



University of British Columbia
CPSC 111, Intro to Computation
Jan-Apr 2006

Tamara Munzner

Class Design III

Lecture 8, Tue Jan 31 2006

based on slides by Paul Carter

<http://www.cs.ubc.ca/~tmm/courses/cpsc111-06-spr>

Reading This Week

- Chap 3 (today)
- Re-read Chapter 4.3-4.5 (Thursday)
- reminder - code examples created in class posted by slides and assigned reading

News

- Assignment 1 due today 5pm
- Wed office hours 11:30-12:30 not 11-12
 - reminder: in X661
- Windows home setup guide posted to WebCT
- Reminders
 - CSLC is available if you need help
 - Check ugrad email account regularly (or forward to active account)
 - grade info sent there

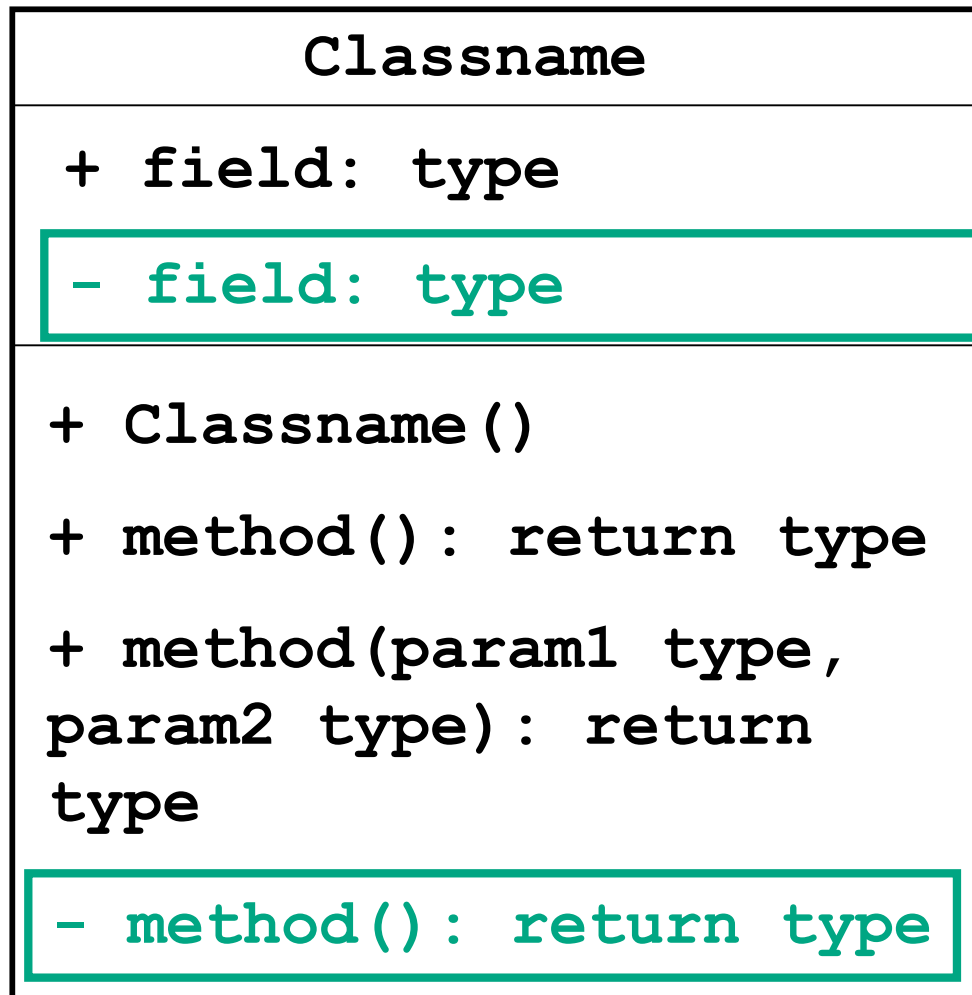
Exam

- Midterm reminder: Tue Feb 7, 18:30 - 20:00
 - Geography 100 & 200
- Exam conflict: email me today

