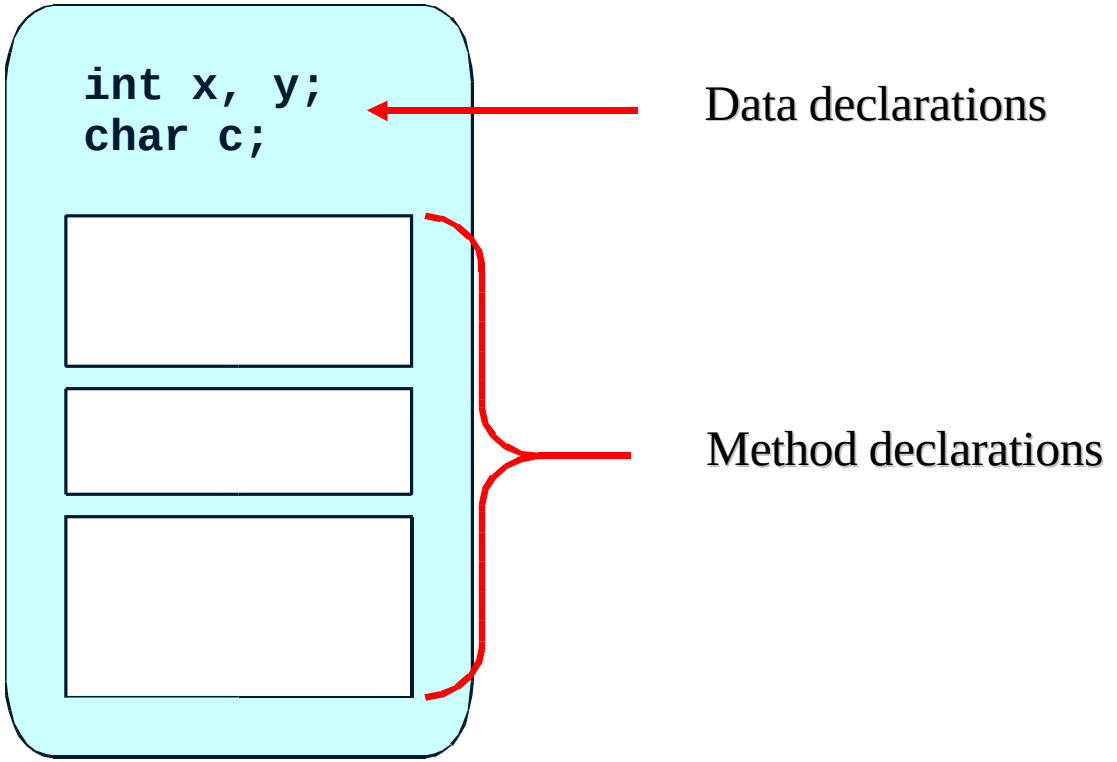
Object-Oriented Programming

- Classes
- Object Creation and Destruction
- Equality

Classes in Java

- A class encapsulates a set of properties
  - Some properties are hidden
  - The remaining properties are the interface of the class

```
class ClassName {
   dataDeclaration
   constructors
   methods
}
```



