Programming and Software Development
COMP90041

Lecture 6

Mutable Objects

Dr. Thomas Christy
thomas.christy@unimelb.edu.au

Slides and materials were developed by Prof Peter Schachte

Copyright © The University of Melbourne

Next Week - Assessment 3

TESTING

Programming Testing

- It is very difficult to write correct software, even for experienced programmers
- Even software that seems to work OK often harbours hidden bugs
- Flawless software often grows bugs as it is modified
- You <u>must</u> test your programs thoroughly
- Plan how to test your program as (or even before) your write it

The Psychology of Testing

- Programmers tend to test their code with the goal of not finding any defects
- Testers test the code with the goal of uncovering defects
- A successful test is one that detects an unknown defect
- Many software houses have separate QA teams for that reason
- You don't have that luxury

How to Test

- Unit testing: testing that individual classes and methods behave correctly
- For each aspect of the specification, write code to establish the conditions of that aspect, and check that the result is as expected
- Regression testing: running a suite of tests whenever you modify your code, looking for new bugs
- Ensure that corner cases (situations where things happen differently than usual) are tested
- Where there are boundary conditions, test just before, at, and just after the boundary
- Write extra tests for the most complicated aspects

THE UNIVERSITY OF MELBOURNE

Last week

this

- Inside a method, special keyword `this` always holds the object the current message was sent to
- An implicit parameter to instance methods
- *E.g.*, if message was `foo.bar(baz)`, while executing method `bar`, `this` is `foo`
- Inside constructor, `this` is the object being created

```
public Person(String givenName,
              String familyName) {
    this.givenName = givenName;
    this.familyName = familyName;
}
```

Copyright © University of Melbourne 2018

Immutable Objects

- Immutable objects cannot be changed once they are created
- An Immutable class is one all of whose instances are immutable
- To be immutable, a class must not have any public methods that modify the object (just constructor)
- In Java, the `String` class is immutable
- **Primitive types** are also immutable
- Bad Things can happen when you modify objects (as we will see)
- Best practice: make classes immutable if possible

Copyright © University of Melbourne 2018

Finality

- Java can help you make classes immutable
- Add `final` keyword to instance variables before type
- Then Java will only let you set it once

```
public class ImmutablePerson {
    private final String familyName;
    private final String givenName;

    public ImmutablePerson(String familyName,
            String givenName) {
        this.familyName = familyName;
        this.givenName = givenName;
    }

   ⋮
```

Copyright © University of Melbourne 2018

Finality Protects You From Yourself

Adding this method:

```
    public void bogus(String s) {
        this.familyName = s;
    }
```

gives this compiler error:

```
ImmutablePerson.java:12: error: cannot assign
 a value to final variable familyName
        this.familyName = s;
            ^
1 error
```

Copyright © University of Melbourne 2018

Class vs. Instance Messages

- An instance message is sent an individual object
- A class message is sent to the whole class
  - Not to any individual object
- So a class method does not have (cannot use) `this`
- Class method cannot access instance variables!
- Class methods cannot use instance methods (without specifying *`object.`* in front)
- but instance methods can use class methods

Copyright © University of Melbourne 2018

The `toString` method

- *E.g.*, if `p` is an `ImmutablePerson`, what should `"Welcome, " + p` produce?
- What should `System.out.print(p)` print?
- You can determine how Java converts instances of your class to a String
- Define a public method `String toString()`

```
public String toString() {
    return givenName + " " + familyName;
}
```

Copyright © University of Melbourne 2018

Summary

- Use `new Class(...)` to create object
- Use `object.methodName(...)` to send message to `object`
- Can use `null` in place of an object, but don't send message to it!
- Declare instance variables and methods like class ones, without `static` keyword
- Always make instance variables private
- Write constructor to initialise instance variables
- Make classes immutable when you can

Copyright © University of Melbourne 2018

Programming and Software Development
COMP90041

Lecture 6

Mutable Objects

Dr. Thomas Christy
thomas.christy@unimelb.edu.au

Slides and materials were developed by Prof Peter Schachte

Copyright © The University of Melbourne

Mutable Objects and Classes

- Instance variables not declared `final` can be modified by methods
- Objects and classes that permit modification are mutable
- They are common in object-oriented programming
- Immutable objects can be thought of as values, like primitive values
- Mutable objects must be understood in terms of their behaviour in memory

Memory

- Computer memory is a long sequence of bytes
- Computer can access any byte by its address — its position in the sequence
- A `short` is stored over 2 bytes, an `int` over 4, *etc.*
- Computer reads multiple bytes at once and assembles into a `short`, `int`, `double`, *etc.*

0000	23
0001	41
0002	97
0003	07
0004	41
0005	76
0006	00
0007	71
.	.
.	.
.	.

