Object-Oriented Programming

Creating Java Classes

Computer Science and Technology
United International College

Review

What we have studied:

Java Programming Essentials

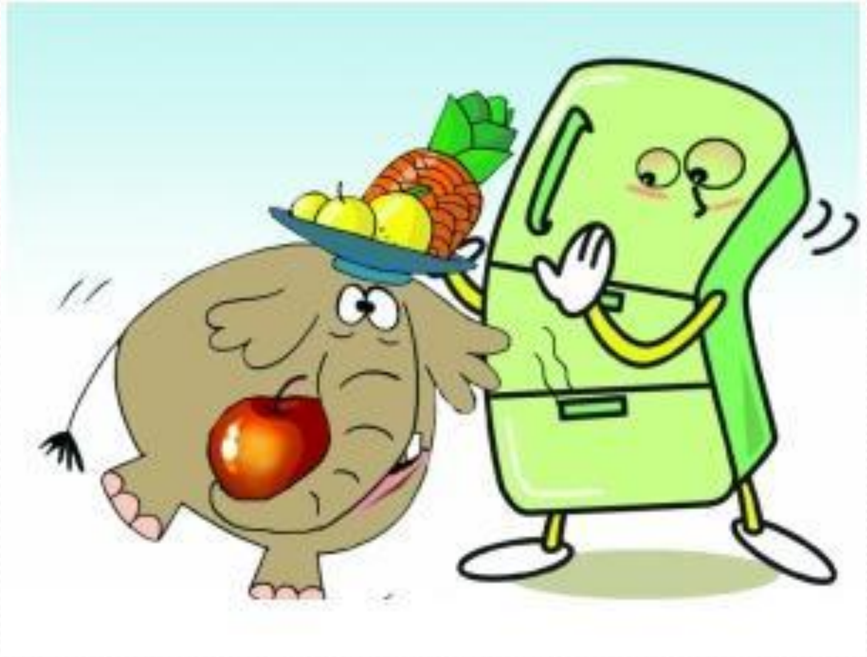
- **Variable Declarations and Initialization;**
- **Data Type and Type casting;**
- **Conditional Statements** (if-then-else, switch case);
- **Loops** (for, while, do-while).

Outline

- **Structured programming vs. OOP**
- **Objects, Classes, Methods**
- **Creating Classes**
- **Instance Variables vs Local Variables**
- **Memory analysis**

Structured Programming vs. OOP

- How to put an elephant into a refrigerator?
  - In the way of structured programming:



Step1: Open the refrigerator

Step2:…

Step3:…

…

Structured Programming vs. OOP

- How to put an elephant into a refrigerator?
  - In the way of OOP:

Don't push me, I can walk!

Come here, I am waiting for you!

Class – Object - Program

- Classes are the most important language feature that make **object-oriented programming** (***OOP***) possible.
- Programming in Java consists of defining a number of classes and creating (instantiating) **objects** of each **class type**.
- A Java program carries out its computations by object-to-object communication through **method calls**.

Objects

- Objects have **state** and **behavior**.

  Example: **Dog**

  - **State**: Color, Name, Breed.
  - **Behaviors**: Fetch stick, Drink water, Wag tail, Bark.

- Software objects also have state and behaviors.
  - State is stored in **instance variables.**
  - Behaviors are accomplished by **methods.**

Benefits of OOP Approach

- **Modularity**: The source code for an object can be written and maintained independently of the source code for other objects.
- **Information-hiding**: By interacting only with an object’s methods, the details of its internal implementation are hidden from the outside world.
- **Code re-use**: If an object already exists you can use that object in your program. This allows specialists to implement/test/debug complex, task-specific objects, which you can then trust to run in your own code.

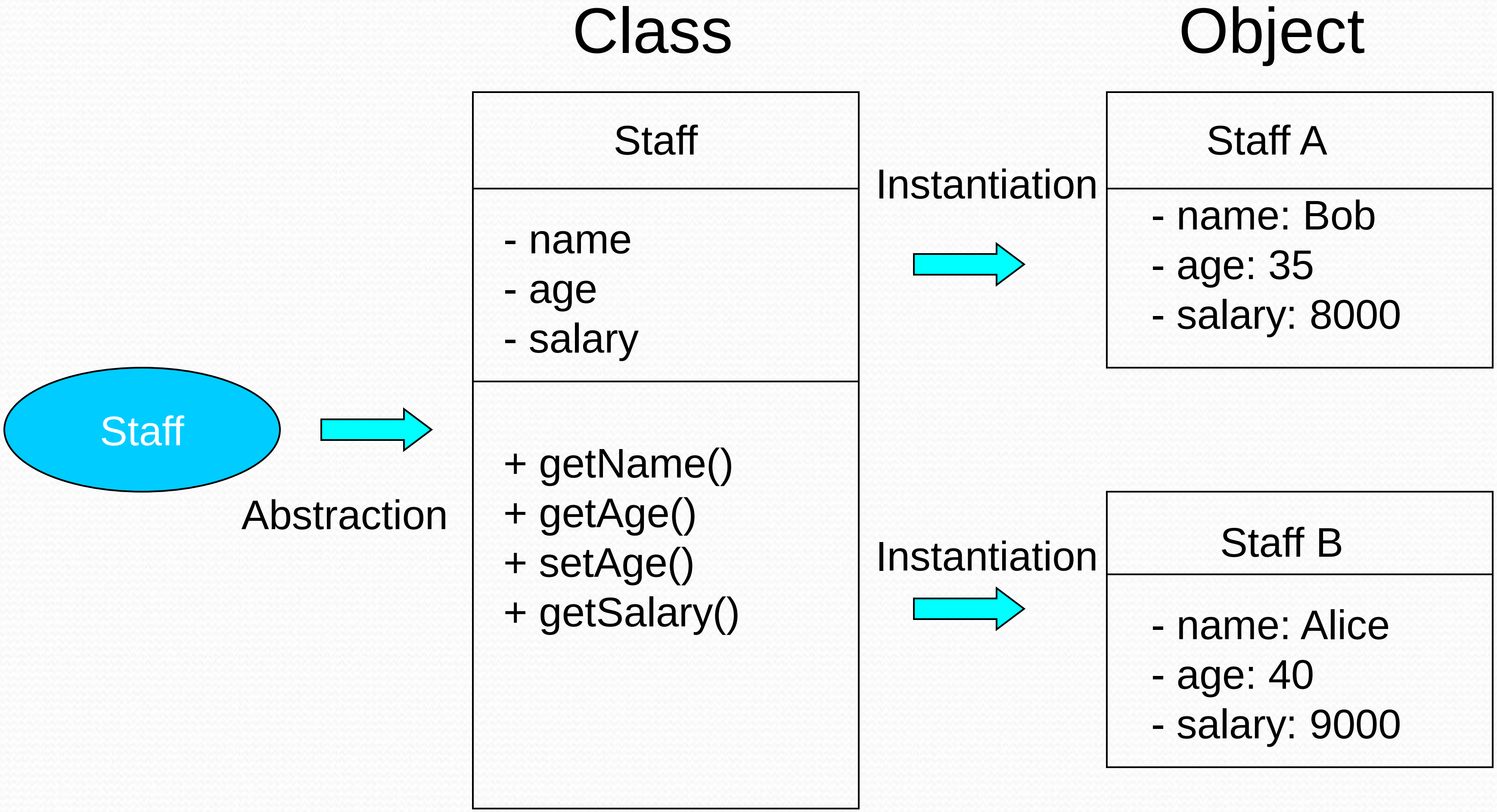
From Java Tutorial -- https://docs.oracle.com/javase/tutorial/java/concepts/object.html

Classes vs. Objects

- A class is a programmer-defined **type.**
  - It is a blueprint or a template for later creating different objects.
  - A class defines instance variables and methods to describe the properties and behaviors that the objects later will have.

- A value of a class type is called an **object** or an **instance** of the class: an object exhibits the properties and behaviors defined by the object's class.