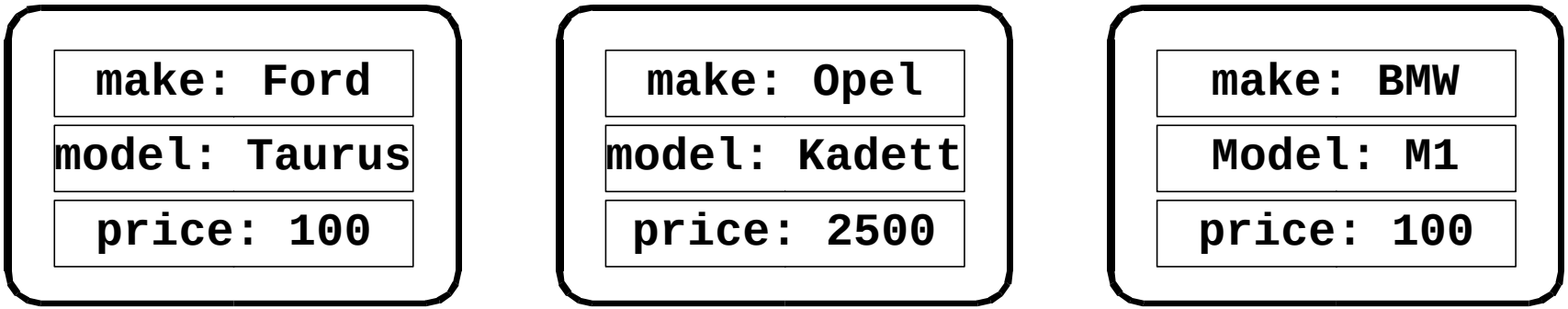
Example of a Class

```
public class Coin { // [Source Lewis and Loftus]
   public static final int HEADS = 0;
   public static final int TAILS = 1;
   private int face;
   public Coin ()   {         // constructor
      flip();
   }
   public void flip (){       // method “procedure”
      face = (int) (Math.random() * 2);
   }
   public int getFace (){     // method “function”
      return face;
   }
   public String toString(){ // method “function”
      String faceName;
      if (face == HEADS)
         faceName = "Heads";
      else
         faceName = "Tails";
      return faceName;
   }
}
```

Instance Variables

- An *instance variable* is a data declaration in a class. Every object instantiated from the class has its own version of the instance variables.

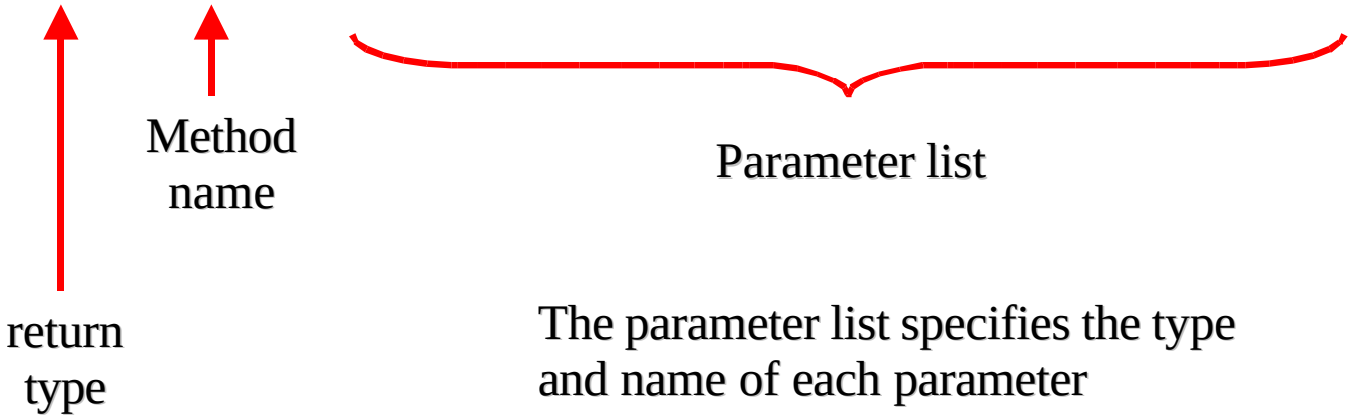
```
class Car {
   private String make;
   private String model;
   private double price;
}
```



Methods in Java

- A *method* is a function or procedure that reads and/or modifies the state of the class.

```
char calc (int num1, int num2, String message)
```



The parameter list specifies the type and name of each parameter

The name of a parameter in the method declaration is called a *formal argument*

Methods in Java, cont.

- All methods have a return type
  - **`void`** for procedures
  - A primitive data type or a class for functions
- The return value
  - Return stop the execution of a method and jumps out
  - Return can be specified with or without an expression
- Parameter are pass-by-value
  - Class parameter are pass as a reference

```
public double getPrice() {
   return this.price;
}

public void increaseCounter() {
   counter = counter + 1;
}
```

```
public double getError() {
   int a := 0;
   a++;
   // compile-error
}
```

Scope

```
public int myFunction (){                    // start scope 1
   int x = 34;
   // x is now available
   {                                         // start scope 2
      int y = 98;
      // both x and y are available
      // cannot redefine x here compile-time error
   }                                         // end scope 2
   // now only x is available
   // y is out-of-scope
   return x;
}                                            // end scope 1
```

- The redefinition of **x** in scope 2 is allowed in C/C++

Object Creation in General

- Object can be created by
  - Instantiating a class
  - Copying an existing object

- Instantiating
  - *Static*: Objects are constructed and destructed at the same time as the surrounding object.
  - *Dynamic*: Objects are created by executing a specific command.

