

Inheritance & Abstract Classes

15-121 Fall 2020

Margaret Reid-Miller

Today

Hw7 is due Wednesday, Nov 4 at 11:55pm.

Today:

- Inheritance
- Abstract Classes
- (Clone)

Object Oriented Programming Inheritance

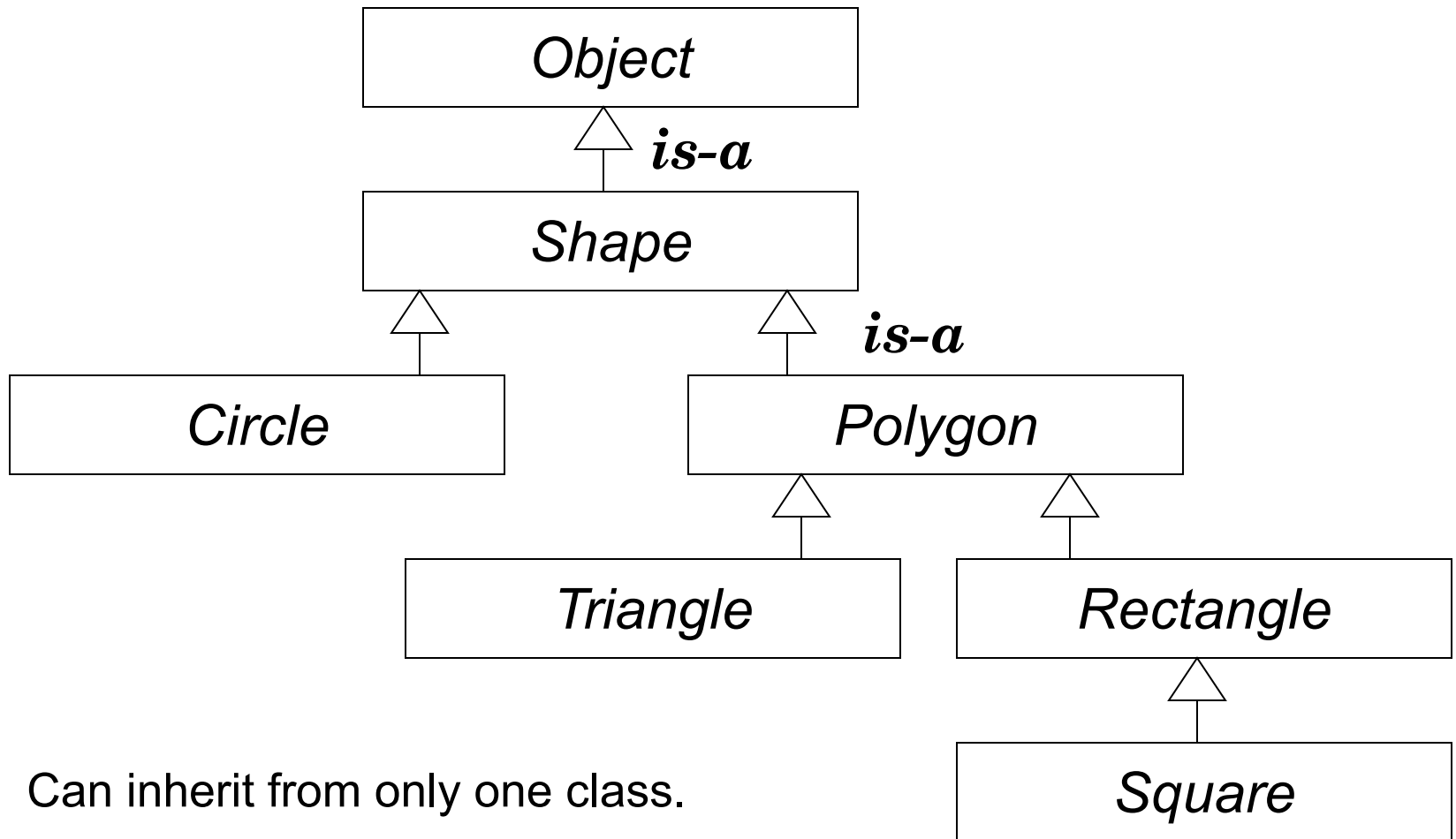
Inheritance enables defining a new class based on an existing class.

