Java modifiers

Madhavan Mukund

https://www.cmi.ac.in/~madhavan

Programming Concepts using Java

Week 3

Modifiers in Java

- Java uses many modifiers in declarations, to cover different features of object-oriented programming

Modifiers in Java

- Java uses many modifiers in declarations, to cover different features of object-oriented programming
- `public` vs `private` to support encapsulation of data

Modifiers in Java

- Java uses many modifiers in declarations, to cover different features of object-oriented programming
- `public` vs `private` to support encapsulation of data
- `static`, for entities defined inside classes that exist without creating objects of the class

Modifiers in Java

- Java uses many modifiers in declarations, to cover different features of object-oriented programming
- `public` vs `private` to support encapsulation of data
- `static`, for entities defined inside classes that exist without creating objects of the class
- `final`, for values that cannot be changed

Modifiers in Java

- Java uses many modifiers in declarations, to cover different features of object-oriented programming
- `public` vs `private` to support encapsulation of data
- `static`, for entities defined inside classes that exist without creating objects of the class
- `final`, for values that cannot be changed
- These modifiers can be applied to classes, instance variables and methods

Modifiers in Java

- Java uses many modifiers in declarations, to cover different features of object-oriented programming
- `public` vs `private` to support encapsulation of data
- `static`, for entities defined inside classes that exist without creating objects of the class
- `final`, for values that cannot be changed
- These modifiers can be applied to classes, instance variables and methods
- Let's look at some examples of situations where different combinations make sense

public vs private

- Faithful implementation of encapsulation necessitates modifiers `public` and `private`
  - Typically, instance variables are `private`
  - Methods to query (accessor) and update (mutator) the state are `public`

public vs private

- Faithful implementation of encapsulation necessitates modifiers `public` and `private`
  - Typically, instance variables are `private`
  - Methods to query (accessor) and update (mutator) the state are `public`
- Can `private` methods make sense?

public vs private

- Faithful implementation of encapsulation necessitates modifiers `public` and `private`
  - Typically, instance variables are `private`
  - Methods to query (accessor) and update (mutator) the state are `public`
- Can `private` methods make sense?
- Example: a `Stack` class
  - Data stored in a private array
  - Public methods to push, pop, query if empty

```
public class Stack {
  private int[] values; // array of values
  private int tos;      // top of stack
  private int size;     // values.length

  /* Constructors to set up values array */

  public void push (int i){
    ....
  }

  public int pop (){
    ...
  }

  public boolean is_empty (){
    return (tos == 0);
  }
}
```

private methods

- Example: a `Stack` class
  - Data stored in a private array
  - Public methods to push, pop, query if empty

```
public class Stack {
  private int[] values; // array of values
  private int tos;      // top of stack
  private int size;     // values.length

  /* Constructors to set up values array */

  public void push (int i){
    ....
  }

  public int pop (){
    ...
  }

  public boolean is_empty (){
    return (tos == 0);
  }
}
```

private methods

- Example: a `Stack` class
  - Data stored in a private array
  - Public methods to push, pop, query if empty
- `push()` needs to check if stack has space

```
public class Stack {
  ...
  public void push (int i){
    if (tos < size){
      values[tos] = i;
      tos = tos+1;
    }else{
      // Deal with stack overflow
    }
    ...
  }
  ...
}
```

private methods

- Example: a `Stack` class
  - Data stored in a private array
  - Public methods to push, pop, query if empty
- `push()` needs to check if stack has space
- Deal gracefully with stack overflow
  - `private` methods invoked from within `push()` to check if stack is full and expand storage

```
public class Stack {
  ...
  public void push (int i){
    if (stack_full()){
      extend_stack();
    }
    ... // Usual push operations
  }
  ...
  private boolean stack_full(){
    return(tos == size);
  }

  private void extend_stack(){
    /* Allocate additional space,
       reset size etc */
  }
}
```

Accessor and mutator methods

- Public methods to query and update private instance variables

Accessor and mutator methods

- Public methods to query and update private instance variables
- `Date` class
  - Private instance variables `day`, `month`, `year`
  - One public accessor/mutator method per instance variable

```
public class Date {
  private int day, month year;

  public void getDay(int d) {...}
  public void getMonth(int m) {...}
  public void getYear(int y) {...}

  public void setDay(int d) {...}
  public void setMonth(int m) {...}
  public void setYear(int y) {...}
}
```

Accessor and mutator methods

- Public methods to query and update private instance variables
- `Date` class
  - Private instance variables `day`, `month`, `year`
  - One public accessor/mutator method per instance variable
- Inconsistent updates are now possible
  - Separately set invalid combinations of `day` and `month`

```
public class Date {
  private int day, month year;

  public void getDay(int d) {...}
  public void getMonth(int m) {...}
  public void getYear(int y) {...}

  public void setDay(int d) {...}
  public void setMonth(int m) {...}
  public void setYear(int y) {...}
}
```