Objects

- An object is stored in memory by storing its instance variables
- An object at memory address 3270 with 2 `int` variables holding 5 and 10 might appear in memory as at right
- Actual values of a byte are 0–255 or -128–127, not 0–99
- Actual addresses are much larger than this illustration

3270	00
3271	00
3272	00
3273	05
3274	00
3275	00
3276	00
3277	10
.	.
.	.
.	.

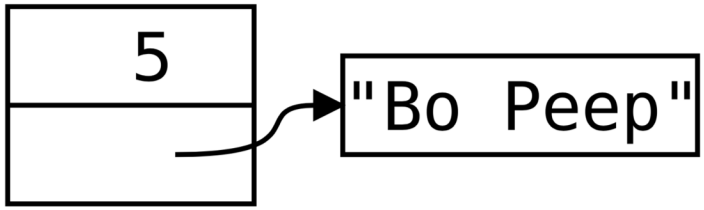
References (Pointers)

- Object types are stored by storing their address
- *E.g.*, to store a `String`, a variable holds its address, not its content
- We show the string stored at 7192, but it could be anywhere in memory
- *E.g.* an object with `int` and `String` variables holding 5 and `"Bo Peep"` would be stored as at right
- The address of an object is called a reference or pointer to it
- Classes are called reference types

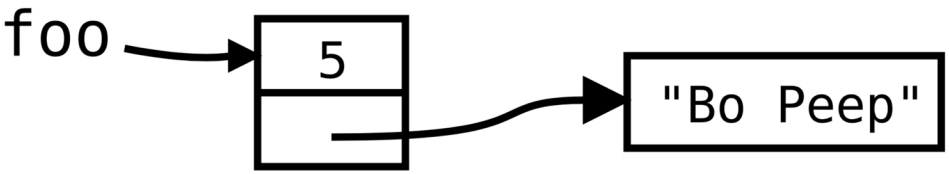
3270	00	
3271	00	
3272	00	
3273	05	
3274	00	
3275	00	
3276	71	
3277	92	
⋮	⋮	
7192	66	B
7193	111	o
7194	32	
⋮	⋮	⋮

Depicting Objects

- Such an object is more abstractly viewed as



- If a variable `foo` holds (a reference to) this object, it could be depicted as



Changing Objects

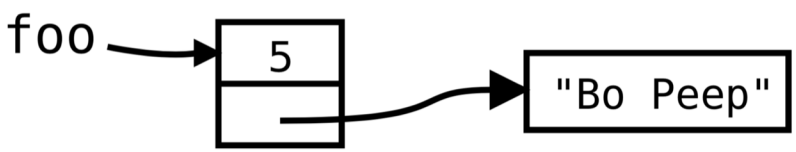
```
public class Person
    private int age;
    private String name;
    public Person(int age, String name) {
        this.age = age; this.name = name;
    }
    public String toString() {
        return name + ", age " + age;
    }
    public static void main(String args[]) {
        Person foo = new Person(5, "Bo Peep");
        ⋮
    }
}
```

Changing Objects 2

- With immutable `Person` class, only way to change the object `foo` points to is assign a new object

```
public static void main(String args[]) {
   Person foo = new Person(5, "Bo Peep");
   foo = new Person(3, "Shaun Sheep");
}
```

After first assignment:

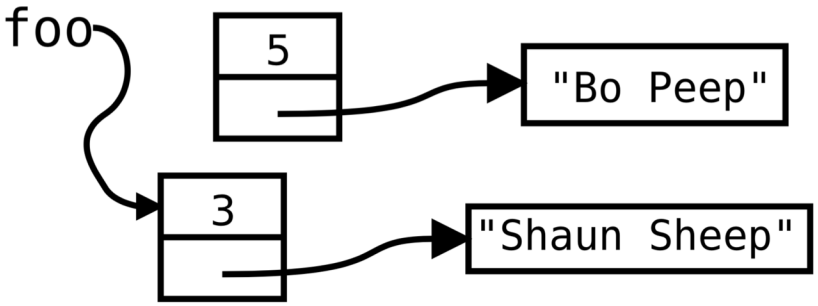


Changing Objects 2

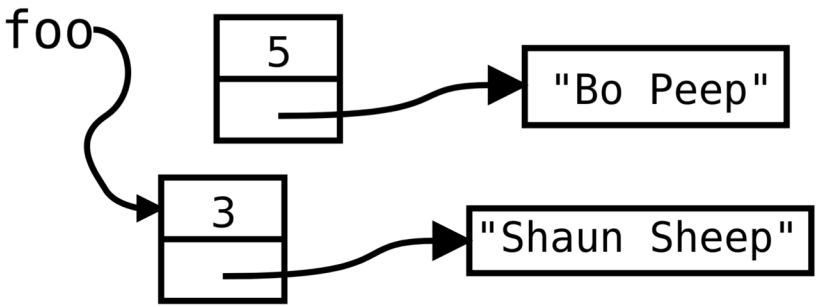
- With immutable `Person` class, only way to change the object `foo` points to is assign a new object

```
public static void main(String args[]) {
   Person foo = new Person(5, "Bo Peep");
   foo = new Person(3, "Shaun Sheep");
}
```

After second assignment:



Garbage Collection



- No variable (or object) points to original object
- Can never be used again, as no way to get to it
- Java has a garbage collector (GC) that "reclaims" such objects so the memory can be used again
- No need to worry about it, it all happens automatically

Mutators

- If instance variable is not declared `final`, methods can reassign it
- Simplest such method takes new value for instance variable as argument and assigns it
- This is called a mutator or setter
- Convention: for instance variable *`instanceVar`*, name mutator `set`*`InstanceVar`*

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

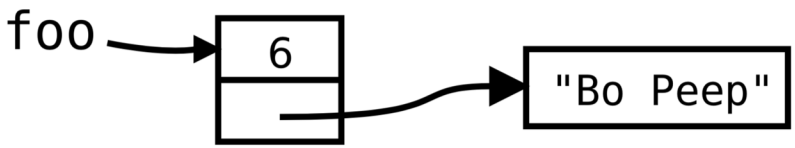
- Using a mutator modifies the existing object, without making a new one
- Only add appropriate setters for the class

Mutating Objects

Mutator gives a second way to change the object referred to by `foo`: mutate it

```
public static void main(String args[]) {
   Person foo = new Person(5, "Bo Peep");
   foo.setAge(6);
}
```

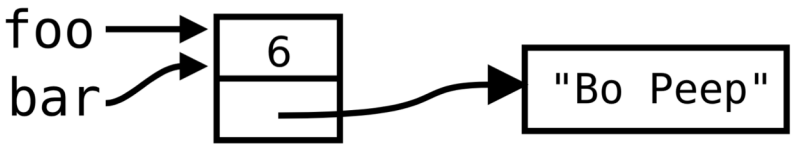
After second statement, only one object has been created, but it has been mutated:



Mutating aliased objects

- If multiple variables refer to same object, they are called aliases
- Changing one alias changes all of them

```
public static void main(String args[]) {
   Person foo = new Person(5, "Bo Peep");
   Person bar = foo;
   foo.setAge(6);
}
```



What does this print?

If this is in the `Person` class:

```
public static void main(String args[]) {
    Person foo = new Person(5, "Bo Peep");
    Person bar = foo;
    foo.setAge(6);
    System.out.println(bar.age);
}
```

A. 5
B. 6
C. Won't compile: `age` is `private`
D. Won't compile: can't put `main` method in `Person` class

Answer at end of slides!

What does this print?

```
public static void main(String args[]) {
    int foo = 5;
    int bar = foo;
    foo = 6;
    System.out.println(bar);
}
```

A. 5
B. 6

What does this print?

```
public static void main(String args[]) {
    int foo = 5;
    int bar = foo;
    foo = 6;
    System.out.println(bar);
}
```

A. 5
B. 6

Setting a variable changes only that variable; mutating an object changes that object for every variable or object that references it

Answer at end!

