HTA Lecture

Nickie, Marley, Alexa, & Nick
Essential Java Topics

Visibility Modifiers, Final & Static Keywords, Anonymous Classes, Exceptions
Visibility Modifiers (1/2)

● Can apply to classes, methods, and instance variables
● **public** can be accessed from any class – use for methods (not variables) that you want to expose, i.e., all but a few **private** helper methods
● **protected** can be accessed from a subclass or any other class in the package (can’t apply to classes, just methods and variables)
  o Java unfortunately doesn’t have an access level which only allows the subclass access
  o Some of you discovered **protected** while doing Tetris. We do not encourage its use because it violates encapsulation principles (we’ll take off if we see it in final projects)
Visibility Modifiers (2/2)

● Default or package protected (no access modifier) can be accessed from any class within the package
  ○ Violates encapsulation as well, and not allowed in CS15

● `private` can only be accessed within the class and any inner class
What’s so bad about public?

- **public** instance variables can be accessed using the same “dot notation” as methods:
  - `<instance>.<variable name>`

- **But**, in Java the inheritance rules for instances variables and methods are different

- What’s printed by this printline?

- **public** instance variables are bad style for a reason!

```java
public class A {
    public int _x = 1;
    public int f() {
        return _x + 10;
    }
    public int g() {
        return _x + this.f();
    }
}
public class B extends A {
    public int _x = 100;
    public int f() {
        return _x;
    }
}
/* additional code elided */
A ab = new B();
System.out.println(ab._x + ab.g());
```
Final Classes and Methods (1/2)

- **final** keyword can be used for classes, methods, and variables

- Similar meaning for classes and methods, but different for variables

- A **final** class cannot be subclassed and a **final** method cannot be overridden

```java
public final class DoNotSubclass{
    //some code
}

public class Example {
    public final void reallyComplicated(){
        //some code
    }
}
```
Final Classes and Methods (2/2)

- Whenever a class is not intended by the programmer to be subclassable then it should be final
  - Ex: PaneOrganizer

- A method should be final if it is integral to the proper functioning of the class

- Prevents someone else from incorrectly trying to extend a class or override a method

```java
public final class DoNotSubclass{
    //some code
}

public class Example {
    public final void reallyComplicated(){
        //some code
    }
}
```

Rule of thumb: Design for inheritance or prohibit it!*

* Source: Effective Java (Joshua Bloch)
Final Variables (1/2)

- You have used `static final` variables when declaring constants
- A `final` variable can only be assigned a value once and cannot be reassigned
- Must be assigned by the time the object has been constructed

```java
public class Example {
    public final int _myInt;
    public final Point _myPoint;

    public Example() {
        _myPoint = new Point(5,6);
        _myPoint.move(5,0); //okay
        _myPoint = new Point(2,3); //error!
        //other error: _myInt not assigned
    }
}
```
Final Variables (2/2)

- If the variable is a reference to an object, the variable cannot be changed but the object can still be mutated

- If you know that a variable should never be changed, good practice to make it final
  - Ex: your `Timeline`

```java
class Example {
    public final int _myInt;
    public final Point _myPoint;

    public Example() {
        _myPoint = new Point(5,6);
        _myPoint.move(5,0); //okay
        _myPoint = new Point(2,3); //error!
        //other error: _myInt not assigned
    }
}
```

Rule of Thumb: **Minimize Mutability!**

* Source: *Effective Java* (Joshua Bloch)
Static Methods (1/2)

- You’ve already seen static methods. Ex:
  - `Math.random()`
  - `Color.rgb(...)`

- **Static** methods are invoked on a class, not an instance
  - So no need to ever call “new Math()” etc.
  - Also means that they cannot access instance variables, since those could be different for each instance
Static Methods (2/2)

- You probably haven’t written a static method yourself so far
  
  - Syntax: `<visibility> static <return type> (<parameters){   }`  

- When should make your method static?
  