- The key idea is to refine an existing class to derive a new class, by adding new fields, and adding new methods or redefining existing methods.
- It enables code reuse because the new class *inherits* the all the fields and methods of the existing class.
- The original class is called the **superclass** and the class based on the superclass class is called the **subclass**.

Inheritance

- Every class inherits (implicitly) from the `Object` class in Java.
- Every class *is-a* `Object`
- There are no fields in `Object`, but there are methods such as `equals` and `toString`.
- All classes are arranged in a hierarchy (a tree) with `Object` at the top of the hierarchy.

Superclasses & Subclasses




Java's inheritance keywords

- A class that **extends** another class is a subclass that inherits all fields and methods (but not constructors) of the superclass.
- The subclass has direct access to all fields that are **public** and **protected**.
- The subclass can **override** the definitions of inherited methods with new implementations (using the same signature) and can access overridden methods using the **super** keyword.

Example: A superclass

```
public class BankAccount {  
    private double balance;  
  
    public BankAccount() {...}  
    public BankAccount(double initBalance) {...}  
    public void deposit(double amount) {...}  
    public void withdraw(double amount) {...}  
    public double getBalance() {...}  
    public String toString() {...}  
}
```



automatically inherits
from Object if no other
superclass is specified

Example: A subclass

```
public class SavingsAccount extends BankAccount
{
    private double interestRate;
```

- The subclass need only define the fields and methods that distinguishes the subclass from the superclass.
- E.g., SavingsAccount inherits the balance field and methods from BankAccount.
- It does not inherit constructors, though.

Inheritance: extends

- **Fields** of a subclass comes for two sources:
 - It inherits all the fields of all its ***ancestor*** classes.
 - It can define additional fields of its own.
- That is, objects of the subclass may store more data than objects of the superclass.
- **Methods** of a subclass can be defined as follows:
 - ***inherits*** from *ancestor* classes automatically,
 - ***override*** (redefine) inherited methods,
 - **add instructions** to inherited methods,
 - **define** new methods, possibly ***overloading*** inherited methods.

In the constructor, `super()` calls the constructor of the superclass

```
public SavingsAccount(double initBalance,  
                        double initRate) {  
    super(initBalance);  ← must be first statement  
    interestRate = initRate;  
}
```

```
public SavingsAccount(double initRate) {  
    super();             ← the superclass initializes balance  
    interestRate = initRate;  
}
```

You can define new methods in a subclass.

```
public void addInterest() {  
    deposit(getBalance()*interestRate);  
}
```



Use public method to get
balance from super class.

Field and Method visibility

- Classes (and their parts) have ***visibility modifiers***:
 - **public**: accessible to everyone
 - **protected**: inside package, inside class, inside subclass
 - **package-private** (default, no modifier used): inside package, inside class
 - **private**: accessible only within the class

A subclass can access a field in an ancestor class that is protected.

In BankAccount:

```
protected double balance;
```

In SavingsAccount:

```
public void addInterest() {  
    deposit(balance*interestRate);  
}
```

Overriding a method redefines an inherited method.

- If an inherited method does not do what the subclass needs, the subclass can **redefine** it.
- A method ***overrides*** the inherited method if it has the **same signature** and **return type** as the parent's definition.
- An object of the superclass will have the superclass method definition.
- An object of a subclass will have the subclass method definition; the superclass' definition of the method is not visible to the object of a subclass.

Use `super.` to call the superclass method that is overridden.

