University of British Columbia CPSC 111, Intro to Computation 2009W2: Jan-Apr 2010

Tamara Munzner

Objects, Class Design

Lecture 8, Fri Jan 22 2010

borrowing from slides by Paul Carter and Wolfgang Heidrich

http://www.cs.ubc.ca/~tmm/courses/111-10

News

- If you have a midterm conflict with first midterm, let me know by end of day on Monday at the latest
  - Mon 2/8 6:30-8pm

Recap: Primitive Types vs. Classes

Primitive Types	Classes
Pre-defined in Java	Written by other programmers or by you
Simplest things, e.g., `int`	Can be arbitrarily complex
Operators: +, -, …	Methods
Values belong to types. E.g., 3 is an `int`, 3.14159 is a `double`	Objects belong to classes E.g., you are a UBC Student
Literals	Constructors

Recap: String - Literal or Constructor

```
public class StringTest
{
    public static void main (String[] args)
    {
        String firstname;
        String lastname;
        firstname= “Kermit”;
        lastname = new String (“the Frog");
        System.out.println("I am not " + firstname
                                   + " " + lastname);
    }
}
```

String is the only class that supports both literals and constructors!

Recap: Importing Packages

- Collections of related classes grouped into packages
  - tell Java which packages to keep track of with import statement
  - again, check API to find which package contains desired class
- No need to import `String`, `System.out` because core `java.lang` packages automatically imported

Recap: Scanner Class Example

```
import java.util.Scanner;

public class Echo
{
    public static void main (String[] args)
    {
        String message;
        Scanner scan = new Scanner (System.in);
        System.out.println ("Enter a line of text: ");
        message = scan.nextLine();
        System.out.println ("You entered: \""
                             + message + "\"");
    }
}
```

- Print out the message on the display

Scanner Class Example

- Let’s try running it

Scanner Class Methods

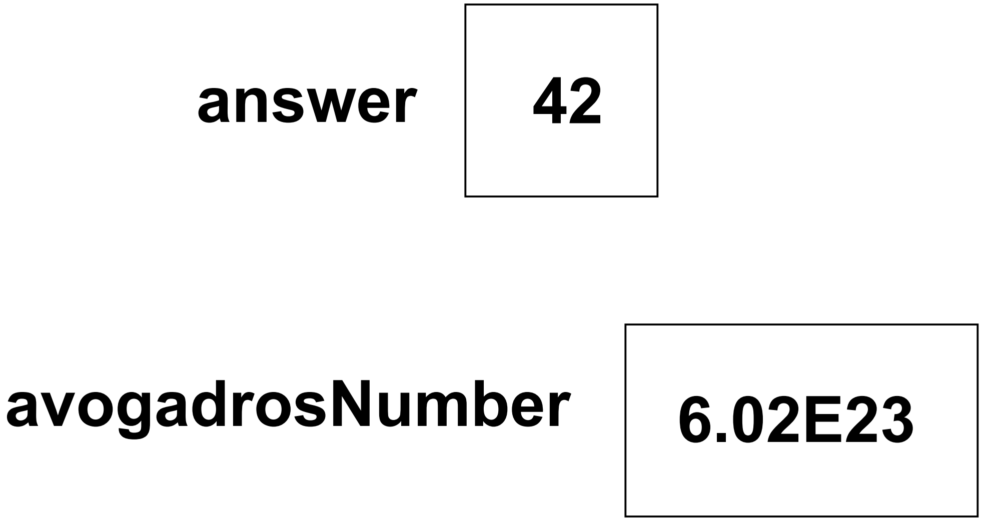
- The Scanner class has other methods to read other kinds of input, e.g.,
  - nextInt()
  - nextDouble()
- See section 4.7 in your book for more.

More on Object References

- Important distinction
  - For primitive types, variables hold the value.
  - For classes, variables hold reference to object

Primitive Types: Variables Hold Values

- Java primitive types are small and simple.
- Java variables hold values for primitive types.