  - Following two conditions should be met
    - It doesn’t modify program state
    - The result that you are returning depends on nothing but the parameters that you are taking in
  
  - Ex: Say you were writing a Geometry class that helps you do math with points
    - `public static double getDistanceBetweenPoints(Point2D p1, Point2D p2)`
Static Variables (1/2)

- Progression in scope:
  - Local variables are known in a single method
  - Instance variables are known to all methods of a class
  - **static** variables are known to all instances of a class (sometimes referred to as **class variables**)

- If we want all instances of a class to share the same value for a variable, declare it **static**

```java
public class CS15Student {
    private static int _numStudents = 0;

    public CS15Student (){
        _numStudents++;
    }

    public static int getNumStudents(){
        return _numStudents++;
    }
}
```

From outside the class, we can call `System.out.println (CS15Student.getNumStudents())` to print the current number of CS15Student instances.
Static Variables (2/2)

- Be careful with these!
  - not very object oriented - violates object encapsulation
  - cannot be garbage collected

- Therefore, generally only use in combination with final for constants

```java
class CS15Student {
    private static int _numStudents = 0;

    public CS15Student (){
        _numStudents++;
    }

    public static int getNumStudents(){
        return _numStudents++;
    }
}
```

From outside the class, we can call `System.out.println(CS15Student.getNumStudents())` to print the current number of CS15Student instances.
Anonymous Classes (1/2)

- Anonymous classes are classes without names
- Allow you to define a class and instantiate it in one step
- Use when you need only one instance of an inner class
- Has to either implement an interface or extend from another class

Using a Predefined Class

```java
stage.setScene(new Scene(root));
```

Using an Anonymous Class

```java
button.addEventHandler(new EventHandler<ActionEvent>() {
    @Override
    public void handle(ActionEvent e) {
        System.exit(0);
    }
});
```
Anonymous Classes (2/2)

Pros

● Can use in place of an inner class
● Allows you to define a subclass or implement an interface on the fly
● Super concise

Cons

● Can make your code more difficult to read
● Code is not reusable
● makes Alexa sad :
Anonymous Class Syntax (1/5)

- Syntax:
  ```java
  new X() {
      // class definition
  }
  ```

- Where X is the superclass of or the interface implemented by the anonymous class
Anonymous Class Syntax (2/5)

Using Private Inner Classes:

```java
public class PaneOrganizer {

    public PaneOrganizer() {
        /* initialization code elided */
        button.addEventHandler(new QuitHandler());
    }

    private class QuitHandler implements EventHandler<ActionEvent> {
        @Override
        public void handle(ActionEvent e) {
            System.exit(0);
        }
    }
}
```
Anonymous Class Syntax (3/5)

Using Anonymous Classes:

```java
public class PaneOrganizer {
    public PaneOrganizer() {
        /* initialization code elided */
        button.addEventHandler(new EventHandler<ActionEvent>() {
            @Override
            public void handle(ActionEvent e) {
                System.exit(0);
            }
        });
    }
}
```
Anonymous Class Syntax (4/5)

Side by Side:

```java
public class PaneOrganizer {
    public PaneOrganizer() {
        /* initialization code elided */
        button.addEventHandler(
            new QuitHandler());
    }

    private class QuitHandler implements EventHandler<ActionEvent> {
        @Override
        public void handle(ActionEvent e) {
            System.exit(0);
        }
    }
}
```

```java
public class PaneOrganizer {
    public PaneOrganizer() {
        /* initialization code elided */
        button.addEventHandler(
            new QuitHandler());
    }

    private class QuitHandler implements EventHandler<ActionEvent> {
        @Override
        public void handle(ActionEvent e) {
            System.exit(0);
        }
    }
}
```
Anonymous Class Syntax (5/5)

- If the superclass constructor requires parameters, pass in the appropriate arguments inside the parentheses right before the first {

```java
public abstract class Person {
    public Person(String name) {
        // constructor elided
    }
    public abstract void talk();
}
```

```java
public class PersonApp {
    public PersonApp() {
        Person andy = new Person("Andy van Dam") {
            @Override
            public void talk() {
                System.out.println("The pen’s not working!");
            }
        };
        Person oussama = new Person("Oussama") {
            @Override
            public void talk() {
                System.out.println("Variables aren’t constant!");
            }
        };
    }
}
```
Anonymous Classes: Usage

When to use

● You only need one instance of the inner class
● In JavaFX, commonly used when defining EventHandlers

When not to use

● The class requires a lot of lines (> 5) of code
● You want to create multiple instances of the class
Anonymous Classes: Additional Resources

