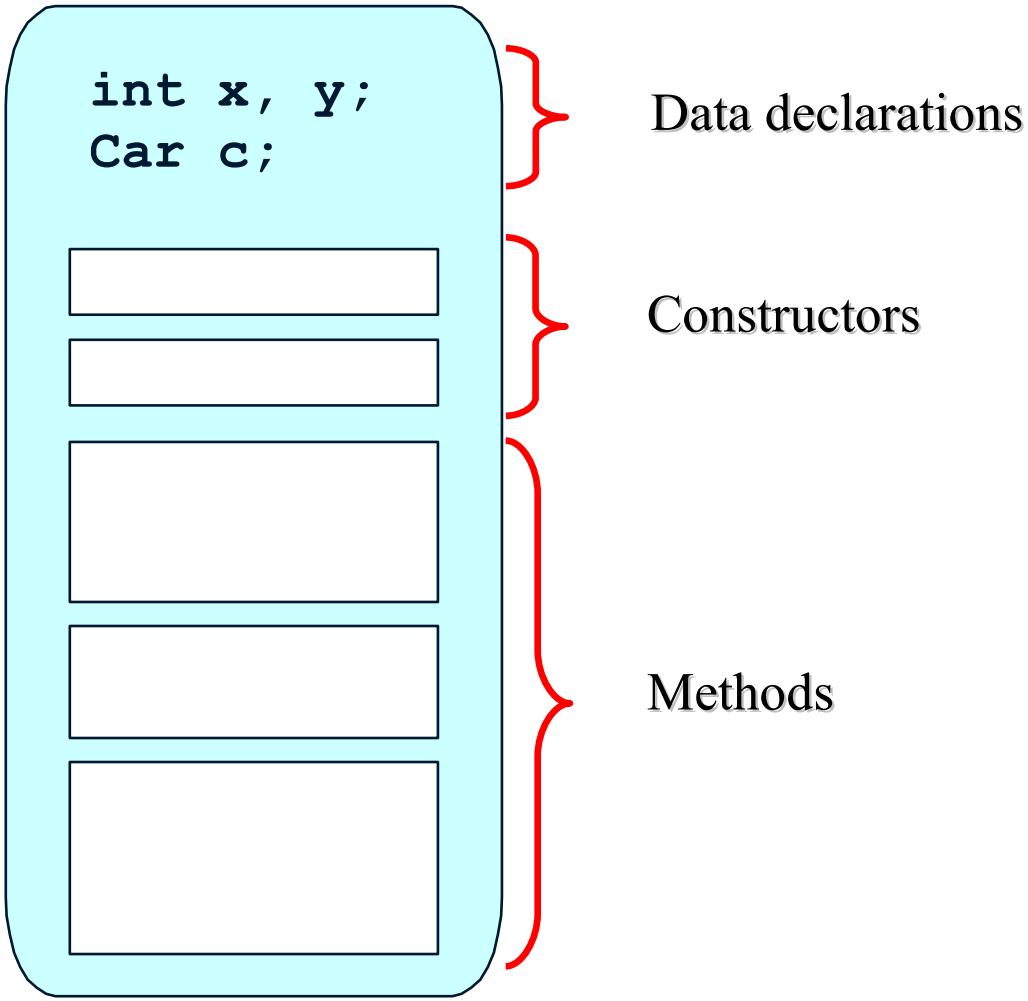
Object-Oriented Programming, Part 1

- Classes
- Methods
  - Arguments and return value
  - Overloading
- Variables
  - Instance variables vs. class variables
  - Scope rules
- Object creation and destruction
  - Constructors
  - Destructors
  - Value vs. object (**`String`** objects are special)
- Equality
  - Three different types of equality

Classes in Java

- A class encapsulates a set of properties (methods and attributes)
  - Some properties are hidden
  - The remaining properties are the interface of the class

```
public class ClassName {
    dataDeclarations
    Constructors
    methods
}
```