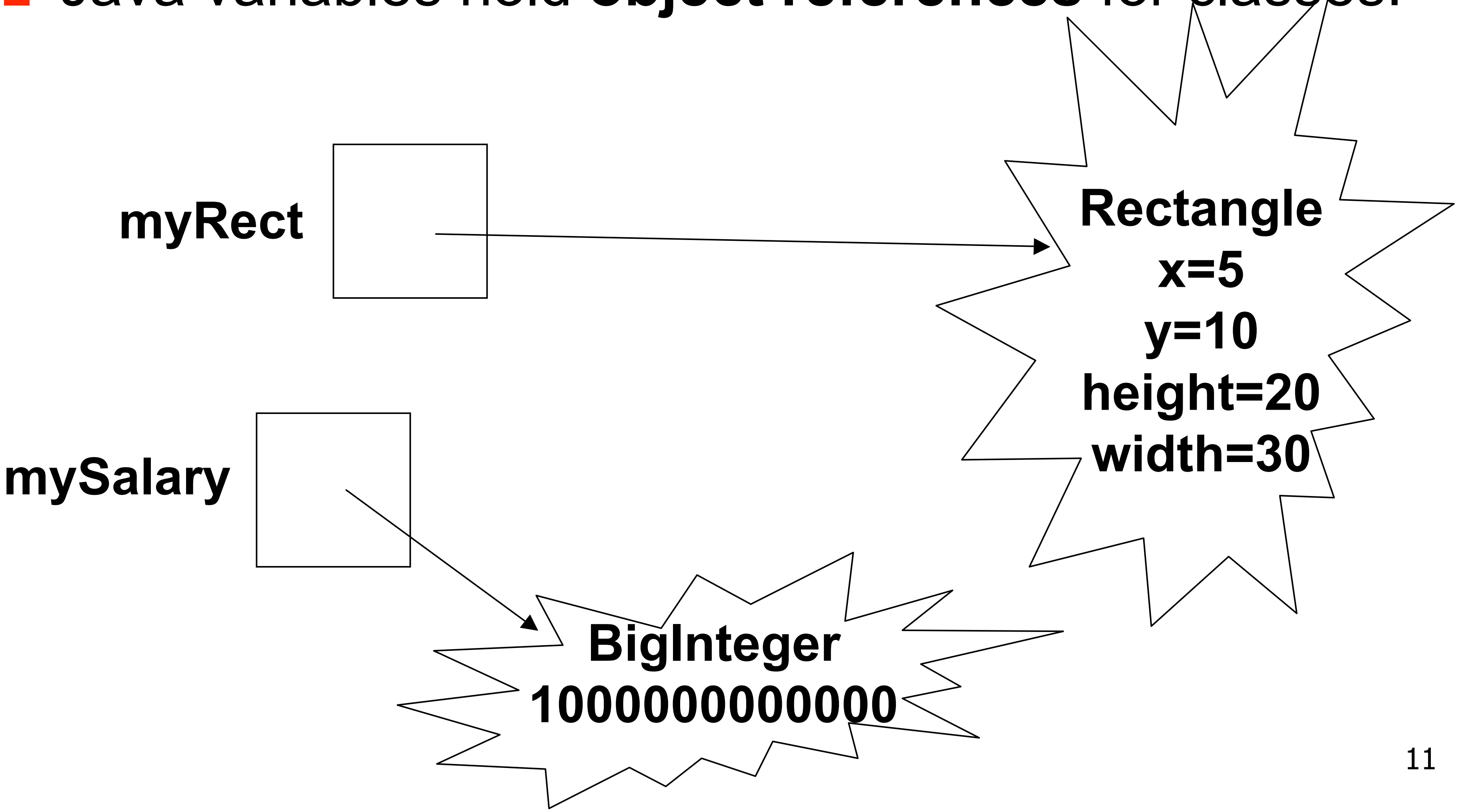


answer 42

avogadrosNumber 6.02E23

Classes: Variables Hold References

- Classes can be arbitrarily big and complex
- Java variables hold **object references** for classes.



Why Care About References vs Values?

- You copy a CD for your friend. Her dog chews it up. Does that affect your CD?
- You and your friend start eating a slice of cake on one shared plate. You get up to make a cup of tea. Her dog jumps on the table and eats the cake. Does that affect your half of the dessert?

Why Care About References vs Values?

- Example using primitive types:

```
int a;
int b;

a= 3;
b= a;
b= b+1;
System.out.println( "a= " + a + " and b=
  " +b );
```

Why Care About References vs Values?

- Example using objects:

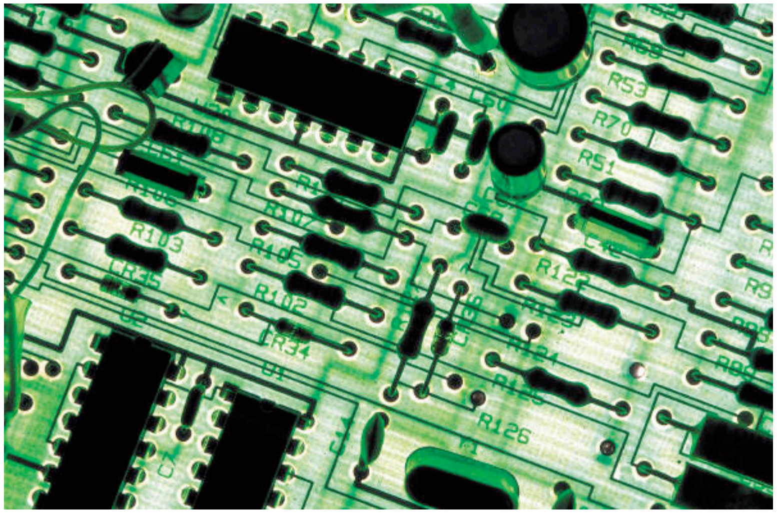
```
Rectangle a;
Rectangle b;

a = new Rectangle(3, 4);
b = a;
b.setSize(5,6);
System.out.println( "a= " + a.getHeight()+
                   ","+a.getWidth()+
                    " and b= " +b.getHeight()+
                    ","+b.getWidth());
```