- A savings account subtracts a fee of \$10 for withdrawals over \$1000.
- We need to override the `withdraw` method and provide a new implementation that is appropriate.

```
public void withdraw(double amount) {  
    if (amount > 1000.0)  
        super.withdraw(amount + 10.0);  
    else  
        super.withdraw(amount);  
}
```

← same signature

← call to superclass method

What happens if you forget to use `super`?

Exercise: Write a `toString` method for the `SavingsAccount` class. Take advantage of code reuse!

Inheritance: extends

- **Fields** of a subclass comes for two sources:
 - It inherits all the fields of all its ***ancestor*** classes.
 - It can define additional fields of its own.
- That is, objects of the subclass may store more data than objects of the superclass.
- **Methods** of a subclass can be defined as follows:
 - ***inherits*** from *ancestor* classes automatically,
 - ***override*** (redefine) inherited methods,
 - **add instructions** to inherited methods,
 - **define** new methods, possibly ***overloading*** inherited methods.

You should override the methods inherited from Object class.

These methods typically do not work properly for our specific subclasses so we must override them.

```
public boolean equals(Object obj) {  
    SavingsAccount other = (SavingsAccount) obj;  
  
    return  
        this.interestRate == other.interestRate  
        && this.balance == other.balance;  
}
```

(assumes balance is protected in the BankAccount class)

Casting an Object type error

In the `equals` method, we used the following cast:

```
SavingsAccount other = (SavingsAccount)obj;
```

What if `obj` is not a `SavingsAccount`?

A `ClassCastException` is thrown at runtime.

We can use the `instanceof` operator to check the object's actual type during runtime to avoid the exception.

Use `instanceof` to prevent a runtime exception.

an operator

```
public boolean equals(Object obj) {  
    if (obj instanceof SavingsAccount) {  
        SavingsAccount other = (SavingsAccount)obj;  
        return  
            this.interestRate == other.interestRate  
            && this.balance == other.balance;  
    }  
    return false; // obj is not a SavingsAccount  
}
```

You should override these 3 Object class methods

- **public boolean equals(Object obj)**
Compares this object with the specified object by comparing the references only.
- **public String toString()**
Returns the class name + "@" + the hexadecimal representation of the object's hashCode.
- **public int hashCode()**
Calculates the hashCode of this object based on its reference only.

Polymorphism:

Which method is called depends on the actual object type not its declared type.

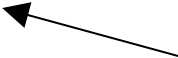
```
BankAccount acct;  
acct = new BankAccount(15121.0);  
acct.withdraw(2000.0);  
System.out.println(acct.getBalance());  
  
acct = new SavingsAccount(15121.0);  
acct.withdraw(2000.0);  
System.out.println(acct.getBalance());
```

The same statement, but are calls to two different methods.
Determined at runtime.

Interfaces (review)

A Java interface (not a GUI) is a means for defining **specifications** for behaviors that are common across classes that are not directly related by inheritance.

```
public interface Comparable<T> {  
    int compareTo(T obj);  
}
```



abstract method(s)

An interface cannot be instantiated directly:

```
Comparable<String> s = new Comparable<String>();
```

WRONG!

Any class that implement an interface must provide *implementations* for the methods specified.

```
public class BankAccount
    implements Comparable<BankAccount> {
    ...

    public int compareTo(BankAccount other) {
        ... // provide implementation
    }
}
```

Inheritance vs Implements

How many classes can a class extend?

- 1 exactly. Why "exactly" one?
- Every class inherits from the Object class

How many interfaces can a class implement?

- Zero or more.
- Provides a form of multiple inheritance.

Java class hierarchy can have Abstract classes

- An **abstract class** can have **abstract methods** like interfaces, but can **also have fields**.
- Abstract classes can have **constructors** that initialize the fields defined in the abstract class.
- Abstract classes can also have **concrete** (implemented) methods.
- An abstract class cannot be instantiated directly.

