that **two references designate the same object or not**

Summary

- Strong typing help detecting certain kinds of errors early
- Comparison operators always compare the value of the expressions
- When comparing references we check that **two references designate the same object or not**
- The method **equals** should be used to **compare the content of the objects**

Summary

- Strong typing help detecting certain kinds of errors early
- Comparison operators always compare the value of the expressions
- When comparing references we check that **two references designate the same object or not**
- The method **equals** should be used to **compare the content of the objects**
- In Java, call-by-value is the mechanism that is used for method calls

Summary

- Strong typing help detecting certain kinds of errors early
- Comparison operators always compare the value of the expressions
- When comparing references we check that **two references designate the same object or not**
- The method **equals** should be used to **compare the content of the objects**
- In Java, call-by-value is the mechanism that is used for method calls
- The scope of variable and parameter names is static in Java

Next lecture

- Introduction to object-oriented programming
  - Role of abstractions
  - Activities of software development
  - UML – Unified Modeling Language
  - Example: Counter

References I


E. B. Koffman and Wolfgang P. A. T.
Data Structures: Abstraction and Design Using Java.
John Wiley & Sons, 2e edition, 2010.


P. Sestoft.
Java Precisely.
The MIT Press, second edition edition, August 2005.


James Gosling, Bill Joy, Guy Steele, and Gilad Bracha.
Java Language Specification.
Addison Wesley, 3rd edition, 2005.

Please don't print these lecture notes unless you really need to!