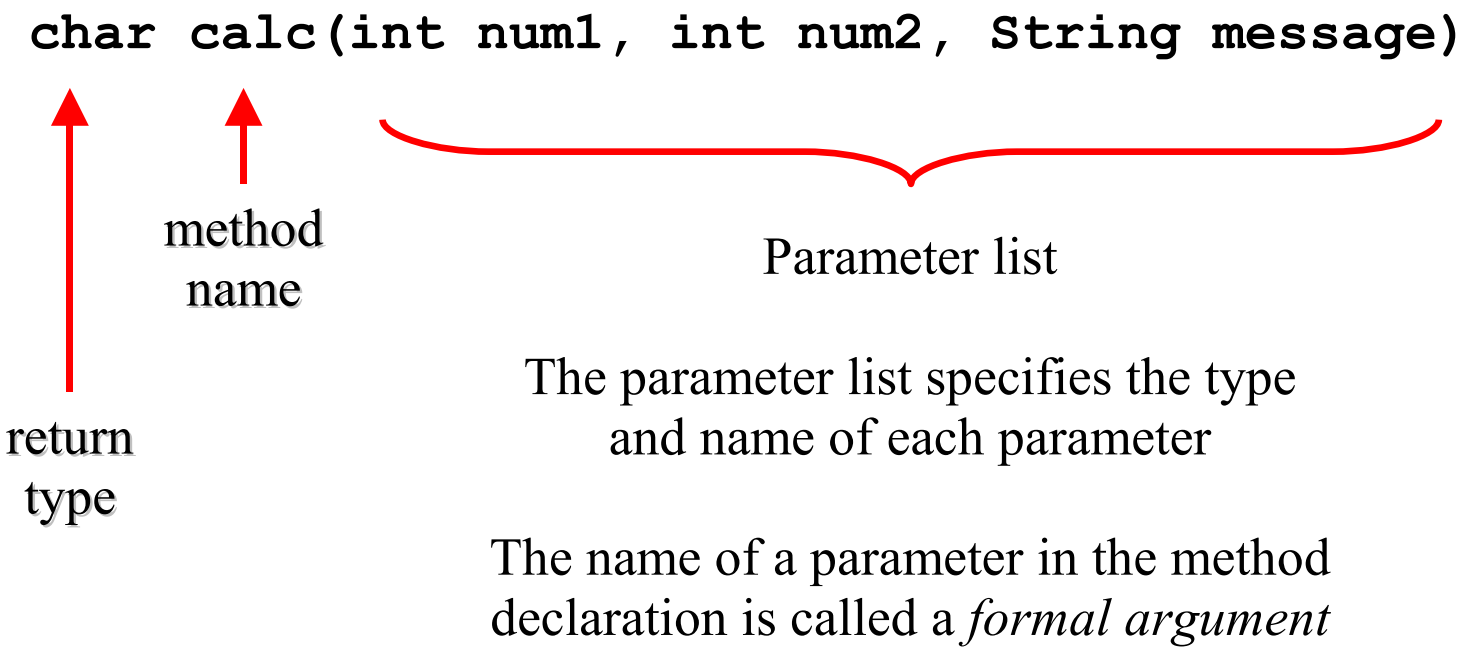


Example of a Class

```
public class Car {
   // Data declaration/state
   private String make;
   private String model;
   private double price;
   // constructor
   public Car(String ma, String mo, double pr) {
      make = ma;
      model = mo;
      price = pr;
   }
   // methods
   public String getMake() {          // "function"
      return make;
   }
   public void setMake(String ma) { // "procedure"
      make = ma;
   }
   public String toString() {         // "function"
      return "make " + getMake() + " model " + model;
   }
}
```

Methods in Java

- A *method* is a function or procedure that reads and/or modifies the state of the class.
  - A function returns a value (a procedure does not).
  - A procedure has side-effects, e.g., change the state of an object.



Method in Java, Example

```
public class Car{
    // snip
    /** Calculates the sales price of the car */
     public int salesPrice(){
         return (int)price;
     }
     /** Calculates the sales price of the car */
     public int salesPrice(int overhead){
         return (int)price + overhead;
     }
     /** Calculates the sales price of the car */
     public double salesPrice(double overheadPercent){
         return price + (overheadPercent * price);
     }

     /** Override the toString method */
     public String toString(){
         return "make " + getMake() + " model "
                 + getModel() + " price " + getPrice();
     }
}
```

Method in Java, Example, cont.

```
public class Car{
    // snip
    // what is wrong here?
     public int salesPrice(){
         return (int)price;
     }
     public double salesPrice(){
         return (double)price;
     }

     // what is ugly here?
     public double salesPrice1(int i, double d){
         // does okay stuff
     }
     public double salesPrice1(double d, int i){
         // does okay stuff
     }
     public static void main(String[] args){
       Car vw = new Car("VW", "Golf", 1000);
       vw.salesPrice();
    }
}
```

Method in Java, Examples, cont.

- What is ugly here?

