Advanced Programming Techniques in Java

COSI 12B

Object Oriented Programming III

Lecture 10

Class Objectives

- Arrays of objects (second subsection of 7.4)
- Method Overloading (last subsection of 3.1)
- Encapsulation (Section 8.4)

Review: Point Class (ver. 5)

Point.java

```
public class Point{
  int x;
  int y;

  // constructor
  public Point(int initialX, int initialY){
    x = initialX;
    y = initialY;
  }

  // constructor
  public Point(){
    x = 0;
    y = 0;
  }

  // shifts points location by the given amount
  public void translate (int dx, int dy){
    x += dx;
    y += dy;
  }

  // computes the distance between two points
  public double distance(Point other){
    int dx = x - other.x;
    int dy = y - other.y;
    return Math.sqrt(dx * dx + dy * dy);
  }
…
```

Point.java (cont.)

```
…
  // computes the distance between a point and the origin
  public double distanceFromOrigin() {
    Point origin = new Point();
    return distance(origin);
  }

  public String toString(){
    return "(" + x + " , " + y + ")";
  }

  // Returns whether o refers to a Point object with
  // the same (x, y) coordinates as this Point object
  public boolean equals(Object o) {
    if (o instanceof Point) {
      Point other = (Point) o
      return x == other.x && y == other.y;
    } else {
      return false;
    }
  }
}
```

Review: Final version of `equals` method

- This version of the `equals` method allows us to correctly compare `Point` objects against any other type of object:

```java
// Returns whether o refers to a Point object with
// the same (x, y) coordinates as this Point object
public boolean equals(Object o) {
    if (o instanceof Point) {
        Point other = (Point) o;
        return x == other.x && y == other.y;
    } else {
        return false;
    }
}
```

you still have to keep the casting

Review: Template for your `equals()` methods

```
public boolean equals (Object o){
  if (o instanceof <type>){
         <type> other = (<type>) o;
         //compare the state and return the result
  }
  else {
         return false;
  }
}
```

Review: The `this` keyword

- **Definition** The **`this`** keyword refers to the current object in a method or constructor
- The `this` keyword is used to eliminate confusion between class attributes and parameters with the same name

Refer to a field:	`this`.**field**
Call a method:	`this`.**method**(**parameters**);
One constructor can call another:	`this`(**parameters**);

- So far, the compiler was converting expressions automatically
  - `x → this.x`
  - `setLocation(10,12) → this.setLocation(10,12)`

Arrays of objects

Arrays of objects

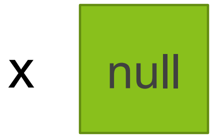
- `String[] words = new String[5];`
- When objects are first constructed their fields are initialized to their default value
  - `int` are initialized to `0`, `char` to `'0'`, `boolean` to `false`
  - Objects are initialized to `null`

```
int[] numbers= new int[4]; // all ints are initialized
System.out.println(numbers[0]); // prints out the number zero

String[] words = new String[4]; /all Strings are initialized
System.out.println(words[0]); // prints out null
```

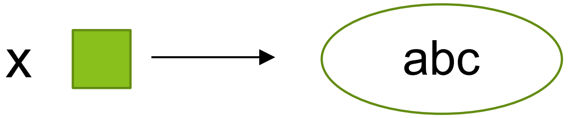
null

- Variables declared of a primitive type stores values
- Variables declared of a reference type store references

```
String x = null
```



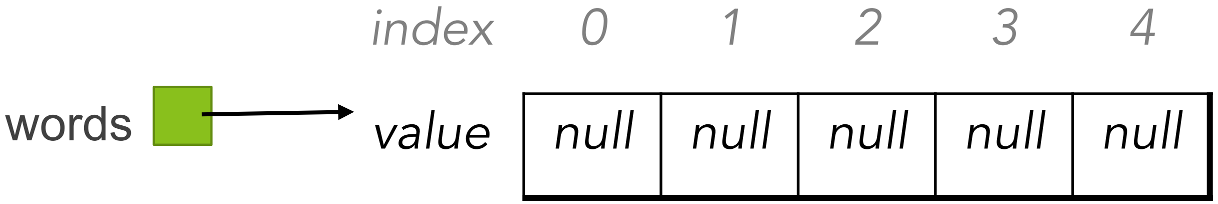
```
String x = "abc"
```



- **Definition** `null` is a value that indicates that the object reference is not currently referring to an object.

