CPS209: COMPUTER SCIENCE II

Objects and Classes

Container class vs. Definition class

- Container class:
  - A collection of **static** methods that are not associated/bound to any particular object
  - These static methods usually have something in common

Container Class - Example

- The **`Math`** class is an example of a *container* class. It is essentially a container for math utility methods: `Math.sqrt()`, `Math.abs()`, `Math.max()`, ...

```
class MathExample
{
  public static void main (String[] args)
  {
    double x1,y1,x2,y2;

    Scanner in = new Scanner(System.in);

    x1 = in.nextDouble(); y1 = in.nextDouble();
    x2 = in.nextDouble(); y2 = in.nextDouble();

    double dist = Math.sqrt((x2 – x1)*(x2 – x1) +
                            (y2 – y1)*(y2 – y1));
    System.out.println("Distance = " + distance);
  }
}
```

Container Class - Example

- Notice how you call Container class methods (i.e. execute them) by writing the name of the ***class*** followed by a **.**
  - i.e. `Math.sqrt()`

Container Classes

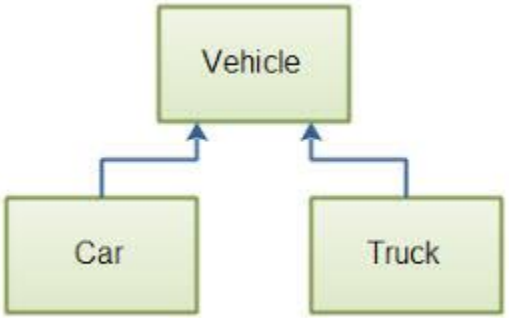
- Container classes allow you to structure your programs as a *collection of static methods (i.e. functions)*
  - This style of program organization – a collection of methods – has a long history and is used in languages like C
- However, for even moderately complex programs a large collection of methods can quickly becomes difficult to manage

Definition Classes

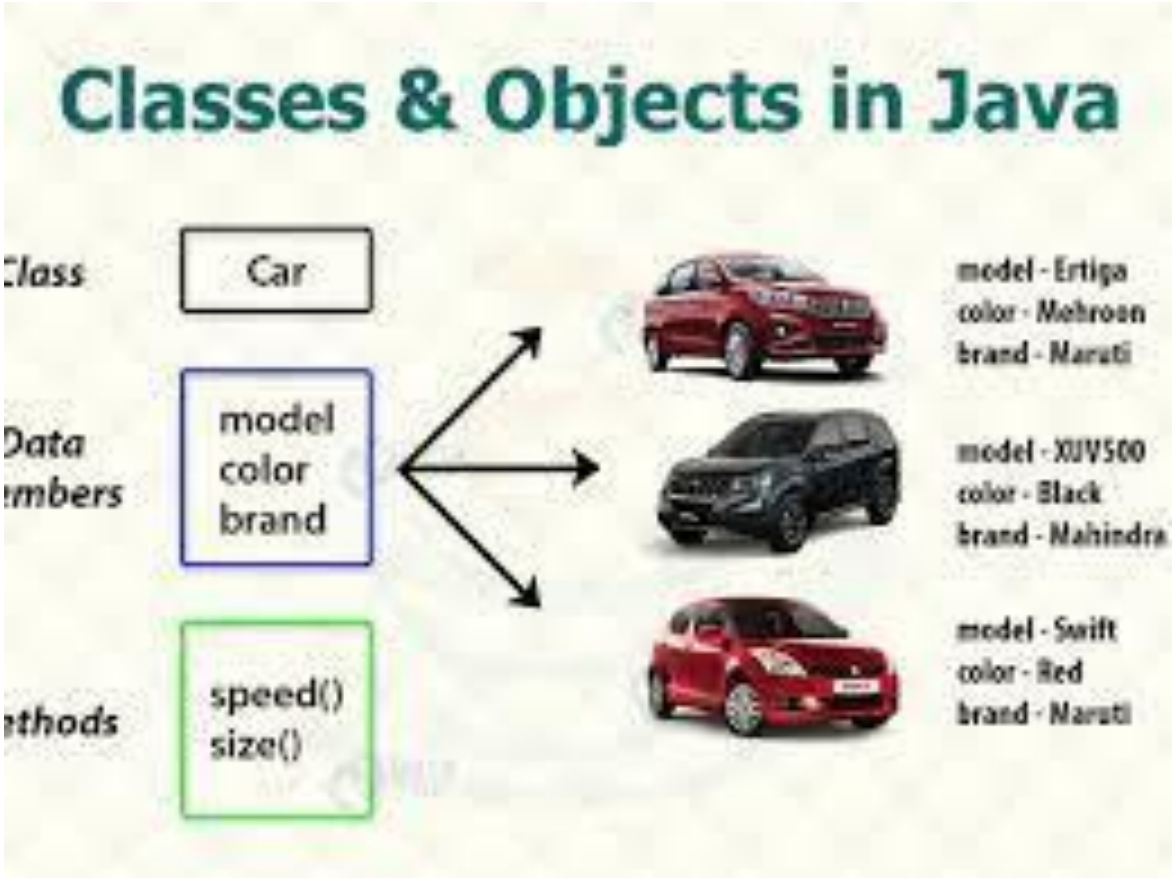
- In cps209 we will focus on **Definition** Classes
  - True Object-Oriented programming!
- Definition classes:
  - These classes define new *objects*
  - Think of Definition class as **instructions for creating a new object**

Objects and (Definition) Classes

- Objects and Classes are a higher level of abstraction than the use of just methods
  - Allows us to more naturally model objects and information in the real world
  - Especially if they are naturally arranged hierarchically
  - How?



Classes and Objects



Objects and Classes

- Gathers both variables (i.e. "data") **and** methods into a container: an ***object***

1. Manages complexity

2. Structures your program by breaking it into "collaborating" "high-level" objects that reflect real world objects and real-world object behavior

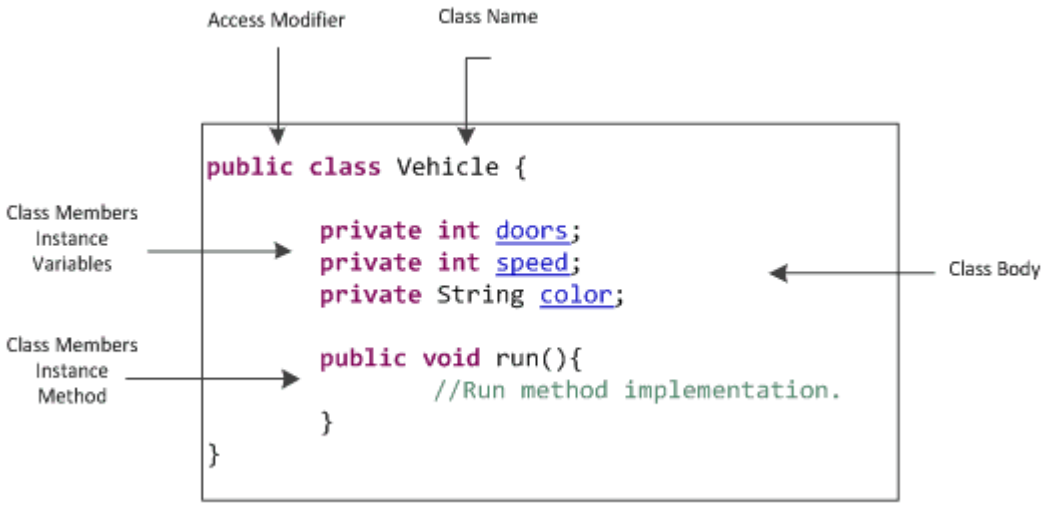
3. *Encapsulates* data and provide well defined public methods to access and modify the data fields
   - i.e. "protects" the data and hides implementation details (which may evolve)

Objects and Classes

- A Definition **Class** can be thought of as a **blueprint** that can be used to build an **object**
  - Analogy: Blueprint of a house (class) and the house itself (object)
  - i.e. a Class is a *set of instructions* for building an object!

Objects and Classes

- From a *practical* point of view, a Class typically **defines**:
  - The type of **data** that will be held inside an object
    - in the form of *variables* (also called *fields*)
  - The names of (non-static) methods (and their parameters)
    - methods describe the *behavior* of an object
  - And the **code** for these methods



Class – An *Abstract* View

- We usually begin by creating an *abstraction* of a Class:
  - What does an object of this class represent?
  - What is the interface of the class?
  - What is the internal state of the class?

- We use ***variables*** to represent (i.e. store) the internal state
  - we need to decide what variables we need and what are their types (int, float etc.)

- We need to decide on the *behavior* (or interface) of the class
  - the class interface is realized (i.e. implemented) via ***methods***

Example: Clock Class Abstraction

- A `Clock` class *represents* a 12-hour clock
- Its *internal state* includes: `hours, minutes, seconds`
- Its *interface* allows telling the clock that a second has elapsed, and querying the clock to retrieve the time:

```
Clock(int hours, int minutes, int seconds)
void secondHasElapsed()
int getSeconds()
int getMinutes()
int getHours()
...
```

Objects and Classes

- To **use a definition class** (i.e. read/set its variables via its methods) you **must!!:**

instantiate (i.e. *create*) an object using the **new** operator!!!

- This is analogous to building a house from the blueprint.

- You can create many objects from a single class
  - i.e. just like you can build many houses from a single blueprint
  - i.e. you can create many objects from a single *set of instructions*

Example: Using a Clock class

```
class ClockTester
{
  static void main (String[] args)
  {
    Clock newYorkTime = new Clock(12,59,59);// Create an object

    for (int j = 0; j < 65; j++)
        newYorkTime.secondHasElapsed();

    System.out.println(newYorkTime.getHours() +
                  ":" + newYorkTime.getMinutes() +
                  ":" + newYorkTime.getSeconds());
  }
}
```

- Note: Where is the code for *class Clock*, you ask??? Typically in another java file.
- We commonly put the tester code, with the `main` method, in its own java file

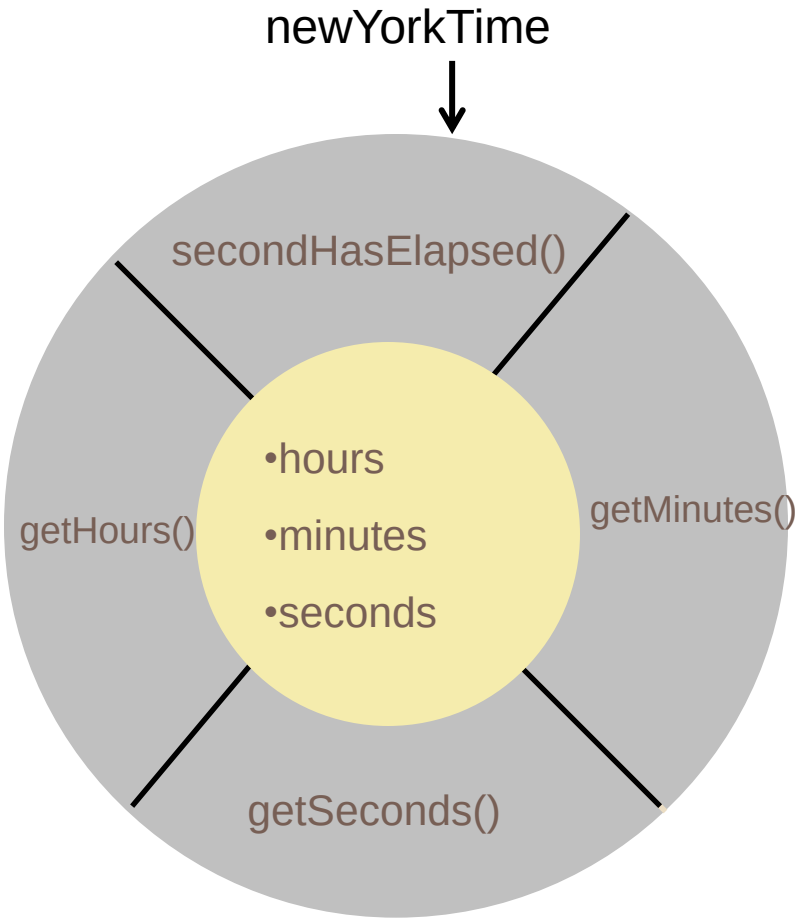
Example: Using a Clock class

```
class ClockTester
{
  static void main (String[] args)
  {
    Clock newYorkTime = new Clock(12,59,59);// Create an object

    for (int j = 0; j < 3; j++)
        newYorkTime.secondHasElapsed();

    System.out.println(newYorkTime.getHours() +
                       ":" + newYorkTime.getMinutes() +
                       ":" + newYorkTime.getSeconds());
  }
}
```

NOTE: notice how we use the name of the *object reference variable* `newYorkTime` to call the method `secondHasElapsed()`, **not the name of the class (Clock)!**

Encapsulation

- *Internal state* of object typically not directly accessible to other parts of the program
- Other parts of the program can only access/modify the object's internal state using its interface
  - i.e. by calling its public methods
- Internal state is *encapsulated* or *hidden*
- Classes and Objects support program modularity



Creating Multiple Objects from a Class

```
public class ClockTester
{
  public static void main(String[] args)
  {
    Clock LosAngeles = new Clock(1,17,52);
    Clock NewYork    = new Clock(4,17,52);
    Clock London     = new Clock(9,17,52);

    // ...


  }
}
```

Creating Multiple Objects from a Class

LosAngeles

Clock Object

hours: 1

minutes: 17

seconds 52

…..

code for methods

NewYork

Clock Object

hours: 4

minutes: 17

seconds 52

…..

code for methods

London

Clock Object

hours: 9

minutes: 17

seconds 52

…..

code for methods

Structure of the Clock class

```
public class Clock
{
    private int hours, minutes, seconds;
    public Clock(int h, int m, int s){ … } // constructor
    public void secondHasElapsed() { … }
    public int getHours() { … }
    public int getMinutes() { … }
    public int getSeconds() { … }
}
```

- *Private:* only visible *within* the class. Usually used for fields (i.e. variables) and "helper" methods.
- *Public:* accessible to all other classes. Usually used for methods.