Example



Class

Object

Staff

Abstraction

Staff

- name
- age
- salary

+ getName()
+ getAge()
+ setAge()
+ getSalary()

Instantiation

Staff A

- name: Bob
- age: 35
- salary: 8000

Instantiation

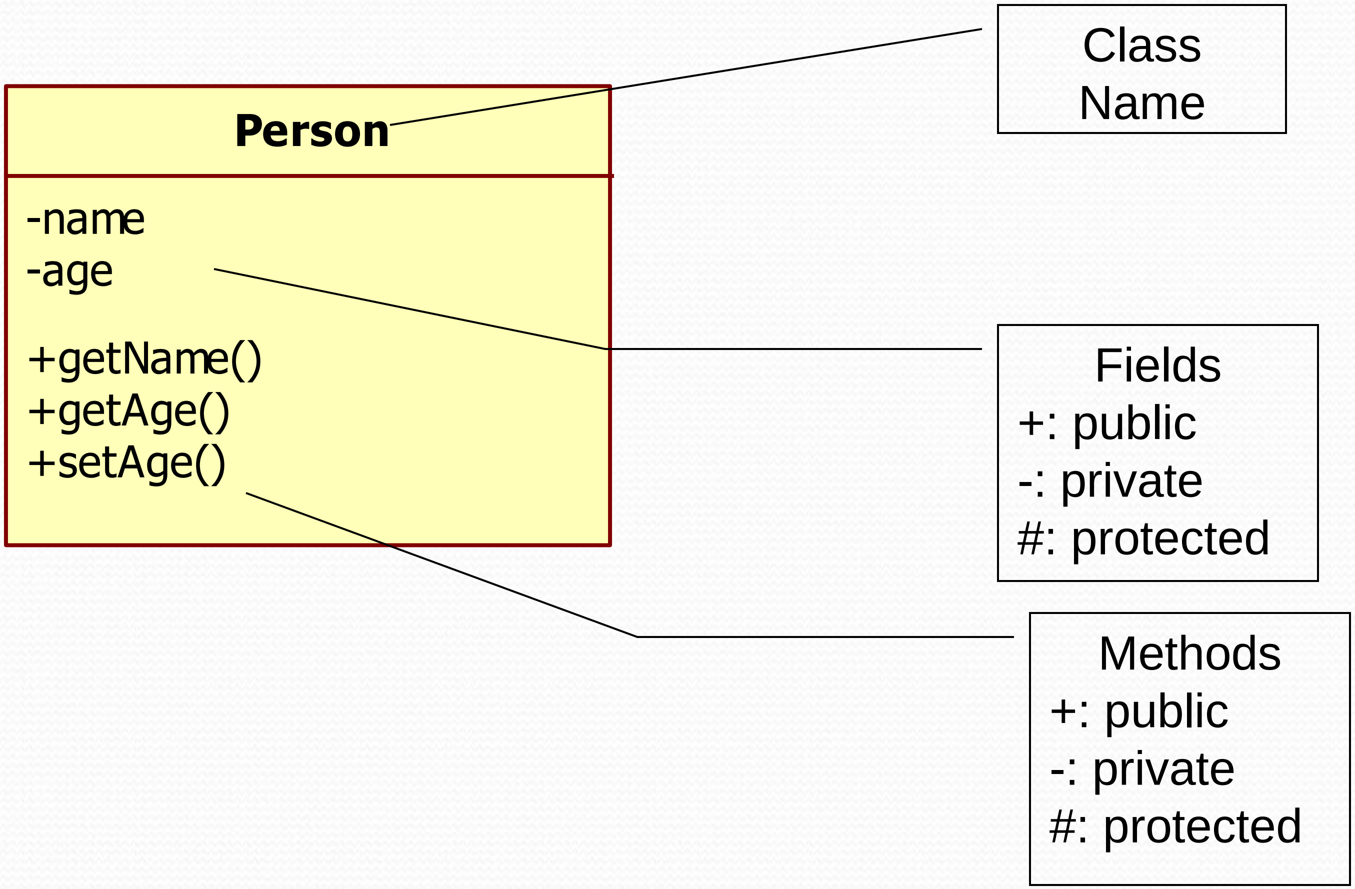
Staff B

- name: Alice
- age: 40
- salary: 9000

Class Definition

- A class definition specifies the data items (state) and methods that all of its objects will have:
  - Data items and methods are sometimes called **members** of the object.
  - Data items are usually called **fields / instance variables / members/ attributes.**
- Instance variable declarations and method definitions can be placed in any order within the class definition, though instance variables are usually written first (just like in UML diagrams).

Class Definition in UML



Person

-name
-age

+getName()
+getAge()
+setAge()

Class
Name

Fields
+: public
-: private
#: protected

Methods
+: public
-: private
#: protected

The **new** Operator

The **new** operator is used to create an object from a particular class:

```
    ClassName classVar = new ClassName();
```

Example: `Person person1 = new Person();`

Note: **person1** is a variable of type **Person** that points to an object which is created from the class **Person** using the **new** operator.

Instance Variables

- Instance variables can be defined as follows:
  - [<modifiers>] type <fields_name> [= defaultValue]
  - E.g.: the `private` modifier (discussed later):

```
private String name = "bob";
private int age;
```

- In order to refer to a particular instance variable, preface it with its object name as follows:

```
person1.name // "." is the period operator
person2.age
```

Instance Variables

- Instance variables
  - instance variables store the state of the object.
  - Each object has its own copy of the variables.
  - Every object has a state that is determined by the values stored in the instance variables of the object.
- Initialization
  - Instance variables which are not initialized explicitly, will be assigned a default value.
  - In practice just always initialize all instance variables yourself in your code.

Instance variables	**Default value**
byte	0
short	0
int	0
long	0L
char	'\u0000'
float	0.0F
double	0.0D
boolean	false
other reference	null

Methods

Method definitions are divided into two parts: a **heading** and a **method body**:

- Declaration:

```
[<modifiers>] <return_type> <name>([argument_list]) {
    [<statement>]
}
```

- E.g.:

```
public void myMethod()            ←          Heading
{
      code  to perform some action        }   Body
      and/or compute a value
}
```

Methods

- Methods are invoked using the name of the object which has the method and the method name as follows:

  ```
  person1.getName();
  ```

- Invoking a method is equivalent to executing the method body.
- Example: `person1.setAge(25);`

"Parameter" and "Argument"

- The **parameters** of a method are defined in a parameter list in the method definition.
- When a method is invoked, the appropriate values must be passed to the method in the form of **arguments.**
- Do not be surprised to find that people often use the terms parameter and argument interchangeably.

Putting it all Together: Person Class Demo

```
public class Person {
  private String name = "Alice";
  private int age = 20;
  public String getName() { return name; }
  public int getAge() { return age; }
  public void setAge(int age) { this.age = age; }
}

public class PersonDemo {
  public static void main(String[] args) {
    Person person1 = new Person();
    System.out.println(person1.getName());
    System.out.println(person1.getAge());
  }
}
```

Putting it all Together: Person Class Demo

- Put each class in a java file with the same name.
  - Example: the `Person` class goes into the `Person.java` file.
- Usually only one class has a `main` method, which is where the program starts executing.
- We create a `Person` object by using the `new` operator.
- We use `this.age = age` in the `setAge()` method. Why?