`null`

- The elements of an array of objects are initialized to `null`
- `String[] words = new String[5];`



index	0	1	2	3	4
value	*null*	*null*	*null*	*null*	*null*

Things you can do with `null`

- Store `null` in a variable or an array element
  - `String s = null;`
  - `words[2] = null;`
- Print a `null` reference
  - `System.out.println(s);    // output: null`
- Ask whether a variable or array element is null
  - `if (words[2] == null) { …`
- Pass `null` as a parameter to a method
  - `System.out.println(null);    // null`
- Return `null` from a method (often to indicate failure)

Dereferencing

- Dereferencing happens using the **.** operator

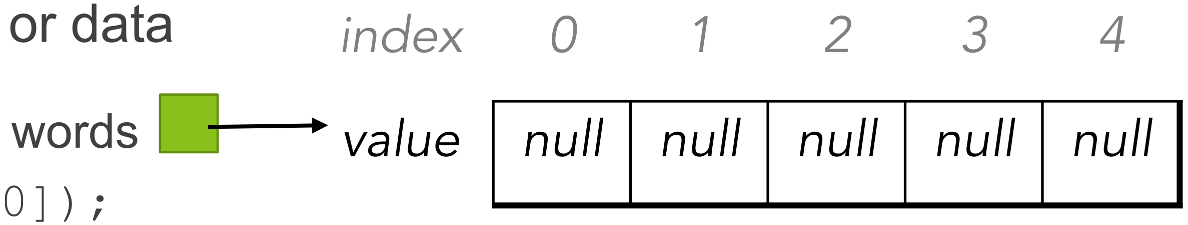
```
String s = "abc”;
int x = s.length();   //s is dereferenced
```

- Dereferencing follows the memory address placed in a reference, to the place in memory where the actual object is located
  - When an object has been found the requested method is called
  - If the reference has value `null`, dereferencing results in a `NullPointerException`

NullPointerException

- It is illegal to dereference `null` (causes an exception)
- `null` is not any object, so it has no methods or data

```
String[] words = new String[5];
System.out.println("word is: " + words[0]);
words[0] = words[0].toUpperCase();
```



index	0	1	2	3	4
value	null	null	null	null	null

- **Output**

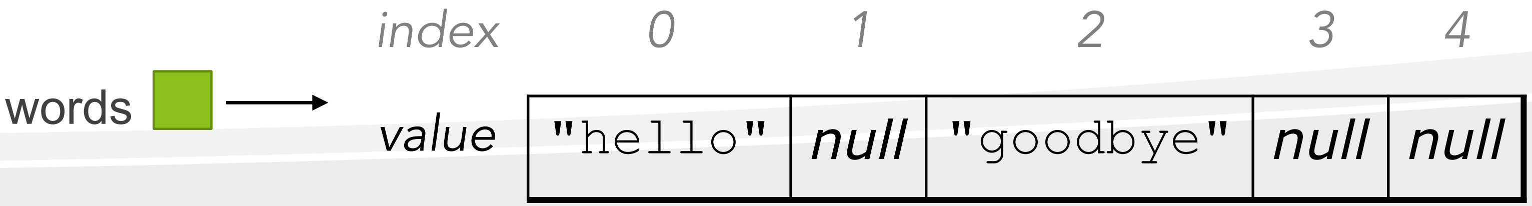
```
word is: null
Exception in thread "main" java.lang.NullPointerException
at Example.main(Example.java:8)
```

Looking before you leap

- You can check for `null` before calling an object's methods

```
String[] words = new String[5];
words[0] = "hello";
words[2] = "goodbye";    // words[1], [3], [4] are null
```

```
for (int i = 0; i < words.length; i++) {
    if (words[i] != null) {
        words[i] = words[i].toUpperCase();
    }
}
```