- The Java Tutorials: Anonymous Classes
- Passing Parameters to Anonymous Classes
- StackOverflow
  - Is usage of anonymous classes in Java considered bad style or good?
  - How are Anonymous (inner) classes used in Java?
Exceptions

Exception
Exceptions Overview

● Exceptions indicate that a fundamental assumption that your code is making is false, and therefore, the program cannot continue.
  ○ Examples:
    ■ Values inputted as parameters are unusable
      ● `Double.parseDouble("hello")` → “hello” can never be a double!
    ■ A method can’t finish
      ● Trying to read from a file that doesn’t exist

● This lecture:
  ○ Exceptional Situations
  ○ Syntax of `try, catch, throw`
  ○ Defining and Using Your Own Exceptions
Exceptional Situations

● Exceptions are helpful!
  ○ Provide the programmer with information about why the program cannot continue.
    ■ Example: `NullPointerException` indicates an action was attempted on a `null` reference.

● Exceptions allow the calling method to dictate how the exception should be handled.
  ○ This allows for programs to choose to continue running even when an exception is `thrown`.
    ■ Mechanics for doing so will be explained in the `try catch` slides.
Exceptional Situations in Java

- **Exceptions** are classes that extend **Throwable** and come in two types:
  1. Those that must be handled by the calling code.
     - E.g. **IOException**, an issue reading a file.
  2. Those that do not need to be handled.
    - **RuntimeExceptions** do not need to be handled.
    - E.g. **NullPointerExceptions**, **IndexOutOfBoundsExceptions**, and **ConcurrentModificationExceptions**

**Note:** We will not be covering **Errors** in this lecture! But they indicate systemic problems, like running out of available memory. Your code can cause **Errors**, but your code should not try to handle them.
Exception Handling Syntax (1/3)

- Until now, you have had no control over coping with exceptions. With a catch statement, you have the chance to implement your own exception handling

- Process for handling exceptions
  - try some code, catch exception thrown by tried code, finally, “clean up” if necessary
  - try, catch, and finally are reserved words
Exception Handling Syntax (2/3)

- **try** denotes code that may throw an exception
  - place questionable code within a **try block**.
  - a **try block** must be immediately followed by a **catch block**.
  - **try-catch** blocks always occurs as pairs.

- **catch** an exception if it’s thrown and write special code to handle it
  - catch blocks distinguished by **type** of exception
  - can have several **catch blocks**, each specifying a particular type of exception
  - Once an exception is handled, execution continues after the catch block

- **finally** (optional)
  - special block of code after the **catch block** that is executed whether or not an exception is thrown
Exception Handling Syntax (3/3)

- All parts enclosed in curly braces { }
- `try`, then `catch` then `finally` blocks
  - Put exception in parentheses as in method definition
- Formal parameter of type `java.lang.Exception` is the most general, as it would catch all exceptions (because they are all subclasses)

```java
/* program code elided */
try {
    /* questionable code that could potentially throw an exception */
}
catch (specific_exception_type name) {
    /* code to handle the exception */
}
...
finally {
    /* code that should happen regardless of whether an exception is thrown */
}
```
Exception Handling Example 1

- A method call within a `try` block may set off a chain of method calls, the last of which throws an exception
  - Andy tells Betty to `getADrink()`; Betty tells Alex to `getADrink()`. Alex is asleep and throws a `DrinkNotAvailableException` which is defined elsewhere
  - This exception is not a subclass of `RuntimeException`, so it should be caught

```java
public class Andy {
    /* properties, methods to teach, eat, kayak, etc, elided :) */
    public void getWater() {
        try {
            /* getADrink() might throw a DrinkNotAvailableException so we have to put it in a try block */
            _water = _betty.getADrink();
        }
        catch(DrinkNotAvailableException e){
            this.fire(_betty);
        }
    }
}
```
Exception Handling Example 2

- **try-catch** blocks can be nested!

- If Andy’s call to Betty to getADrink() throws DrinkNotAvailableException, he can ask Michael to getADrink().

- getADrink() may call another method that throws an exception and it will still be caught here - this is called “exception resolution”

/* Andy class code elided */

```java
public void getWater() {
    try {
        _water = _betty.getADrink();
    }
    catch (DrinkNotAvailableException e) {
        this.fire(_betty);
        try {
            _water = _michael.getADrink();
        }
        catch(DrinkNotAvailableException e){
            this.fire(_michael);
        }
    }
}
```
Exception Resolution

● Similar to method resolution in inheritance hierarchies: starts with the method that throws the exception.
  ○ Work back up the chain of method calls until a try-catch block is found for that exception (or a superclass of that exception)
  ○ So, you do not necessarily have to try and catch every exception at every level of calling.

