Java Basics

Object Orientated Programming in Java

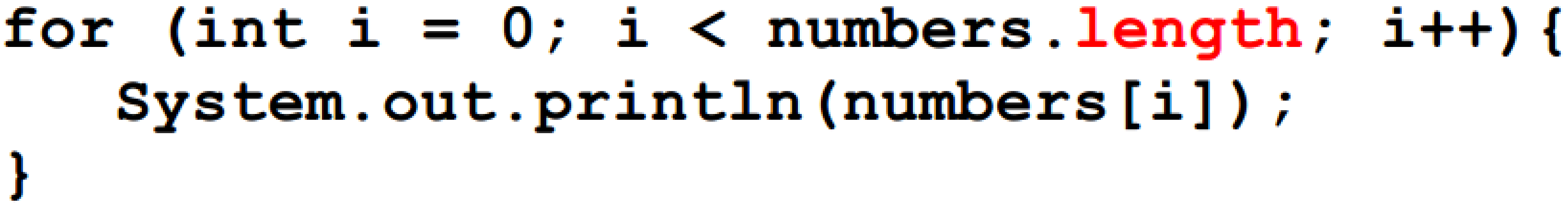
Benjamin Kenwright

Outline

- Review Previous Weeks
- Object Orientated Concepts
  - Classes, Methods, Overloading, Object Creation, Equality, ...
- Today’s Practical
- Review/Discussion

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

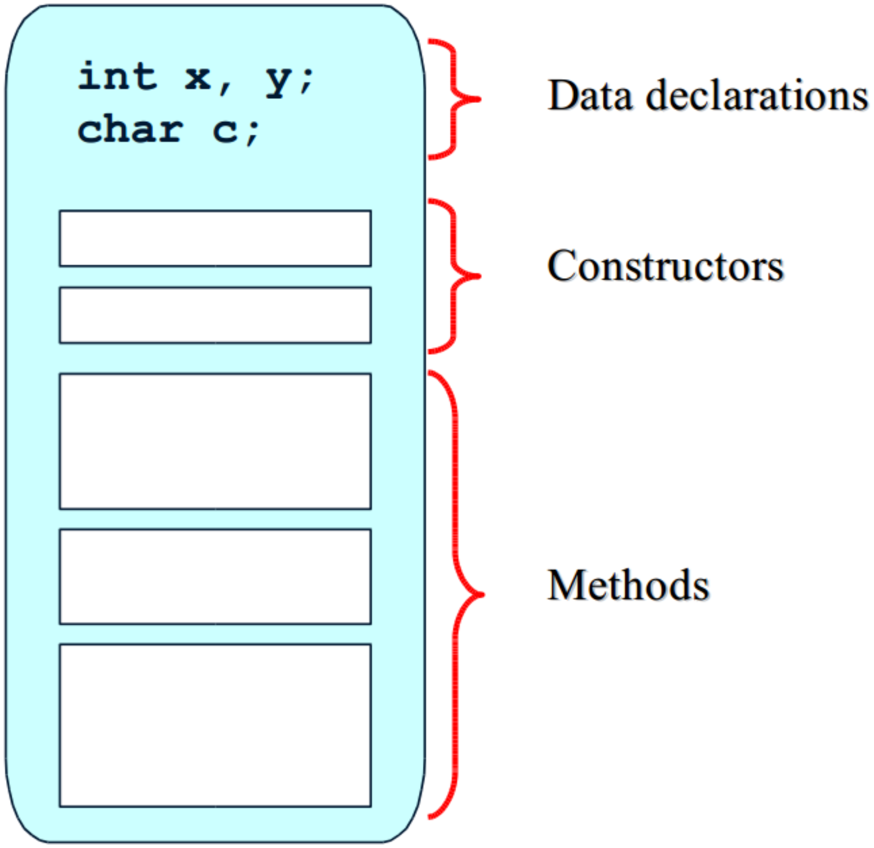


```
for (int i = 0; i < numbers.length; i++){
   System.out.println(numbers[i]);
}
```

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
}
```



```
int x, y;
char c;
```

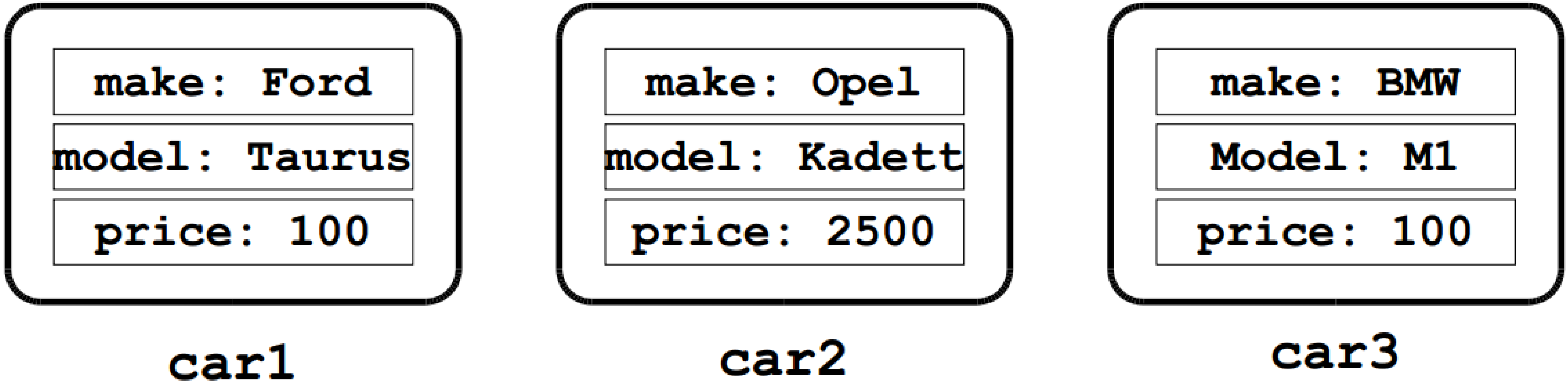
Example of a Class