Abstract Class Example

```
public abstract class Vehicle {  
    private int speed;  
    private String manufacturer;  
    ...  
    public int getSpeed { return speed; }  
    public String getManufacturer  
        { return manufacturer; }  
    public abstract double toll();  
    ...  
}
```

A subclass of an abstract class must implement the abstract methods

```
public class Truck extends Vehicle {  
    private int numWheels;  
    ...  
    public double toll() {  
        return 10.0 * numWheels;  
    }  
    ...  
}
```

You cannot instantiate a Vehicle (it's abstract) but you can a Truck (it's concrete).

A variable can be declared with its interface or its supertypes

Interfaces:

```
String s = new String("Pittsburgh");  
Comparable<String> s2 = new String("Erie");
```

Abstract or super classes:

```
Truck myTruck = new Truck("Good Humor");  
Vehicle myRide = new Truck("Ryder");  
BankAccount bob = new SavingsAccount(1000.0);
```

Comparison

	Actual Class	Abstract Class	Interface
Instances can be created	Y	N	N
Can define fields and methods	Y	Y	N
Can define constants	Y	Y	Y
Number of these a class can extend	0 or 1	0 or 1	0
Number of these a class can implement	0	0	any number
Can extend another class	Y	Y	N
Can declare abstract methods	N	Y	Y
Can declare variables of this type	Y	Y	Y

Copying data

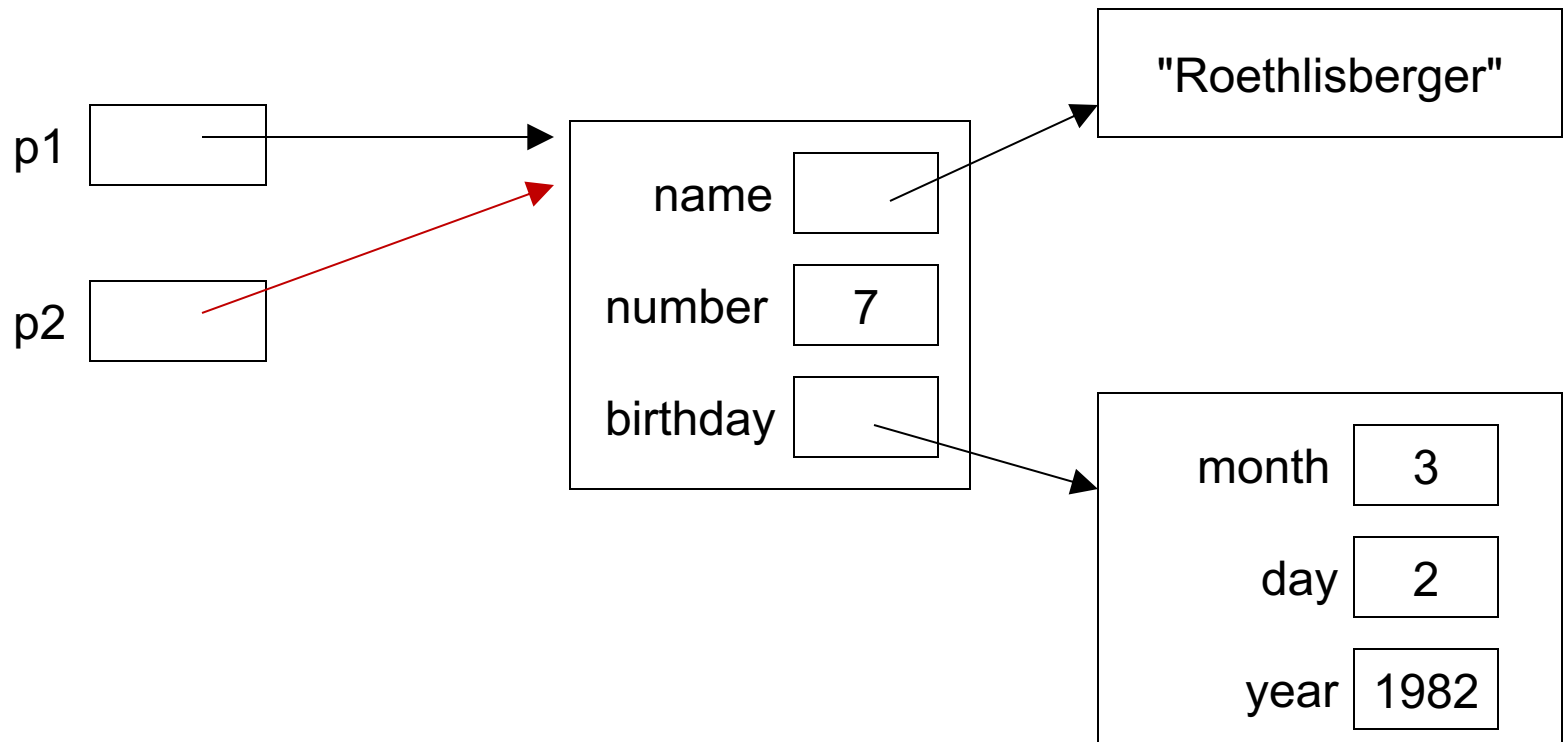
```
public class Person {  
    String name;  
    int number;  
    CalendarDate birthday;  
}
```



```
Person p1 = new Person("Roethlisberger", 7,  
    new CalendarDate(3, 2, 1982));
```