- DRC: Disability Resource Center
 - academic accomodation for disabilities
 - forms due one week before exam (today!)
 - <http://students.ubc.ca/access/drc.cfm>

Correction: UML

- UML diagram representing class design

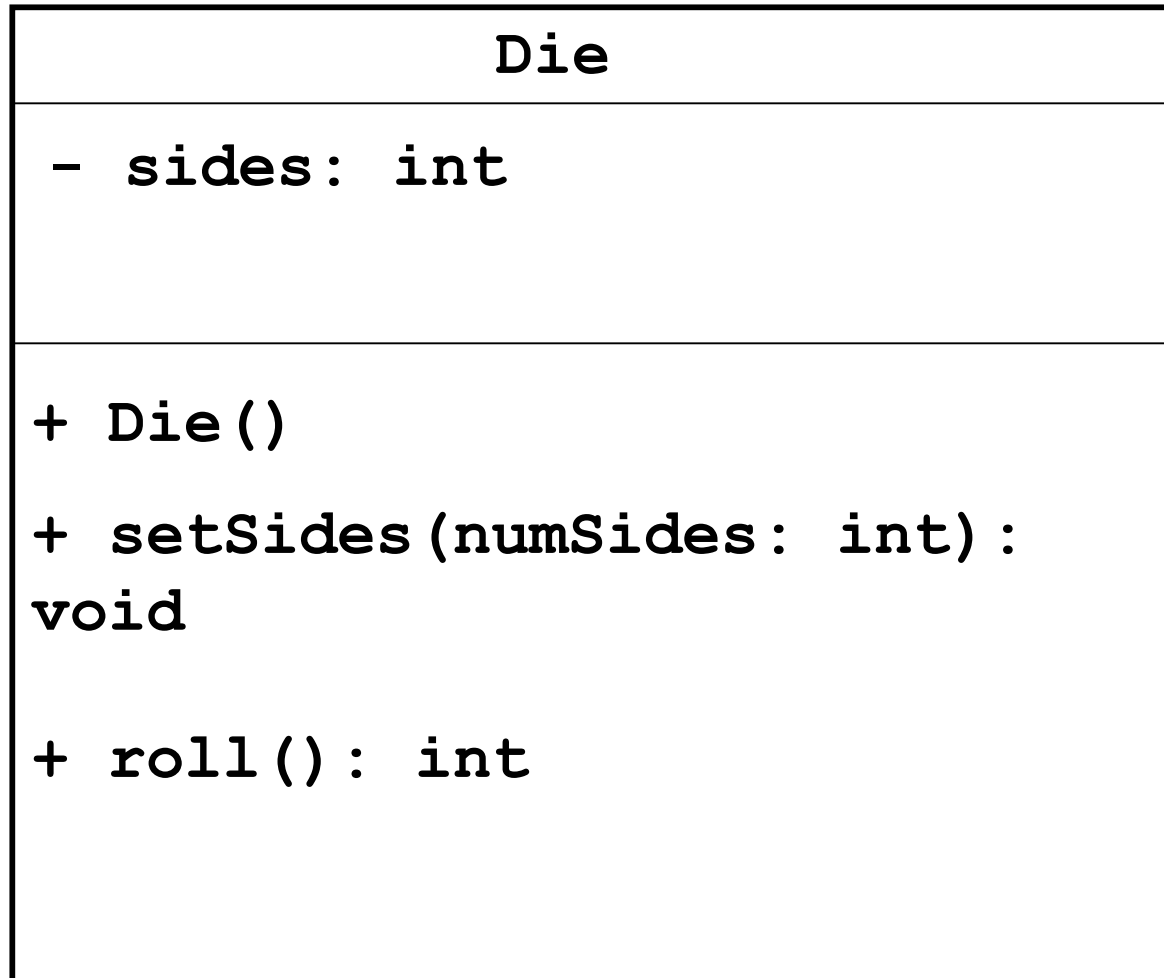


fields

methods

Recap: UML

- UML diagram for `Die` class we designed



fields

methods

Objectives

- understand how to design new classes using abstraction and encapsulation
- understand how to implement new classes in Java
- understand how to comment classes using javadoc conventions
- understand how to create documentation using javadoc
- understand how to finish refining code