The `this` Keyword

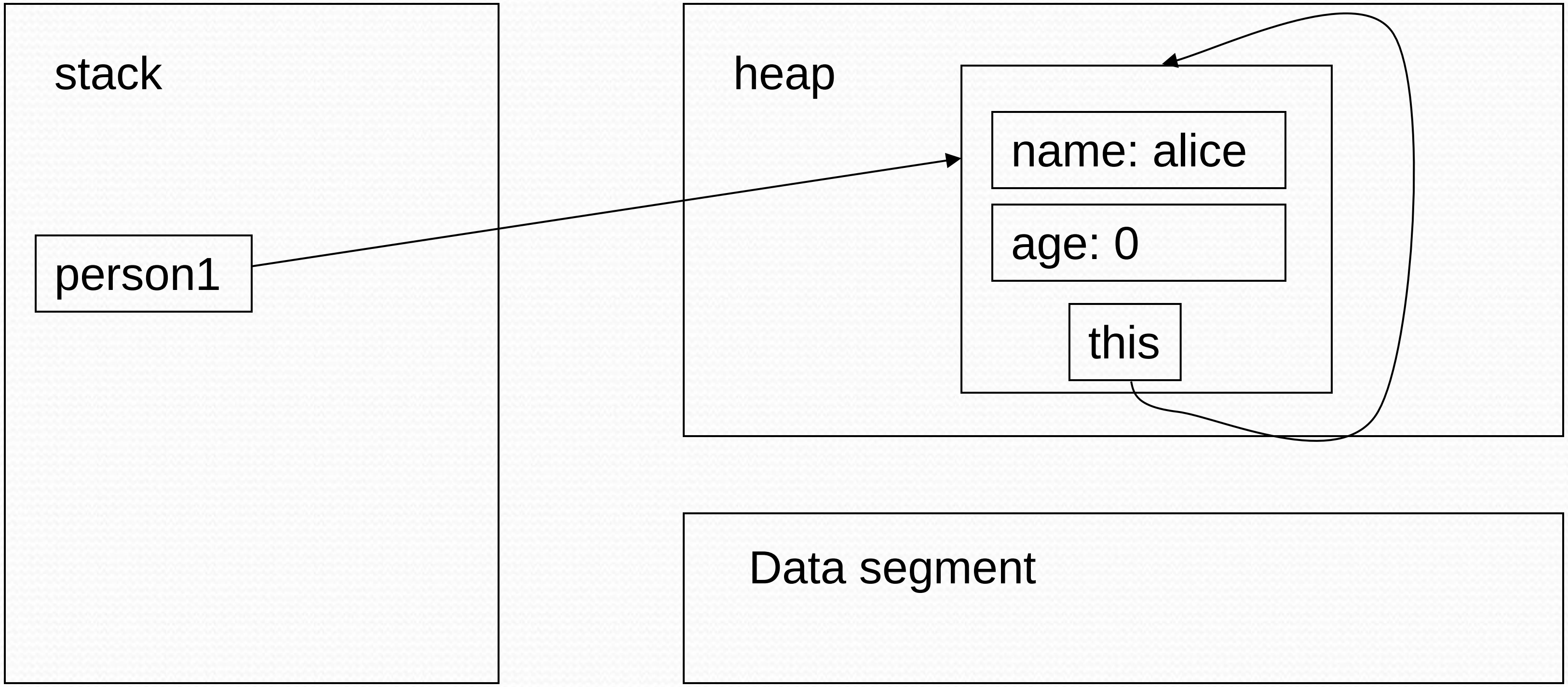
- All instance variables are understood to have **`<the calling object>.`** in front of them
- If an explicit name for the calling object is needed, the keyword **`this`** can be used (just like "I" can be used in English).
- **`myInstanceVariable`** always means, and is always interchangeable with, **`this.myInstanceVariable`**
- Example: **`age`** is an instance variable for **`Person`**

```
void setAge(int newValue) { age = newValue; }
```

Here **`age`** means the same as **`this.age`**

Memory Analysis

```
Person person1 = new Person();
```



The `this` Keyword

- Consider:

```
void setAge(int age) { this.age = age; }
```

- Why do we need to use `this.age`?
- Answer: to eliminate uncertainty over the variable age.
  - In `setAge(int age)`, `age` is a **parameter**, not the **instance variable** for the `Person` class. But, we want to set the `Person` instance variable `age` to be equal to the parameter `age`.
- If we tried `age = age`, this would not work. Why?

The `this` Keyword

- `this` ***must*** be used if a parameter or a local variable with the same name is used in a method.
- Otherwise, all occurrences of the variable name will be interpreted as local.

```
int someVariable = this.someVariable;
```

Local variable

Instance variable

Local Variables

- A variable declared within a method definition is called a **local variable.**
  - The `person1` variable declared in the `main` method of the `PersonDemo` class is a local variable.
- If two methods each have a local variable of the same name, they are still two entirely different variables.
- A local variable is only valid for calculation **inside** its method definition.

Global Variables

- Some programming languages include another kind of variable called a *global* variable.
- The Java language does **not** have global variables.

Constructors

- A **constructor** is a special kind of method that is designed to create a new object and initialize the instance variables of the new object:

```
public ClassName(Parameters){ code }
```

- A constructor **must** have the same name as the class.
- A constructor has **no return type**, not even `void`.
- Constructors are often *overloaded* (there is more than one constructor in the same class).

Constructors

- A constructor is called when an object of the class is created using **`new`**:

  ```
  ClassName objectName = new ClassName(Args);
  ```

- The name of the constructor and its parenthesized list of arguments (if any) **must** follow the **`new`** operator.
- This is the **only** valid way to invoke a constructor: a constructor cannot be invoked directly like an ordinary method.

Constructors

- If you do not write a constructor yourself in a class then the class automatically gets a default constructor with an empty list of parameters.

  Example: `Person` class:

  ```
  Person person1 = new Person();
  ```

- This is a correct construction, even though the Java code for the `Person` class does not have a visible constructor. The constructor is still there, added automatically for you by Java.

Constructors

We can add our own constructors to the `Person` class:

```
public Person() {} // Same as default constructor
public Person(String name) { this.name = name; }
public Person(String name, int value) {
    this.name = name;
    age = value; // Note the lack of "this"
}
```