```
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;
}
```



make: Ford
model: Taurus
price: 100

car1

make: Opel
model: Kadett
price: 2500

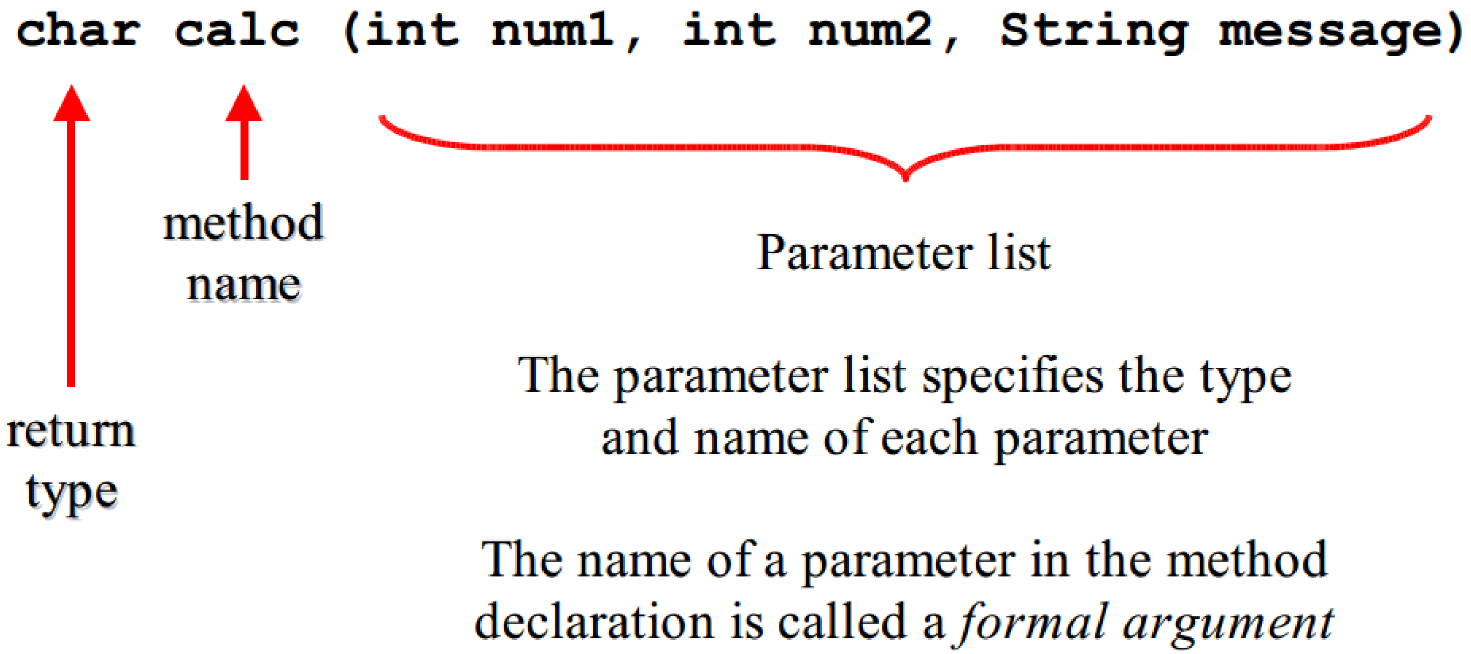
car2

make: BMW
Model: M1
price: 100

car3

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



