Week 14 - Friday

COMP 2000

Last time

- What did we talk about last time?
- Review up to Exam 2
  - Java GUIs
    - **`JOptionPane`**
    - **`JFrame`**
    - Widgets
    - Layout managers
    - Action listeners
  - Recursion
  - Files
    - Text I/O
    - Binary I/O
    - Serialization
  - Networking

Questions?

Project 4

Final exam

- Final exam will be held virtually:
  - Monday, April 27, 2020
  - 10:15 a.m. to 12:15 p.m.
- There will be multiple choice, short answer, and programming questions
- I recommend that you use an editor like Notepad++ to write your answers, since Blackboard doesn't play nice with tabs
- I **don't** recommend that you use Eclipse, since the syntax highlighting features will make you doubt yourself and try to get things perfect when getting them done is more important

Review after Exam 2

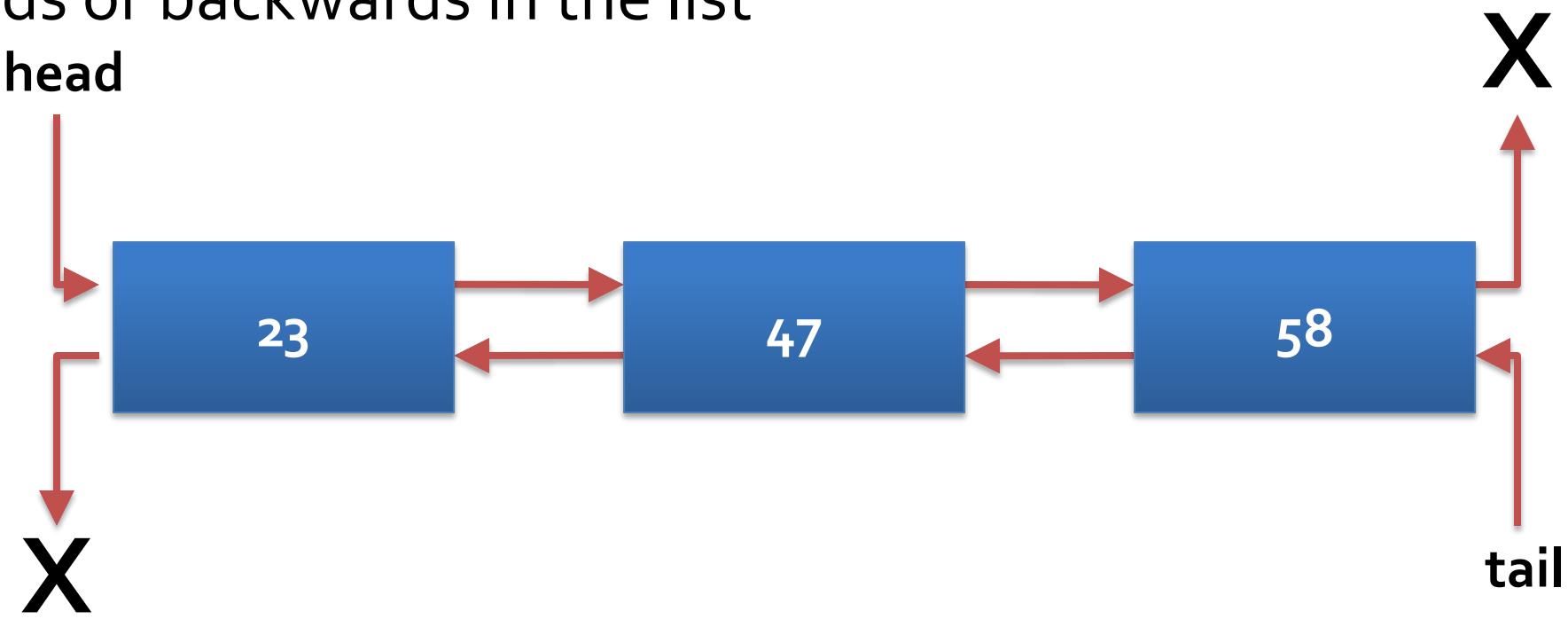
Dynamic Data Structures

Linked list

- A linked list is one of the simplest kinds of dynamic data structures
- You can imagine a linked list as a train
- Each node in the linked list has some cargo, and it can point at the next item in the list
- The last item points at null so that you know that the train has ended
- You can add and remove nodes as much as you want, and nothing needs to be resized

Doubly linked list

- The most common library implementation of a linked list is a **doubly linked list**
- Node consists of data, a next pointer, and a previous pointer
- Because we know the next and the previous, we can move forwards or backwards in the list



Definition

- Simple definition for a doubly-linked list that holds an unlimited number of **`String`** values:

```
public class LinkedList {
   private static class Node {
        public String data;
        public Node next;
        public Node previous;
   }

   private Node head = null
   private Node tail = null;
   private int size = 0;
   …
}
```