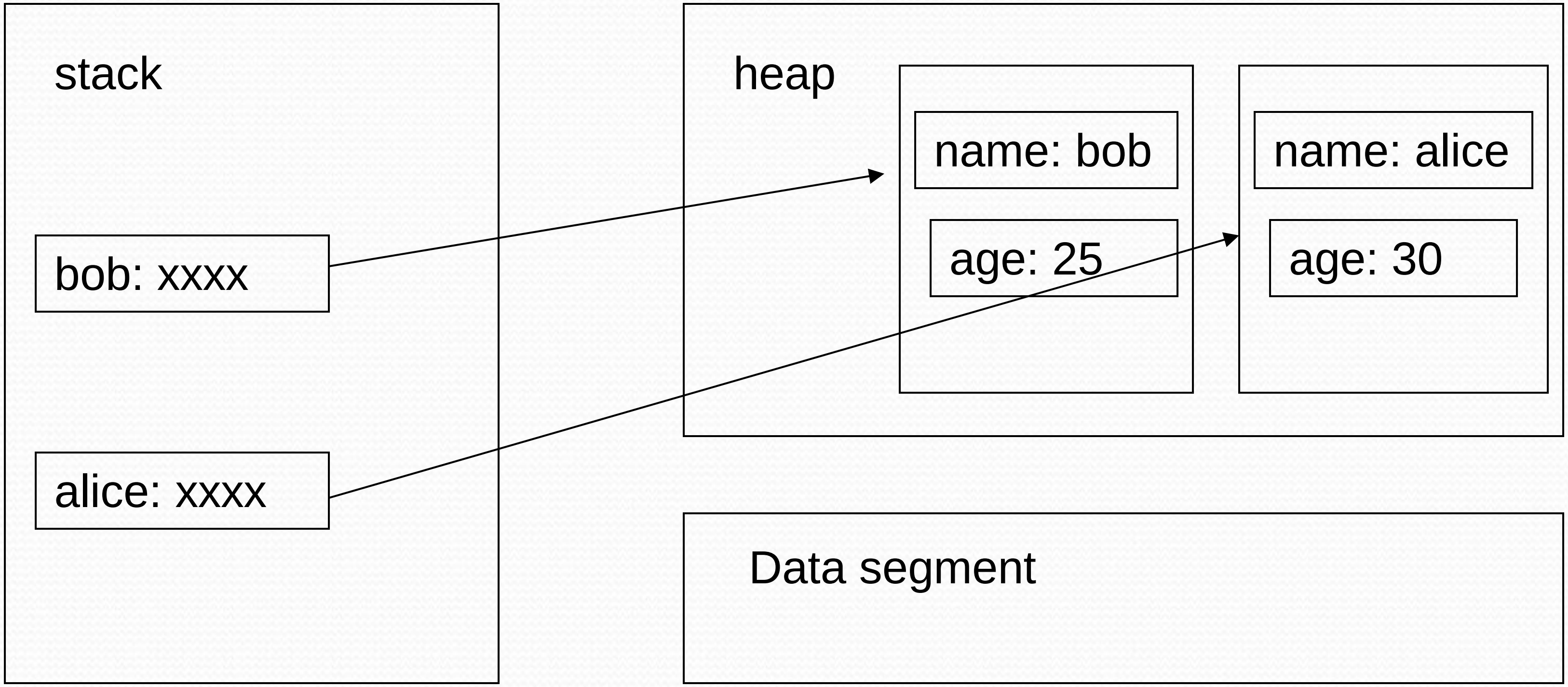
A Constructor Has a `this` Keyword

- Like any ordinary method, every constructor has a `this` keyword.
- Just like in a method, the `this` keyword can be used in a constructor to differentiate between an instance variable of the object being constructed and an argument of the constructor.
- The first action taken by a constructor is to automatically create an object with instance variables.
- Then within the definition of a constructor, the `this` keyword refers to the object being created by the constructor.

Memory Analysis

```
public class Person {
   private String name;
   private int age;
   public Person(String name, int value) { // constructor
      this.name = name;
      age = value;
   }
   public String getName() { return name; }
   public int getAge() { return age; }
   public void setAge(int age) { this.age = age; }
}
…
Person bob = new Person("bob", 25);
Person alice = new Person("alice", 30);
```

Memory Analysis



More on the Default Constructor

- If you do not include any constructor in your class, Java will automatically create a **default** or **no-argument** constructor that takes no argument, performs no initialization, but allows the object to be created.
- **IMPORTANT: If you include even one constructor in your class, Java will not provide the default constructor anymore.**
- **Best strategy:** provide **all** constructors in your code (including a no-argument constructor if you need one).

Naming Convention

- Classes always start with a capital letter.
  - `Person`
- Instance variables always start with a lower case letter.
  - `name`, `age`
- Methods start with lower case letter and Camel case. Methods should be verbs.
  - `setName`, `getName`

Example

```
public class Birthday {
    private int day;
    private int month;
    private int year;
    public Birthday(int d, int m, int y) {
        day = d;
        month = m;
        year = y;
    }
    public void setDay(int d) { day = d; }
    public void setMonth(int m) { month = m; }
    public void setYear(int y) { year = y; }
    public int getDay() { return day; }
    public int getMonth() { return month; }
    public int getYear() { return year; }
    public void display() {
        System.out.println(day + "-" + month + "-" + year);
    }
}
```

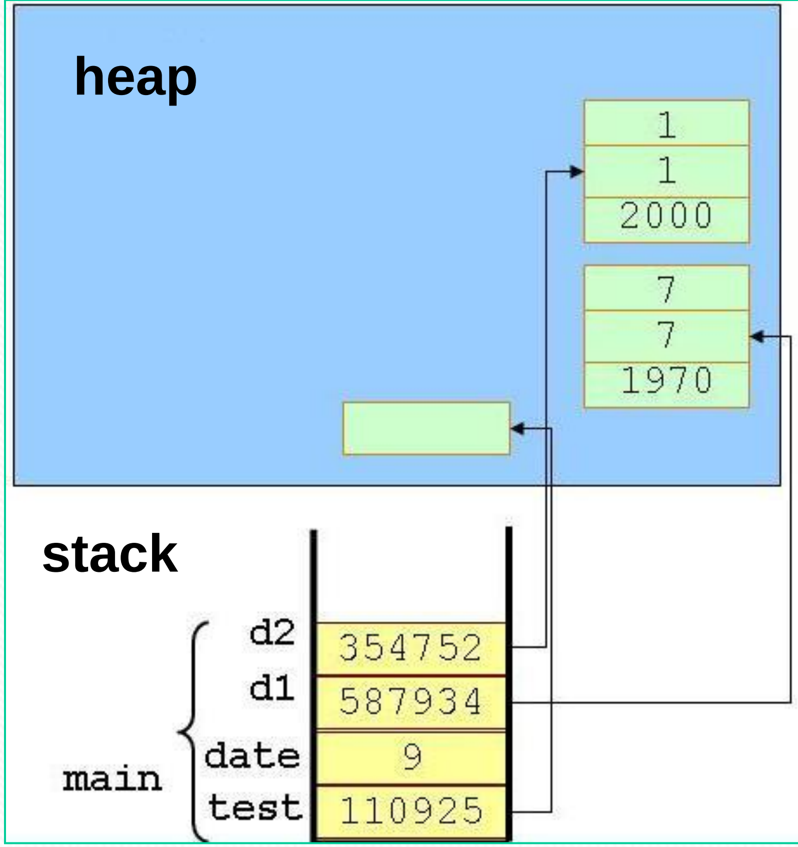
Birthday.java

Example

```
public class Test {
    public static void main(String[] args) {
        Test test = new Test();
        int date = 9;
        Birthday d1 = new Birthday(7, 7, 1977);
        Birthday d2 = new Birthday(1, 1, 2000);
        test.change1(date);
        test.change2(d1);
        test.change3(d2);
        System.out.println("date = "+ date);
        d1.display();
        d2.display();
    }
    public void change1(int i) { i = 1234; }
    public void change2(Birthday b) {
        b = new Birthday(22, 2, 2004);
    }
    public void change3(Birthday b) {
        b.setDay(22);
    }
}
```