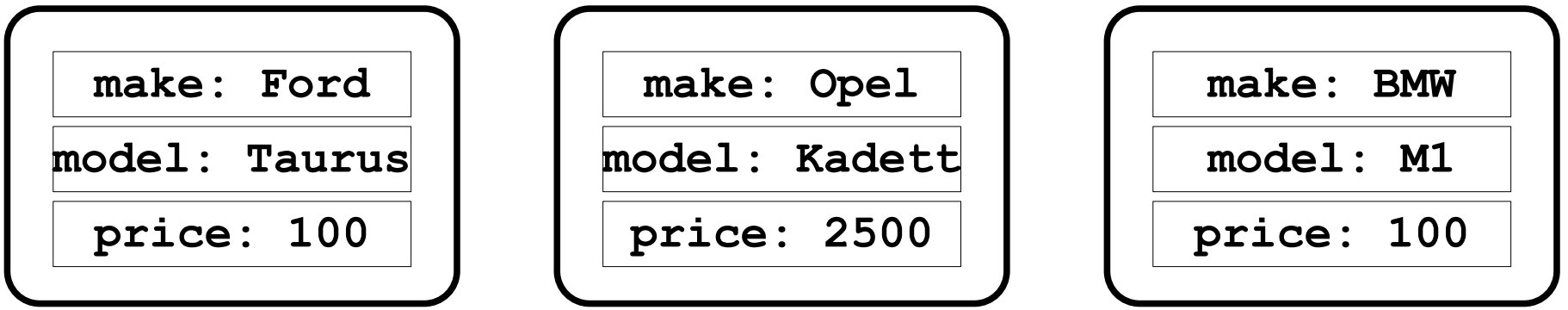
```
// Finds the maximum value
public int max(int x, int y) {
  int result = x; // make a guess on maximum value
  if (x < y) { result = y;}
  return result;
  x++;
}
```

```
// Checks if x > 10
public boolean greaterThan10(int x) {
  boolean result;
  if (x > 10) { result = true;}
  else        { result = false;}
  return result;
}
```

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.

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



Scope

```
public class Car {
    // snip
    private String make; // can be seen in entire class

    public String someMethod() {
        String tmp = "Hello"; // local to this method
        System.out.println(make); // allowed
        return tmp + make;

    }

    public int someOtherMethod() {
        System.out.println(tmp); //not allowed
        System.out.println(make); //allowed

        for(int j = 0; j < 10 j++)
            System.out.println(j); //allowed

        return j; //not allowed
    }
}
```

Scope, cont.

```
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 variable **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 (Java).

- 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

- *Instantiation*
  - A process where storage is allocated for an “empty” object.
- *Initialization*
  - A process where instances variables are assigned start values.

- Dynamic instantiation in Java by calling the **`new`** operator.
- Static instantiation is *not* supported in Java.
- Cloning implemented in Java via the method **`clone()`** in class **`java.lang.Object`**.
- Objects are always allocated on the heap.

- Initialization is done in *constructors* in Java
  - Very similar to the way it is done in C++

Object Destruction in Java

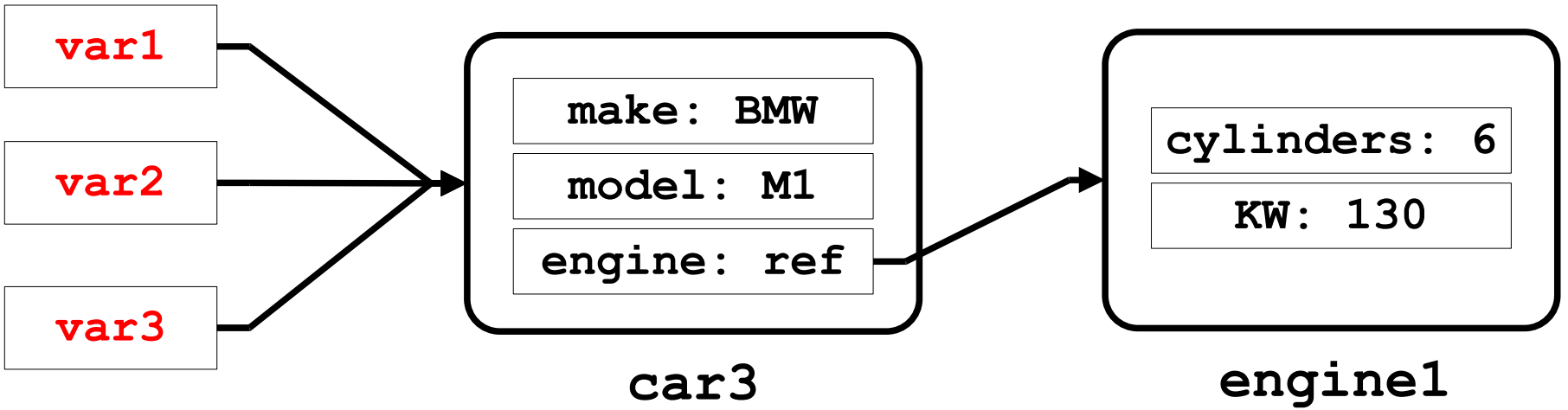
- Object destruction in Java is implicit an done via a *garbage collector*.
  - Can be called explicitly via **`System.gc()`**.

- A special method **`finalize()`** is called immediately before garbage collection.
  - Method in class **`java.lang.Object`**, that can be overridden.
  - Takes no parameters and returns **`void`**.
  - Used for releasing resources, e.g., close file handles.
  - Rarely necessary, e.g., “dead-conditions” for error detection purposes.

- Tip: Avoid to use the **`finalize`** method!

Objects and References

- Variables of non-primitive types that are not initialized have the special value **`null`**.
  - Test: **`var1 == null`**
  - Assignment: **`var2 = null`**
- Objects have identity but no name
  - 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 of a newly created object are initialized with "reasonable" start values.

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

Constructors in Java, cont.

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

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

    /** The default constructor */
    public Car() {
        this("", "", 0.0); // must be the first thing
    }

    /** Constructor that assigns values to instance vars. */
    public Car(String make, String model, double price) {
        this.make = make;
        this.model = model;
        this.price = price;
    }
}
```