```
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 passed as a reference

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

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

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

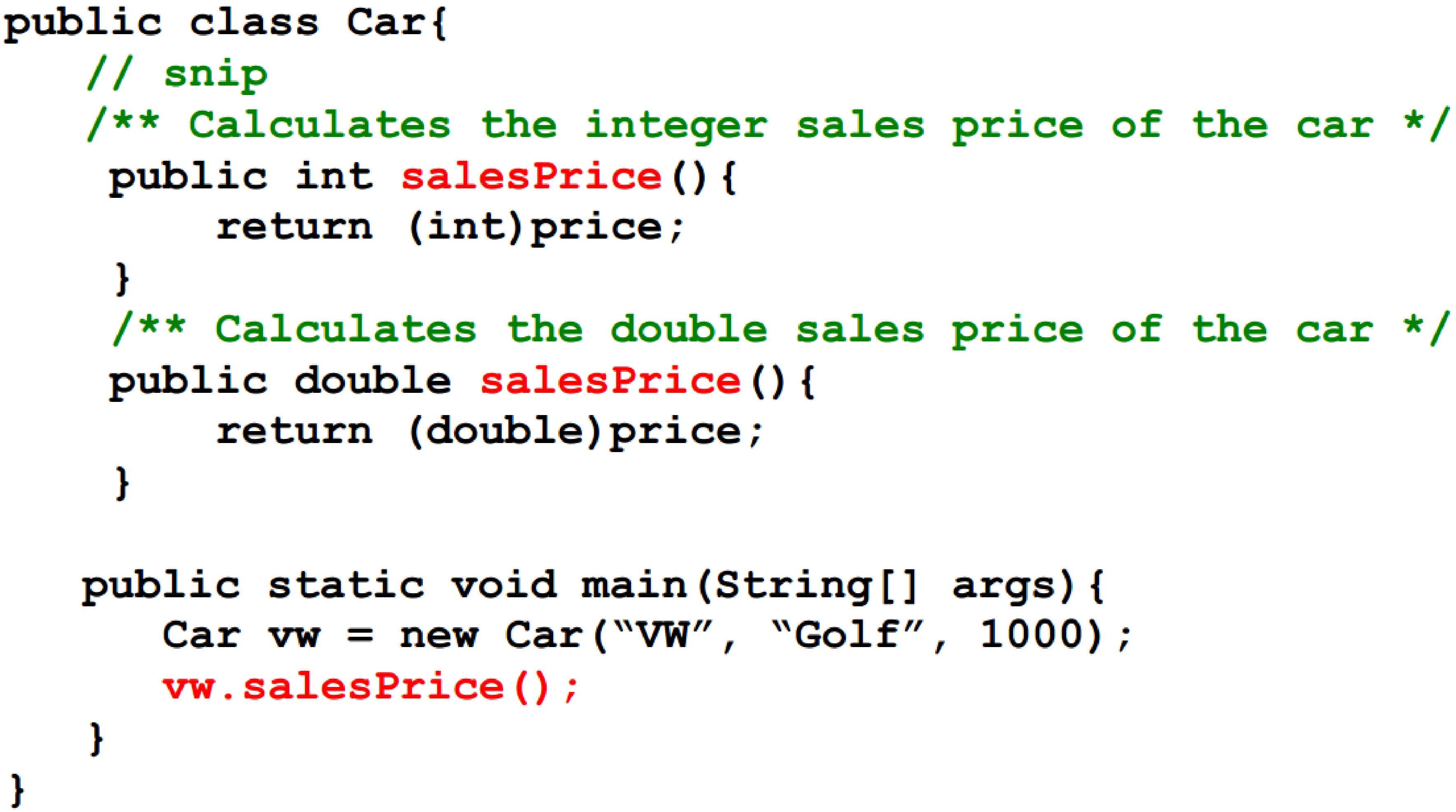
Method in Java, Example

```
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);
    }

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

Method in Java, Example, cont.

- What is wrong here?



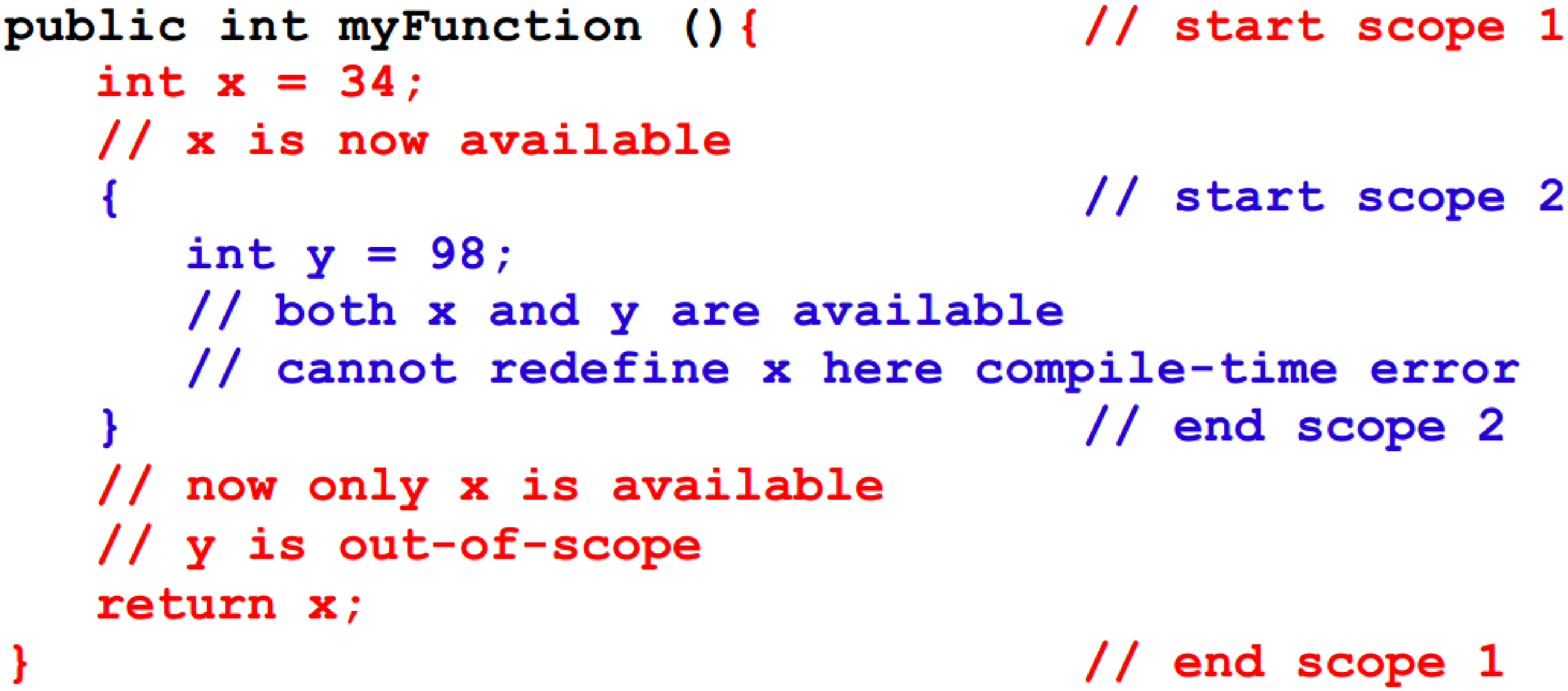
```
public class Car{
   // snip
   /** Calculates the integer sales price of the car */
    public int salesPrice(){
        return (int)price;
    }
    /** Calculates the double sales price of the car */
    public double salesPrice(){
        return (double)price;
    }