Instance Variables

- Internal state variables declared *outside of methods*, but *inside the class* { ...}
  - State variables are also called *instance variables* or *fields*.

```
public class Clock
{
Public static int num = 0;
private int hours, minutes, seconds;
    // … methods …
}
```

Constructor Methods

- Objects must be initialized before they can be used.
  - i.e. initialize its instance variables!
- A *constructor* is a method that initializes instance variables
  - It has exactly the same name as the class
  - It is automatically called when an object is created

```
public class Clock
{
    private int hours, minutes, seconds;

    public Clock(int h, int m, int s) // constructor
    {
       hours = h;
       minutes = m;
       seconds = s;
    }
    // other methods …
}
```

Constructors never return values, but do not use void in their declaration

Constructor Methods

- A class can have **many** constructor methods (or **none**!)
  - Each constructor method must have different parameter types
- If you do not supply any constructors, the compiler will make a default constructor automatically
  - It takes no parameters
  - It initializes all instance variables

By default, numbers are initialized to 0, booleans to false, and objects as null.

- You cannot directly call a constructor method like other methods
  - It is called for you when you create an object using new

Example:
Clock Class Constructor Method

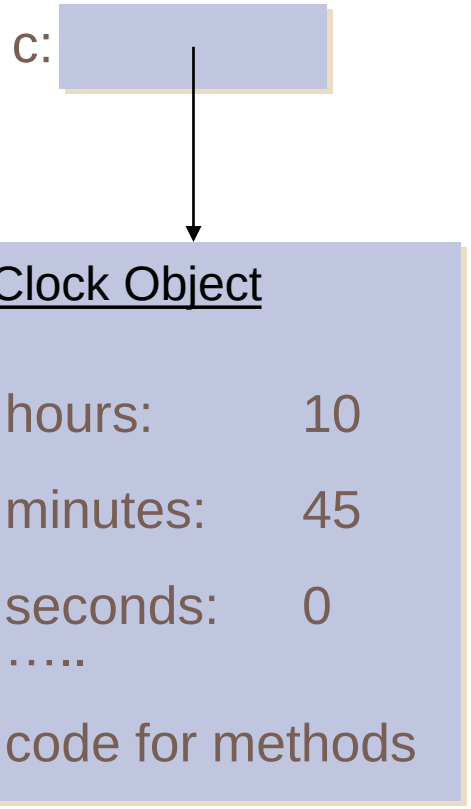
```
public class Clock
{
    private int hours, minutes, seconds;

    public Clock(int h, int m, int s) // constructor
    {
       hours = h;
       minutes = m;
       seconds = s;
    }
    // other methods of Class Clock…
}
```

Example: Clock Class Constructor Method

```
public class Clock
{
    private int hours, minutes, seconds;

    public Clock(int h, int m, int s) // constructor
    {
        hours = h;
        minutes = m;
        seconds = s;
    }
    // other methods …
}
```

Example: Instantiating a Clock object and calling its constructor method

```
public class ClockTester
{
  public static void main(String[] args)
  {
    Clock c;
    c = new Clock(10,45,0);
    // ...
   }
}
```



NOTE: Memory Space for holding the object instance variables and code is only allocated **when the object is constructed** (i.e. **new**).

Object References

- Object reference variables store the ***location*** of an object in memory, **not the object itself!!**
- That is, the new operator returns the location (a memory address) of the new object and stores this location in the reference variable

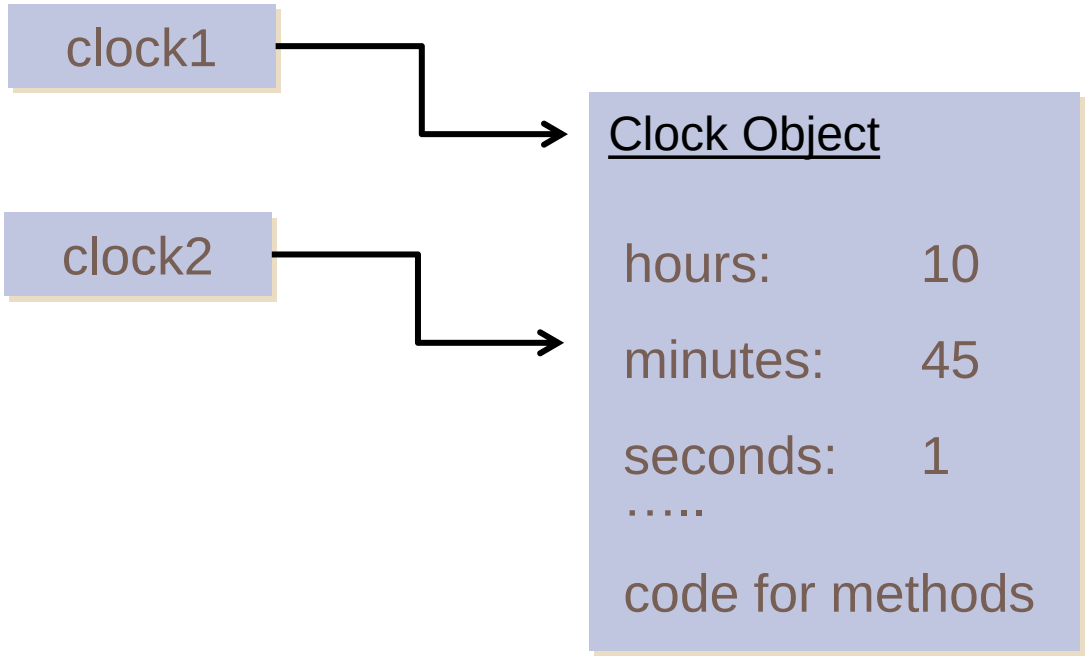
```
Clock clock1 = new Clock(10, 45, 0);
```

- Multiple reference variables can refer to the same object!!

```
Clock clock2 = clock1;
clock2.secondHasElapsed();
```