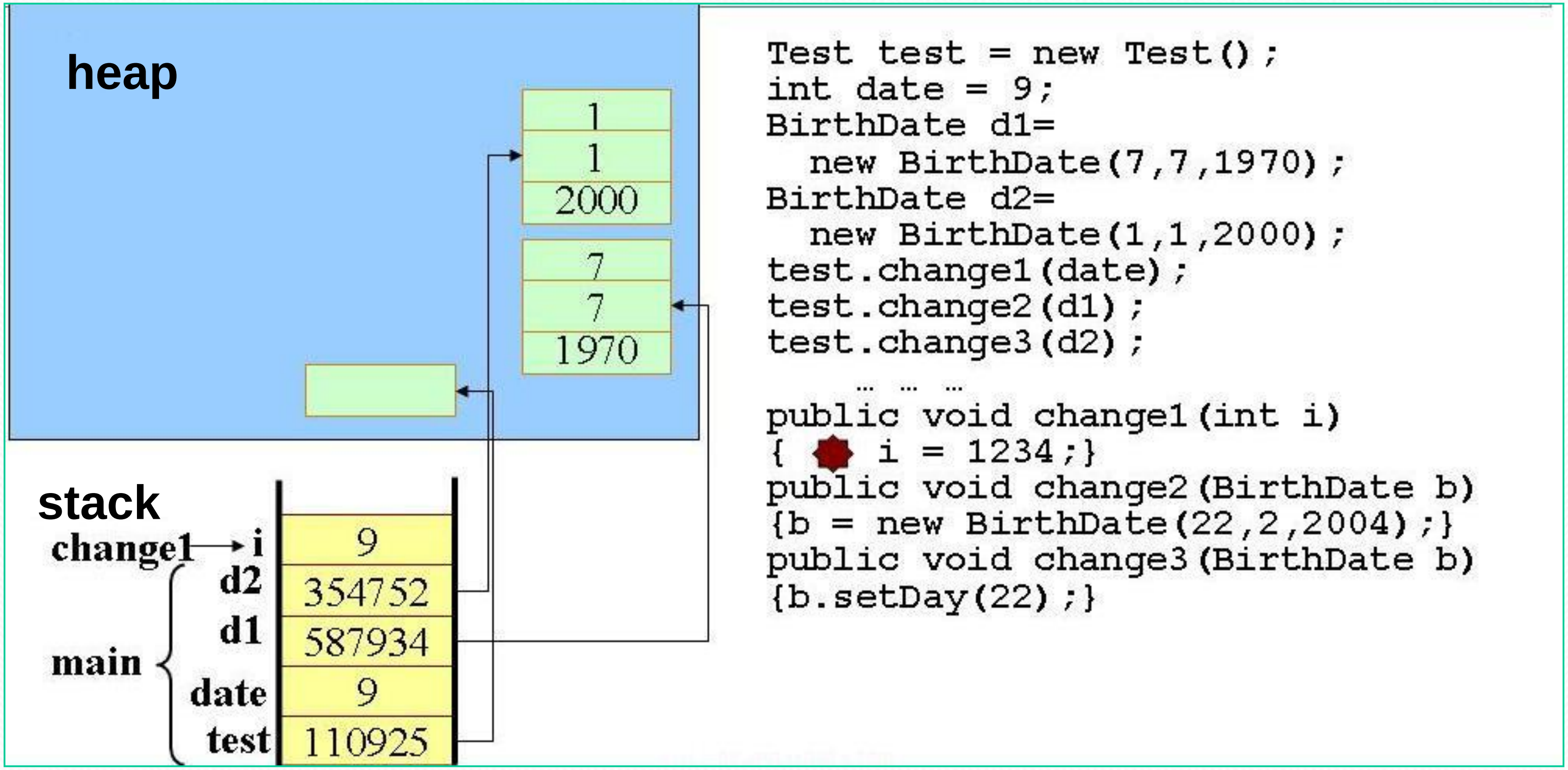
Test.java

Memory Analysis (1)



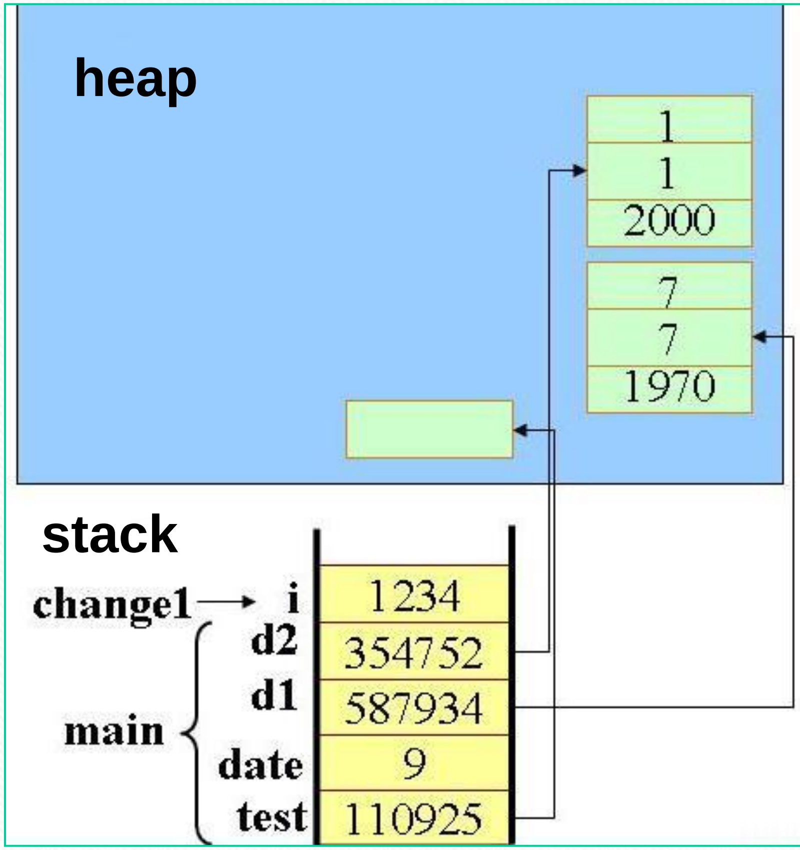
```
Test test = new Test();
int date = 9;
BirthDate d1=
  new BirthDate(7,7,1970);
BirthDate d2=
  new BirthDate(1,1,2000);
test.change1(date);
test.change2(d1);
test.change3(d2);
      … … …
public void change1(int i)
{i = 1234;}
public void change2(BirthDate b)
{b = new BirthDate(22,2,2004);}
public void change3(BirthDate b)
{b.setDay(22);}
```

Memory Analysis (2)



```
Test test = new Test();
int date = 9;
BirthDate d1=
  new BirthDate(7,7,1970);
BirthDate d2=
  new BirthDate(1,1,2000);
test.change1(date);
test.change2(d1);
test.change3(d2);
      … … …
public void change1(int i)
{  ●  i = 1234;}
public void change2(BirthDate b)
{b = new BirthDate(22,2,2004);}
public void change3(BirthDate b)
{b.setDay(22);}
```

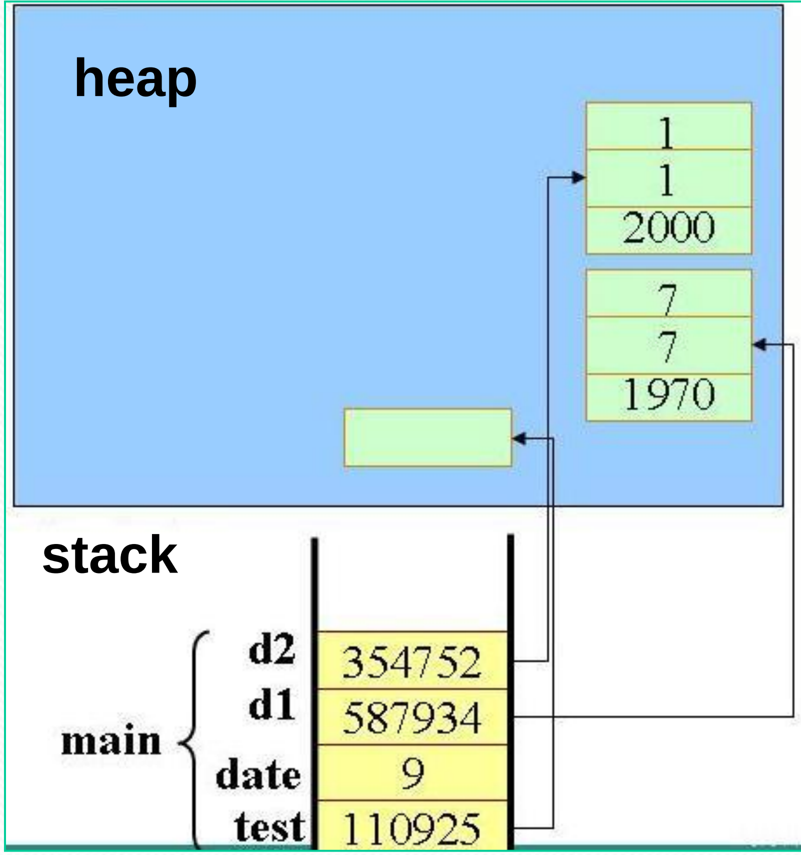
Memory Analysis (3)



```
Test test = new Test();
int date = 9;
BirthDate d1=
  new BirthDate(7,7,1970);
BirthDate d2=
  new BirthDate(1,1,2000);
test.change1(date);
test.change2(d1);
test.change3(d2);
      … … …
public void change1(int i)
{i = 1234;     }
public void change2(BirthDate b)
{b = new BirthDate(22,2,2004);}
public void change3(BirthDate b)
{b.setDay(22);}
```