   public static void main(String[] args){
      Car vw = new Car(“VW”, “Golf”, 1000);
      vw.salesPrice();
   }
}
```

- Ambiguous function overloading (only different by return type)
    - int salesPrice()
    - double salesPrice()

Scope

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



```
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
```

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

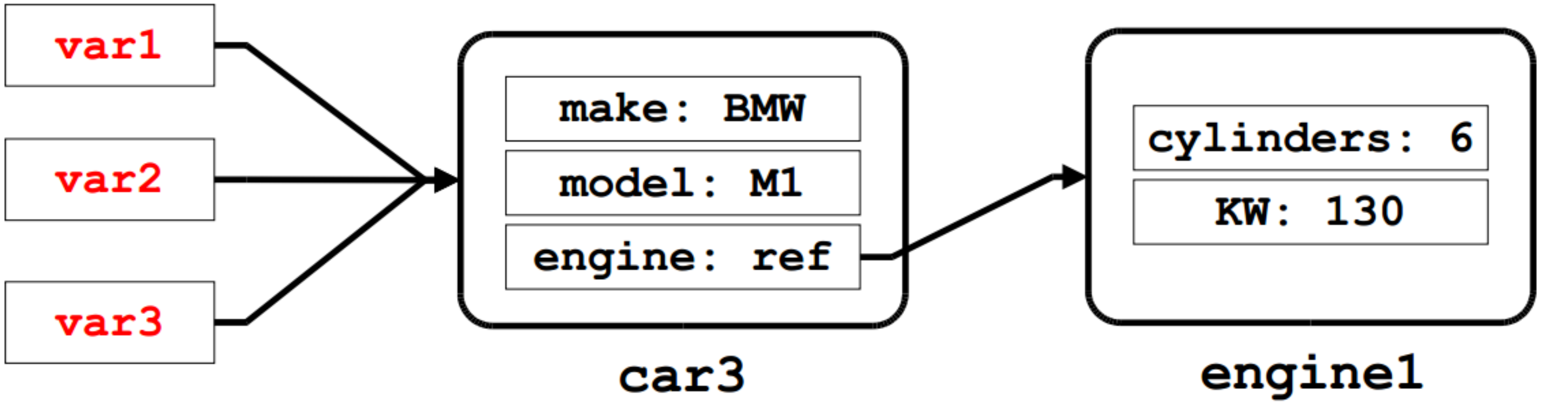
- Instantiazion: A process where storage is allocated for an “empty” object.
- Initialization: A process where instances variables are assigned a start value.
- Dynamic instantiazion in Java by calling the *new* operator
- Static instantiazion is not supported in Java.
- Cloning implemented in Java via the method *clone()* in class