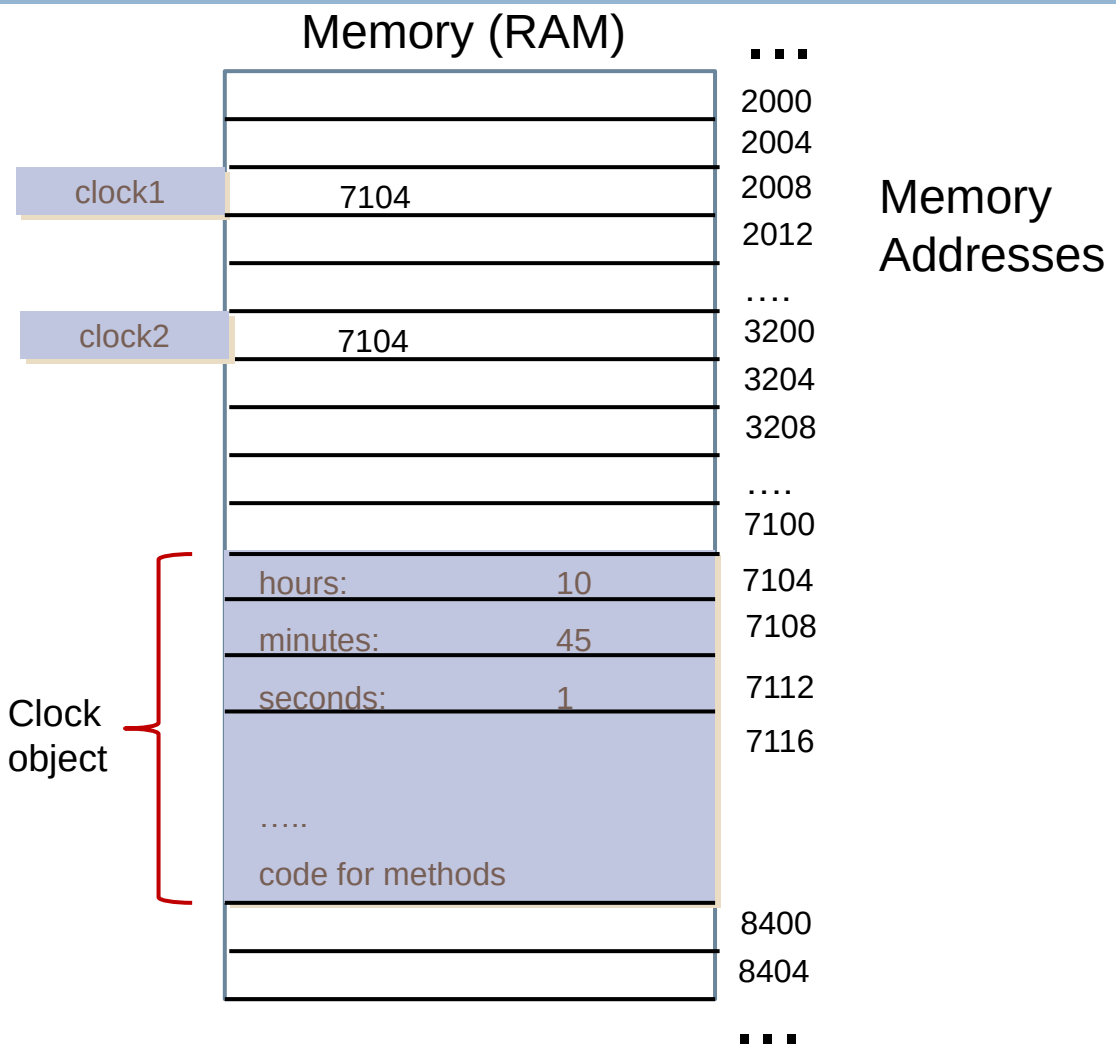
Reference Variables: Example

- Two Clock references variables referring to the same object



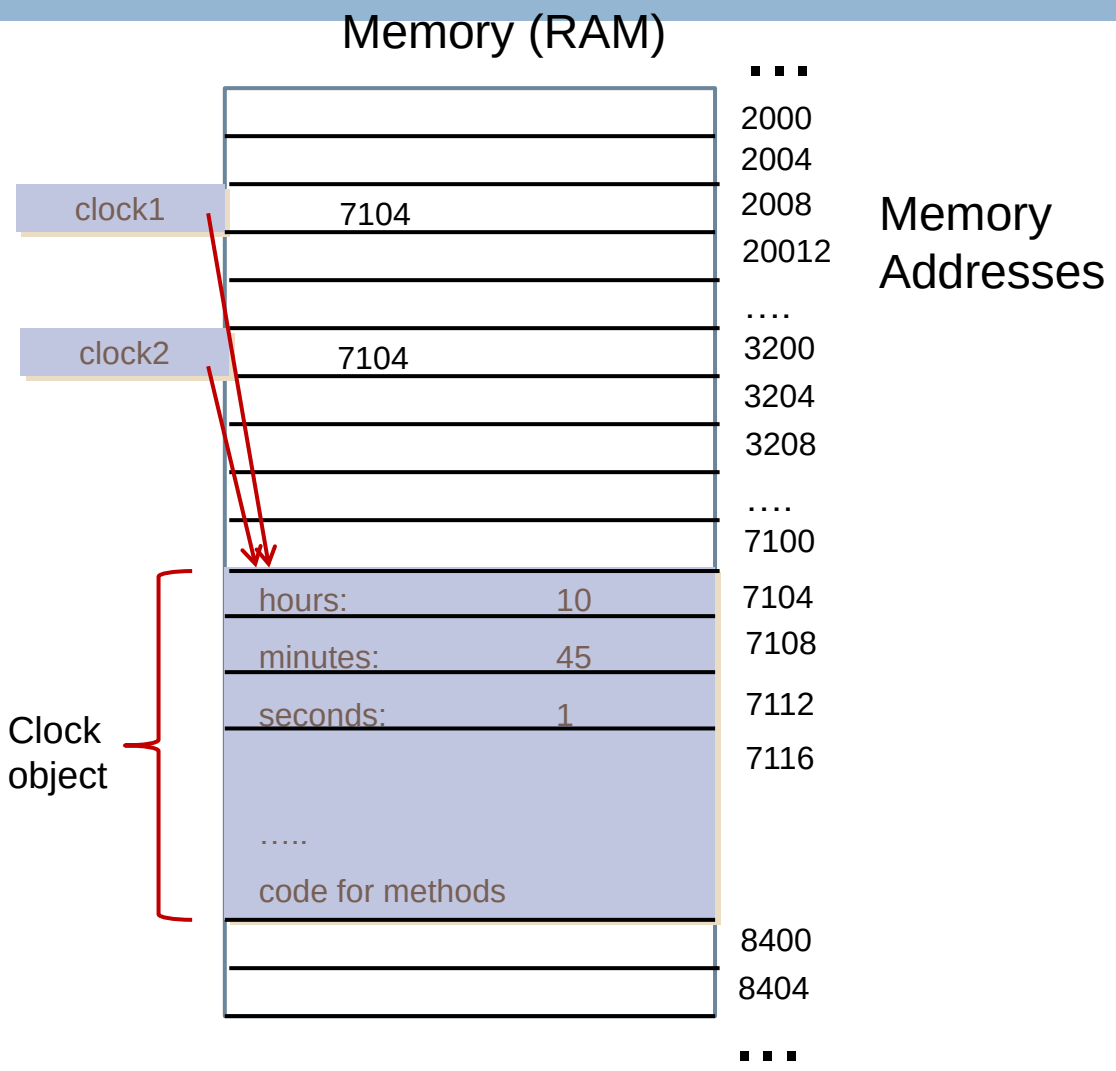
Reference Variables

- **Example Memory View**
- Note: random memory locations chosen in this example
- See how the reference variables (clock1, clock2) store the same address of the Clock object



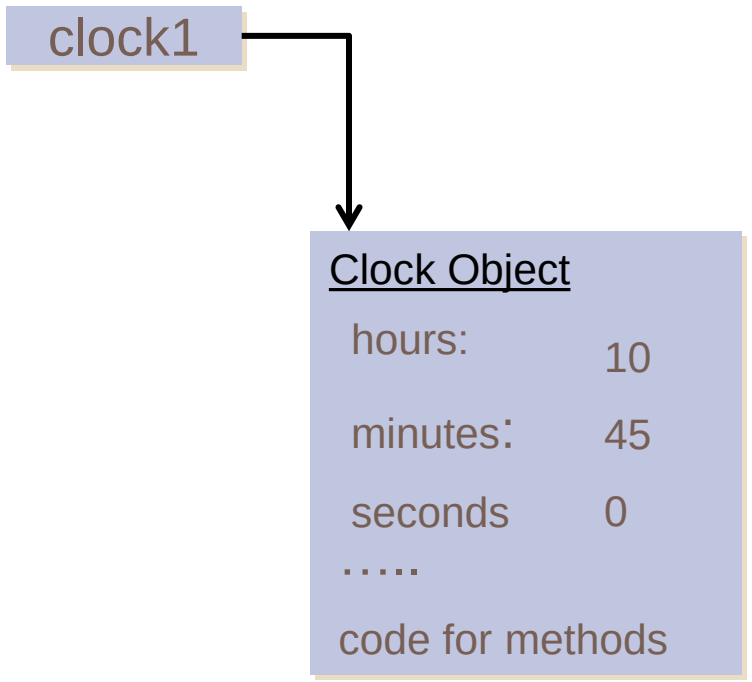
Reference Variables

- **Example Memory View**
- Note: random memory locations chosen in this example
- See how the reference variables (clock1, clock2) store the same address of the Clock object



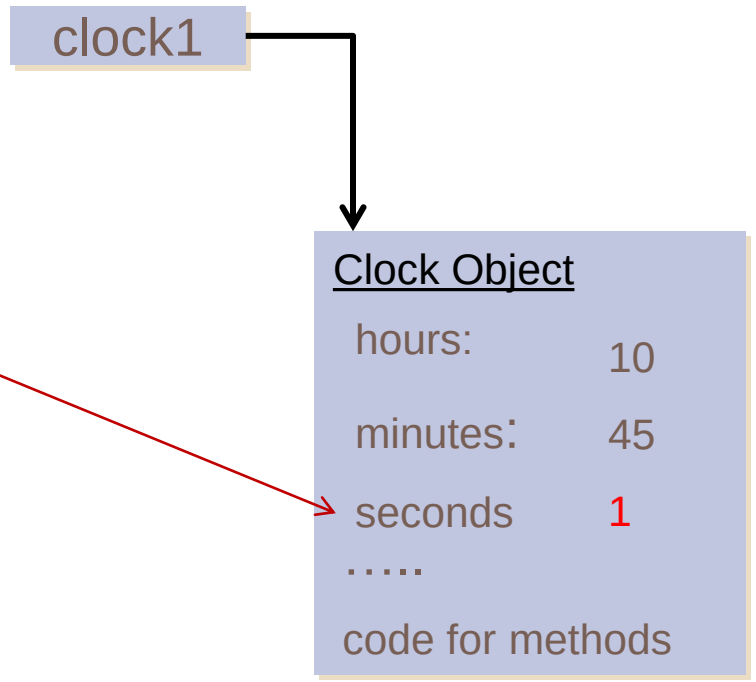
Reference Variables: Example

```
class ClockTester
{
  static void main (String[] args)
  {
    Clock clock1 = new Clock(10,45,0);
  }
}
```



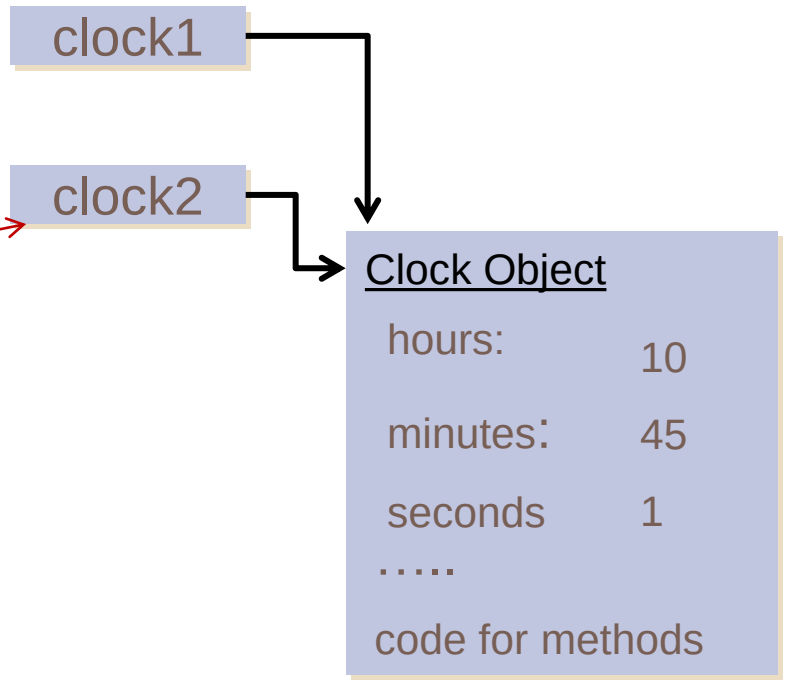
Reference Variables: Example

```
class ClockTester
{
  static void main (String[] args)
  {
    Clock clock1 = new Clock(10,45,0);
    clock1.secondHasElapsed();
  }
}
```



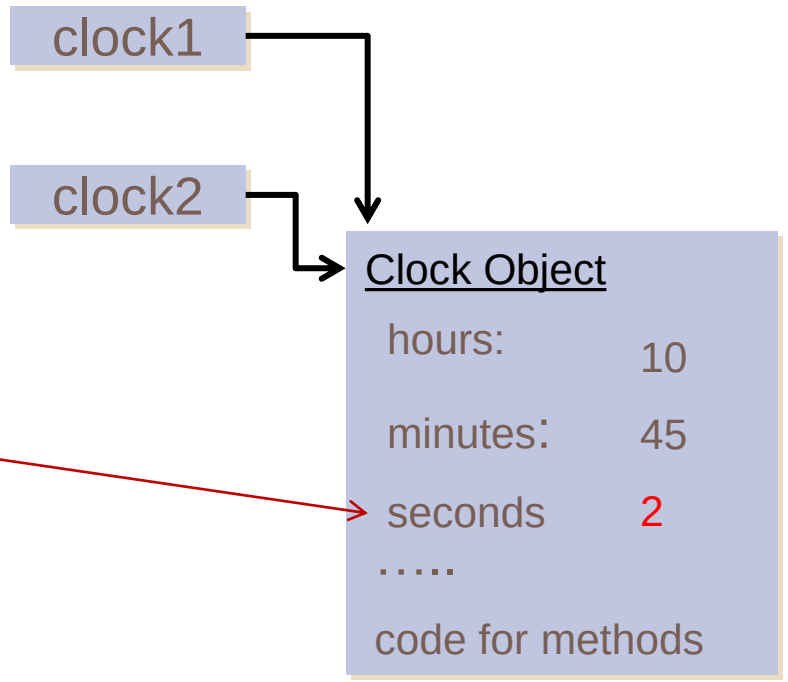
Reference Variables: Example

```
class ClockTester
{
  static void main (String[] args)
  {
    Clock clock1 = new Clock(10,45,0);
    clock1.secondHasElapsed();

    Clock clock2 = clock1;
  }
}
```



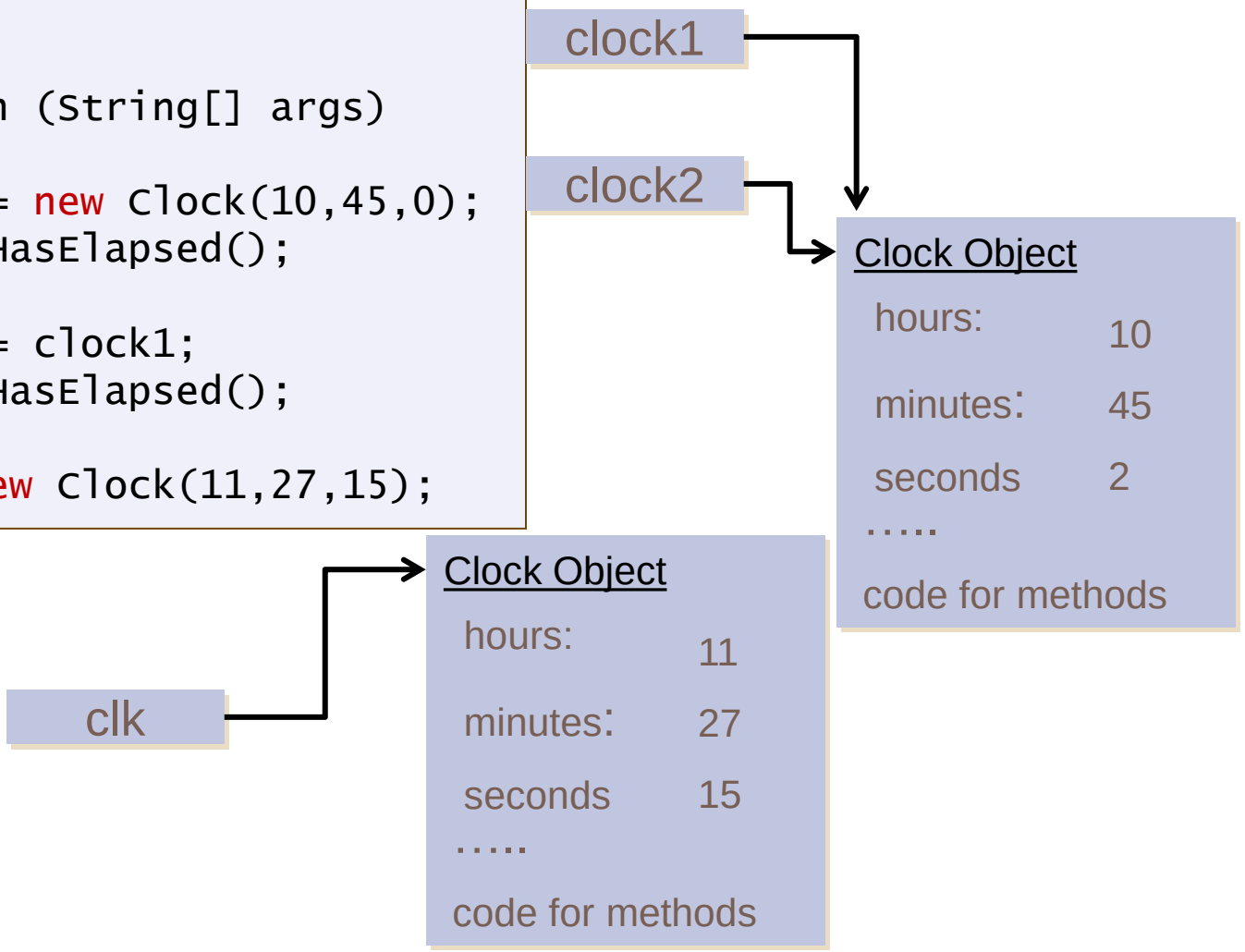
Reference Variables: Example

```
class ClockTester
{
  static void main (String[] args)
  {
    Clock clock1 = new Clock(10,45,0);
    clock1.secondHasElapsed();

    Clock clock2 = clock1;
    clock2.secondHasElapsed();
  }
}
```