Creating Classes

- So far you’ve seen how to use classes created by others
- Now let’s think about how to create our own
- Example: rolling dice
  - doesn’t exist already in Java API
  - we need to design
  - we need to implement
- Start with two design principles

Abstraction

- Abstraction: process whereby we
  - hide non-essential details
  - provide a view that is relevant
- Often want different layers of abstraction depending on what is relevant



Encapsulation

- Encapsulation: process whereby
  - inner workings made inaccessible to protect them and maintain their integrity
  - operations can be performed by user only through well-defined interface.
  - aka information hiding
- Cell phone example
  - inner workings encapsulated in hand set
    - cell phone users can't get at them
  - intuitive interface makes using them easy
    - without understanding how they actually work

Information Hiding

- Hide internal details from user of object.
  - maintains integrity of object
  - allow us flexibility to change them without affecting users
- Parnas' Law:
  - "Only what is hidden can by changed without risk."

Designing `Die` Class

- Blueprint for constructing objects of type **`Die`**
- Think of manufacturing airplanes or dresses or whatever
  - design one blueprint or pattern
  - manufacture many instances from it
- Consider two viewpoints
  - client programmer: wants to use **`Die`** object in a program
  - designer: creator of **`Die`** class

Client Programmer

- What operations does client programmer need?
  - what methods should we create for **`Die`**?

Designing Die

```
public class Die
{










}
```

Designing `Die` -- Better

```
/**
  Provides a simple model of a die
  (as in pair of dice).
*/
public class Die
{






}
```

Designer

- Decide on inner workings
  - implementation of class
- Objects need state
  - attributes that distinguish one instance from another
  - many names for these
    - state variables
    - fields
    - attributes
    - data members
  - what fields should we create for **`Die`**?

Implementing `Die`

```
/**
  Provides a simple model of a die
  (as in pair of dice).
*/
public class Die
{





}
```

Random Numbers

- `Random` class in `java.util` package
  - `public Random()`
    - Constructor
  - `public float nextFloat()`
    - Returns random number between 0.0 (inclusive) and 1.0 (exclusive)
  - `public int nextInt()`
    - Returns random integer ranging over all possible int values
  - `public int nextInt( int num )`
    - Returns random integer in range 0 to (num-1)

Implementing `Die`

```
/**
  Provides a simple model of a die
  (as in pair of dice).
*/
public class Die
{




}
```

`return` Statement

- Use the **`return`** statement to specify the return value when implementing a method:

```
 int addTwoInts (int a, int b) {
    return a+b;
 }
```

- Syntax: **`return`** *expression* **;**
- The method stops executing at that point and "returns" to caller.

Implementing Die

```
/**
  Provides a simple model of a die
  (as in pair of dice).
*/
public class Die
{





}
```

Information Hiding

- Hide fields from client programmer
  - maintain their integrity
  - allow us flexibility to change them without affecting code written by client programmer
- Parnas' Law:
  - "Only what is hidden can by changed without risk."

Public vs Private

- `public` keyword indicates that something can be referenced from outside object
  - can be seen/used by client programmer
- `private` keyword indicates that something cannot be referenced from outside object
  - cannot be seen/used by client programmer
- Let’s fill in public/private for `Die` class

Public vs. Private Example

```
public class Die {
...
  public int roll()
  ...
  private void cheat(int nextRoll)
...
}
```

Public vs. Private Example

```
Die myDie = new Die();

int result = myDie.roll();  // OK
myDie.cheat(6);           //not allowed!
```

Implementing Die

```
/**
  Provides a simple model of a die
  (as in pair of dice).
*/
public class Die
{







}
```

Trying It Out!

- `Die` class has no main method.
- Best is to write another class that instantiates some objects of your new class and tries them out.
  - Sometimes called a “tester” or “testbench”

Implementing `RollDice`

```
public class RollDice
{
   public static void main ( String [] args)
   {









}
```