Accessor and mutator methods

- Public methods to query and update private instance variables
- `Date` class
  - Private instance variables `day`, `month`, `year`
  - One public accessor/mutator method per instance variable
- Inconsistent updates are now possible
  - Separately set invalid combinations of `day` and `month`
- Instead, allow only combined update

```
public class Date {
  private int day, month year;

  public void getDay(int d) {...}
  public void getMonth(int m) {...}
  public void getYear(int y) {...}

  public void setDate(int d, int m, int y) {
    ...
    // Validate d-m-y combination
  }

}
```

`static` components

- Use `static` for components that exist without creating objects
  - Library functions, `main()`, ...
  - Useful constants like `Math.PI`, `Integer.MAX_VALUE`

`static` components

- Use `static` for components that exist without creating objects
  - Library functions, `main()`, ...
  - Useful constants like `Math.PI`, `Integer.MAX_VALUE`
- These `static` components are also `public`

`static` components

- Use `static` for components that exist without creating objects
  - Library functions, `main()`, ...
  - Useful constants like `Math.PI`, `Integer.MAX_VALUE`
- These `static` components are also `public`
- Do `private static` components make sense?

static components

- Use `static` for components that exist without creating objects
  - Library functions, `main()`, ...
  - Useful constants like `Math.PI`, `Integer.MAX_VALUE`
- These `static` components are also `public`
- Do `private static` components make sense?
- Internal constants for bookkeeping
  - Constructor sets unique id for each order

```
public class Order {
  private static int lastorderid = 0;

  private int orderid;
  ....

  public Order(...) {
    lastorderid++;
    orderid = lastorderid;
    ...
  }
```

static components

- Use `static` for components that exist without creating objects
  - Library functions, `main()`, ...
  - Useful constants like `Math.PI`, `Integer.MAX_VALUE`
- These `static` components are also `public`
- Do `private static` components make sense?
- Internal constants for bookkeeping
  - Constructor sets unique id for each order

```
public class Order {
  private static int lastorderid = 0;

  private int orderid;
  ....

  public Order(...) {
    lastorderid++;
    orderid = lastorderid;
    ...
  }
```

- `lastorderid` is private static field

static components

- Use `static` for components that exist without creating objects
  - Library functions, `main()`, ...
  - Useful constants like `Math.PI`, `Integer.MAX_VALUE`
- These `static` components are also `public`
- Do `private static` components make sense?
- Internal constants for bookkeeping
  - Constructor sets unique id for each order

```
public class Order {
  private static int lastorderid = 0;

  private int orderid;
  ....

  public Order(...) {
    lastorderid++;
    orderid = lastorderid;
    ...
  }
```

- `lastorderid` is private static field
- Common to all objects in the class

static components

- Use `static` for components that exist without creating objects
  - Library functions, `main()`, ...
  - Useful constants like `Math.PI`, `Integer.MAX_VALUE`
- These `static` components are also `public`
- Do `private static` components make sense?
- Internal constants for bookkeeping
  - Constructor sets unique id for each order

```
public class Order {
  private static int lastorderid = 0;

  private int orderid;
  ....

  public Order(...) {
    lastorderid++;
    orderid = lastorderid;
    ...
  }
```

- `lastorderid` is private static field
- Common to all objects in the class
- Be careful about concurrent updates!

`final` components

- `final` denotes that a value cannot be updated

`final` components

- `final` denotes that a value cannot be updated
- Usually used for constants (`public` and `static` instance variables)
  - `Math.PI`, `Integer.MAX_VALUE`

`final` components

- `final` denotes that a value cannot be updated
- Usually used for constants (`public` and `static` instance variables)
  - `Math.PI`, `Integer.MAX_VALUE`
- What would `final` mean for a method?
  - Cannot redefine functions at run-time, unlike Python!

`final` components

- `final` denotes that a value cannot be updated
- Usually used for constants (`public` and `static` instance variables)
  - `Math.PI`, `Integer.MAX_VALUE`
- What would `final` mean for a method?
  - Cannot redefine functions at run-time, unlike Python!
- Recall overriding
  - Subclass redefines a method available with the same signature in the parent class

`final` components

- `final` denotes that a value cannot be updated
- Usually used for constants (`public` and `static` instance variables)
  - `Math.PI`, `Integer.MAX_VALUE`
- What would `final` mean for a method?
  - Cannot redefine functions at run-time, unlike Python!
- Recall overriding
  - Subclass redefines a method available with the same signature in the parent class
- A `final` method cannot be overridden

Summary

- `private` and `public` are natural artefacts of encapsulation
  - Usually, instance variables are `private` and methods are `public`
  - However, `private` methods also make sense
- Modifiers `static` and `final` are orthogonal to `public`/`private`
- Use `private static` instance variables to maintain bookkeeping information across objects in a class
  - Global serial number, count number of objects created, profile method invocations, ...
- Usually `final` is used with instance variables to denote constants
- Also makes sense for methods
  - A `final` method cannot be overridden by a subclass
- Can also have `private` classes, later