Recap: Separation and Modularity

- Design possibilities
 - `Die` and `RollDie` as separate classes
 - one single class that does it all
- Separation allows code **re-use** through **modularity**
 - another software design principle
- One module for **modeling** a die: `Die` class
- Other modules can **use** die or dice
 - we wrote one, the `RollDice` class
- Modularization also occurs at file level
 - modules stored in different files
 - also makes re-use easier

Recap: Control Flow Between Modules

- So far, easy to understand **control flow**: order in which statements are executed
 - march down line by line through file
- Now consider control flow between modules

Client code

```
int rollResult;  
myDie.setSides();  
rollResult = myDie.roll();
```

Die class methods

```
public int roll()  
{  
    ...  
}  
  
public void setSides()  
{  
    ...  
}
```

Key Topic Summary

Borrowed phrasing from Steve Wolfman

- Generalizing from something concrete
 - fancy name: abstraction
- Hiding the guts from the outside
 - fancy name: encapsulation
- Keeping one part from stomping on another
 - fancy name: modularity
- Breaking down a problem
 - fancy name: functional decomposition

Implementing Point and PointTest

```
public class Point {
```

```
}
```

Commenting Code

■ Conventions

- explain what classes and methods do
- plus anywhere that you've done something nonobvious
 - often better to say why than what

- not useful

```
int wishes = 3; // set wishes to 3
```

- useful

```
int wishes = 3; // follow fairy tale convention
```

javadoc Comments

- Specific format for method and class header comments
 - running javadoc program will automatically generate HTML documentation
- Rules
 - `/**` to start, first sentence used for method summary
 - `@param` tag for parameter name and explanation
 - `@return` tag for return value explanation
 - other tags: `@author`, `@version`
 - `*/` to end
- Running

```
% javadoc Die.java
% javadoc *.java
```

javadoc Method Comment Example

```
/**
 * Sets the die shape, thus the range of values it can roll.
 * @param numSides the number of sides of the die
 */
public void setSides(int numSides) {
    sides = numSides;
}

/**
 * Gets the number of sides of the die.
 * @return the number of sides of the die
 */
public int getSides() {
    return sides;
}
```

javadoc Class Comment Example

```
/** Die: simulate rolling a die
 * @author: CPSC 111, Section 206, Spring 05-06
 * @version: Jan 31, 2006
 *
 * This is the final Die code. We started on Jan 24,
 * tested and improved in on Jan 26, and did a final
 * cleanup pass on Jan 31.
 */
```

Cleanup Pass

- Would we hand in our code as it stands?
 - good use of whitespace?
 - well commented?
 - every class, method, parameter, return value
 - clear, descriptive variable naming conventions?
 - constants vs. variables or magic numbers?
 - fields initialized?
 - good structure?
 - follows specification?
- ideal: do as you go
 - commenting first is a great idea!
- acceptable: clean up before declaring victory

Formal vs. Actual Parameters

- **formal** parameter: in declaration of class
- **actual** parameter: passed in when method is called
 - variable names may or may not match
- if parameter is primitive type
 - **call by value**: value of actual parameter copied into formal parameter when method is called
 - changes made to formal parameter inside method body will not be reflected in actual parameter value outside of method
- if parameter is object: covered later

Scope

- Fields of class are have **class scope**: accessible to any class member
 - in `Die` and `Point` class implementation, fields accessed by all class methods
- Parameters of method and any variables declared within body of method have **local scope**: accessible only to that method
 - not to any other part of your code
- In general, scope of a variable is block of code within which it is declared
 - **block** of code is defined by braces { }