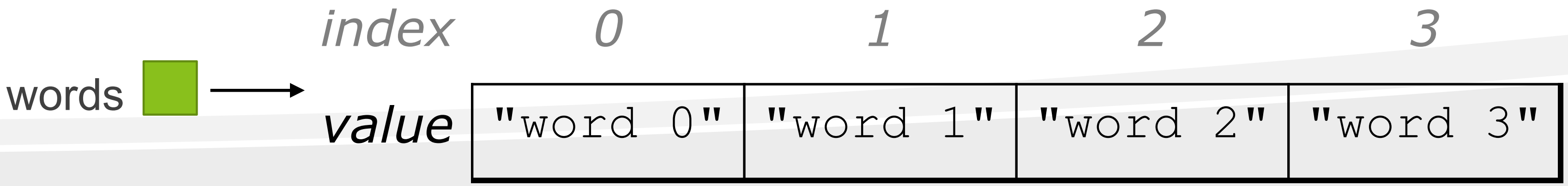
words →

index	*0*	*1*	*2*	*3*	*4*
value	"hello"	*null*	"goodbye"	*null*	*null*

Two-phase initialization

- Initialize the array itself (each element is initially `null`)
- Initialize each element of the array to be a new object

```
String[] words = new String[4];     // phase 1
for (int i = 0; i < words.length; i++) {
      words[i] = "word " + i;         // phase 2
}
```



words

index	0	1	2	3
value	"word 0"	"word 1"	"word 2"	"word 3"

Method Overloading

Method Overloading

- Java allows you to **overload a method**
- **Method overloading** is a feature that allows a class to have more than one method with the same name but different argument lists
    - Note: overloaded methods can only differ in their parameters **not the return types**
- **Constructor overloading** allows a class to have more than one constructor with different argument lists

Method Overloading

- There are three ways to overload a method
  - Number of parameters

    ```
    add (int, int)
    add (int, int, int)
    ```

  - Data type of parameters

    ```
    add (int, int)
    add (int, double)
    ```

  - Sequence of data type of parameters

    ```
    add(int, double)
    add(double, int)
    ```

Method Overloading: example

- Method 1:

```
public double calcInt(double balance, double rate){
        return balance * rate;
}
```

```
calcInt(1000.00, 0.04)
```

- Method 2:

```
public double calcInt(double balance, int rate){
     double ratePercent = rate/100.0;
     return balance * ratePercent;
}
```

```
calcInt(1000.00, 4)
```

Method Overloading: example (cont.)

- Method 1:

```
public double calcInt(double balance, double rate){
        return balance * rate;
}
```

Could `calcInt(1000.00, 4)` call method 1?

- Method 2:

```
public double calcInt(double balance, int rate){
     double ratePercent = rate/100.0;
     return balance * ratePercent;
}
```

Compiler recognizes a more exact match for the method call that uses the integer parameter and uses method 2

Method Overloading: example (cont.)

- Let’s assume we only have this method:

```
public double calcInt(double balance, double rate){
        return balance * rate;
}
```

- What happens if you call `calcInt(1000.0, 4)`?
  - The method still compiles, and it works (but not correctly)
  - Compiler will cast 4(integer) to 4.0

- When a data type of smaller size is promoted to the data type of bigger size then this is called type promotion

Method Overloading: example (cont.)

- Let's assume we have the following methods:

```
public double calcInt(int balance, double rate)
public double calcInt(double balance, int rate)
```

- What happens if you call `calcInt (300, 6)`?
  - There is no exact match! Compiler will complain

- **There is always risk when overloading methods. But still it is considered good programming style (more convenient)**

The `final` keyword for parameters

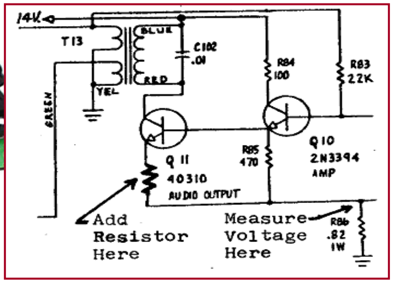
```
public double calcInt(final int balance, double rate)

public double calcInt(int balance, double rate)
```

- `final` keyword means the balance parameter is meant not to be changed inside this method
  - balance is considered a constant within the method `calcInt`
- These two methods are not overloaded. The compiler assumes they are the same
  - results in compiler/syntax error.