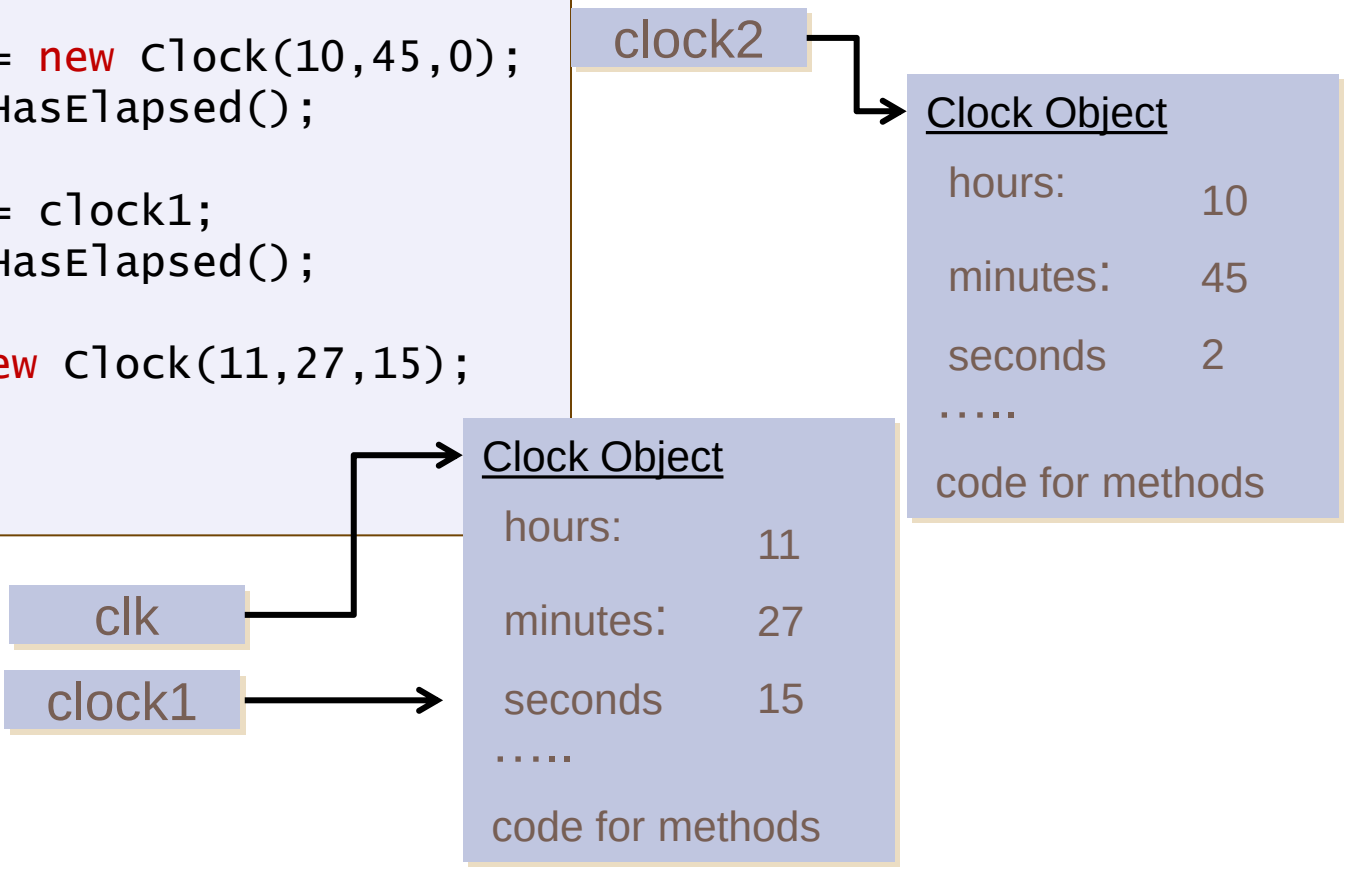
Reference Variables: Example

```
class ClockTester
{
  static void main (String[] args)
  {
    Clock clock1 = new Clock(10,45,0);
    clock1.secondHasElapsed();

    Clock clock2 = clock1;
    clock2.secondHasElapsed();

    Clock clk = new Clock(11,27,15);
  }
}
```



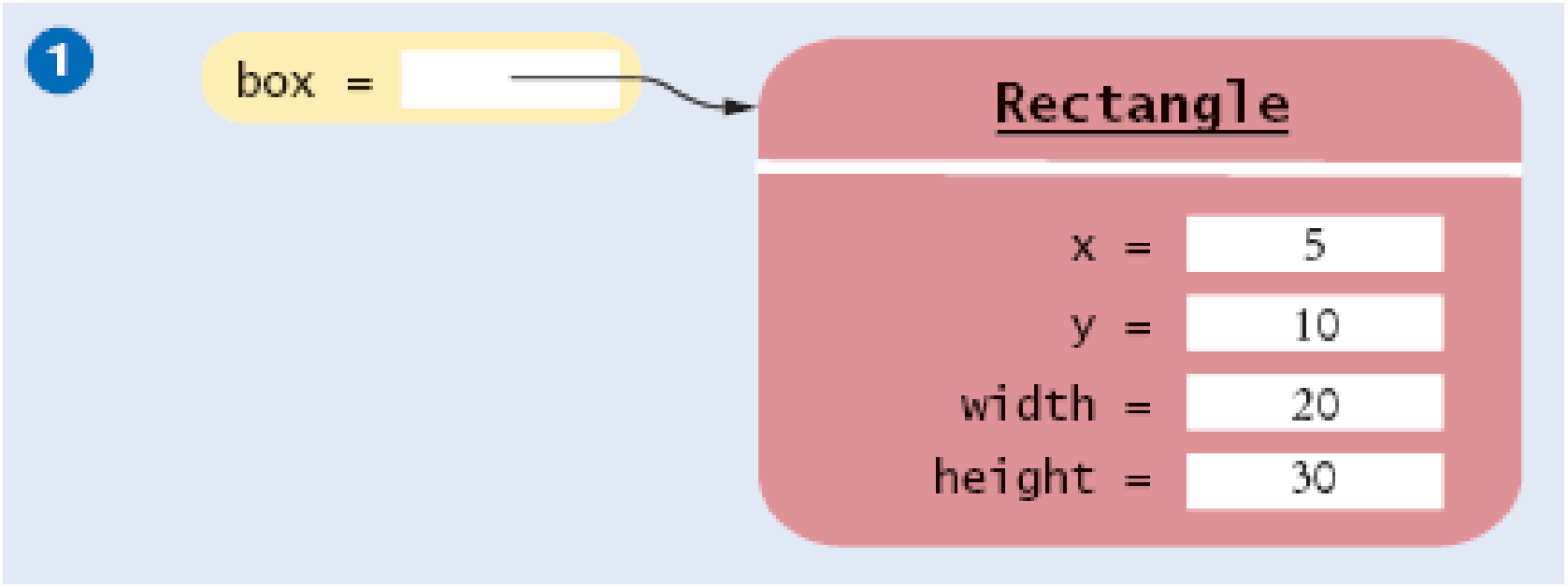
Reference Variables: Example

```
class ClockTester
{
  static void main (String[] args)
  {
    Clock clock1 = new Clock(10,45,0);
    clock1.secondHasElapsed();

    Clock clock2 = clock1;
    clock2.secondHasElapsed();

    Clock clk = new Clock(11,27,15);
    clock1 = clk;

  }
}
```



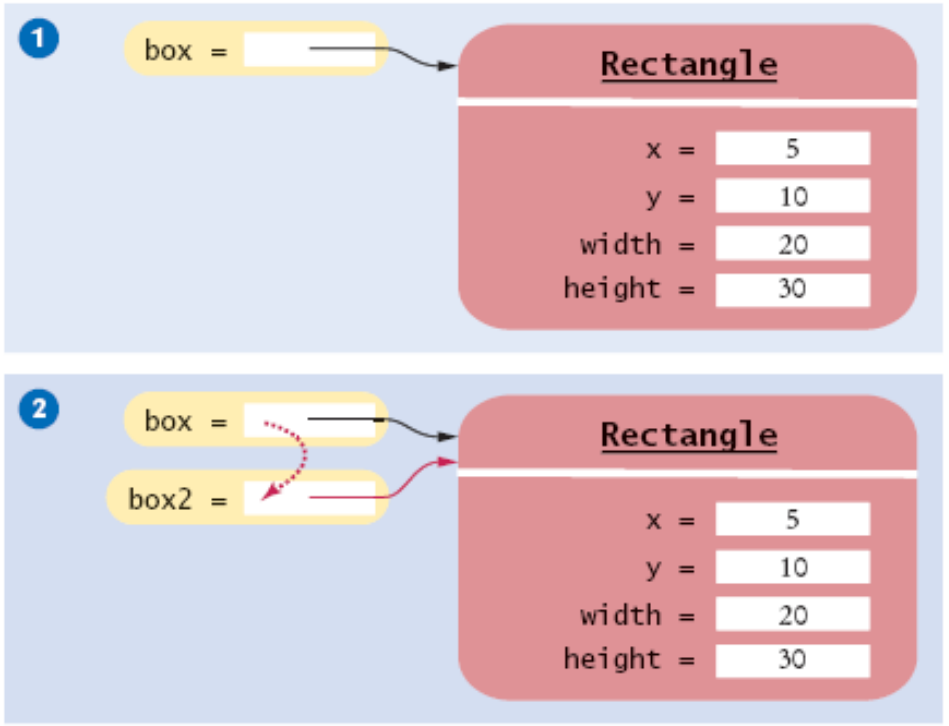
Reference Variables: Example 2

- `Rectangle box = new Rectangle(5, 10, 20, 30);`



Reference Variables: Example 2

```
Rectangle box = new Rectangle(5, 10, 20, 30);
Rectangle box2 = box;
```



Reference Variables: Example 2

```
Rectangle box = new Rectangle(5, 10, 20, 30); 1
Rectangle box2 = box; 2
box2.translate(15, 25); 3
```