Copying data

```
Person p2 = p1;
```

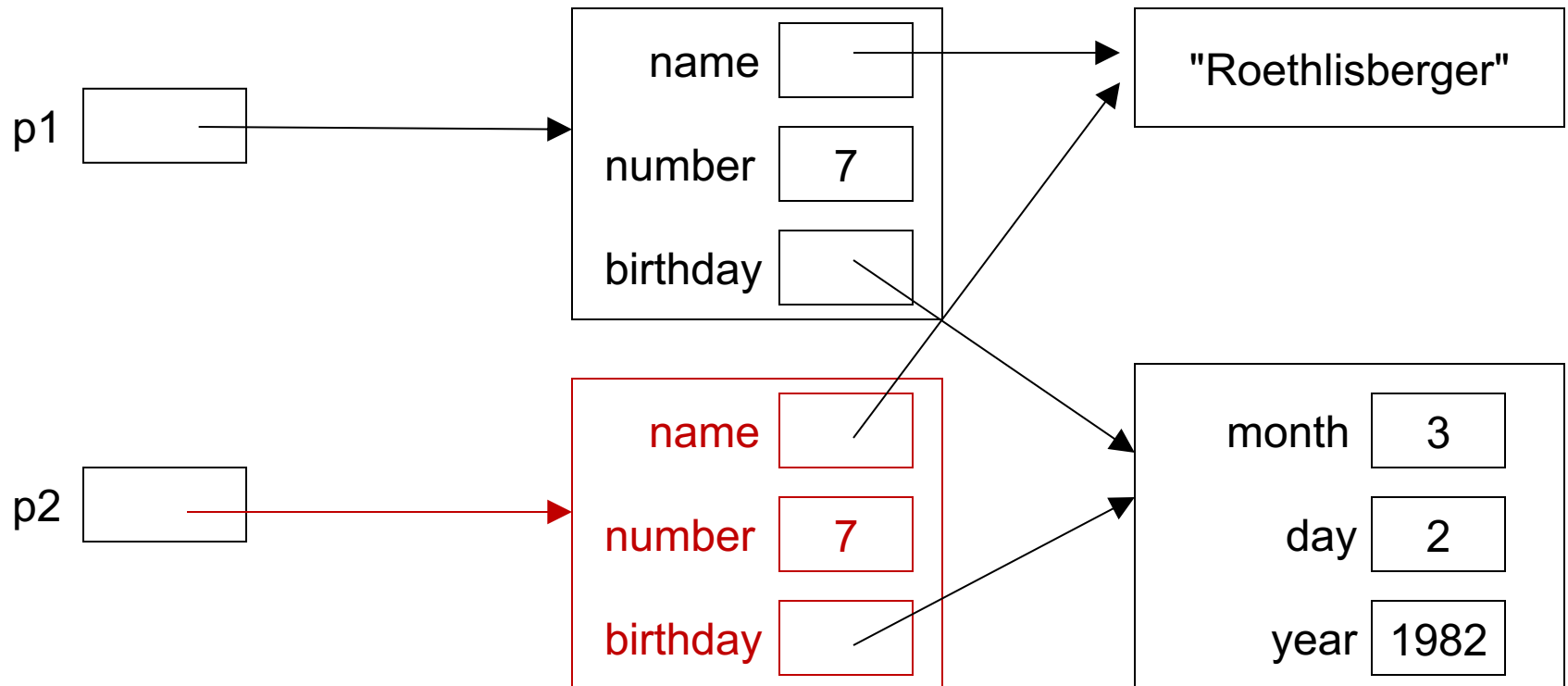


Cloning: Shallow Copy

```
public Object clone() {  
    Object newObj  
        = new Person(name, number, birthday);  
    return newObj;  
}
```

Cloning: Shallow Copy

```
Person p2 = (Person)p1.clone();
```



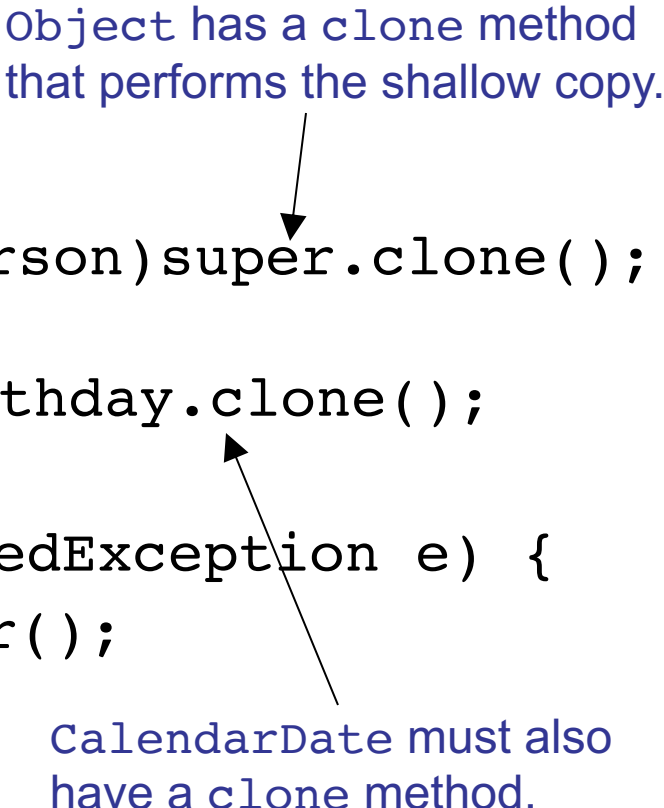
copies references only

Cloning: Deep Copy

```
@ Override
public Object clone() {
    try {
        Person newPerson = (Person)super.clone();
        newPerson.birthday =
            (CalendarDate)birthday.clone();
        return newPerson;
    } catch (CloneNotSupportedException e) {
        throw new InternalError();
    }
}
```

Object has a clone method
that performs the shallow copy.

CalendarDate must also
have a clone method.



Cloning: Deep Copy

```
@ Override
public Object clone() {
    try {
        CalendarDate newDate =
            (CalendarDate)super.clone();
        return newDate;
    } catch (CloneNotSupportedException e) {
        throw new InternalError();
    }
}
```

Cloning: Deep Copy

The method `Object.clone` will generate a `CloneNotSupportedException` if it is called in a class that does not implement the `Cloneable` interface.