● If an exception must be caught, then you’d better be sure that you catch the exception at some point!
  ○ If the exception is not caught, the program will crash or not perform as expected.
Creating Exceptions

- You can define and throw your own exceptions.
  - Useful for handling to special cases that are not accurately covered by pre-existing exceptions.

Example:

```java
public class GhostPenEmptyException extends Exception {
    public GhostPenEmptyException(String message) {
        super("Error: " + message);
    }
}
```

- The `String` passed to `super` is printed to the output window for debugging when `getMessage()` is called by the user.
Throwing Exceptions (1/2)

- Every method that throws an `Exception` and is not a subclass of `RuntimeException` must declare what exceptions it throws.
  - Remember! `RuntimeException`s don’t need to be handled.

```java
// Defined in the GhostPen class
public void releaseGhost() throws GhostPenEmptyException {
    if(_queue.isEmpty()) {
        throw new GhostPenEmptyException("Could not dequeue, GhostPen is empty");
    }
    /* elided */
}
```
Throwing Exceptions (2/2)

- When calling a method declared to \texttt{throw} an exception must have a \texttt{try} block surrounding the method call and an accompanying \texttt{catch} block to handle the possible exception.

```java
public void moveGhosts() {
    while(pacman.isAtLarge()) {
        try {
            _ghostPen.releaseGhost();
        } catch (GhostPenEmptyException e) {
            System.out.println(e.getMessage());
        }
    }
}
```
Exception: Pros and Cons

● Pros:
  ○ Cleaner code: rather than returning a `boolean` up chain of calls to check for exceptional cases, throw an exception!
  ○ Use return value for meaningful data, not error checking
  ○ Factor out error-checking code into one class, so it can be reused

● Cons:
  ○ Throwing exceptions requires extra computation
  ○ Can become messy if not used sparingly
  ○ Can accidentally cover up a serious exception, such as a `NullPointerException`, by catching them
In Conclusion

● Words of Wisdom:
  ○ Never try to “fix up” your program by catching all exceptions
    ■ “Oh… \texttt{NullPointerException}… let me just catch it, so the TAs
      won’t know I have buggy code! Hahahaha!!!”
  ○ Best to throw an exception when an error occurs that you cannot deal
    with yourself, but can be better handled by some method on the stack

Wow, what an Exception-al lecture!
CS Past 15

Industry, Research, Involvement at Brown, Culture
Working in Industry

● Lots of companies come to the Career Fairs (both Tech and General)
  ○ 73 came to the Tech Fair
  ○ 78 came to the General Fair
  ○ some overlap between the two fairs

● 28 of these companies have recruiting relationships with the Brown CS department through the Industry Partners Program

● Both tech and non-tech companies have opportunities for you!
Types of Careers in Industry (1/3)

● Software Engineer/Developer
  ○ Focus on creating and coding the software
  ○ Variety of specialties: Test, UI, etc.
  ○ Not a code monkey, not coding 10-12 hours a day
    ■ Often in meetings collaborating on design, setting requirements, and talking to prospective customers
    ■ Depends on company/job, so research/ask about it during process
Types of Careers in Industry (2/3)

• Project/Program/Product Manager
  ○ Focus defining what the product should be and what features it should have
  ○ Includes some level of project management/coordination
  ○ Work with both prospective users and software developers
  ○ Technical position
    ■ Some PMs code and make prototypes
  ○ Can’t just tell everyone what to do. Have to convince the engineers that your plans are the best for the product
Types of Careers in Industry (3/3)

- Other careers
  - UX (User Experience) Designer
  - UI (User Interface) Designer
    - **UX vs. UI**
  - Quality Assurance Engineer
  - Data Scientist
  - Systems Programmer
  - IT Architect
The Road to a Job/Internship*

- Research companies
- Submit resume
- Technical interview (phone/skype)
- Onsite Interview
- Offer

*approximate - some companies will have slightly different steps to receiving an offer!
Personal Experience (1/3)