Typical Errors in Constructors

```
public class Car {
    // instance variables
    private String make;
    private String model;
    private double price;
    /** what is wrong here? */
    public Car(String make, String model, double price) {
        make = make; model = model; price = price;
    }
    /** what is wrong here? */
    public Car(String make, String model, double price) {
        this.make = make;
        this.model = model;
        this.price = price;
        return this;
    }
    /** what is wrong here? */
    public Car(String make, String model) {
        this.make = make;
        this.model = model;
    }
}
```

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();
   }
}
```

Constructor vs. Method

Similarities

- Can take arguments
  - all pass-by-value
- Can be overloaded
- Access modifiers can be specified (e.g., **`private`** or **`public`**)
- Can be **`final`** (covered later)

Dissimilarities

- Has fixed name (same as the class)
- No return value
  - "returns" a reference to an object
- Special call via new operator
  - **`new Car()`**
  - Cannot be called by methods
- Default constructor can be synthesized by the system
- Cannot be declared **`static`**
  - It is in fact a static method!

Cloning in Java

```
public class Car {
    // instance variables
    private String make;
    private String model;
    private double price;
    // snip
    /** Constructor that assigns values to instance vars. */
    public Car(String make, String model, double price) {
        this.make = make;
        this.model = model;
        this.price = price;
    }

    /** Cloning in Java overrides Object.clone() */
    public Object clone() {  // note the return type
        return new Car(make, model, price);
    }
}
```

- Recommendations for what is allowed in a **`clone`** method.
  - Will be covered later in the course.

Object Destruction in Java, cont.

```
public class Car{
   // snip

   /** Overwrite the finalize method.
    * @see java.lang.Object#finalize()
    */
   public void finalize(){
      // write who is dying, pretty stupid just an example
      System.out.println("I'm dying " + this);
   }
}
```

- No guarantee that **`finalize`** method is ever called!
  - If you do not run out of main memory!
- Note that garbage collection only cleans-up memory
  - not open files, open network connections, open database connections

Object Destruction in Java, cont.