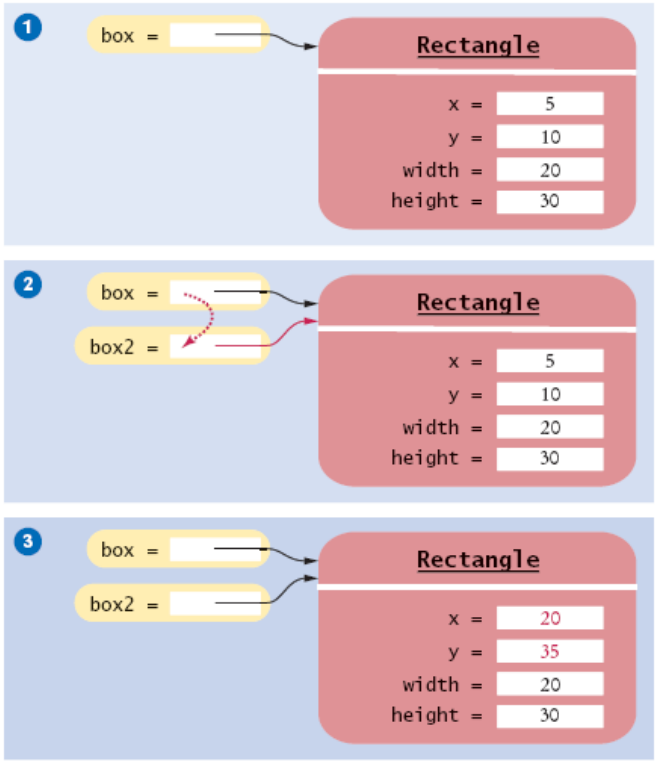


Figure 20 Copying Object References

Null Reference

- A reference variable may be set to point to 'no' object
  - You cannot call methods of an object using a reference variable that is pointing to nothing (i.e. == null)
  - Causes a run-time error!

```
class ClockTester
{
  static void main (String[] args)
  {
    Clock clk   = null;
    int seconds = clk.getSeconds(); // Runtime error!!
    System.out.println("Seconds = " + seconds);
  }
}
```

Garbage Reference

- A reference variable that is not initialized refers to a random location in memory
- You can only call methods using the reference variable if it is referring to a valid object

```
class ClockTester
{
  static void main (String[] args)
  {
    Clock clk;
    int seconds = clk.getSeconds(); // Runtime error!!
    System.out.println("Seconds = " + seconds);
  }
}
```

Null Reference

- Sometimes must check if a reference is null before using it

```
class ClockTester
{
  static void main (String[] args)
  {
    Clock clk = . . . // e.g. read Clock object from a file

    if (clk != null)
      System.out.println("Seconds = clk.getSeconds());
    }
}
```

- In this example we might be calling a method that reads hours, minutes and seconds from a file, creates a Clock object, and then returns a reference to it. The file could be corrupt or empty

Reminder: Definition Class

- We are creating objects and executing methods via a reference variable which "points" to the object
- **We are not creating a container class of static methods!**

```
class ClockTester
{
  static void main (String[] args)
  {
    // Error! getSeconds is not static!!
    int seconds = Clock.getSeconds();
    System.out.println("Seconds = " + seconds);
  }
}
```

String Objects: Reminder

- Strings ***are objects*** created by using ***class String!***

```
class StringExample
{
  static void main (String[] args)
  {
    String fname = “john”; // this is a java shortcut!

    String lname = new String(“smith”); // true expanded form

    fname = “jane”; // an entirely new String object!!

    System.out.println(“Name is “ + fname + lname);
  }
}
```

ArrayList Objects: Reminder

- ArrayLists are objects!!

Methods

- To make an object useful, we must provide *methods* that define its behavior.
- We create *instance methods*, which operate on a specific *instance* of an object
  - i.e. the methods change the variables of a specific object
- Static methods, like `main`, are not bound to a specific object

Example: secondHasElapsed() method

```
public void secondHasElapsed() {
  if (seconds < 59) seconds++;
  else {
    seconds=0;
    if (minutes < 59) minutes++ ;
    else {
      minutes = 0;
      hours = hours < 12 ? hours+1 : 1;
    }
  }
}
```

Return Types

- Methods may *return* a value
- The *return type* of a method indicates the type of value that the method sends back to the calling method (e.g. float, int)
  - The return-type of `getHours()` is `int`.
- A method that does not return a value (such as `secondHasElapsed()` has a `void` return type
- `return` statement specifies the value that should be returned,
  - The value type must conform with the method return type

Clock *Accessor* Methods

```
public int getHours() {
   return hours;
}

public int getMinutes() {
   return minutes;
}

public int getSeconds() {
   return seconds;
}
```

Method Context

- The `getHours()` and `secondHasElapsed()` methods are *instance methods*,
  - Which means they act on a particular instance (i.e. object) of the class:
  - i.e. they change or access the variables stored inside the object

- Methods cannot be called "out of the blue". They must be called for a particular object using the reference to the object:

```
Clock c = new Clock(1,2,3);
getHours();              // Error!! which clock object?
int h = c.getHours();    // Correct! will return 1
```

ClockTest example

```
public class ClockTester
{
  public static void main(String[] args)
  {
    Clock swatch = new Clock(12,59,59);
    Clock rolex  = new Clock(12,59,59);

    System.out.println(swatch.getHours()); // 12
    System.out.println(rolex.getHours());  // 12

    swatch.secondHasElapsed();

    System.out.println(swatch.getHours()); // 1
    System.out.println(rolex.getHours());  // 12
  }
}
```

Method Parameters

- A method can be defined to accept zero or more parameters
- Each parameter is defined by its *type* and *name*
- Parameters in the *method definition* are called *formal parameters*
  - Values passed to a method when it is called are called *actual parameters*
- The name of the method together with the list of its formal parameters is called the ***signature of the method***
- A new method for class Clock

```
public void setTime(int h, int m, int s)
```

Method setTime()

```
public void setTime(int h, int m, int s)
{
  if ((s >= 0) && (s < 60) &&
      (m >= 0) && (m < 60) &&
      (h > 0) && (h <= 12)) {
     hours = h;
     minutes = m;
     seconds = s;
  }
 // Note: no effect if parameter values are
 // not in valid range.
}
```

Accessor and Mutator Methods

- Many methods fall into two categories:
  - Accessor Methods: '**get**' methods
    - Asks the object for information
    - Normally return a value of some variable
    - e.g. `int getHours()`
  - Mutator Methods: '**set**' methods
    - Changes value of a variable in the object
    - Usually take a parameter that will change an instance variable
    - Normally returns void
    - e.g. `void setTime(int h, int m, int s)`

Variable Scope

- (Non-static) variables may be declared inside a:
  1. Class – instance variables (outside of methods)
  2. Method/Constructor – local variables (and parameters)
  3. Inner block of a method { … } – also local variables

- A variable is recognized throughout the "block" {…} in which it was defined. This is called the ***scope*** of the variable.
  - same variable name may be used in different scopes, and are in fact totally different variables
  - If the same name is used in an outer and inner scope, then the inner scope definition "hides" the outer one.

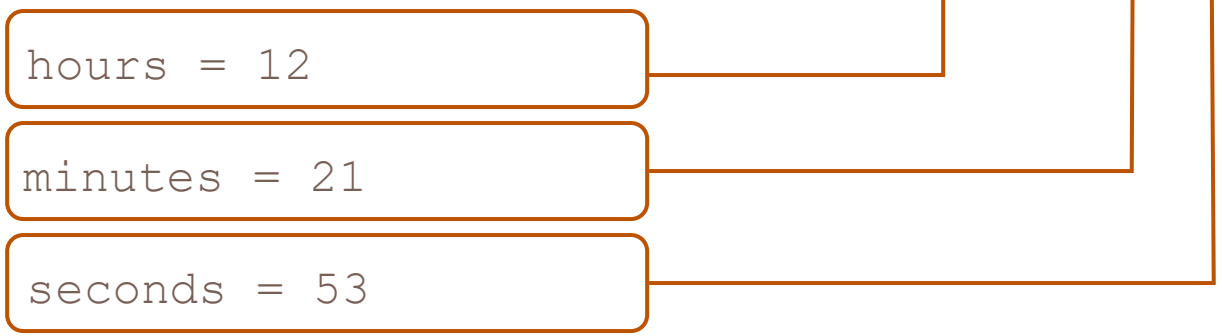
Passing Parameters

```
public class ClockTester
{
  public static void main(String[] args)
  {
    Clock breitling = new Clock(1,21,53);
    int lunchHour = 12;

    breitling.setTime(lunchHour, 21, 53);

    System.out.println(breitling.getHours()); // 12

  }
}
```

Passing Parameters by Value

- When a parameter is passed, a copy of the **value** is made and assigned to the formal parameter:

```
Clock breitling = new Clock(1,21,53);
int lunchHour = 12;
breitling.setTime(lunchHour,21,53);
```



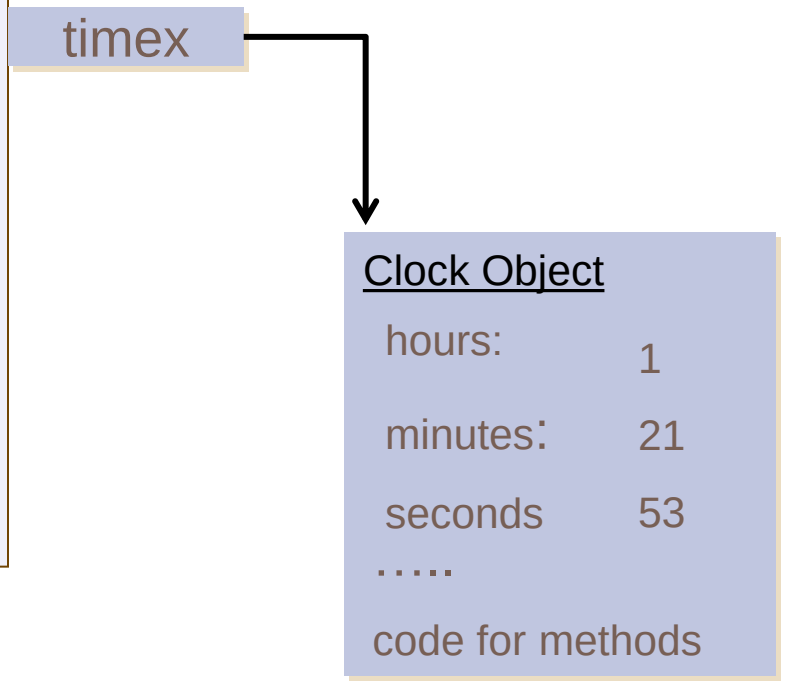
Passing Object Parameters

- When an Object reference is a parameter, only the **reference to the object** is passed by value. The parameter now points to the object.
- Let's add another `setTime` method to class `Clock` with a different signature

```
public void setTime(Clock other)
{
     hours   = other.getHours();
     minutes = other.getMinutes();
     seconds = other.getSeconds();
}
```

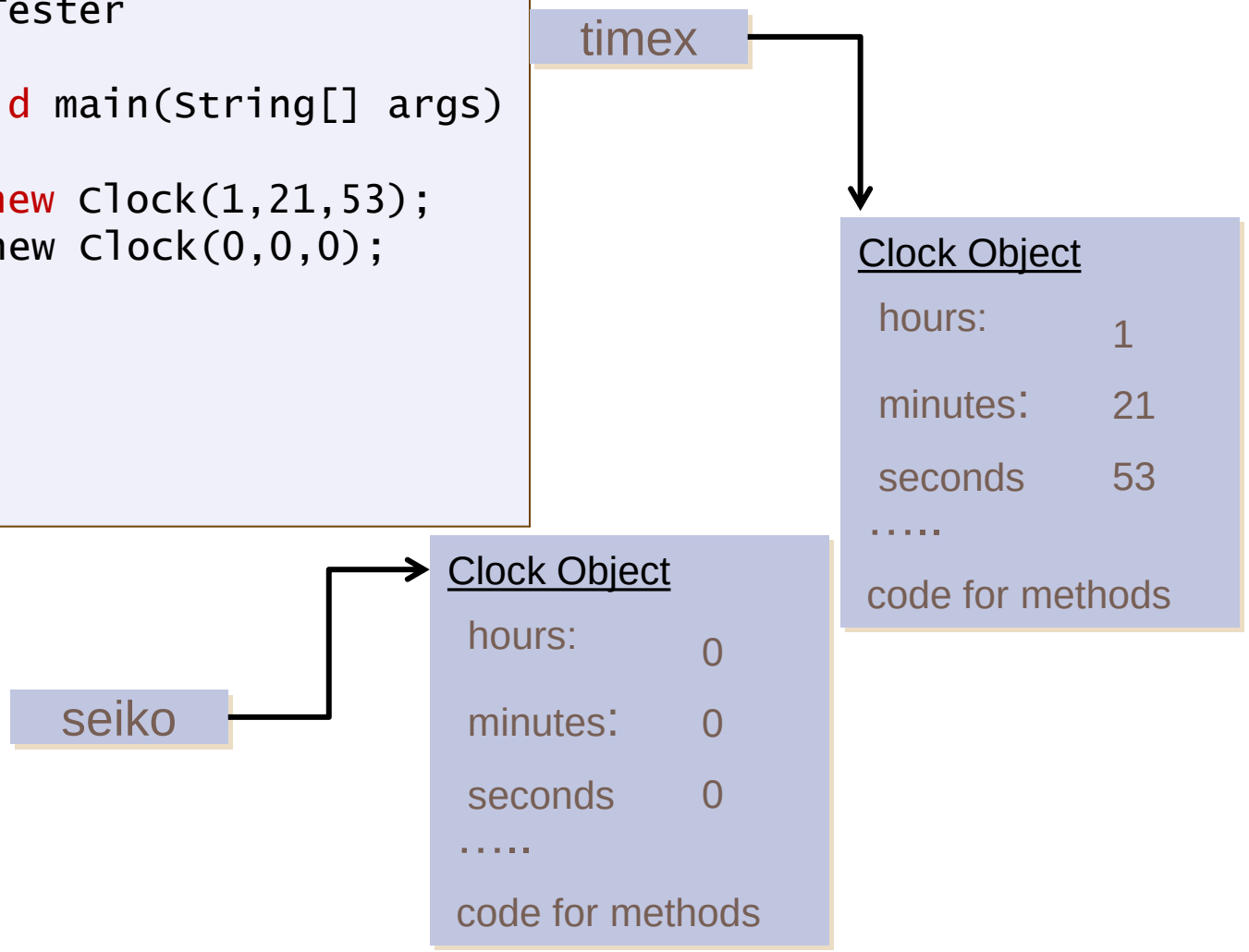
Passing Object Parameters

```
public class ClockTester
{
  public static void main(String[] args)
  {
    Clock timex = new Clock(1,21,53);

  }
}
```