Class Invariants

- We carefully declared instance variables `private`, so other classes can't mess with them
- This allows us to ensure all instances have certain properties
- *E.g.*, may want to ensure we have a name and location for a person who is seeking a lost loved one
- Condition like this is called a class invariant

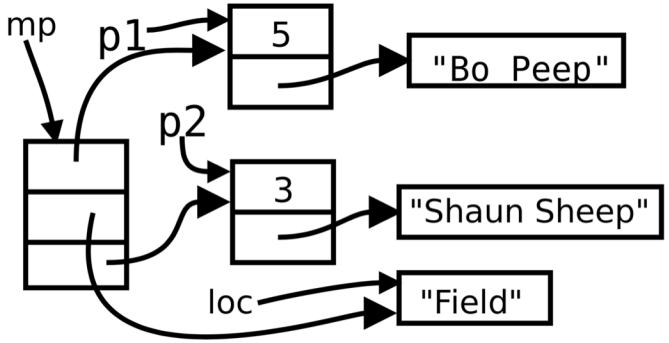
Class Invariants

```
public class MissingPerson {
   private Person seeker;
   private String location;
   private Person sought;
   public MissingPerson(Person seeker, String location,
                        Person sought) {
      if (seeker.getName().isEmpty()
          || location.isEmpty()) {
          System.out.println("Missing name/location");
          System.exit(1);
       }
       this.seeker = seeker;
       this.location = location;
       this.sought = sought;
   }
   ⋮
```

Privacy Leaks

But if an instance variable holds a mutable object and any method of another class can access that object, it can violate conditions of that object we want to ensure

```
Person p1 = new Person(5, "Bo Peep");
Person p2 = new Person(3, "Shaun Sheep");
String loc = "Field";
MissingPerson mp = new MissingPerson(p1, loc, p2);
```



Privacy Leaks

What if that code is followed by this:

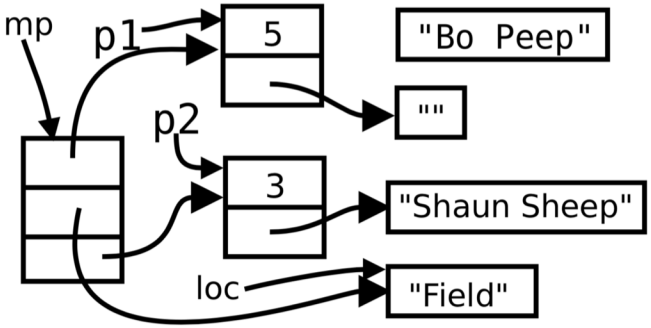
```
p1.setName("");
```

Privacy Leaks

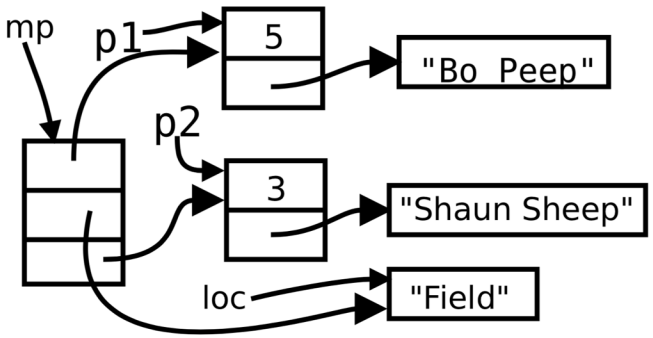
What if that code is followed by this:

```
p1.setName("");
```

Now `seeker.getName().isEmpty()` is true for `mp`!
That violates our class invariant!



Shared mutable objects



- This problem occurs because `mp` shares its `seeker` with `p1`, and that object is mutable
- If either `p1` or `mp.seeker` is changed, the other changes, too (because they are the same object)
- Note `mp.location` shares with `loc`, but that is not a problem because strings are immutable

Shared mutable objects Example

```
public static void main(String[] args) {
        Person p1 = new Person(5, "Bo Peep");
    Person p2 = new Person(3, "Shaun Sheep");
    MissingPerson mp =
            new MissingPerson(p1,"Field",p2);
    System.out.println(mp);
    p1.setName("Bad Wolf");
    p1.setAge(10);
    System.out.println(mp);
}
```

```
Bo Peep, age 5, at Field, seeks Shaun Sheep, age 3
Bad Wolf, age 10, at Field, seeks Shaun Sheep, age 3
```

Defensive Copying

- Best solution: make `Person` class immutable
- Alternative: stop instance variables from sharing with any outside variables or objects
- Do this by copying objects passed in to methods and storing the copy instead of the original
- Same if mutable object stored in instance variable is returned by method: caller can modify it
- Must also copy objects stored in instance variables and return the copy instead of original object
- This is called defensive copying
- Commonly needed in constructors, getters, and setters