  - ▷java.lang.Object
- 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 Object, that can be redefined.
  - 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

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

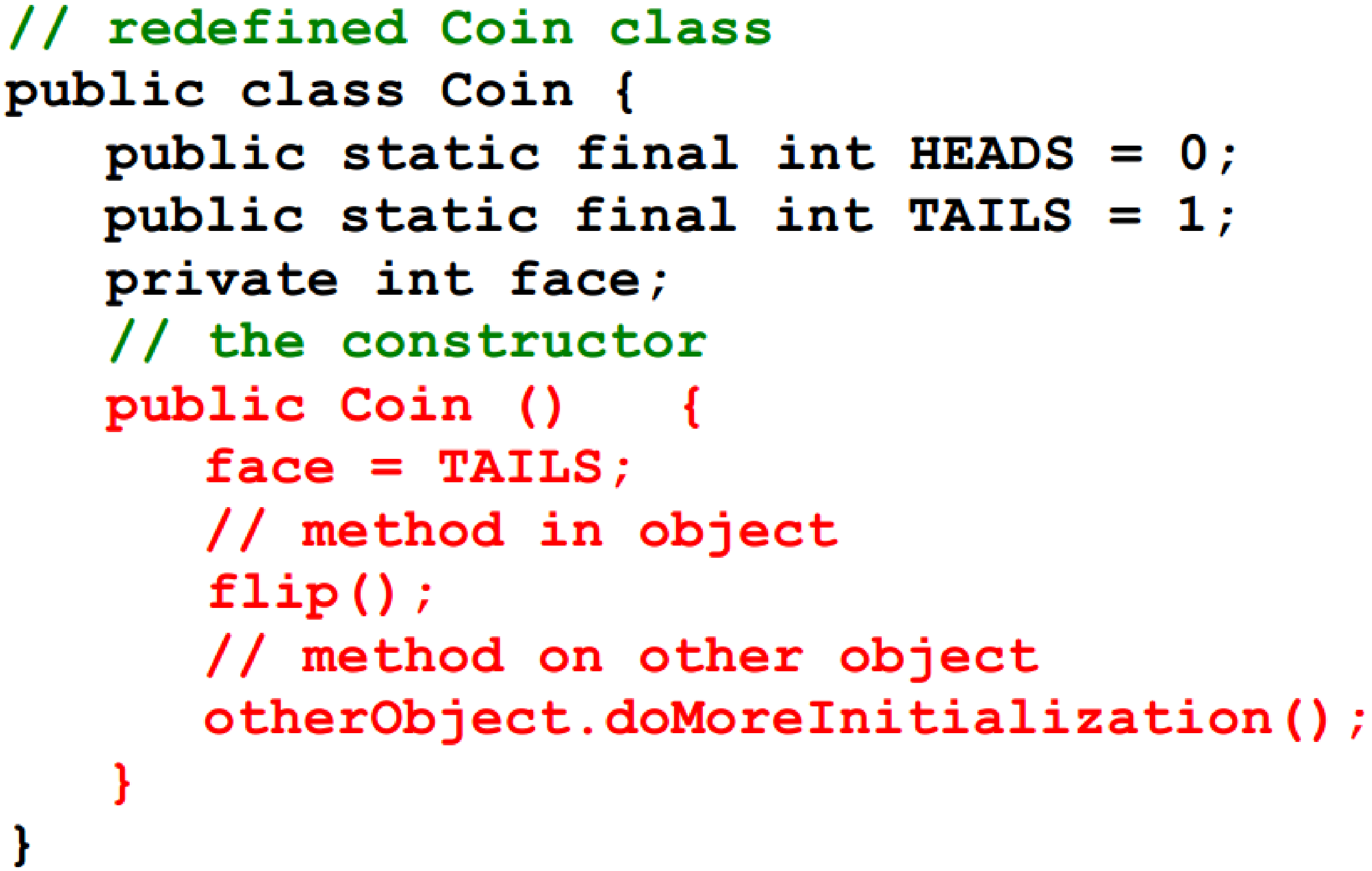


Constructors in Java

- A constructor is a special method where the instance variables 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.
  - 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)

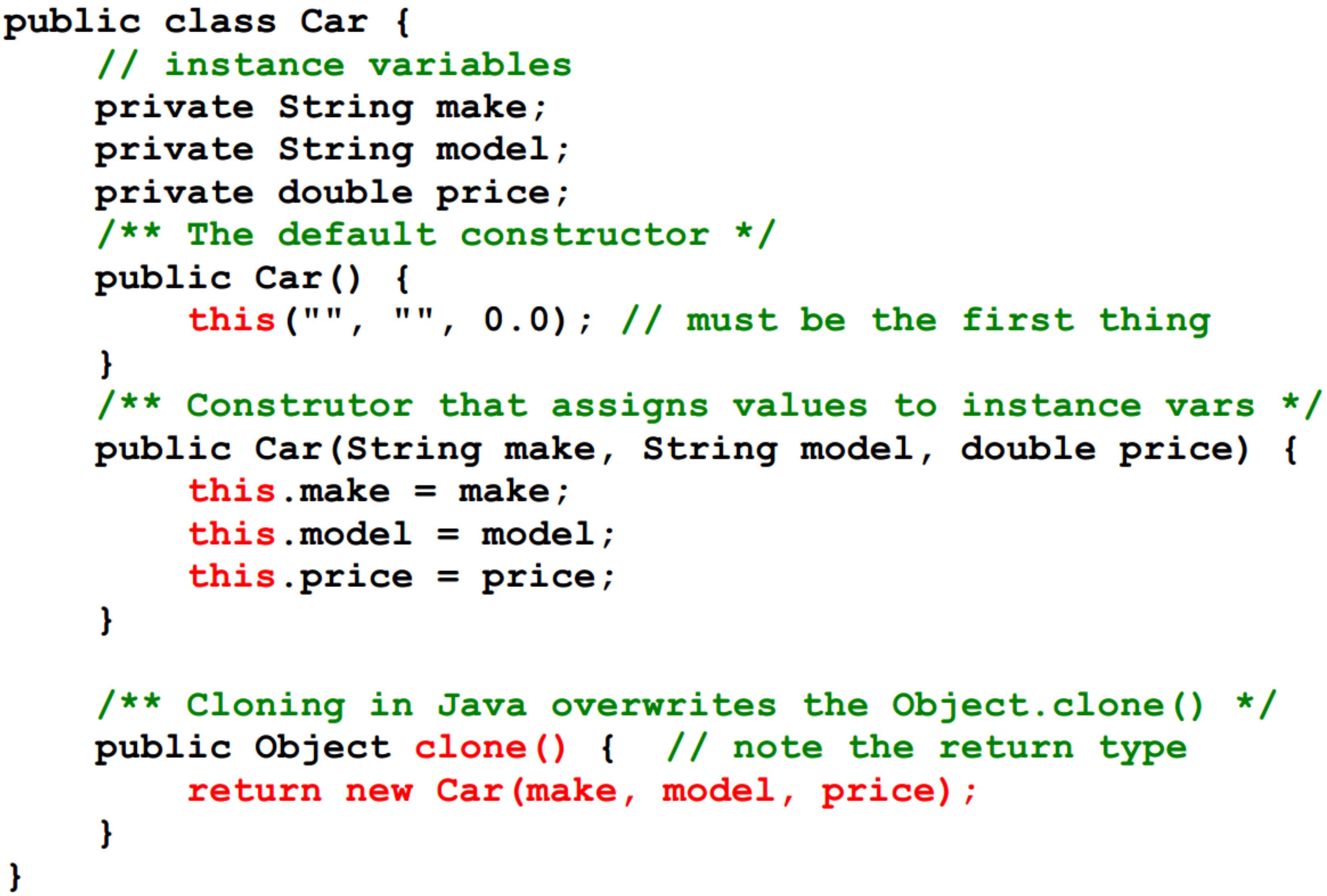
Constructors in Java, cont.

- 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;
   // the constructor
   public Coin ()   {
      face = TAILS;
      // method in object
      flip();
      // method on other object
      otherObject.doMoreInitialization();
   }
}
```