- Copying
  - Often called *cloning*

Object Destruction in General

- Object can be destructed in two way.
  - *Explicit*, e.g., by calling a special method or operator (C++).
  - *Implicit*, when the object is no longer needed by the program.

- Explicit
  - An object in use can be destructed.
  - Not handling destruction can cause memory leaks.
- Implicit
  - Objects are destructed automatically by a *garbage collector*.
  - There is a performance overhead in starting the garbage collector.
  - There is a scheduling problem in when to start the garbage collector.

Object Creation in Java

- *Instantiazion*: A process where storage is allocated for an "empty" object.
- *Initialization*: A process where instances variables are assigned a start value.

- Dynamic instantiazion in Java by calling the **`new`** operator.
- Static instantiazion is not supported in Java.
- Cloning implemented in Java via the method **`clone()`** in class **`Object`**.

- Initialization is done in *constructors* in Java.

Object Destruction in Java

- Object destruction in Java is implicit an done via a *garbage collector*.

- A special method **`finalize`** is called immediately before garbage collection.
  - Method in class **`Object`**, that can be redefined.
  - Takes no parameters and returns **`void`**.
  - Used for releasing resources, e.g., close file handles.
  - Rarely necessary.

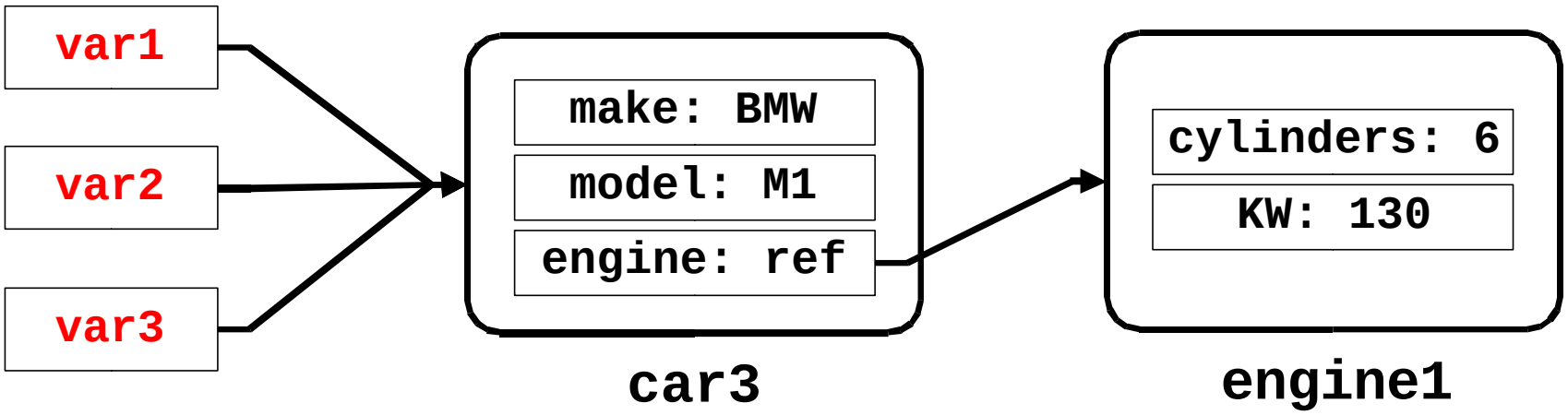
Objects and References

- Variables of non-primitive types that are not initialized have the special value **`null`**.
  - Test: **`var1 == null`**
  - Assignment: **`var2 = null`**

Object have identity but no name,

- i.e., not possible to identify an object O1 by the name of the variable referring to O1.

- *Aliasing*: Many variables referring to the same object



Constructors in Java

- A *constructor* is a special method where the instance variables or a newly created object are initialized with "reasonable" start values.