```
class MemoryUsage{              /* Dummy class to take up mem. */
   int id;                      /* Id of object */
   String name;                 /* Name of object */
   MemoryUsage(int id){         /* Constructor */
      this.id = id;
      this.name = "Name: " + id;
   }
   /** Overwrite the finalize method */
   public void finalize(){
      System.out.println("Goodbye cruel world " + this.id);
   }
}
public class Cleanup{
   public static void main(String[] args){
      for (int i = 0; i < 999; i++){
         // allocate object and discard it again
         MemoryUsage m = new MemoryUsage(i);
         if (i % 100 == 0){ System.gc(); }
      }
   }
}
```

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?

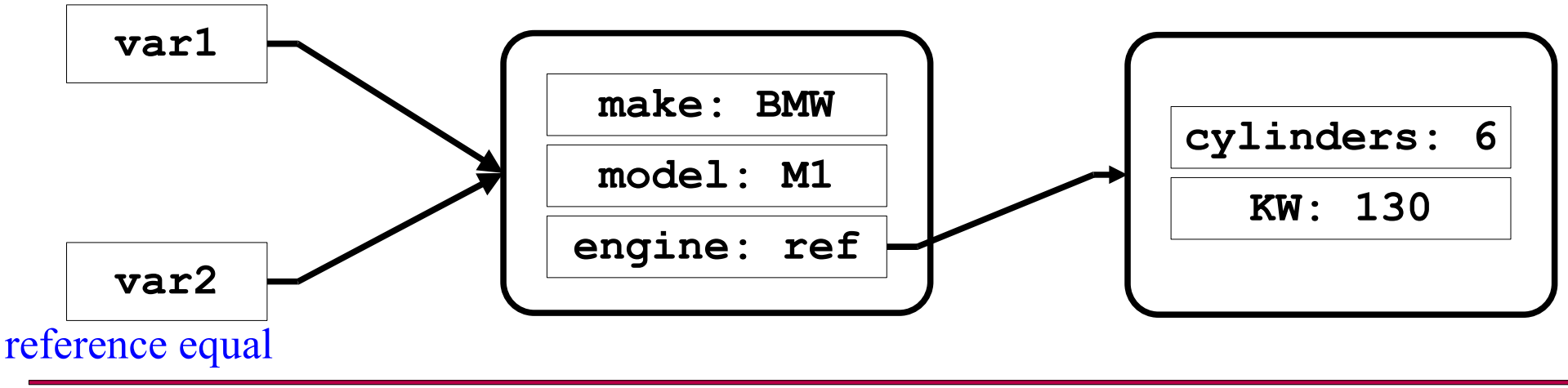
- Values in Java are of the primitive type **`byte`**, **`short`**, **`int`**, **`long`**, **`float`**, **`double`**, **`boolean`**, and **`char`**.
- Wrapper classes exists in Java for make the primitive type act as objects.
  - **`Character`** for **`char`**
  - **`Integer`** for **`int`** etc.
- Auto-boxing available in Java 1.5

Value vs. Object, Strings in Java

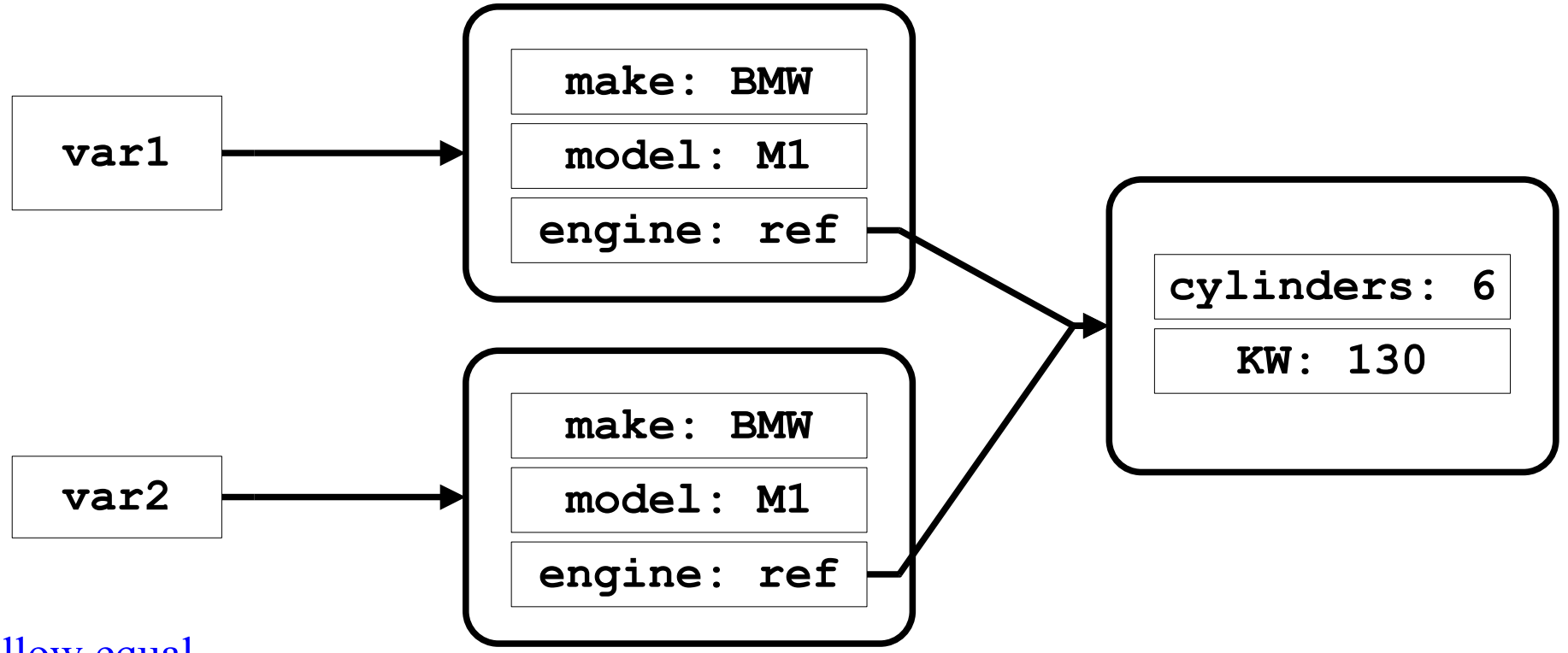
- Strings in Java are of the class **`String`**.