- Google - [Engineering Practicum Intern](#)
- Google Plus - iOS
  - Poll rendering in the stream
- 40 hours/week
  - *not* always coding!
  - lots of design discussions, doc write-ups, asking questions
- Besides working…
  - intern lunches/events
  - team offsites
Personal Experience (2/3)

Things I Loved

- Co-workers’ willingness to help
- Wearing a t-shirt and shorts
- THE (FREE) FOOD

Things I Didn’t Love

- Size
  - ~11,000 people
  - ~1,700 interns
- Team vibe
  - but every team is different
- Intern events
  - not run super well
  - probably related to size
Personal Experience (3/3)

Things I Learned

● Never be afraid to ask questions!
● Be humble, but know that your inexperience can also be a strength - you may think of new ways to do things because you’re seeing the team’s problems for the first time
● Personal discoveries
  ○ I <3 Programming
  ○ Team dynamic is really important for me
  ○ Company size matters
  ○ Google is not the “end all be all”
Why Research?

- Often different from classwork, opens up the black box
- Get to know professors and undergraduates, M.Sc. and Ph.D. students better
- Lots of interdisciplinary opportunities
- Path to Graduate School
- Be independent and creative
- Learn how to read the latest research literature
- Beef up your resume
- FUN!
What Can You Do with Research?

- Get into Grad School! Get awesome jobs!
- On the path to becoming an Academic
- Work on cutting-edge problems in industry
- Microsoft Research (est. in 1991) is one of the largest, most respected software research organizations
  - They work in a variety of areas like systems, security, data mining and big data, computer graphics and computer vision, speech and natural language understanding, UI and UX, algorithms
  - Typically need a Ph.D. to work there, though some researchers do have developers working with them
  - GoogleX, IBM Research, FXPAL, …
How To Get Involved With Research (1/2)

● Ask specific professors about projects
  ○ the best way to find interesting options is by talking to professors
  ○ be prepared to talk about your interests and background

● Take a 1000-level course
  ○ Taking upper-level CS courses is a great way to explore your interests and discover what you like

● Take a graduate course
  ○ Often have a research component to it
How To Get Involved With Research (2/2)

● Many open opportunities for undergrads!
  ○ Attend the Town Hall meetings every semester
  ○ Browse cs.brown.edu for info about specific professors and projects

● Typically start after a few years of CS, but some faculty (e.g., Andy) take outstanding students after 15/16 or 17/18
Involvement at Brown (1/3)

● The UTA Program
  ○ 60% of Brown CS concentrators TA at least one semester
  ○ Applicants open in October for the next Spring semester and March for the next Fall semester
  ○ Can TA for credit, pay, or both

● Hack@Brown
  ○ Started and run by a team of undergrads
  ○ Run workshops and events year-round
  ○ Registration for this year open now!
Involvement at Brown (2/3)

● **WiCS** - Women in Computer Science
  - Mentorship program, meetings and events
  - Supports the *Artemis Project*, a free summer camp for rising ninth-grade girls from the Providence area who show interest in science and technology.

● **Mosaic+**
  - New student group this year, advocating for diversity within Brown's CS community
  - Big-little system, workshops, group study
Involvement at Brown (3/3)

● Collaboration in future CS classes!

● Some options for next semester
  ○ CS16: Algorithms and Data Structures
  ○ CS22: Discrete Math and Probability
  ○ CS180: Cybersecurity and International Relations
  ○ CS145: Probability and Computing

● No single Brown CS class is required!
Computer Science Culture at Brown

Myth: You have to eat/sleep/breathe Computer Science

My experience:

● Brown CS is incredibly friendly to studying abroad
● Take plenty of courses in other departments
● Don’t have to pull all nighters to be a “real” CS student
  ○ Note: There is no such thing as a “real” CS student
Computer Science Culture at Brown

Myth: An [ ScB / AB ] is the better option.

My experience:

- Really doesn’t matter
- Take classes that you find interesting!
In Conclusion

● Good luck with final projects and the rest of your semesters
  ○ You’ve got this!
  ○ Reflect on how much you’ve learned - go back and look at AndyBot, we promise it will be an enlightening experience

● We <3 Brown CS
  ○ This people in this department are awesome
  ○ This field is awesome
  ○ Computers are awesome
  ○ We <3 you