- A class must have a constructor
  - A default is provided implicitly.
- A constructor must have the same name as the class.
- A constructor has no return value.
  - That's why it is as special method
- A constructor can be overloaded.
- A constructor can call other methods (but not vice-versa).
- A constructor can call other constructors (via **`this`**).

Constructors in Java, cont.

- Every class should have a programmer defined constructor, that explicitly guarantees correct initialization of new objects.

```
// redefined Coin class
public class Coin {
   public static final int HEADS = 0;
   public static final int TAILS = 1;
   private int face;
   public Coin ()   {
      face = TAILS;
      // method in object
      flip();
      // method on other object
      otherObject.doMoreInitialization();
   }
}
```

Constructor Examples

```
public class Car {
   private String make;
   private String model;
   private double price;

   // default constructor
   public Car() {
      this ("", "", 0.0);
   }

   // give reasonable values to instance variables
   public Car(String make, String model, double price){
      this.make  = make;
      this.model = model;
      this.price = price;
   }

}
```

Constructor Initialization

```
public class Garage {
   Car car1 = new Car();          //
   static Car car2 = new Car();  // created on first access
}
```

```
public class Garage1 {
   Car car1;
   static Car car2;
   // Explicit static initialization
   static {
      car2 = new Car();
   }
}
```

Value vs. Object

- A *value* is a data element without identity that cannot change state.
- An *object* is an encapsulated data element with identity, state, and behavior.

- An object can behave like value (or record). Is it a good idea?

- Values in Java are of the primitive type **`byte`**, **`short`**, **`int`**, **`long`**, **`float`**, **`double`**, **`boolean`**, and **`char`**.
- Wrapper classes exists in Java for make the primitive type act as objects.

Strings in Java

- Strings in Java are of the class **`String`**.
- Objects of class **`String`** behave like values.

- Characteristics of Strings
  - The notation "fly" instantiates the class String and initialize it with the values "f"", "l", and "y".
  - The class **`String`** has many different constructors.
  - Values in a string cannot be modified (use **`StringBuffer`** instead).
  - Class **`String`** redefines the method **`equals()`** from class **`Object`**.

Arrays in Java

- Not pointers like in C,
- Bounds checking at run-time
- `int[] numbers; // equivalent`
  `int   number[];`
- `int[] numbers = {1, 2, 3, 4, 5, 6, 7};`
  - The size is fixed at compile-time!
- `int[] numbers = new Integer[getSize()];`
  - The size is fixed at run-time!
  - Cannot be resized

```
for (int i = 0; i < numbers.length; i++){
   System.out.println(numbers[i]);
}
```