Containers

- When you write a container class (like a list), you have to write it to contain something
  - A list of **`String`** values
  - A list of **`Wombat`** values
  - A list of **`int`** values
- What if we could design a list class and not specify what its contents are?
- Someone has to say what it contains only when they make a particular list

Generics

- That's the idea behind **generics** in Java
  - The name is because it lets you make a generic list instead of a specific kind of list
- You can make classes (often, but not always, containers)
- These classes have one or more **type parameters**
- The type parameters are like variables that hold type information
- When you make such an object, you have to say what its types are

Angle brackets

- Influenced by templates in C++, Java puts type parameters in angle brackets (**<>**)
- For example, we can declare the following **`LinkedList`** objects defined in the Java Collections Framework

```
LinkedList<String> words = new LinkedList<String>();
LinkedList<Wombat> zoo = new LinkedList<Wombat>();
LinkedList<Integer> numbers = new LinkedList<Integer>();
```

- For technical reasons, you can only use reference types for type parameters, never primitive types

Boxing and unboxing

- If you use the wrapper class as the type parameter, Java will automatically convert primitive types to and from the wrapper class
- This is called boxing and unboxing
- For example:

```
LinkedList<Integer> numbers = new LinkedList<>();
numbers.add(7);
numbers.add(15);
int value = numbers.get(0);   // Holds 7
```

- For the most part, it magically works
- However, storing primitive types is less efficient

Type parameter syntax

- When declaring a generic class, put angle brackets and the type parameter after the name of the class
- The type parameter is often called **T**, standing for type
- Consider a simple generic class that holds a pair of…anything

```
public class Pair<T> {
    private T x;
    private T y;
    public Pair(T x, T y) {
        this.x = x;
        this.y = y;
    }
}
```

Definition

- Instead of **`String`** values, we can write a doubly linked list class that holds anything

```
public class LinkedList<T> {
   private static class Node<T> {
        public T data;
        public Node<T> next;
        public Node<T> previous;
   }

   private Node<T> head = null
   private Node<T> tail = null;
   private int size = 0;
   …
}
```

Java Collections Framework

Container interfaces

- **`Collection`** Parent interface of most containers
- **`Iterable`** A collection that can be iterated over
- **`List`** A collection that contains items in an order
- **`Queue`** A collection that supports FIFO operations
- **`Set`** A collection of unordered objects
- **`Map`** A collection of (key, value) pairs

Container classes

- **`LinkedList`** List implementation using a linked list
- **`ArrayList`** List implementation using a dynamic array
- **`Stack`** FILO data structure
- **`Vector`** Like an ArrayList, but thread-safe
- **`HashSet`** Set implementation using a hash table
- **`TreeSet`** Set implementation using binary search trees
- **`HashMap`** Map implementation using a hash table
- **`TreeMap`** Map implementation using binary search trees

Tools

- Collections
  - sort()
  - max()
  - min()
  - replaceAll()
  - reverse()
- Arrays
  - binarySearch()
  - sort()

List<E> methods

- The **List<E>** interface is one of the biggest you'll ever see
- Here are a few important methods in it

Returns	Method	Description
boolean	**add(E element)**	Adds **element** to the end of the list
void	**add(int index, E element)**	Adds **element** before **index**
boolean	**addAll(Collection<? extends E> collection)**	Adds everything from **collection** to this list
void	**clear()**	Removes everything from this list
boolean	**contains(Object object)**	Returns **true** if this list contains **object**
E	**get(int index)**	Return the element at **index**
int	**indexOf(Object object)**	Returns the first index where something that equals **object** can be found
boolean	**isEmpty()**	Returns **true** if the list is empty
boolean	**remove(int index)**	Remove the element at **index**
E	**set(int index, E element)**	Set the item at location **index** to **element**
int	**size()**	Returns the size of the list

`ArrayList` vs. `LinkedList`

- As you will learn (or have learned) in COMP 2100, **`ArrayList`** uses an array inside to store datay
  - When you need more space, it makes a new array and copies all the old stuff into the new array
- **`LinkedList`** uses a (wait for it) linked list to store the data
- In principle, **`LinkedList`** is faster for lots of unpredictable adds and removals
  - Especially adds and removals at the beginning of the list
- In practice, **`ArrayList`** is almost always faster
  - Modern machines are really good at ripping through arrays

Maps

- Maps are a kind of data structure that holds a (key, value) pair
- For example, a map might use social security numbers as keys and have **`Person`** objects as the value
- In a map, the keys must be unique, but the values could be repeated
- Both Java and C++ use the name map for the symbol table classes in their standard libraries
- Python calls it a dictionary (and supports it in the language, not just in libraries)
- Maps are also called symbol tables