Constructors and Cloning in Java



```
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
    }
    /** Construtor 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 overwrites the Object.clone() */
    public Object clone() {  // note the return type
        return new Car(make, model, 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();
    }
}
```

Constructor vs. Method

Constructor vs. Method

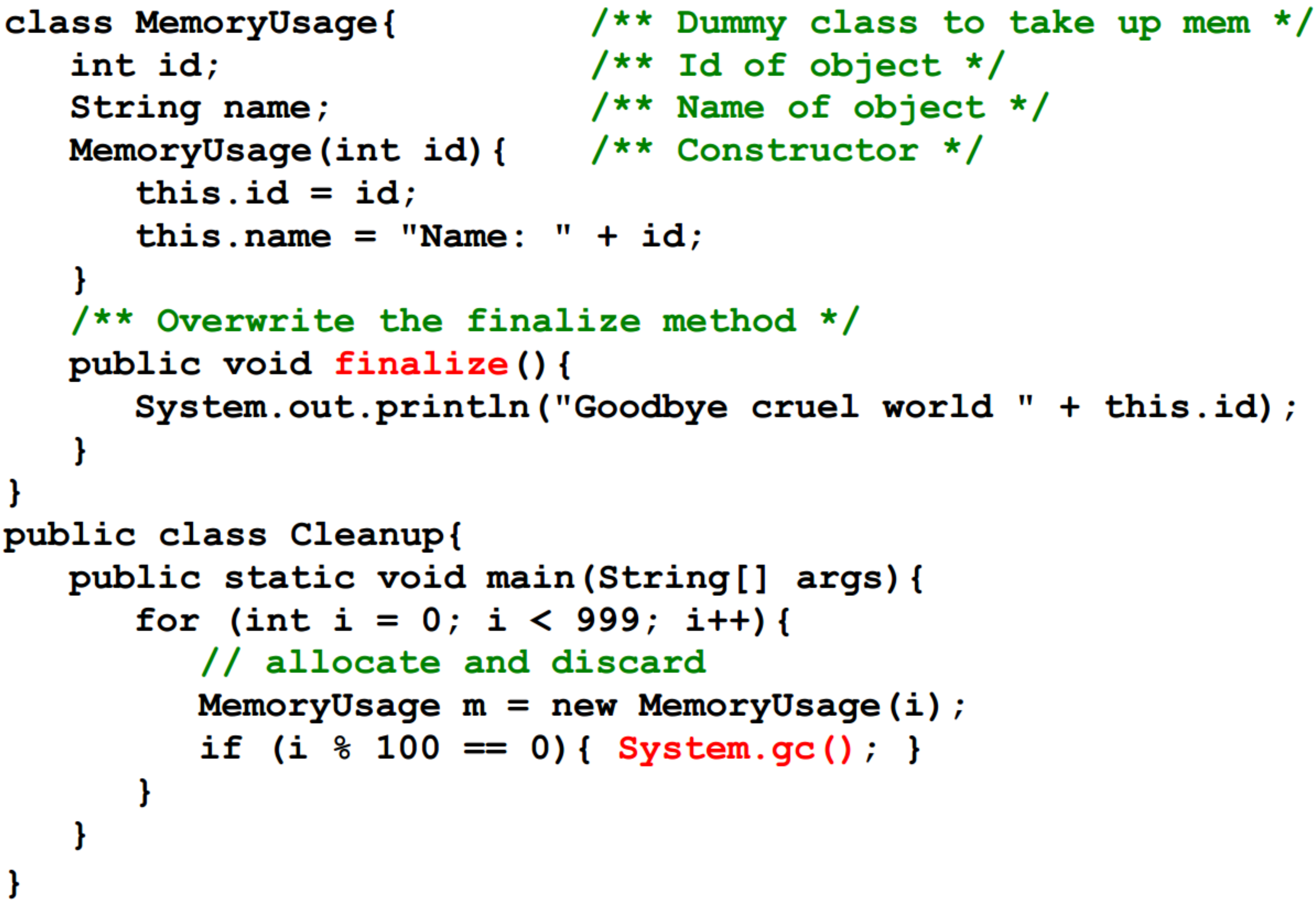
Similarities

- 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)

Dissimilarties

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

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 and discard
         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