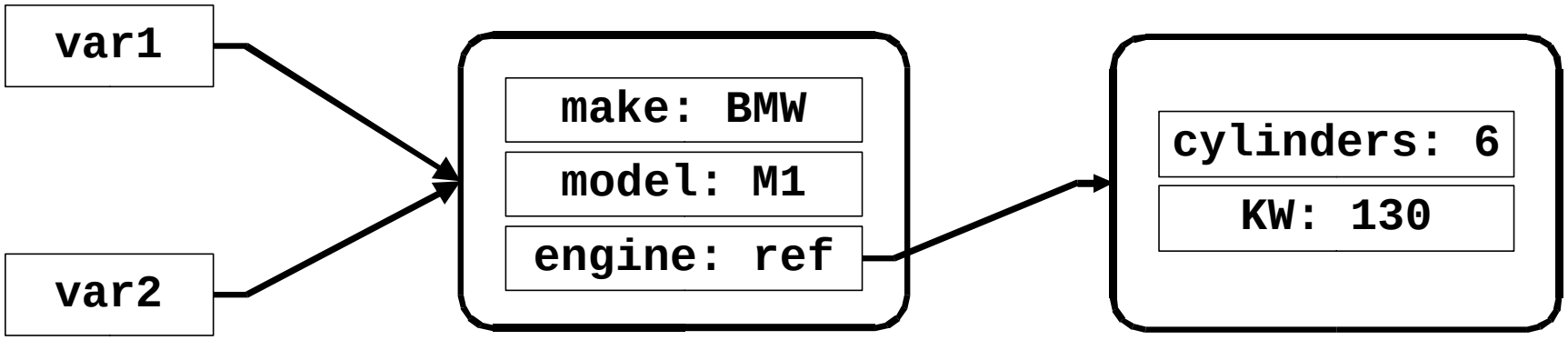
Equality

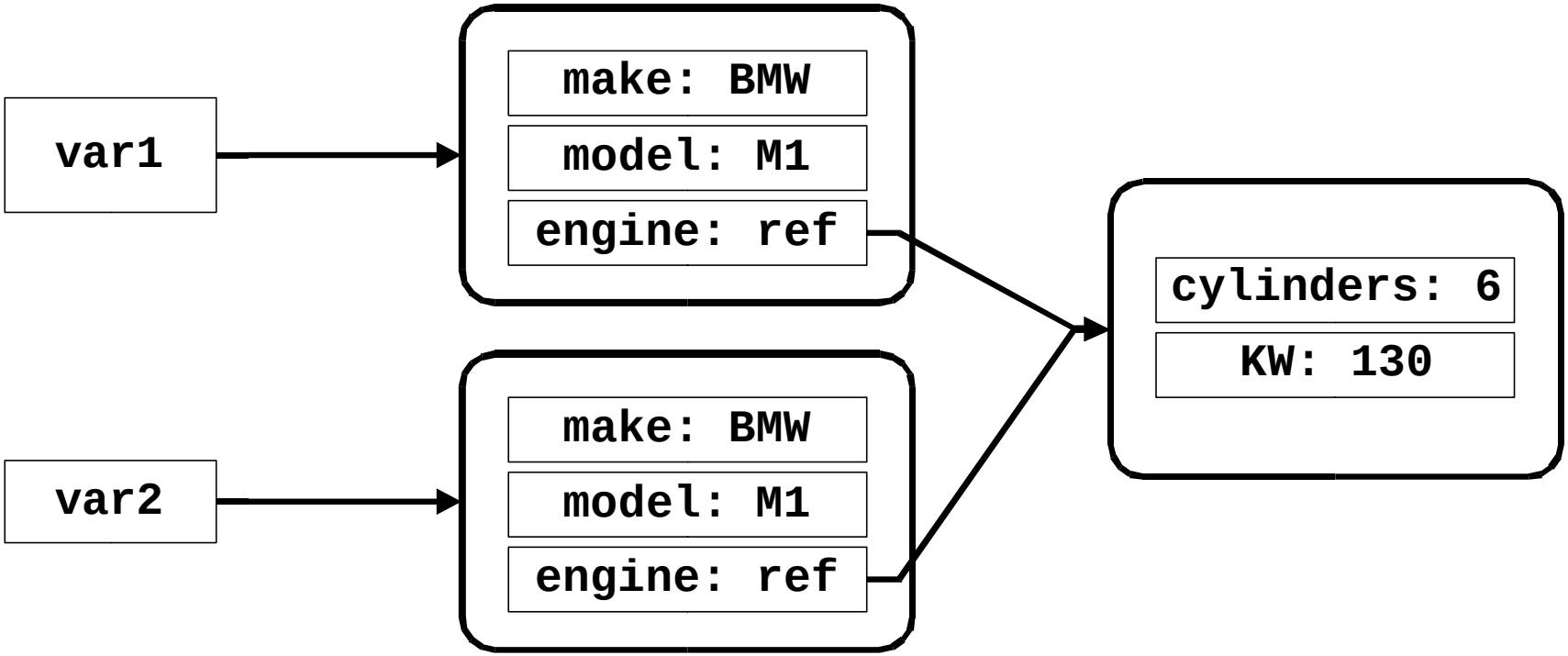
- Are the references **a** and **b** equal?

- *Reference Equality*
  - Returns whether **a** and **b** points to the same object.
- *Shallow Equality*
  - Returns whether **a** and **b** are structurally similar.
  - One level of object are compared.
- *Deep Equality*
  - Returns where **a** and **b** have object-network that is structurally similar.
  - Multiple level of objects are compared recursively.