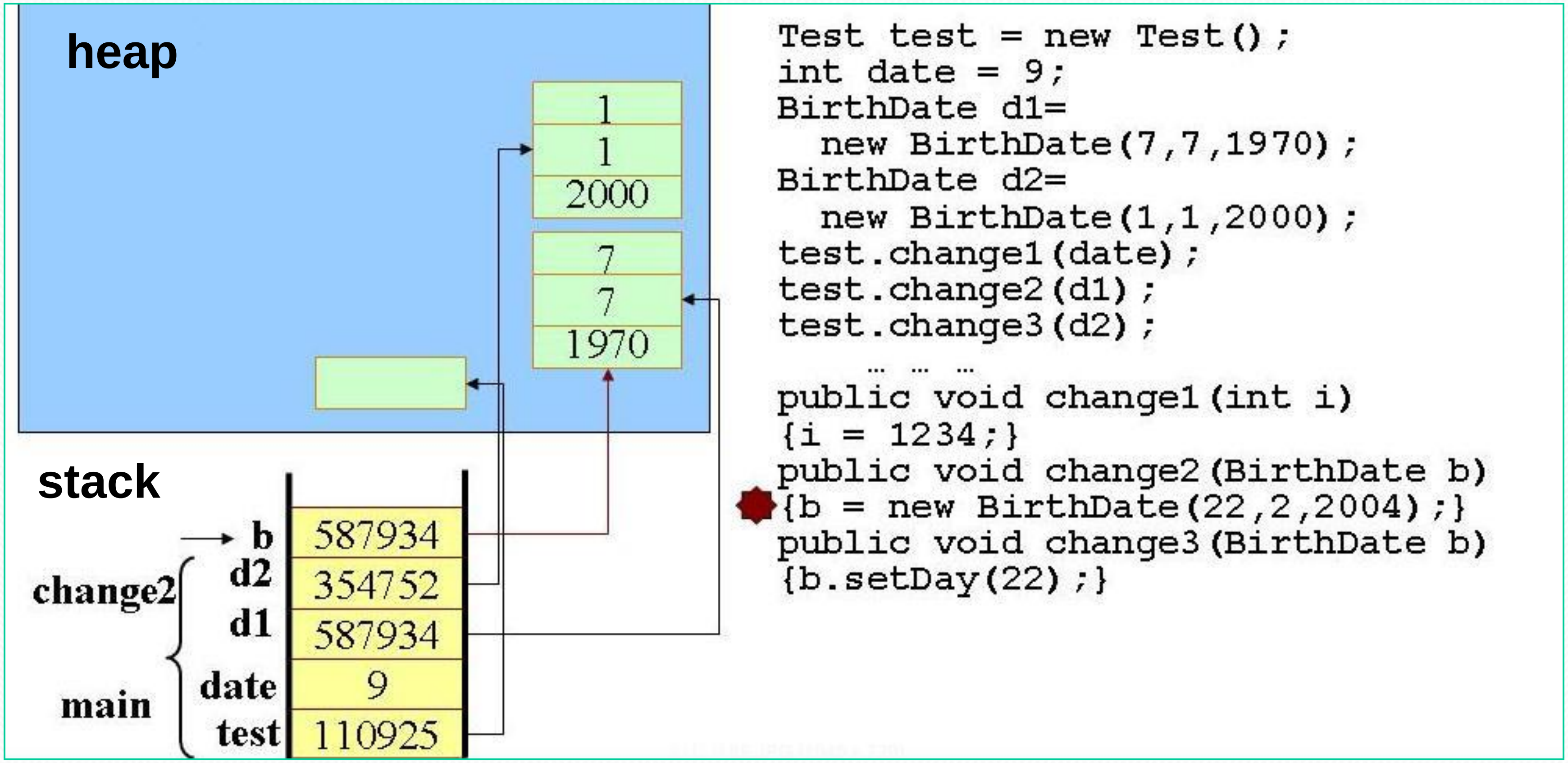
Valid period for a local variable!

Memory Analysis (4)



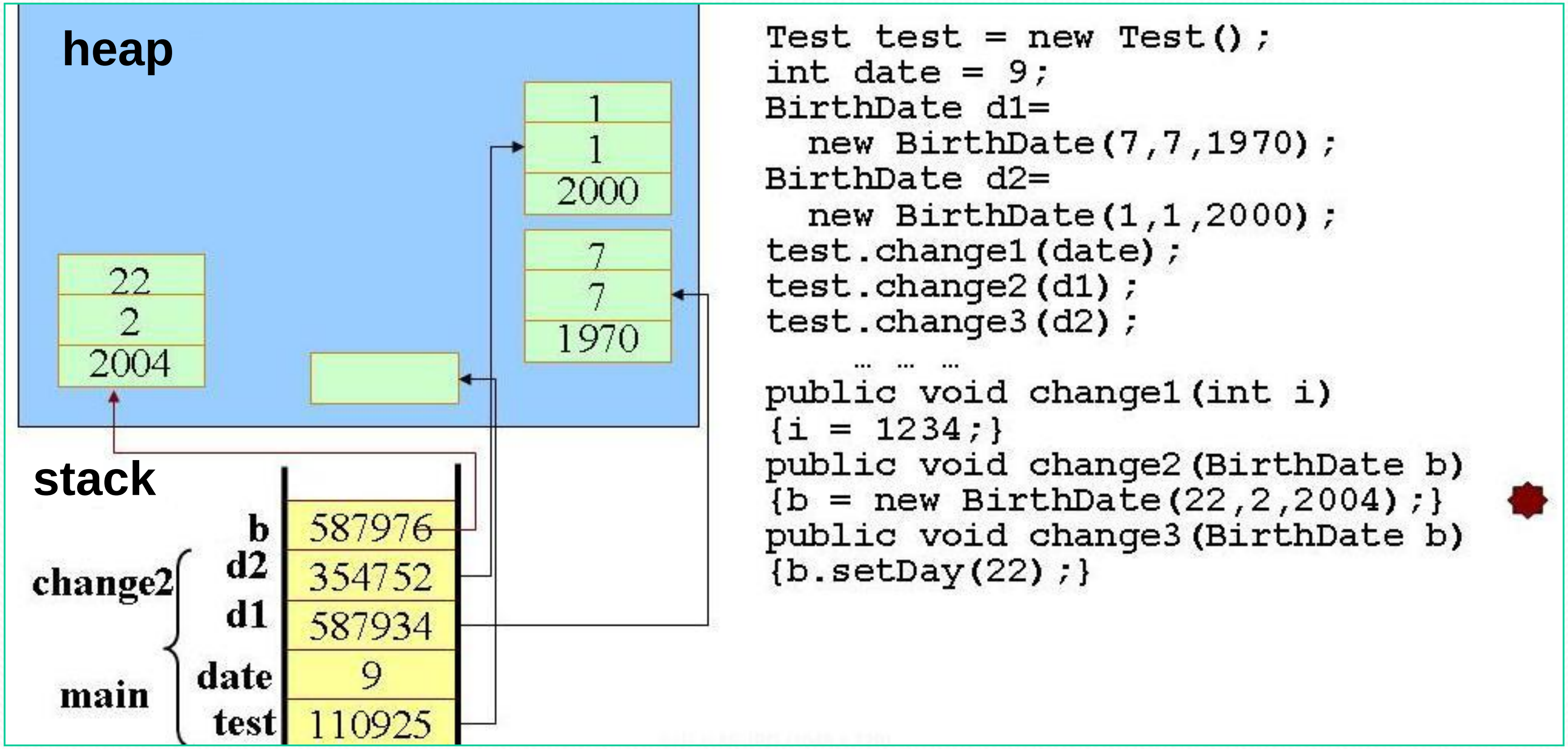
```
Test test = new Test();
int date = 9;
BirthDate d1=
   new BirthDate(7,7,1970);
BirthDate d2=
   new BirthDate(1,1,2000);
test.change1(date);
test.change2(d1);
test.change3(d2);
      … … …
public void change1(int i)
{i = 1234;}
public void change2(BirthDate b)
{b = new BirthDate(22,2,2004);}
public void change3(BirthDate b)
{b.setDay(22);}
```