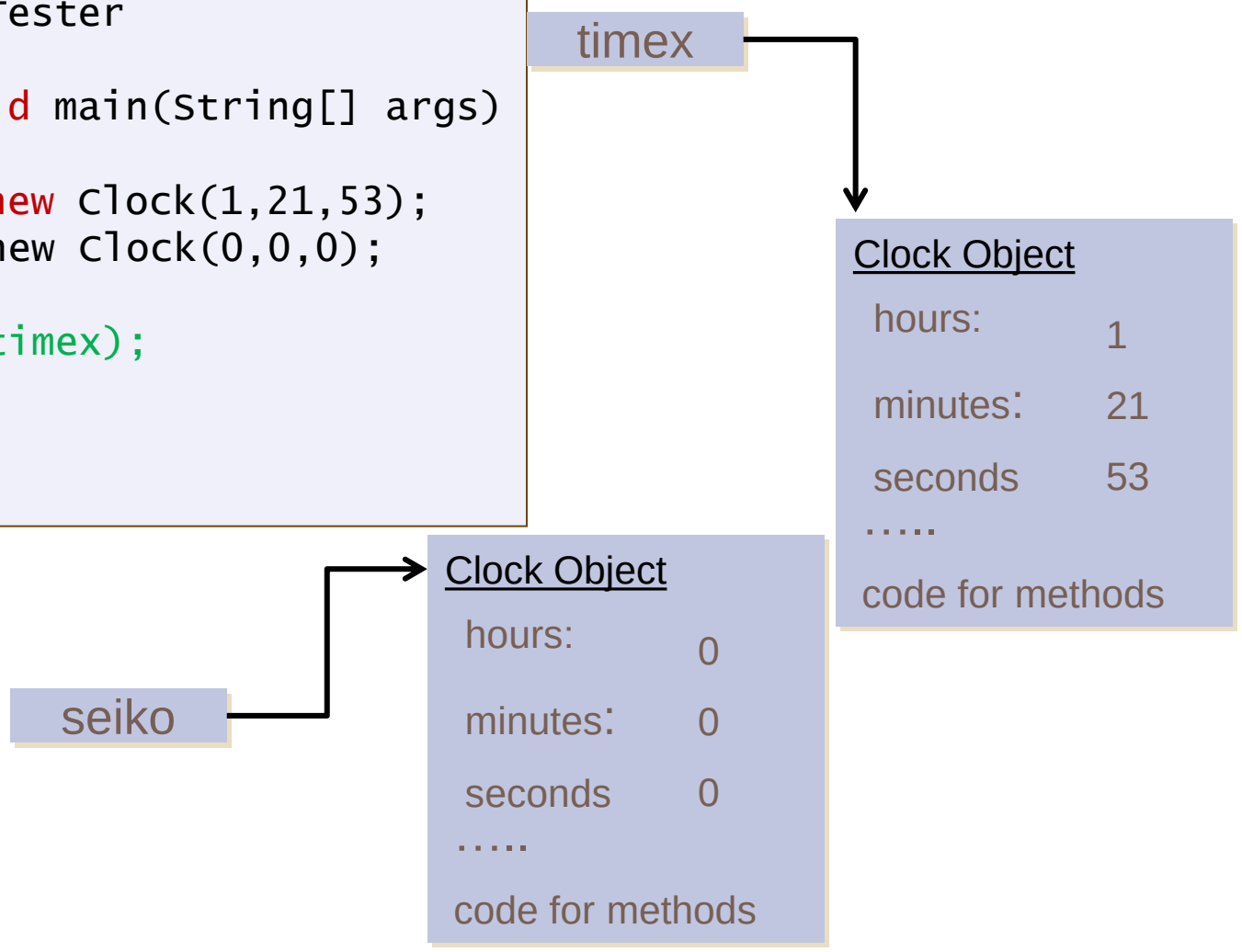
Passing Object Parameters

```
public class ClockTester
{
  public static void main(String[] args)
  {
    Clock timex = new Clock(1,21,53);
    Clock seiko = new Clock(0,0,0);

  }
}
```



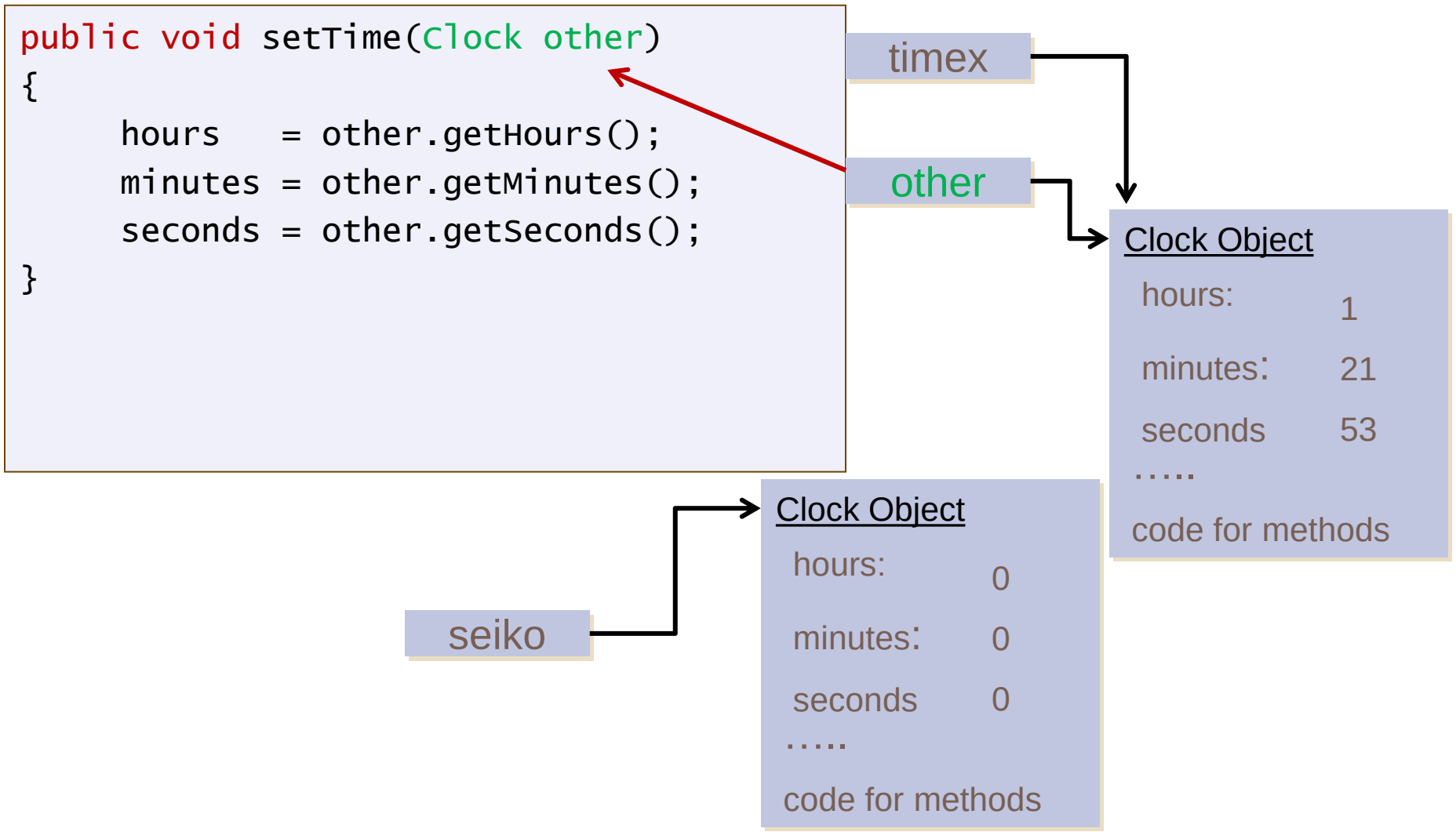
Passing Object Parameters

```
public class ClockTester
{
  public static void main(String[] args)
  {
    Clock timex = new Clock(1,21,53);
    Clock seiko = new Clock(0,0,0);

    seiko.setTime(timex);
  }
}
```



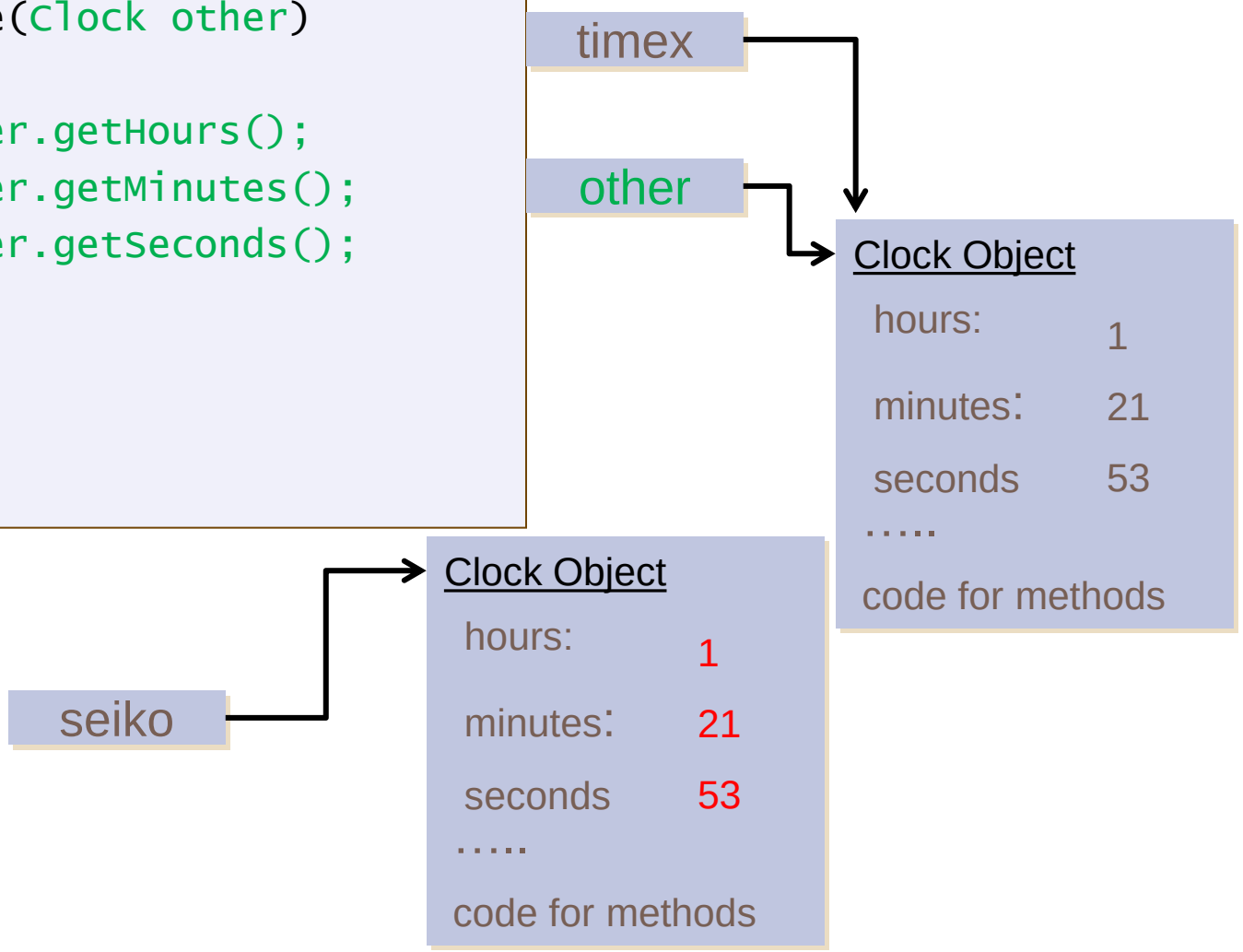
Passing Object Parameters

```
public void setTime(Clock other)
{
      hours   = other.getHours();
      minutes = other.getMinutes();
      seconds = other.getSeconds();
}
```