Concrete example

- Maps are for you can imagine storing as data with two columns, a key and a value
- In this way you can look up the weight of anyone
- However, the keys **must** be unique
  - Ahmad and Carmen might weigh the same, but Ahmad cannot weight two different values
- There are multimaps in which a single key can be mapped to multiple values
  - But they are used much less often
  - All you really need is a map whose values are lists

Name (Key)	Weight (Value)
Ahmad	210
Bai Li	145
Carmen	105
Deepak	175
Erica	205

JCF Map

- The Java interface for maps is, unsurprisingly, **`Map<K,V>`**
  - **`K`** is the type of the key
  - **`V`** is the type of the value
  - Yes, it's a container with **two** generic types
- Any Java class that implements this interface can do the important things that you need for a map
  - **`get(Object key)`**
  - **`containsKey(Object key)`**
  - **`put(K key, V value)`**

JCF implementation

- Because the Java gods love us, they provided two main implementations of the **`Map`** interface
- **`HashMap<K,V>`**
  - **Hash table** implementation
  - To be useful, type **`K`** must have a meaningful **`hashCode()`** method
- **`TreeMap<K,V>`**
  - **Balanced binary search tree** implementation
  - To work, type **`K`** must implement the **`compareTo()`** method
  - Or you can supply a comparator when you create the **`TreeMap`**

JCF Set

- Java also provides an interface for sets
- A set is like a map without values (only keys)
- All we care about is storing an unordered collection of things
- The Java interface for sets is **`Set<E>`**
  - **`E`** is the type of objects being stored
- Any Java class that implements this interface can do the important things that you need for a set
  - **`add(E element)`**
  - **`contains(Object object)`**

JCF implementation

- As with maps, there are two main implementations of the **`Set`** interface
- **`HashSet<E>`**
  - **Hash table** implementation
  - To be useful, type **`E`** must have a meaningful **`hashCode()`** method
- **`TreeSet<E>`**
  - **Balanced binary search tree** implementation
  - To work, type **`E`** must implement the **`compareTo()`** method
  - Or you can supply a comparator when you create the **`TreeSet`**

Sorting

Sorting arrays

- Every language has its own libraries for sorting
- Let's start with sorting arrays
- It would be nice if every array just had a **`sort()`** method
  - But it doesn't!
- Instead, there's an **`Arrays`** (note the **`s`**) class with a number of useful static methods (with versions for arrays of every primitive type as well as **`Object`**):
  - **`sort()`**
  - **`binarySearch()`**
  - **`toString()`**
- To use it, import **`java.util.Arrays`**

Array sorting example

- Calling **`Arrays.sort()`** will sort arrays of **`byte`**, **`char`**, **`short`**, **`int`**, **`long`**, **`float`**, **`double`**, and **`String`**, always in ascending order
- Calling **`Arrays.toString()`** also produces a nice printable version of an array

```
// Obviously, data could also be input from the user or file
int[] numbers = {98, 50, 25, 30, 10, 56, 79, 86, 18, 92};
Arrays.sort(numbers);
// Output: [10, 18, 25, 30, 50, 56, 79, 86, 92, 98]
System.out.println(Arrays.toString(numbers));

String[] words = {"The", "quick", "brown", "fox", "jumps", "over",
"the", "lazy", "dog"};
Arrays.sort(words);
// Output: [The, brown, dog, fox, jumps, lazy, over, quick, the]
// Don't forget that uppercase letters have lower ASCII values
System.out.println(Arrays.toString(words));
```

Sorting other collections

- If you're sorting a collection (meaning **`List`**, **`LinkedList`**, **`ArrayList`**, **`Vector`**, etc.), you can use **`Collections.sort()`**
- When a collection has its own **`sort()`** method (as **`ArrayList`** does), use that, since it's tuned for performance on that collection

```
Scanner file = new Scanner(new File(fileName));
LinkedList<String> words = new LinkedList<>();
while(file.hasNext())
      words.add(file.next());
file.close();
// Print out all the words in the file, sorted
Collections.sort(words);
for(String word : words)
      System.out.println(word);
```

Comparable<T>

- If you want to sort an array or a list of some object, it must implement the **Comparable<T>** interface, where **T** is usually the type of the object itself
- The **Comparable<T>** interface has one method in it:

```
int compareTo(T other);
```

- An object that implements **Comparable<T>** will return:
  - A negative number if it comes before **other** in order
  - A positive number if it comes after **other** in order
  - Zero if it is equivalent to **other**
- It's usually not important what the values are, just whether they are positive or negative

Wombat example

- Wombat's are prized for their cuddliness, so we want to compare them by how fat they are
- By subtracting the other **`Wombat`** object's weight from our own, we get negative if we're smaller, positive if we're bigger, and zero if we weigh the same

```
public class Wombat implements Comparable<Wombat> {
	private int weight;
	private String name;
	public Wombat(String name, int weight) {
		this.name = name;
		this.weight = weight;
	}
	public int compareTo(Wombat other) {
		return weight - other.weight;
	}
	public int String getName() {
		return name;
	}
}
```

What if things weren't designed to be sorted?