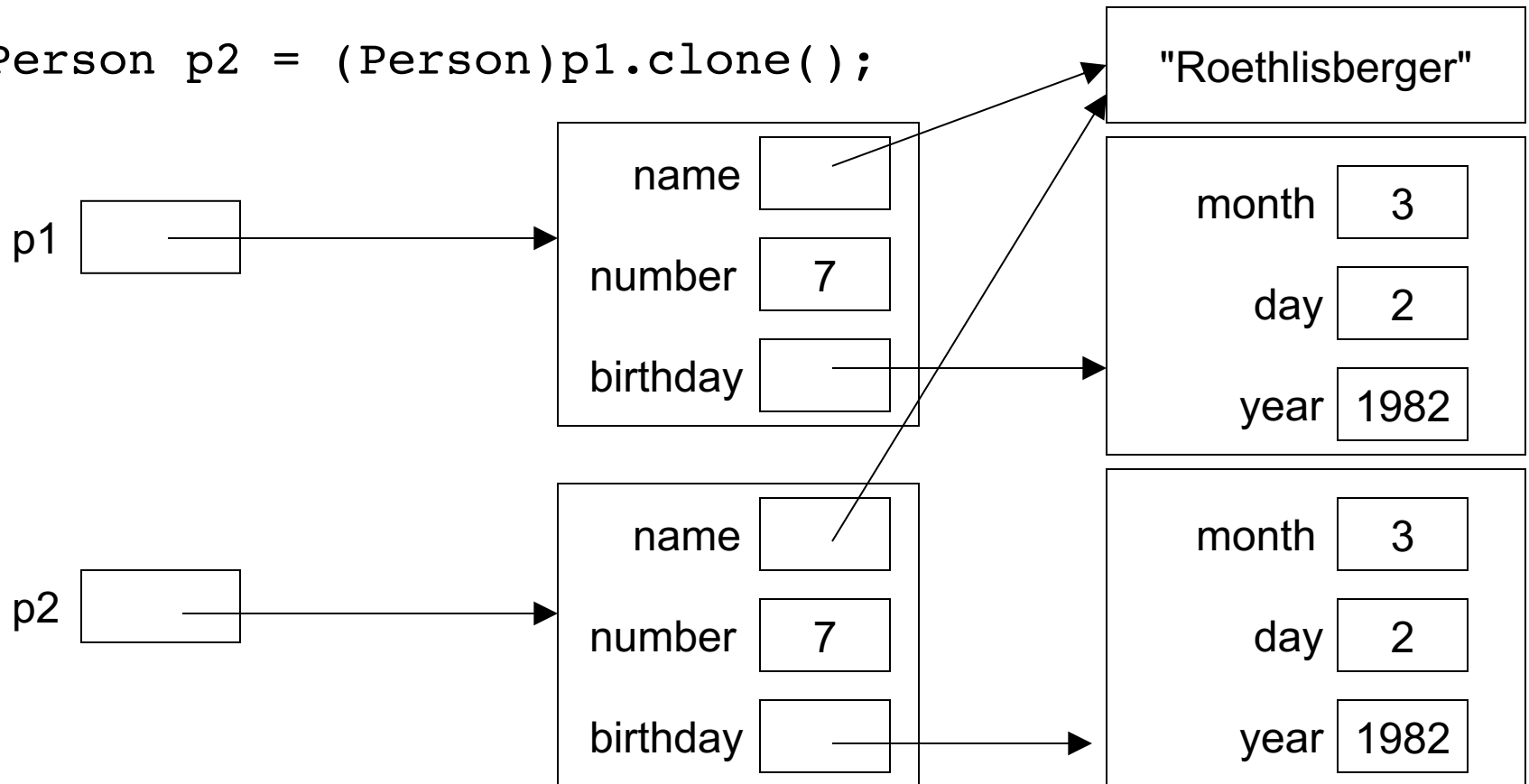
Therefore:

```
public class Person implements Cloneable {  
    ...  
}  
public class CalendarDate implements Cloneable {  
    ...  
}
```

Cloning: Deep Copy

Strings are
immutable

```
Person p2 = (Person)p1.clone();
```



Cloning: Deep Copy