  - Objects of class **`String`** behave like values.
- Characteristics of Strings
  - The notation "Ski" instantiates the class **`String`** and initialize it with the values 'S', 'k', and 'i'.
  - The class **`String`** has many different constructors.
  - Values in a **`String`** cannot be modified (use **`StringBuffer`**).
  - Class **`String`** redefines the method **`equals()`** from class **`Object`**.

`String`
`length()`
`charAt(int)`
`indexOf(char)`
`substring(int)`
`toLowerCase()`
`toUpperCase()`
`trim()`
`endsWith(String)`
`startsWith(String)`
`intern()`

Equality Examples

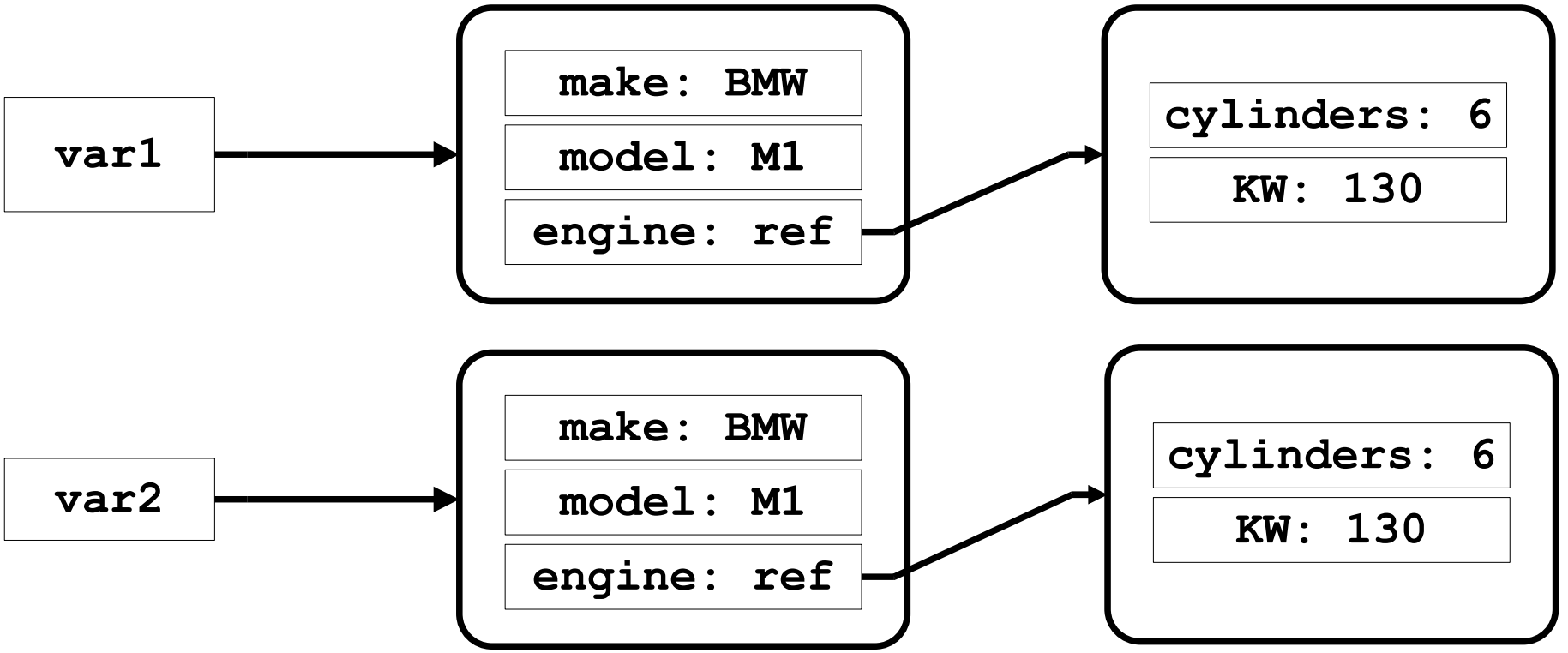


reference equal



shallow equal

Equality Examples, cont.



deep equal

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 objects are compared.
- *Deep Equality*
  - Returns where **a** and **b** have object-networks that are structurally similar.
  - Multiple level of objects are compared recursively.

- *Reference Equality* $\Rightarrow$ *Shallow Equality* $\Rightarrow$ *Deep Equality*

Types of Equality in Java

- `==`
  - Equality on primitive data types
    - `8 == 7`
    - `'b' == 'c'`
  - Reference equality on object references
    - `oneCar == anotherCar`
  - Strings are special