- What if the objects you're working with don't implement the **`Comparable`** interface?
- Or if you want to sort them in some other way?
- You can supply a custom **`Comparator<T>`** object to the **`sort()`** methods that will say how they should be compared
- The **`Comparator<T>`** interface contains one method you have to implement:

```
int compare(T a, T, b);
```

- It should return negative if **`a`** comes before **`b`**, positive if **`a`** comes after **`b`**, and zero if **`a`** and **`b`** are equivalent

Planet example

- Here's a simple class for **`Planet`**, a class they didn't expect to sort

```
public class Planet {
	private String name;
	private double radius;
	public Planet(String name, double radius) {
		this.name = name;
		this.radius = radius;
	}
	public double getRadius() {
		return radius;
	}
	public String getName() {
		return name;
	}
}
```

Sorting `Planet` objects by radius

- Since the **`Planet`** class doesn't implement **`Comparable`**, we have to make a **`Comparator`** to pass to the **`sort()`** method
- We have to make an anonymous inner **`Comparable`** class, using the **`Double.compare()`** method to help use order by radius

```
List<Planet> planets = new ArrayList<>();

planets.add(new Planet("Venus", 6051.8));
planets.add(new Planet("Earth", 6371.0));
planets.add(new Planet("Mars", 3389.5));
Comparator<Planet> comparator = new Comparator<Planet>() {
      int compare(Planet a, Planet b) {
            return Double.compare(a.getRadius(), b.getRadius());
      }
};
Collections.sort(planets, comparator);
// Order: Mars, Venus, Earth
```

Sorting `Planet` objects in Java 8

- Using Java 8 style, we could also create the **`Comparator`** object with the quicker (but slightly more confusing) **`->`** syntax

```
List<Planet> planets = new ArrayList<>();

planets.add(new Planet("Venus", 6051.8));
planets.add(new Planet("Earth", 6371.0));
planets.add(new Planet("Mars", 3389.5));

// Order: Mars, Venus, Earth
Collections.sort(planets, (a, b) ->
Double.compare(a.getRadius(), b.getRadius()));
```