Encapsulation

Encapsulation

- **Definition** **Encapsulation** refers to the concepts of hiding implementation details of an object from the clients of the object



- Protects the integrity of an object's data
- Focusing on the iPod’s external behavior enable us to use it easily while ignoring the details of its inner workings

Encapsulation (cont.)

- Encapsulation is a principle of wrapping data (variables) and code together as a single unit
- It is one of the four OOP concepts
  - Encapsulation
  - Inheritance
  - Polymorphism
  - Abstraction

Encapsulation example 1

```
public class Account {
    int account_number;
    int account_balance;
    ...
    public void showData() {
        //code to show data
    }
    public void deposit(int a) {
       account_balance = account_balance + a;
    }
    ...
}
```

- Suppose a hacker managed to gain access to the code of your bank account and she tries to deposit amount -100.
- Is that possible?

Encapsulation example 1 (cont.)

```
public class Hacker {
    public static void main (String[] args){
        ...
        Account a = new Account();
        a.account_balance = -100;
        ...
    }
}
```

- The whole idea behind encapsulation is to hide the implementation details from users

`private` Fields

- To encapsulate the fields of an object, so they cannot be accessed from outside the class they need to be declared `private`

- **Syntax**

```
private <type> <name>;
```

- If a field is `private` it means it can only be accessed within the same class
- No outside class can access `private` data member of other classes

Back to example 1

```
public class Hacker {
   ...
   Account a = new Account();
   a.account_balance = -100;
}
```

```
public class Account {
   private int account_number;
   private int account_balance;

   public void showData() {
      //code to show data
   }

   public void deposit(int a) {
       account_balance = account_balance + a;
   }
}
```

- Suppose a hacker managed to gain access to the code of your bank account and she tries to deposit amount -100.
- Is that possible?

```
public class Hacker {
   ...
   Account a = new Account();
   a.account_balance = -100;
}
```

Back to example 1

```
public class Account {
   private int account_number;
   private int account_balance;

   public void showData() {
      //code to show data
   }

   public void deposit(int a) {
       account_balance = account_balance + a;
   }
}
```

Approach 1

- Fields are private, it means they can only be accessed within the same class

Back to example 1

```
public class Hacker {
    ...
    Account a = new Account();
    a.deposit(-100);
}
```

```
public class Account {
    private int account_number;
    private int account_balance;

    public void showData() {
        //code to show data
    }

    public void deposit(int a) {
        account_balance = account_balance + a;
    }
}
```

- Suppose a hacker managed to gain access to the code of your bank account and she tries to deposit amount -100.
- Is that possible?

Back to example 1

```
public class Hacker {
   ...
   Account a = new Account();
   a.deposit(-100);
}
```

```
public class Account {
   private int account_number;
   private int account_balance;

   public void showData() {
      //code to show data
   }

   public void deposit(int a) {
       if (a < 0){
          //show error
       } else {
          account_balance = account_balance + a;
       }
}
```

Back to example 1

```
public class Hacker {
    ...
    Account a = new Account();
    a.deposit(-100);
}
```

```
public class Account {
    private int account_number;
    private int account_balance;

    public void showData() {
        //code to show data
    }

    public void deposit(int a) {
         if (a < 0){
            //show error
         } else {
            account_balance = account_balance + a;
         }
}
```

Approach 2

- The `deposit` method has a check for negative values. Approach 2 fails

Back to example 1

```
public class Account {
   private int account_number;
   private int account_balance;

   public void showData() {
      //code to show data
   }

   public void deposit(int a) {
       if (a < 0){
          //show error
       } else {
          account_balance = account_balance + a;
       }
}
```