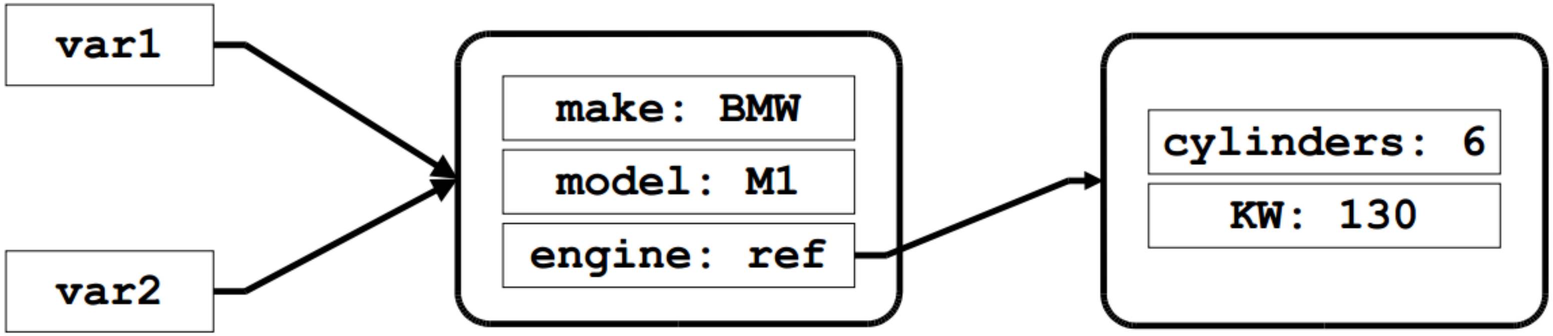
Strings in Java

- 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

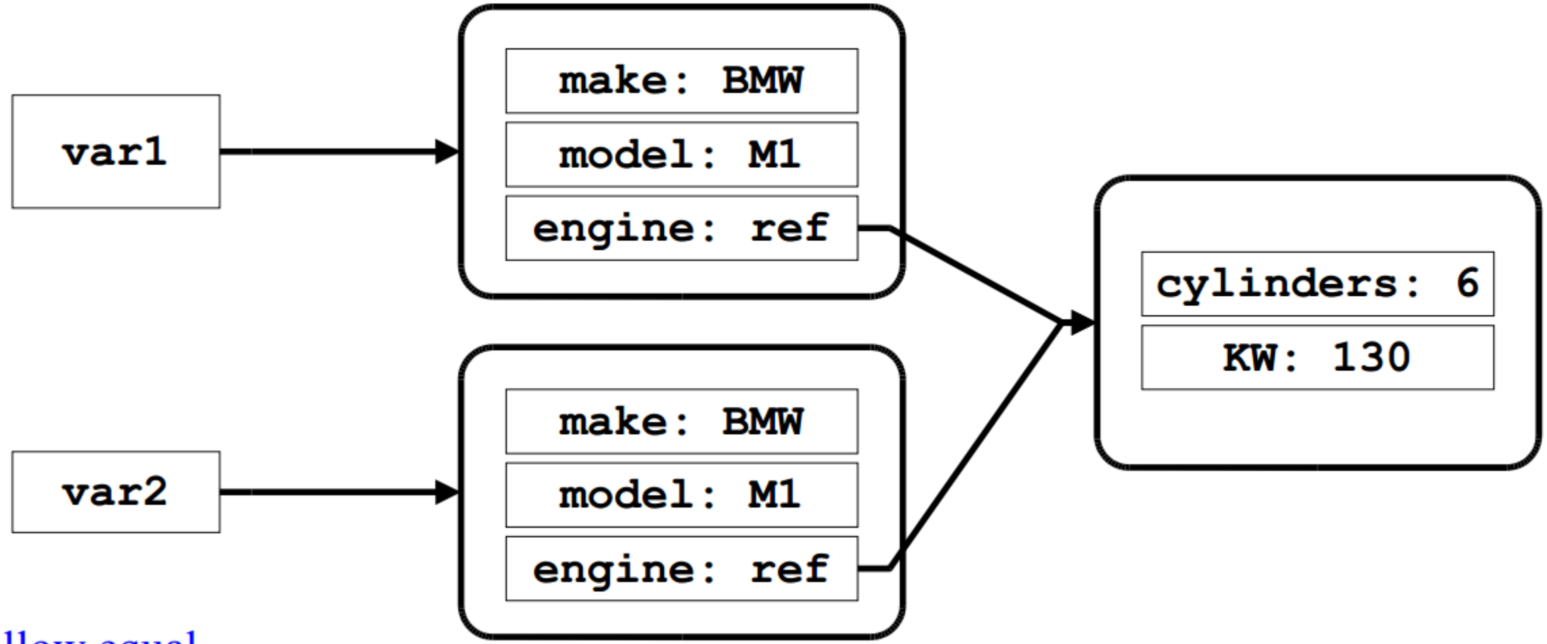
Equality

- 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

Equality Examples

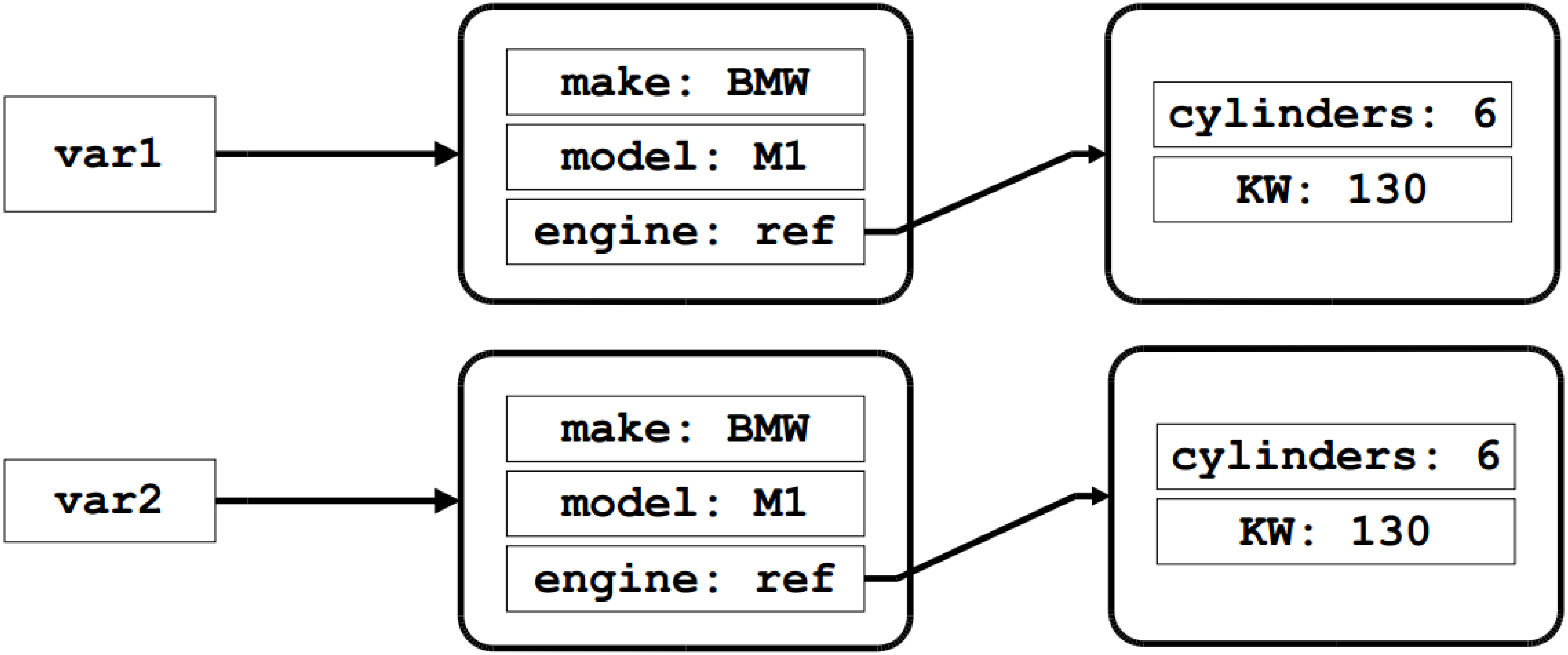


reference equal



shallow equal

Equality Examples, cont.



Types of Equality in Java

- ==
  - Equality on primitive data types
    - 8 == 7
    - 'b' == 'c'
  - Reference equality on object references
    - onePoint == anotherPoint
  - Strings are special
    - String s1 = "hello"; String s2 = "hello";
    - if (s1 == s2){
      System.out.println(s1 + " equals" + s2);}
- *equals*
  - Method on the class *java.lang.Object*
  - Default works like reference equality.
  - Can be refined in subclass
- □ onePoint.equals(anotherPoint)

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() == this.make
            && ((Car) o).getModel() == this.model
            && ((Car) o).getPrice() == this.price;
            // relies on “short circuiting”
    }
```

Summary

- Overview Essential OOP Java Principles
- Instance variables, Strings, …
- Methods, Overloading
- Initialization
- Garbage collection
- Equality
- Working with Classes & Objects

This Week

- Read Chapters 8, 9, 10
- Review Slides
- Work through Java Exercises

Questions/Discussion