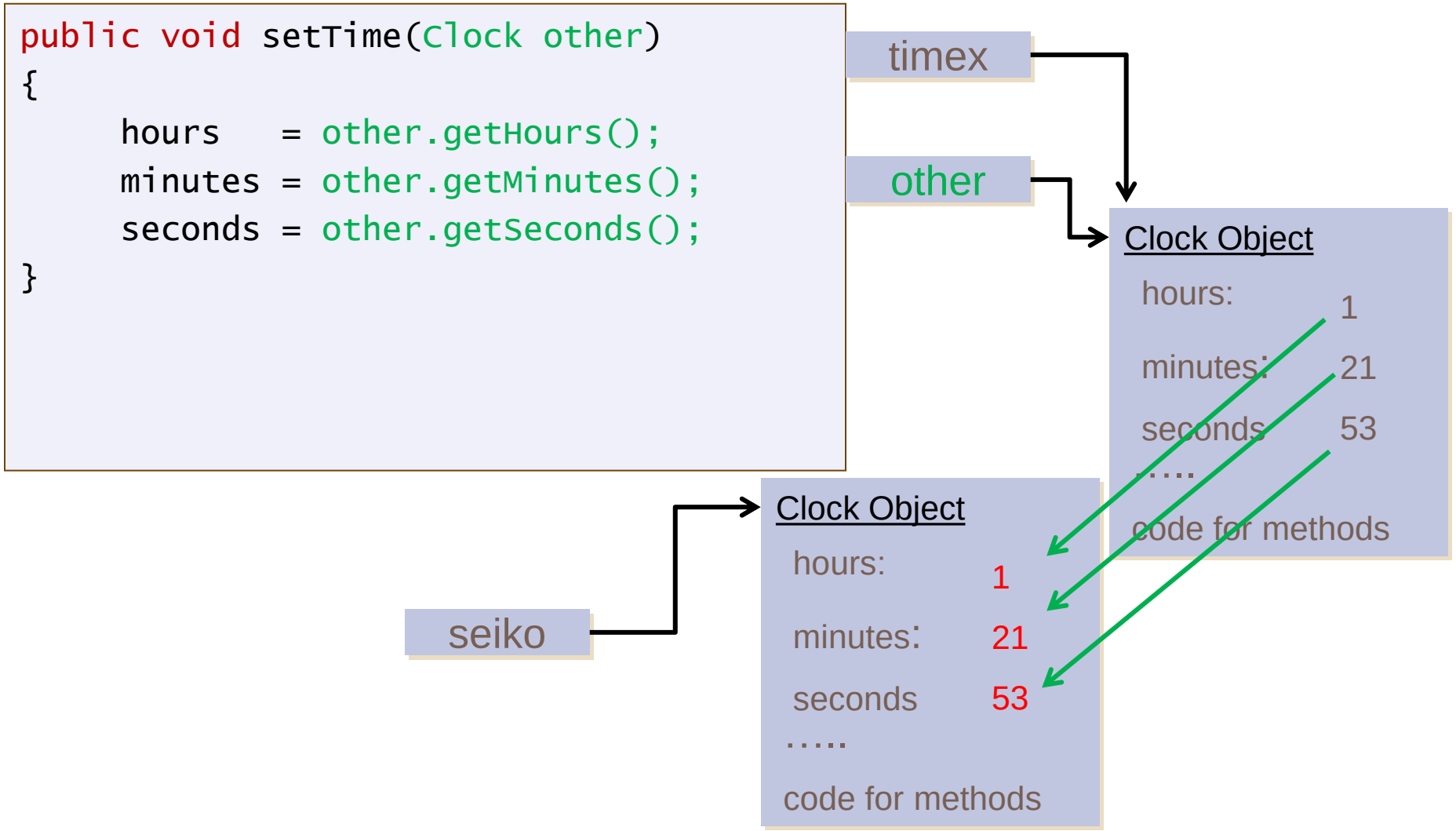
Passing Object Parameters

```
public void setTime(Clock other)
{
      hours   = other.getHours();
      minutes = other.getMinutes();
      seconds = other.getSeconds();
}
```



Passing Object Parameters

```
public void setTime(Clock other)
{
      hours   = other.getHours();
      minutes = other.getMinutes();
      seconds = other.getSeconds();
}
```



The `this` reference variable

- When appearing inside an instance method, the `this` keyword is a *reference variable* pointing to the object that the method is acting upon.
- It is like an invisible parameter that is passed to the method
- The following are equivalent

```
public int getHours() {
  return this.hours;
}
```

```
public int getHours() {
  return hours;
}
```

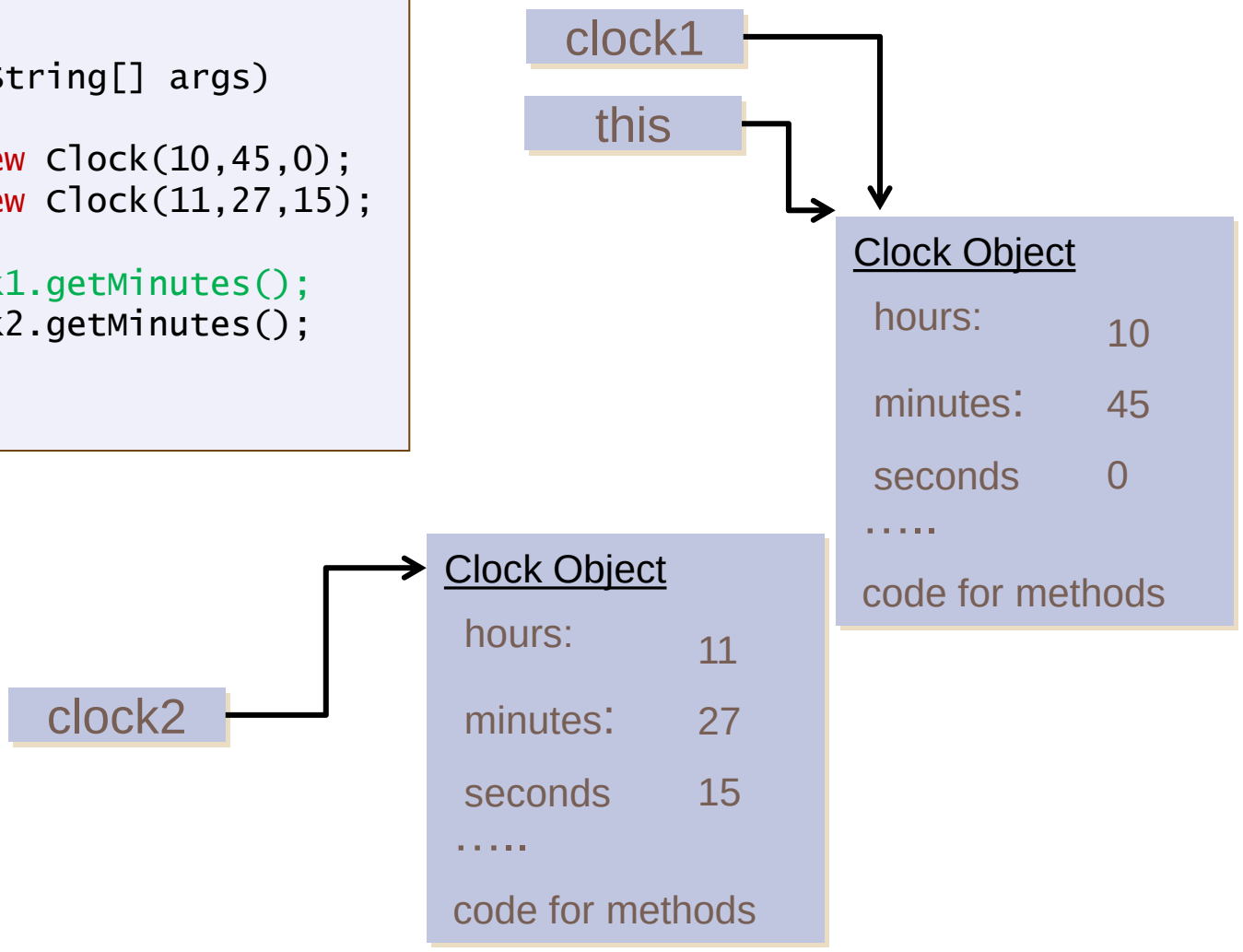
The `this` reference variable

```
class ClockTester
{
  static void main (String[] args)
  {
    Clock clock1 = new Clock(10,45,0);
    Clock clock2 = new Clock(11,27,15);

    int mins1 = clock1.getMinutes();
    int mins2 = clock2.getMinutes();
  }
}
```

```
public class Clock
{
  private int hours;
  private int minutes;
  private int seconds;

  public Clock(int h, int m, int s)
  {
     this.hours = h;
     this.minutes = m;
     this.seconds = s;
  }
  public int getMinutes()
  {
    return this.minutes;
  }
  // other methods …
}
```

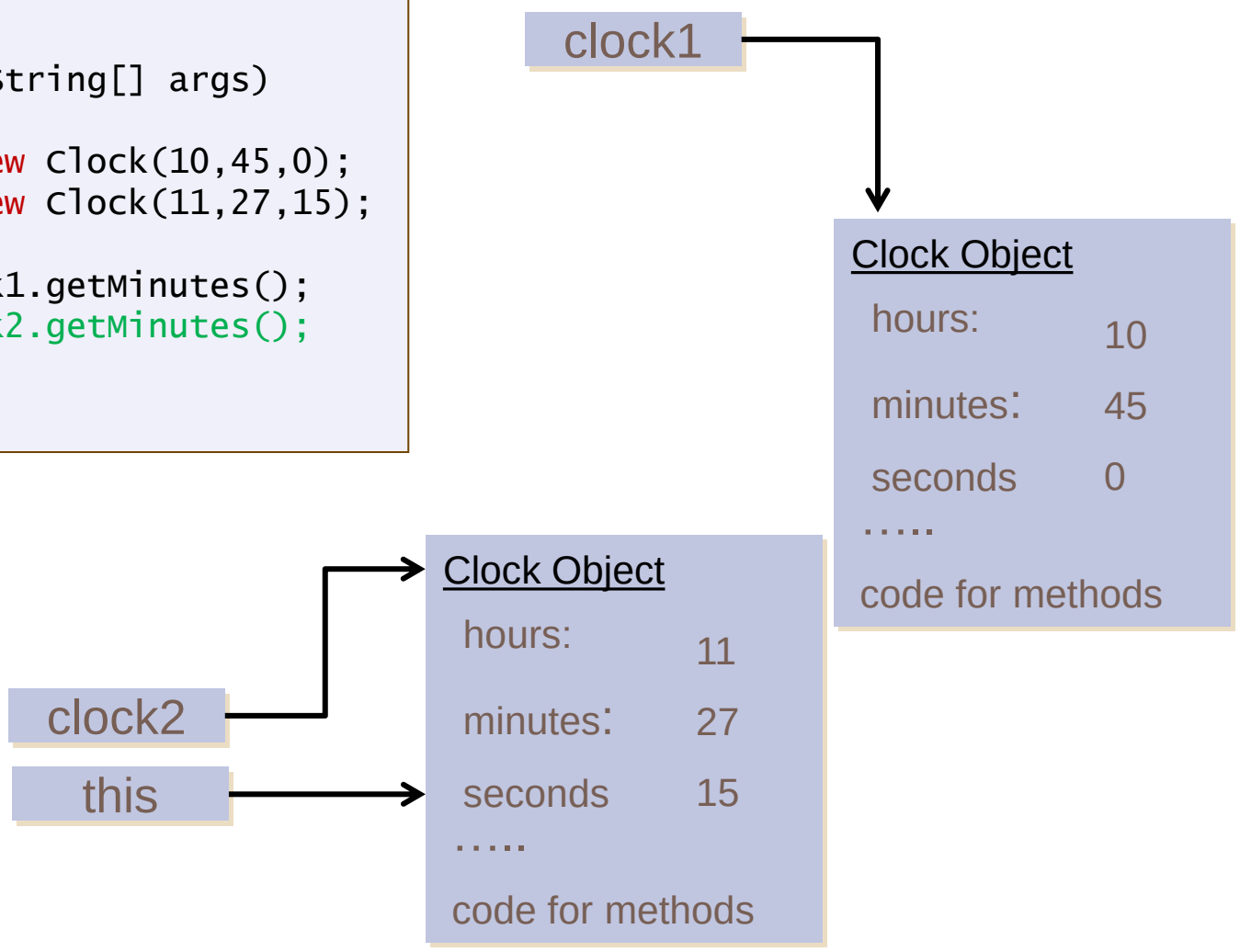
The `this` reference variable

```
class ClockTester
{
  static void main (String[] args)
  {
    Clock clock1 = new Clock(10,45,0);
    Clock clock2 = new Clock(11,27,15);

    int mins1 = clock1.getMinutes();
    int mins2 = clock2.getMinutes();
  }
}
```