- Approach 1 and Approach 2 fail
- You never expose your data to an external party (which makes your application secure)
- The entire code can be thought as capsule

Point Class (ver. 6)

Point.java

```
public class Point{
  private int x;
  private int y;

  // constructor
  public Point(int initialX, int initialY){
    x = initialX;
    y = initialY;
  }

  // constructor
  public Point(){
    x = 0;
    y = 0;
  }

  // shifts points location by the given amount
  public void translate (int dx, int dy){
    x += dx;
    y += dy;
  }

  // computes the distance between two points
  public double distance(Point other){
    int dx = x - other.x;
    int dy = y - other.y;
    return Math.sqrt(dx * dx + dy * dy);
  }
…
```

Point.java (cont.)

```
…
   // computes the distance between a point and the origin
  public double distanceFromOrigin() {
    Point origin = new Point();
    return distance(origin);
  }

  public boolean equals(Object o) {
    if (o instanceof Point) {
      Point other = (Point) o;
      return x == other.x && y == other.y;
    } else {
      return false;
    }
  }

  public String toString(){
    return "(" + x + " , " + y + ")";
  }
}
```

`private` Fields

- Declaring fields `private` encapsulates the state of the object
- `private` fields are visible to all the code inside the `Point` class, but not anywhere else

```
public class PointMain {
     public static void main(String[] args){
          //Create a Point objects
          Point p1 = new Point(5, 2);

          //Print each point
          System.out.println("p1.x is "+ p1.x);
     }

}
```

```
PointMain.java:6: error: x has private access in Point
        System.out.println("p1.x is " + p1.x);
                                          ^
1 error
```

Accessing `private` fields

- Data members declared private can only be accessed within the same class
- No outside class can access them
- If you need to access these variables, you must use public "getter" and "setter" methods
  - The “getter” are used to **retrieve** fields
  - The ”setter” are used to **modify** fields

get and set for the Account class

```
public class Account{
        private int account_number;
        private int account_balance;

        // getter method
        public int getBalance() {
           return this.account_balance;
        }

        // setter method
        public void setNumber(int num) {
           this.account_number = num;
        }
}
```

Accessing `private` fields

- We need to provide a way for the client code to access/set a Point object’s field values

```
//A "read-only" access to the x field ("accessor")
public int getX(){
        return x;
}
```

```
// Allows clients to change the x field("mutator")
public void setX(int newX){
        x = newX;
}
```

- Client code will look more like this:

```
System.out.println(p1.getX());
p1.setX(14);
```

get and set methods

- Typically are used to retrieve or modify fields of a class
- Not all fields need a get /set methods
  - BUT, if you want to make sure that you restrict how your client programs can get or change fields you should think about using these methods
- Example `Point` class

```
int getX() {return x;}

int getY() {return y;}

void setX(int xVal) {x = xVal;}

void setY (int yVal) {y = yVal;}
```

Not very useful because usually we change both coordinates with `setLocation(int x, int y)`

Template of a well encapsulated object

```
public class <class name> {
   // fields
   private <type> <name>;
   private <type> <name>;
   ....
   //constructors
   public <class name>(<type> <name>, ...,<type> <name>) {
           <statement>;
           .....
   }
   //methods
   public <type> <name>(<type> <name>,...,<type> <name>){
           <statement>;
           .....
   }
}
```

1. fields on top & private

2. constructors

3. methods

Point class

```
public class Point{
     private int x;
     private int y;

     public Point(){
          this(0, 0);
     }

     public Point(int x, int y){
          setLocation(x, y);
     }

     public double distanceFromOrigin(){
          return Math.sqrt(x * x + y * y);
     }

     public int getX(){
          return x;
    }
…
```

```
…
     public int getY(){
          return y;
     }

     public void setLocation(int x, int y){
          this.x = x;
          this.y = y;
     }

     public String toString(){
          return "(" + x + "," + y + ")";
     }

     public void translate (int dx, int dy){
          setLocation(x + dx, y + dy);
    }


…
}
```