```
String s1 = "hello"; String s2 = "hello";
 if (s1 == s2){
     System.out.println(s1 + " equals" + s2);}
```

- **`equals`**
  - Method on class **`java.lang.Object`**.
  - Default works like reference equality.
  - Can be overridden

Requirements to `equals` Method

- *Reflexsive*
  - **`x.equals(x)`** is always true
- *Symmetry*
  - For all objects **`x`** and **`y`**, **`x.equals(y)`** is true iff **`y.equals(x)`**
- *Transitivify*
  - For all objects **`x`** and **`y`**, and **`z`** if **`x.equals(y)`** and **`y.equals(z)`** then **`x.equals(z)`** must be true
- *Consitency*
  - For all objects **`x`** and **`y`**, **`x.equals(y)`** should return true (or false) consistently if the states of are **`x`** and **`y`** unchanged
- *Non-nullable*
  - For all objects **`x`**, **`x.equals(null)`** should return false

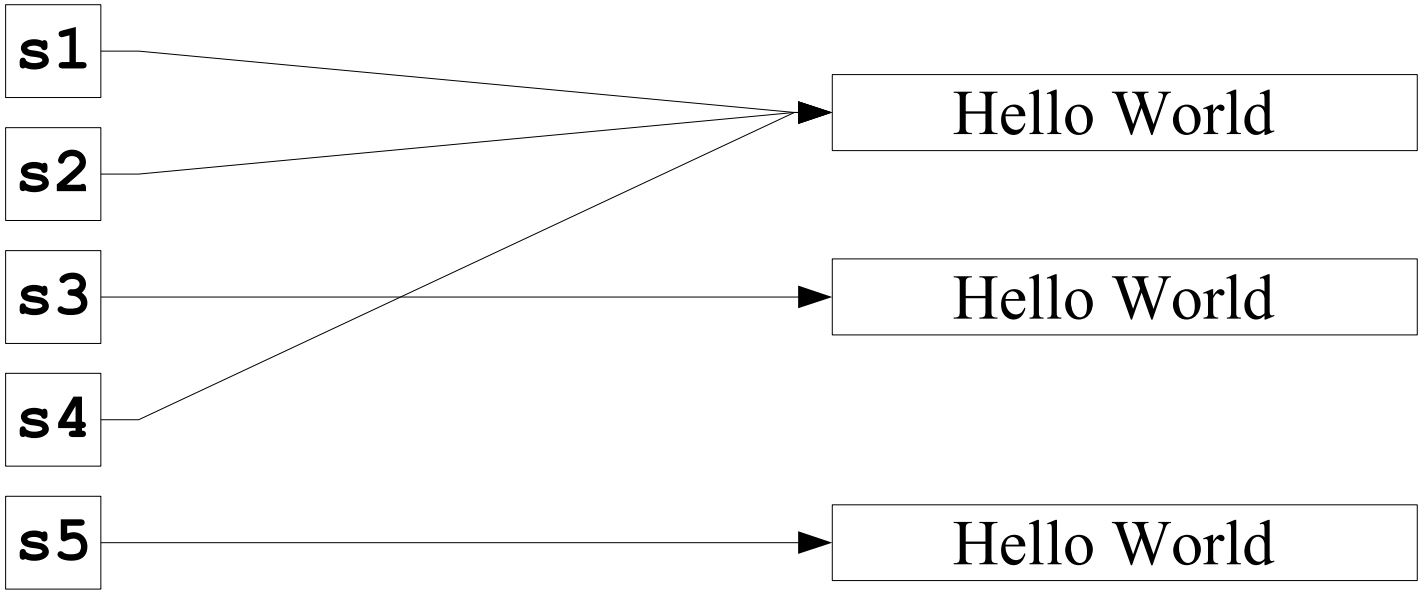
`equals` example