Memory Analysis (5)



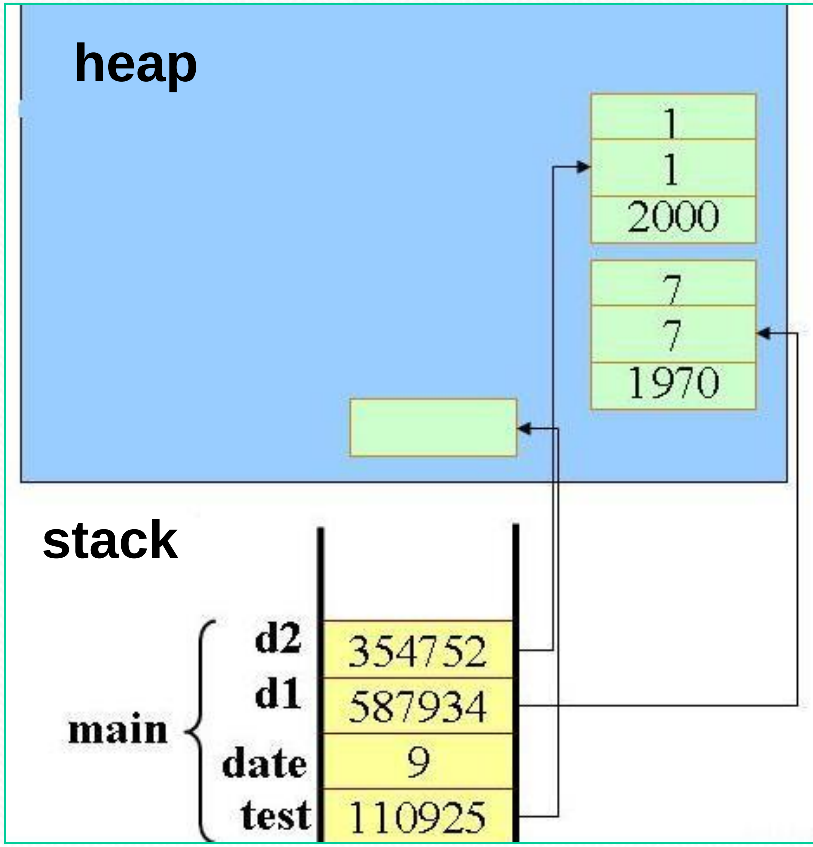
```
Test test = new Test();
int date = 9;
BirthDate d1=
  new BirthDate(7,7,1970);
BirthDate d2=
  new BirthDate(1,1,2000);
test.change1(date);
test.change2(d1);
test.change3(d2);
    … … …
public void change1(int i)
{i = 1234;}
public void change2(BirthDate b)
{b = new BirthDate(22,2,2004);}
public void change3(BirthDate b)
{b.setDay(22);}
```

Memory Analysis (6)



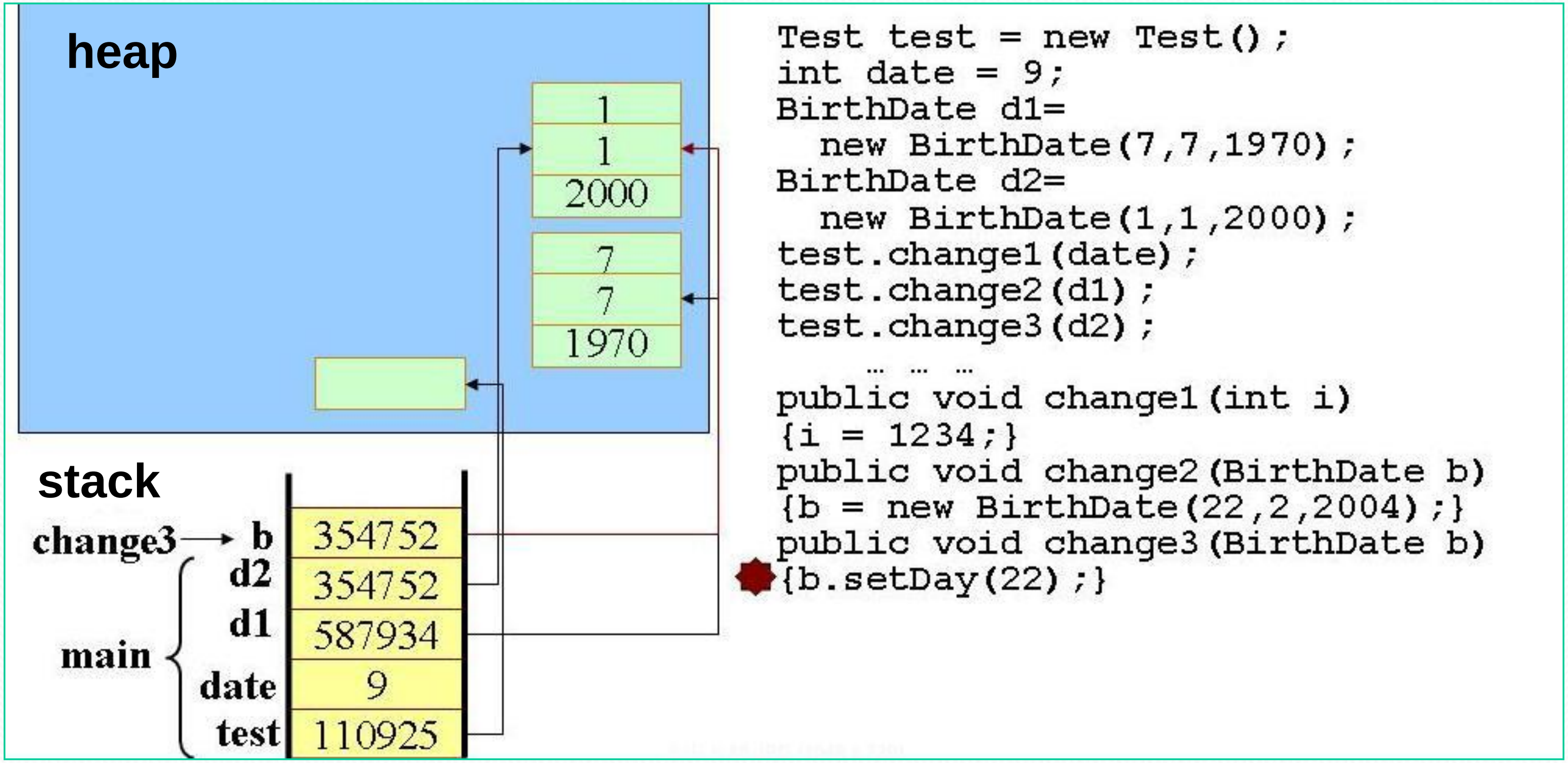
```
Test test = new Test();
int date = 9;
BirthDate d1=
  new BirthDate(7,7,1970);
BirthDate d2=
  new BirthDate(1,1,2000);
test.change1(date);
test.change2(d1);
test.change3(d2);
      … … …
public void change1(int i)
{i = 1234;}
public void change2(BirthDate b)
{b = new BirthDate(22,2,2004);}
public void change3(BirthDate b)
{b.setDay(22);}
```

Memory Analysis (7)