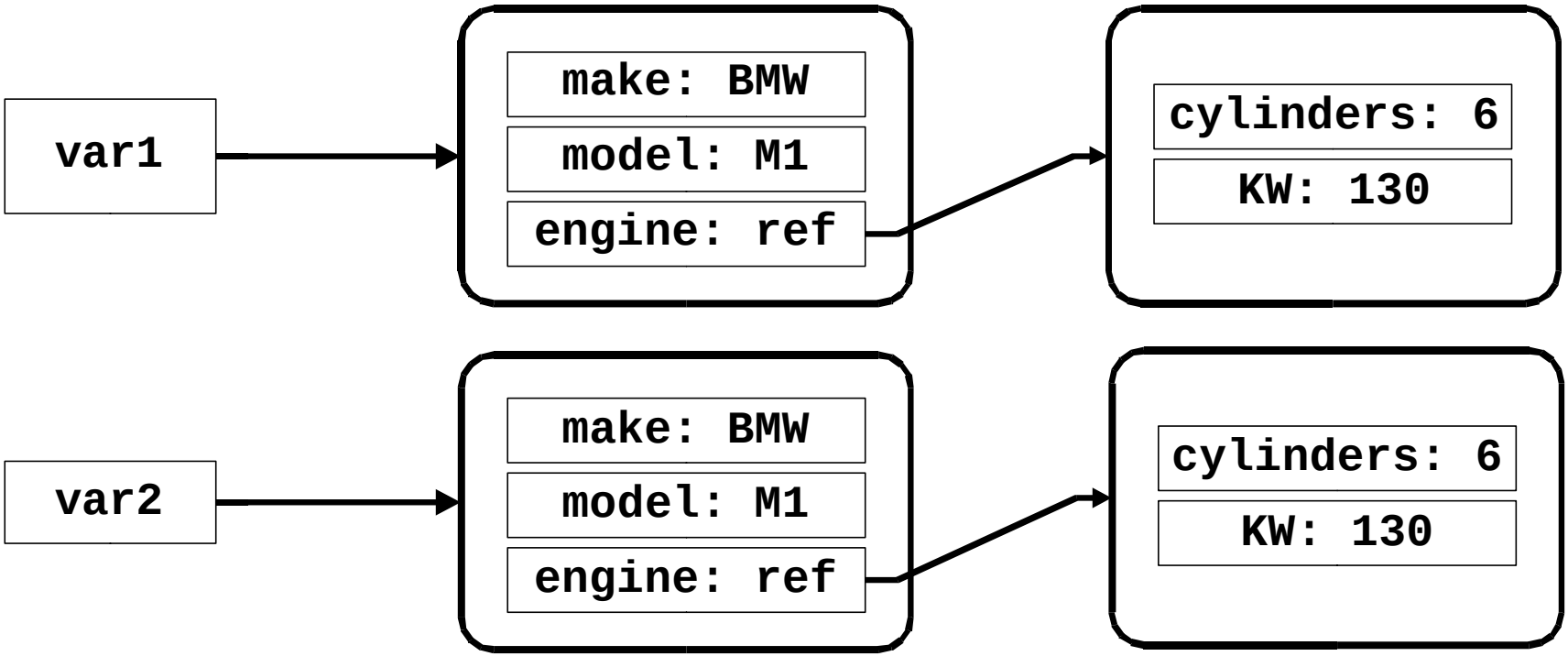
- *Reference Equality Þ Shallow Equality Þ Deep Equality*

Equality Examples





Equality Examples, cont.



Types of Equality in Java

- `==`
  - Equality on primitive data types
    - **`8 == 7`**
    - **`'b' == 'c'`**
  - Reference equality on object references
    - **`onePoint == anotherPoint`**
- **`equals`**
  - Method on the class **`Object`**.
  - Default works like reference equality.
  - Can be refined in subclass
    - **`onePoint.equals(anotherPoint)`**

Summary

- Instance variables
- Strings are treated specially in Java
- Initialization is critical for objects
  - Java guarantees proper initialization using constructors
  - Source of many errors in C
- Java helps clean-up with garbage collection
  - Only memory is clean, close those file handles explicitly!
  - No memory leaks, "show stopper" in C/C++ project!
- Equality (three types of equality)