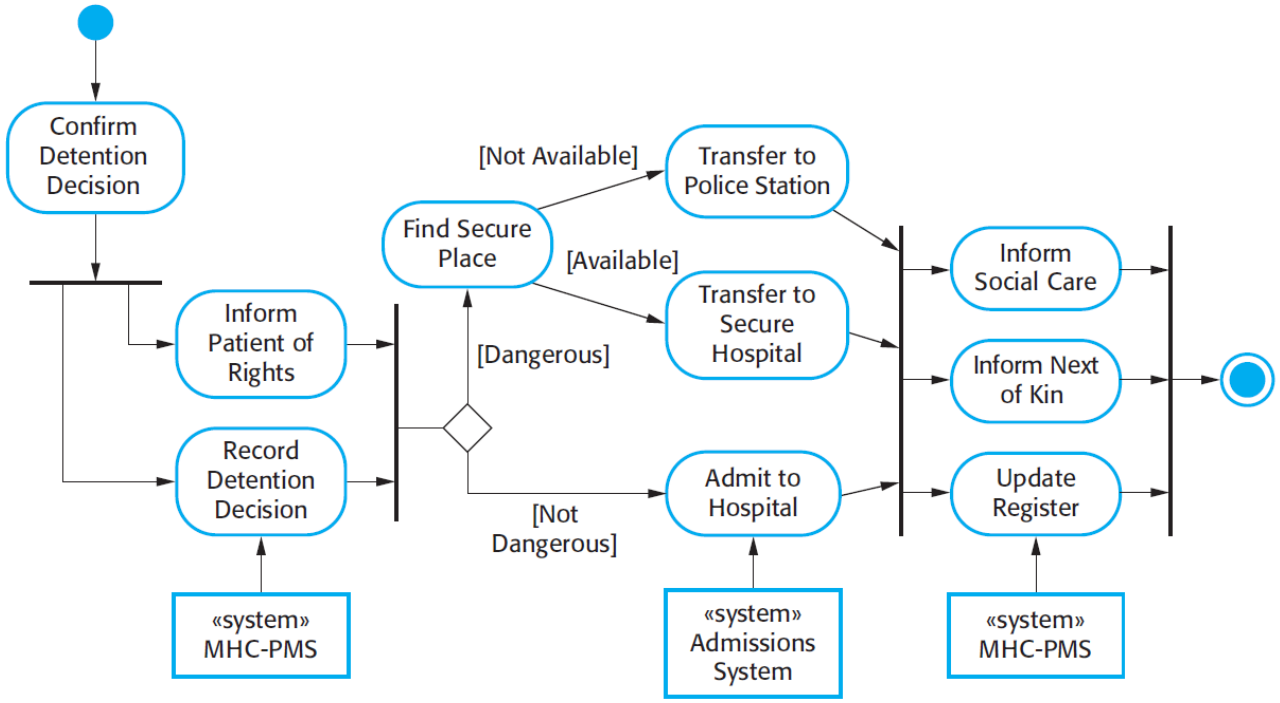
UML

UML

- The **Unified Modeling Language** (UML) is an international standard for graphical models of software systems
- A few useful kinds of diagrams:
  - Activity diagrams
  - Use case diagrams
  - Sequence diagrams
  - State diagrams
- Class diagrams are important enough that we'll talk about them in greater detail

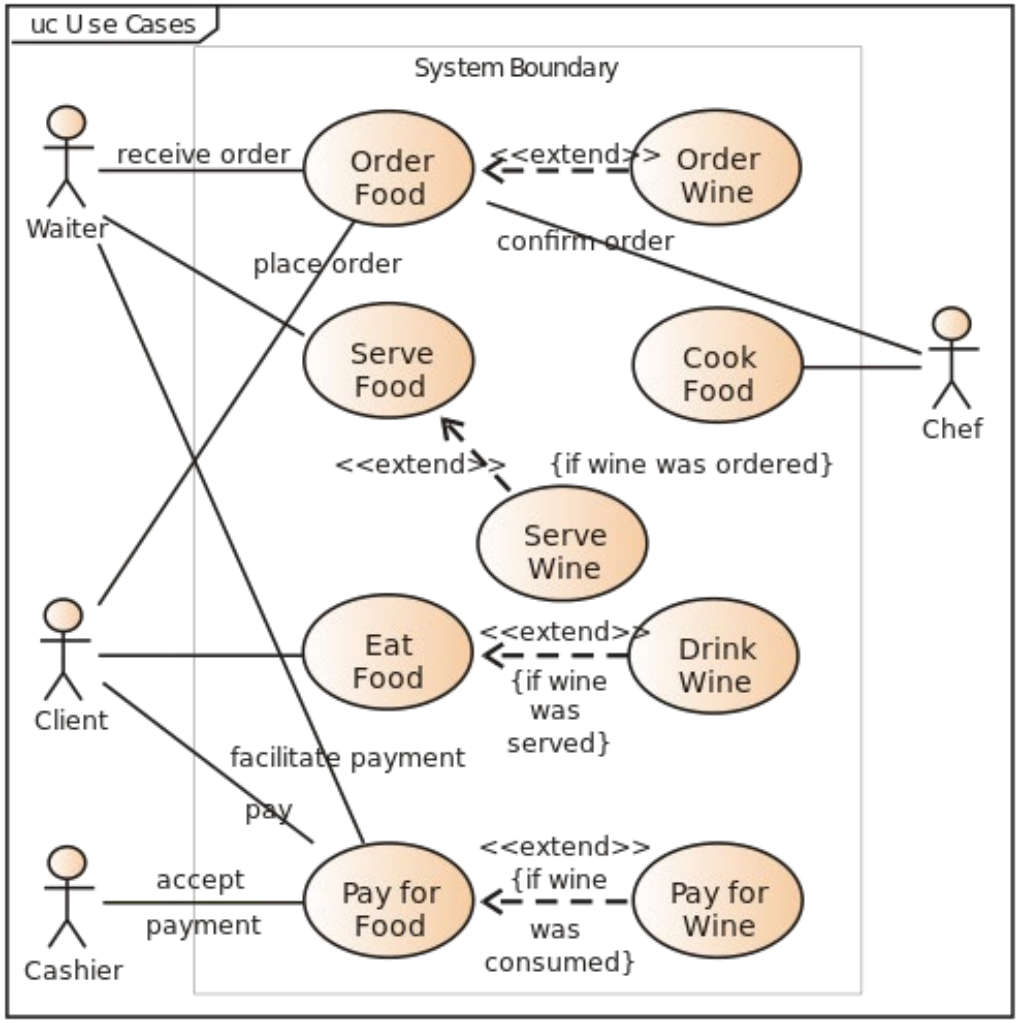
Activity diagrams

- Activity diagrams show the workflow of actions that a system takes
- Formally:
  - Rounded rectangles represent actions
  - Diamonds represent decisions
  - Bars represent starting or ending concurrent activities
  - A black circle represents the start
  - An encircled black circle represents the end