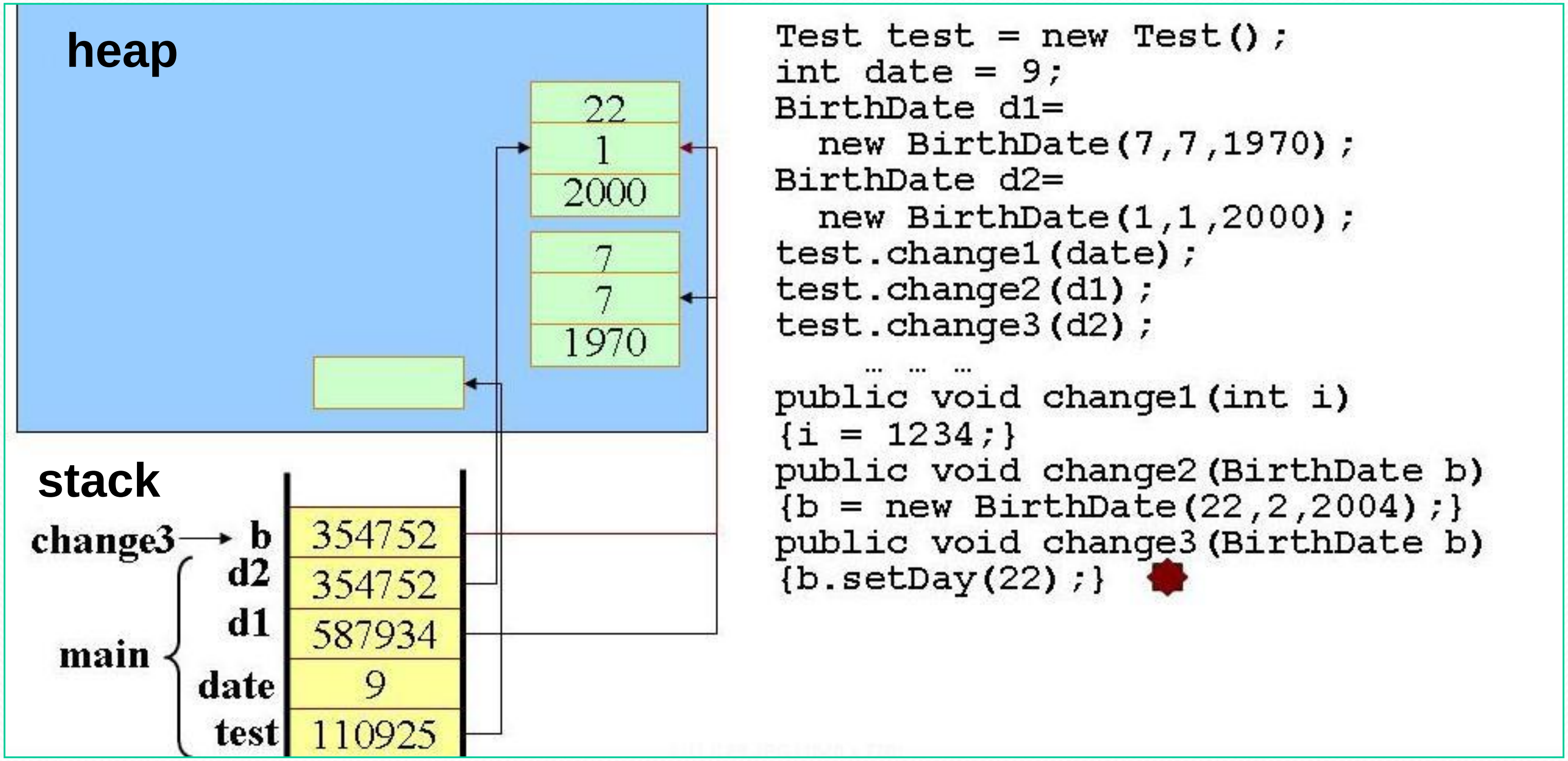
```
Test test = new Test();
int date = 9;
BirthDate d1=
  new BirthDate(7,7,1970);
BirthDate d2=
  new BirthDate(1,1,2000);
test.change1(date);
test.change2(d1);
test.change3(d2);
    … … …
public void change1(int i)
{i = 1234;}
public void change2(BirthDate b)
{b = new BirthDate(22,2,2004);}
public void change3(BirthDate b)
{b.setDay(22);}
```

Memory Analysis (8)



```
Test test = new Test();
int date = 9;
BirthDate d1=
  new BirthDate(7,7,1970);
BirthDate d2=
  new BirthDate(1,1,2000);
test.change1(date);
test.change2(d1);
test.change3(d2);
      … … …
public void change1(int i)
{i = 1234;}
public void change2(BirthDate b)
{b = new BirthDate(22,2,2004);}
public void change3(BirthDate b)
{b.setDay(22);}
```

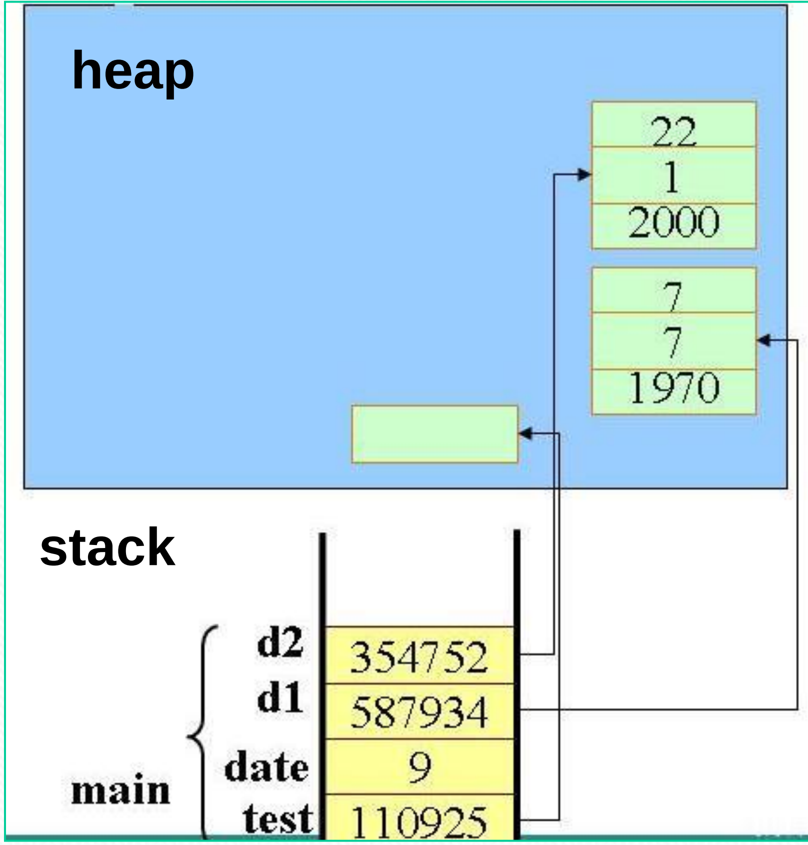
Memory Analysis (9)



```
Test test = new Test();
int date = 9;
BirthDate d1=
  new BirthDate(7,7,1970);
BirthDate d2=
  new BirthDate(1,1,2000);
test.change1(date);
test.change2(d1);
test.change3(d2);
     … … …
public void change1(int i)
{i = 1234;}
public void change2(BirthDate b)
{b = new BirthDate(22,2,2004);}
public void change3(BirthDate b)
{b.setDay(22);}
```

The birthday of **d2** is changed!

Memory Analysis (10)



```
Test test = new Test();
int date = 9;
BirthDate d1=
  new BirthDate(7,7,1970);
BirthDate d2=
  new BirthDate(1,1,2000);
test.change1(date);
test.change2(d1);
test.change3(d2);
    … … …
public void change1(int i)
{i = 1234;}
public void change2(BirthDate b)
{b = new BirthDate(22,2,2004);}
public void change3(BirthDate b)
{b.setDay(22);}
```

Code Analysis

Let’s look at the class CreateObjectDemo

Write and Show Time

- Write a **`Student`** class for a library system.
- Write 4 instance variables.
- Write 2 methods which allows a student to borrow books and return books.
- In the **`Library`** class, which contains the **`main`** method, create a student who borrows three books, and returns one book. Then show how many books that student is keeping.

Summary

- Class and object
- Relationships between classes
- Class definition
- Instance variable and methods
- Key word: `this`
- Constructors
- Memory analysis