```
public class Car {
    // snip
    /** Gets the make inst variable(helper function). */
    public String getMake() {
        return make;
    }
    // snip

    /**
     * Implements the equals method
     * @see java.lang.Object#equals(java.lang.Object)
     */
    public boolean equals(Object o) {
        return o instanceof Car // is it a Car object?
            && ((Car) o).getMake().equals(this.make)
            && ((Car) o).getModel().equals(this.model)
            && ((Car) o).getPrice() == this.price;
            // relies on "short circuiting"
    }
```

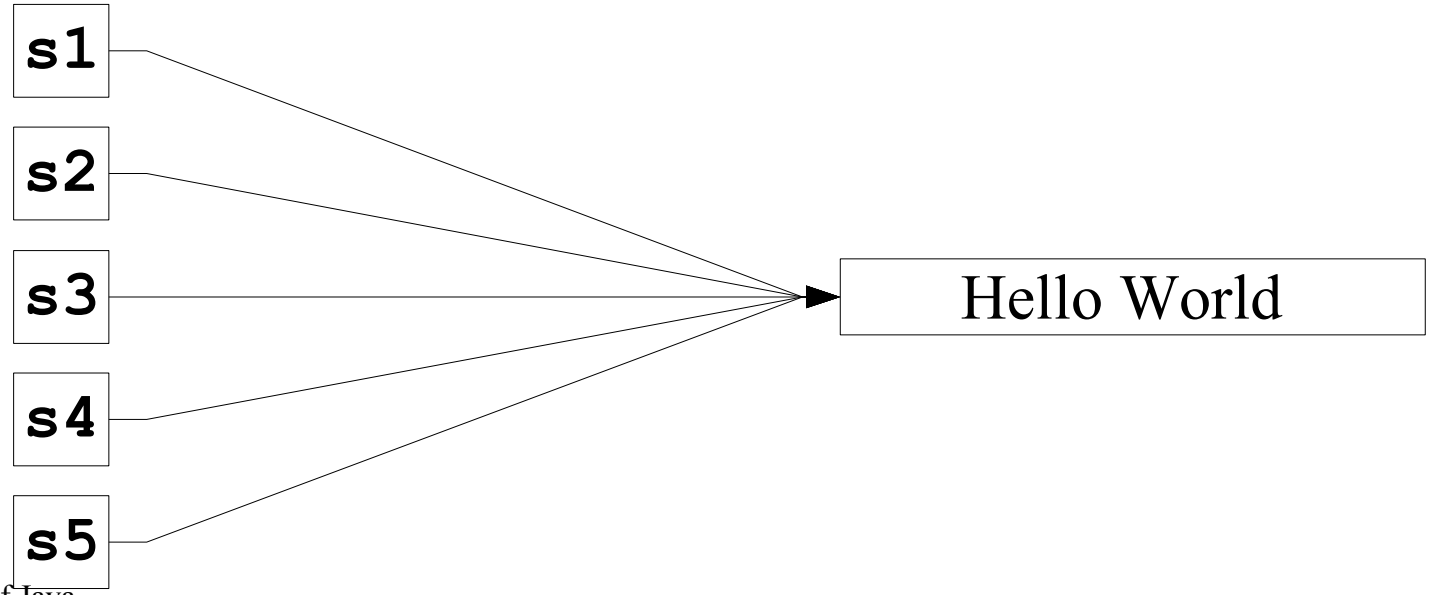
String Equality

```
public static void equal(){
   String s1 = "Hello World"; String s2 = "Hello World";
   String s3 = new String ("Hello World");
   String s4 = s1;
   String s5 = "Hello Worl"; s5 += "d";
   if (s1 == s2) {System.out.println("s1 == s2"); }
   else          {System.out.println("s1 != s2"); }
   if (s1 == s3) {System.out.println("s1 == s3"); }
   else          {System.out.println("s1 != s3"); }
   if (s1 == s4) {System.out.println("s1 == s4"); }
   else          {System.out.println("s1 != s4"); }
   if (s1 == s5) {System.out.println("s1 == s5"); }
   else          {System.out.println("s1 != s5"); }

   // s1.equals(s2)s1.equals(s3)s1.equals(s4)s1.equals(s5)
   // s1 = s1.intern()
}
```

String Equality, cont.



After executing **`s1-5.intern()`**



Summary

- No code is outside classes
  - Instance variables
- Methods
  - Overloading is generally a good thing
- Initialization is critical for objects
  - Source of many errors in C
  - Java guarantees proper initialization using constructors
- Java helps clean-up with garbage collection
  - Only memory is clean, close those file handles explicitly!
  - No memory leaks, “show stopper” in a C/C++ project!
- Equality (three types of equality)
  - **`java.lang.Object.equals()`**
- Strings are treated specially in Java
  - Always compare strings by using **`equals()`**

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;           // data declaration/state
   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;
   }
}
```

Arrays in Java

- Objects and 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]);
}
```

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`** stops 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 passed as a reference (reference is copied)

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

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

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