Copy Constructor

- For mutable classes, write a copy constructor: a constructor that takes one argument of the same type as the object being constructed
- Just makes the new object an exact copy of the input argument
- *E.g.*:

```
public Person(Person orig) {
    this.age = orig.age;
    this.name = orig.name;
}
```

Deep Copying

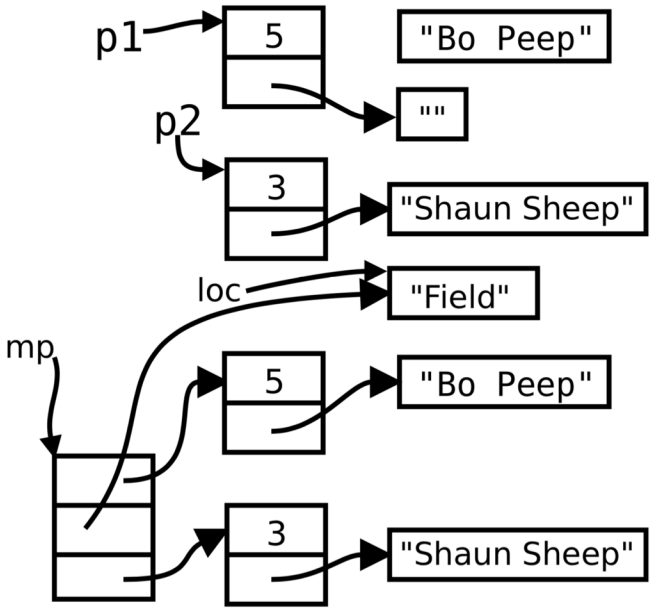
- If any instance variables are mutable classes, copy constructor must copy those objects, too
- This is called a deep copy: it does not share any mutable objects with the original object

```
public MissingPerson(MissingPerson orig) {
    this.seeker = new Person(orig.seeker);
    this.location = orig.location; // immutable!
    this.sought = new Person(orig.sought);
}
```

With Deep Copying

With a deep copy, `p1` and `p2` do not share with `mp`, so this is harmless:

```
p1.setName("");
```



Plugging Privacy Leaks

```
public class MissingPerson {
   private Person seeker;
   private String location;
   private Person sought;
   public MissingPerson(Person seeker, String location,
                        Person sought) {
      if (seeker.getName().isEmpty()
          || location.isEmpty()) {
          System.out.println("Missing name/location");
          System.exit(1);
       }
       this.seeker = new Person(seeker); // copy!
       this.location = location;
       this.sought = new Person(sought); // copy!
   }
   ⋮
```

Plugging Privacy Leaks (2)

```
public MissingPerson(MissingPerson orig) {
   this.seeker = new Person(orig.seeker); // copy!
   this.location = orig.location; // immutable!
   this.sought = new Person(orig.sought); // copy!
}
public Person getSeeker() {
   return new Person(seeker); // copy!
}
public String getLocation() {
   return location; // immutable
}
public Person getSought() {
   return new Person(sought); // copy!
}
⋮
```

Plugging Privacy Leaks (3)

```
    public void setSeeker(Person seeker) {
        this.seeker = new Person(seeker);
    }
    public void setLocation(String location) {
      this.location = location; // immutable
    }
    public void setSought(Person sought) {
        this.sought = new Person(sought);
    }
}
```

With this change, `"Shaun Sheep"` is safe from the `"Bad Wolf"`

Summary

- Instances variables not declared `final` can be modified
- If public methods modify instance variables, the class is mutable
- Mutable classes are common, but dangerous
- When multiple variables/objects refer to an object, mutating the object changes it for all of them
- Privacy leak: when method of another class gets ahold of a mutable object stored in a private instance variable
- Solution: copy mutable object using copy constructor before storing or returning it

THE UNIVERSITY OF
MELBOURNE

© Copyright The University of Melbourne 2018

What does this print?

If this is in the `Person` class:

```
public static void main(String args[]) {
    Person foo = new Person(5, "Bo Peep");
    Person bar = foo;
    foo.setAge(6);
    System.out.println(bar.age);
}
```

A. 5
B. 6
C. Won't compile: `age` is `private`
D. Won't compile: can't put `main` method in `Person` class

What does this print?

```
public static void main(String args[]) {
    int foo = 5;
    int bar = foo;
    foo = 6;
    System.out.println(bar);
}
```

A. 5

B. 6

Setting a variable changes only that variable; mutating an object changes that object for every variable or object that references it