The `this` reference variable

```
class ClockTester
{
  static void main (String[] args)
  {
    Clock clock1 = new Clock(10,45,0);
    Clock clock2 = new Clock(11,27,15);

    int mins1 = clock1.getMinutes();
    int mins2 = clock2.getMinutes();
  }
}
```



Using the this reference in a Constructor

- Sometimes people use the this reference in constructors
  - It makes it very clear that you are setting the instance variable:

```
public class Student
{
   private int id;
   private String name;

   public  Student(int id, String name)
   {
      this.id = id;
      this.name = name;
   }
}
```

Using the `this` reference in a Constructor

- **Beware**: code below does **not** set the instance variables !!!
- It is setting the parameter variable to itself

```
public class Student
{
  private int id;
  private String name;
  public Student(int i, String n)
  {
    id = i;
    name = n;
  }
}
```

- It is usually bad practice to use the same name for an instance variable and a local variable or parameter

Example: A Bank Account Class

- Class BankAccount has the following interface:

```
public BankAccount(long accountNumber)
public void deposit(double amount)
public void withdraw(double amount)
public double getBalance()
public void transfer(double amount,BankAccount targetAccount)
```

Bank Account - example

```
public class BankAccount
{
  private long accountNumber;
  private double balance;

  public BankAccount(long accountNumber)
  {
    this.accountNumber = accountNumber;
    balance = 0;
  }

  public double getBalance()
  {
    return balance;
  }
  // continued in the next slide...
```

Bank Account - example

```
   public void deposit(double amount)
   {
      balance += amount;
   }

   public void withdraw(double amount)
   {
      balance -= amount;
   }

   public void transfer(double amount, BankAccount targetAccount)
   {
      this.withdraw(amount);
      targetAccount.deposit(amount);
   }
}
```

Bank Account – usage example

```
class BankAccountTester
{
  public static void main( String[] args)
  {
    BankAccount aliceAcc = new BankAccount(1398723);
    BankAccount bobAcc   = new BankAccount(1978394);

    aliceAcc.deposit(900);
    aliceAcc.transfer(700,bobAcc);
    // Alice's balance = 200 ; Bob's balance = 700
```

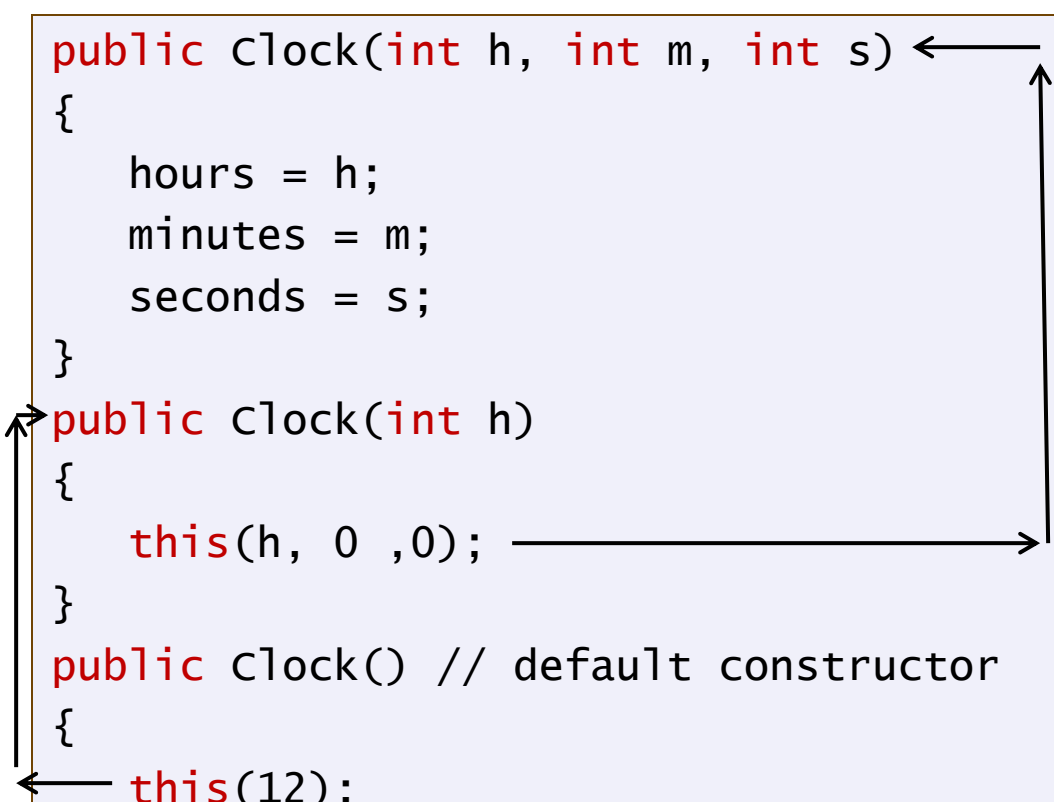
Constructor and Method Overloading

- A class can define several constructor methods** -- several ways to initialize the instance variables
  - constructors differ by the *number* and/or *type* of parameters
- When we construct an object, the compiler decides which constructor to call according to the number and types of the actual arguments passed in
  - A constructor with no parameters is called the *default constructor*
- Similarly, different methods can also use the same name as long as they differ in the number and/or type of parameters.
- When a method is called, the compiler decides which method to call according to the number and types of the actual arguments passed in

a class can also have ***no* constructor methods!

Constructor Overloading example: Three Constructor Methods

```
public Clock(int h, int m, int s)                    1
{
   hours = h;
   minutes = m;
   seconds = s;
}
public Clock(int h)                                  2
{
   this(h, 0 ,0);
}
public Clock() // default constructor                3
{
   this(12);
}
```



Visibility Modifiers

- We accomplish encapsulation through the appropriate use of *visibility modifiers*
- Visibility modifiers specify which parts of an object can be "seen" or modified (i.e. which methods and variables)
- We use the modifier `final` to define a constant
- Java has three visibility modifiers: public, private, and protected
- We will discuss the protected modifier later in the course

Visibility Modifiers - Classes

- A **class** can be defined either with the public modifier or without a visibility modifier.
- If a class is declared as public it can be directly used by methods in any other class
- If a class is declared without a visibility modifier it has a *default visibility*. This sets a limit to which methods in other classes can use this class (classes in the same package). We will discuss default visibility later in the course.
- Classes that define a new type of object that is to be used by methods in other classes should be declared public.

Visibility Modifiers - Members

- A member is a variable, a method or a constructor of the class.
- Members of a class can be declared as **private, protected, public** or without a visibility modifier:

```
private int hours;
int hours;
public int hours;
```

- Members that are declared without a visibility modifier are said to have *default visibility*. We will discuss default and protected visibility later in the course.

Public Visibility

- We *expose* methods that are part of the interface of the class by declaring them as public

- Members that are declared as public can be accessed from any class
  - Example: the methods getHours(), secondHasElapsed() and setTime() are part of the interface of class Clock so we define them as public.

- We typically do not want to reveal the internal representation of the object's data. So we **usually do not** declare its state variables as public (*encapsulation*)

Private Visibility

- A variable or method that is declared as private, can be accessed only by code that is *within* (inside) the class
- We *hide* the internal implementation of the class by declaring its state variables and auxiliary methods as private.
- Data hiding is essential for encapsulation.

Illegal Access - example

```
// Example of illegal access
class BankAccountTest {
  public static void main(String[] args) {
    BankAccount victim = new BankAccount(2398742);
    victim.balance = victim.balance - 500;
    // this will not compile!
  }
}
```

```
public class BankAccount {
  private long accountNumber;
  private double balance;
  // …
```

Accessing Internal State of Objects of Same Class

- Sometimes object instances of the same class need to access each other's "guts" (e.g., for state copying - if we want to create an identical instance of an object we have)

- Example:
  - from within a `BankAccount` object, any private member of a different `BankAccount` object can be accessed.

Encapsulation - example

```
public void transfer(double amount, BankAccount targetAccount)
{
  withdraw(amount);
  targetAccount.deposit(amount);
}

// alternative version (valid, but not so nice – don't do it!)
public void transfer(double amount, BankAccount targetAccount)
{
  balance -= amount;
  targetAccount.balance += amount;
}
```

toString() method

Static and Auxiliary Methods

The Static Modifier

- The `static` modifier can be applied to **variables** or **methods**
- It associates a variable or method with the ***class*** rather than with an ***object***
- *Methods* that are declared static do not act upon any particular object. They just define a task or algorithm.
- As we have learned, we can write a class that is a collection of static methods. Such a class isn't meant to define new type of objects. It is just used as a library of methods that are related in some way.

Example - a `Math` Class

```
/**
 * A library of mathematical methods.
 */
public class Math {
  /**
   * Computes the trigonometric sine of an angle.
   */
  public static double sin(double x) { … }

  /**
   * Computes the logarithm of a given number.
   */
  public static double log(double x) { … }
  // ...
```

Static Variables

- A variable that is declared static is associated with the class itself and not with an instance of it.
- Static variables are also called ***class variables***.
- We use static variables to store information that is not associated with a given object, but is relevant to the class.
- We have already seen such usage - constants of a class (final static)

Static Variables - Example

```
public class BankAccount
{
  private long accountNumber;
  private double balance;
  private static int numberOfAccounts = 0;

  public BankAccount()
  {
    this.accountNumber = ++numberOfAccounts;
    this.balance = 0;
  }

  public static int getNumberOfAccounts
  {
    return numberOfAccounts;
  }
}
```

Static Variables - Example

```
Class BankAccountTester
{
  public static void main(String[] args)
  {
    BankAccount aliceAcc = new BankAccount();
    BankAccount bobAcc   = new BankAccount();
    BankAccount tedAcc   = new BankAccount();

    System.out.println(BankAccount.getNumberOfAccounts());
  }
}
```

NOTE: we call the static method `getNumberOfAccounts` using the **class** name!!

NOTE: There is one copy of a static variable that is shared among all objects of the Class!!!!

Static Methods and Instance Variables

- Static methods:
  - *cannot* reference (read/set) instance variables
  - *can* only reference static variables or local variables (within the method)

- `this` keyword has no meaning inside a static method, thus its use inside a static method is not allowed.

- Instance methods, however, can access static variables.

Auxiliary Methods

- Complicated well-defined pieces of code that define a task or that are repeated often in different methods should be put into an auxiliary (i.e. helper) method
- Only if you can clearly define the task (well defined input and output) should you make it a helper method.
- This results in more readable and manageable code
- This is very helpful when implementing algorithms, or when we need “helper” methods in classes.

Division into methods – interest calculation

```
public class BankAccount {

  final static int DAYS_IN_YEAR = 365;

  public void earnDailyInterest(int days){
    double rate = 1 + interestRate()/DAYS_IN_YEAR;
    for(int j=0 ; j<days ; j++)
      balance *= rate;
  }

  private double interestRate() {
     return (balance>100000) ? 0.06 : 0.05;
  }
```

Division into methods – interest calculation

```
public class BankAccount {

  final static int DAYS_IN_YEAR = 365;

  public void earnDailyInterest(int days){
    double rate = 1 + interestRate()/DAYS_IN_YEAR;
    for(int j=0 ; j<days ; j++)
      balance *= rate;
  }

  private double interestRate() {
     return (balance>100000) ? 0.06 : 0.05;
  }
```

Division into Methods – Printing primes

```
public class PrintPrimes {
  public static void main(String[] args) {
    for(j = 2 ; j < 1000 ; j++){
      if (isPrime(j))
        System.out.println(j);
    }
  }
  private static boolean isPrime(int n){
    for(int j=2 ; j < n ; j++)
      if (n%j == 0)
        return false;
    return true;
  }
}
```