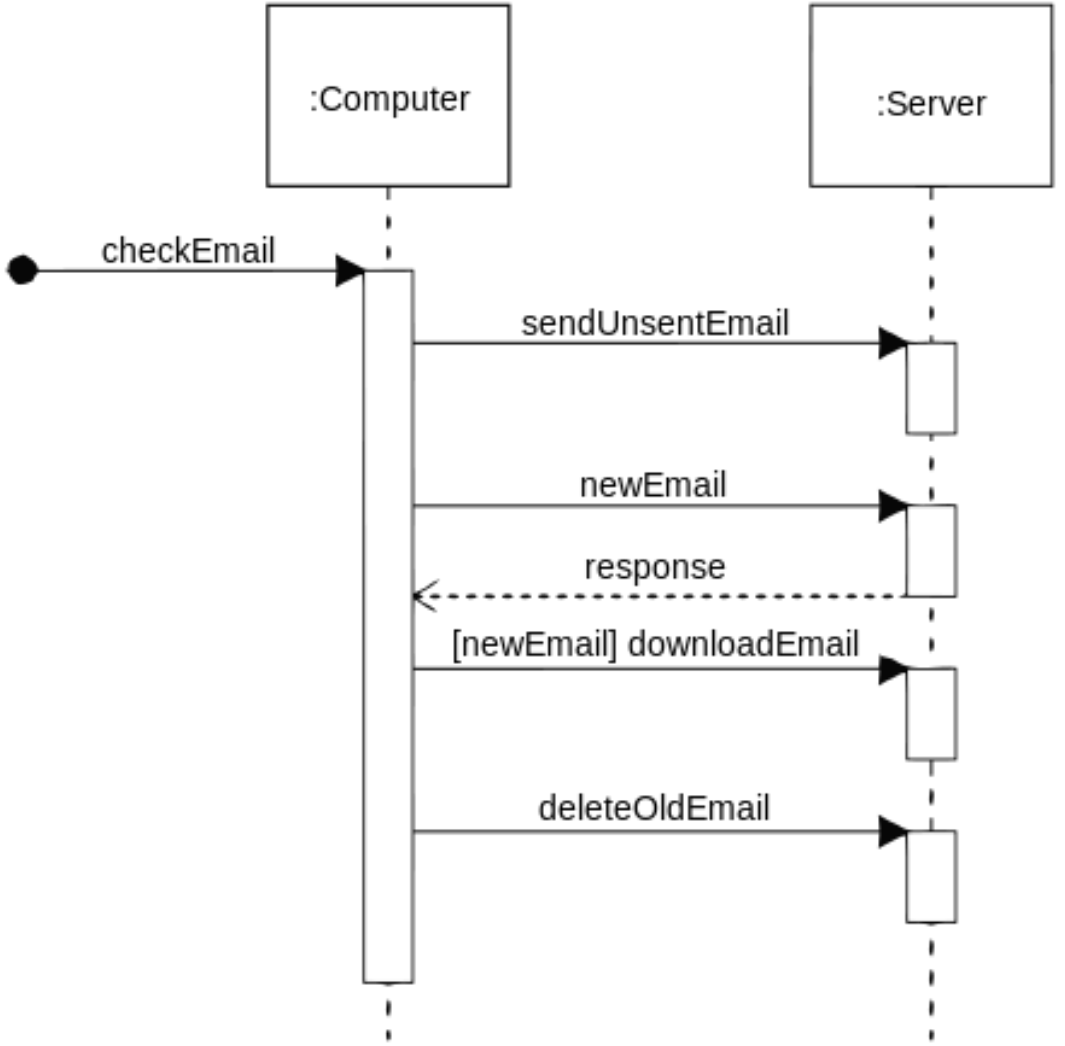
Use case diagrams

- Use case diagrams show relationships between users of a system and different use cases where the user is involved
- Example from Wikipedia:



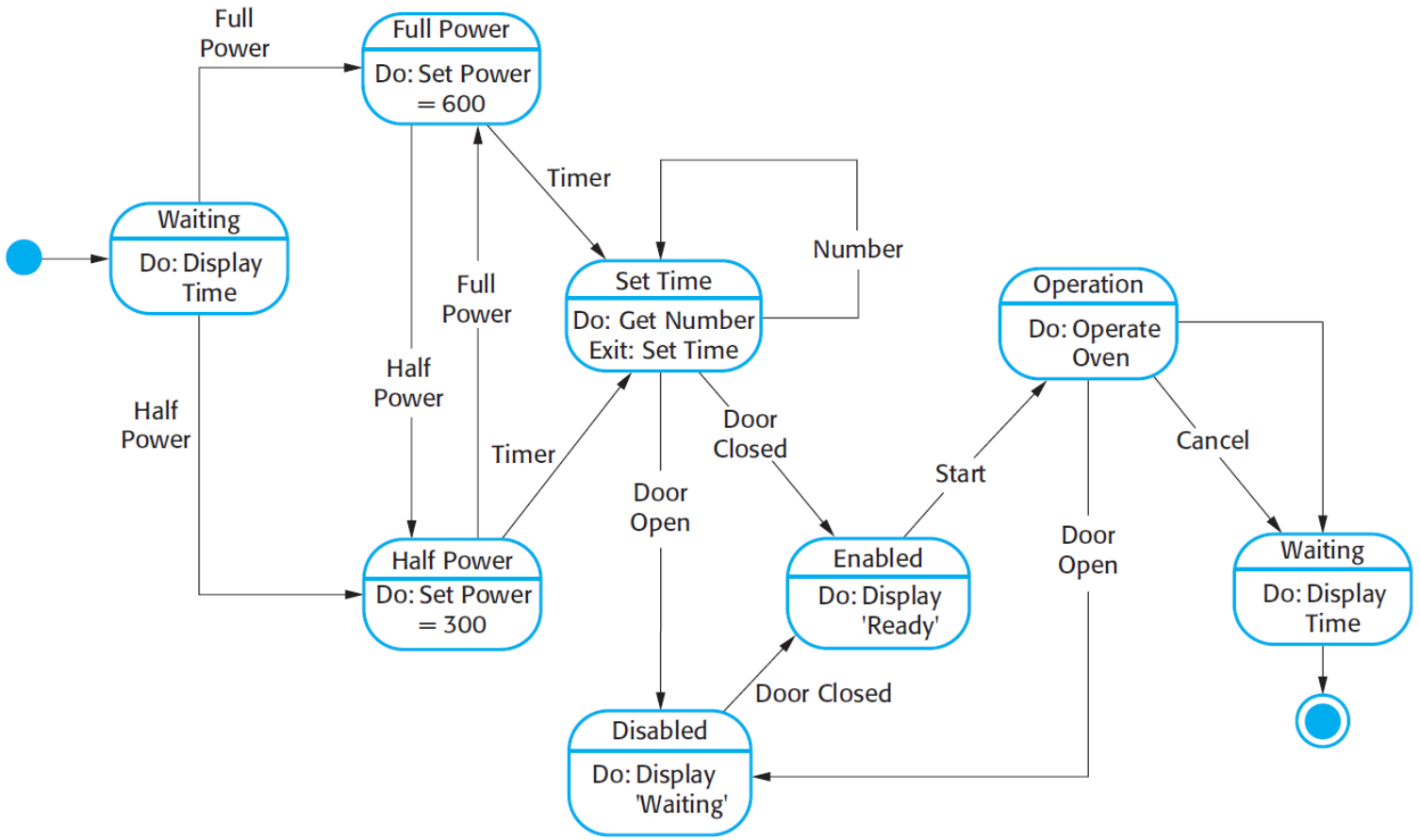
Sequence diagrams

- Sequence diagrams show system object interactions over time
- These messages are visualized as arrows
  - Solid arrow heads are synchronous messages
  - Open arrow heads are asynchronous messages
  - Dashed lines represent replies
- Example from Wikipedia:



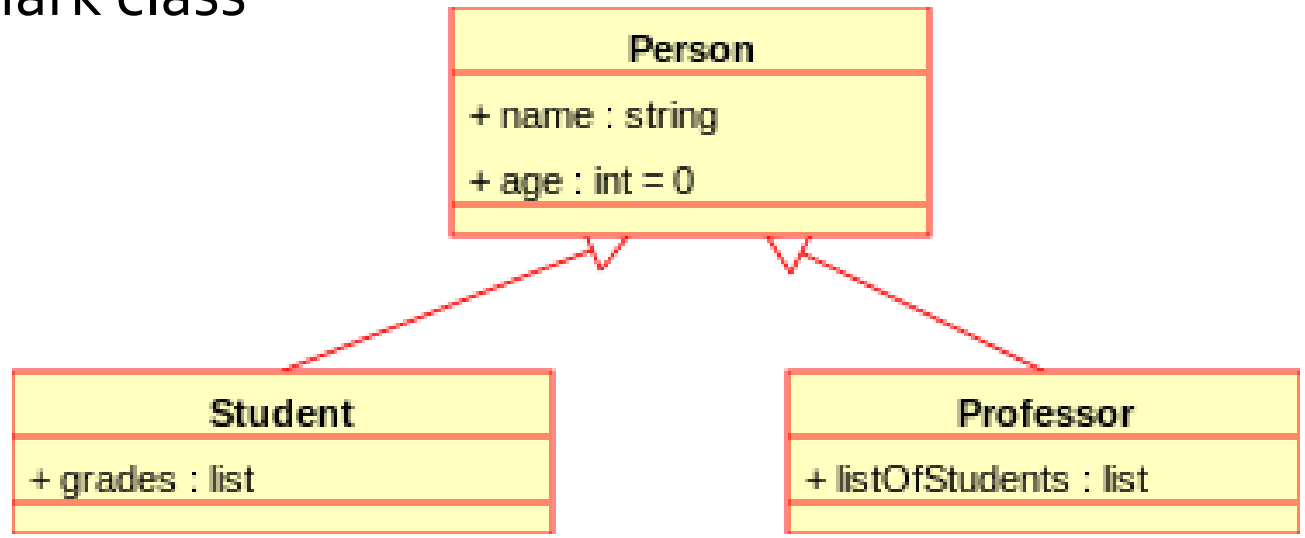
State diagrams

- State diagrams are the UML generalization of finite state automata from discrete math
- They describe a series of states that a system can be in and how transitions between those states happen



Class diagrams

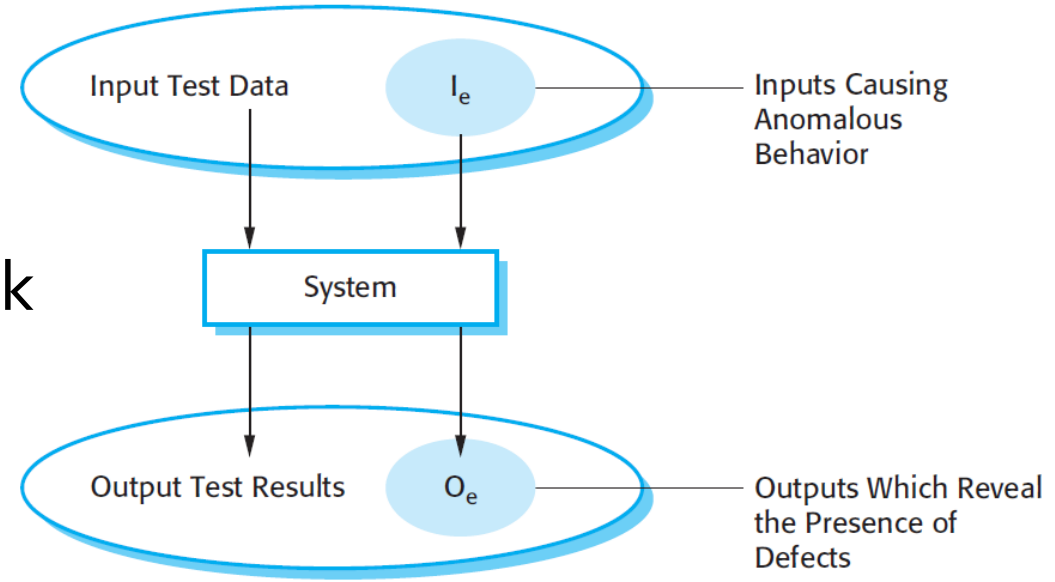
- Class diagrams show many kinds of relationships
- The **classes** being described often (but not always) map to classes in object-oriented languages
- The following symbols are used to mark class members:
  - **+** Public
  - **–** Private
  - **#** Protected
  - **/** Derived
  - **~** Package
  - ***** Random
- Example from Wikipedia:



Testing

Purposes of testing

- There are two almost opposing purposes for testing
- Showing that software meets its requirements
  - **Validation testing**
  - Looking for good outputs
- Finding inputs where software doesn't work
  - **Defect testing**
  - Looking for bad outputs
- When a project is due, students often confuse the two
  - Trying to convince themselves that the code is fine instead of looking for problems



Stages of testing

- Commercial software systems often go through three stages of testing
- Development testing
  - Look for bugs during development
  - Designers and programmers do the testing
- Release testing
  - Test a complete version of the code to see if it meets requirements
  - A separate testing team does the testing
- User testing
  - Users test the system in a real environment
  - Acceptance testing is a special kind of user testing to decide whether or not the product should be accepted or sent back

Development testing

- Development testing is the idea of testing you're most familiar with
  - Testing the software as it's being developed
  - Development testing is focused on defect testing
  - Debugging happens alongside development testing
- Three stages of development testing:
  - Unit testing: testing individual classes or methods
  - Component testing: testing components made from several objects
  - System testing: testing the system as a whole

Unit testing

- Unit testing focuses on very small components
  - Methods or functions
  - Objects
- Unit tests try many different inputs for the methods or objects to make sure that the outputs match

Unit test example

- Broken method to determine if a year is a leap year:

```
public static boolean isLeapYear(int year) {
    return year % 4 == 0 && year % 100 != 0;
}
```

- Tests:
  - `isLeapYear(2016)` → `true` (correct)
  - `isLeapYear(2018)` → `false` (correct)
  - `isLeapYear(1900)` → `false` (correct)
  - `isLeapYear(2000)` → `false` (incorrect)

Black box testing

- One philosophy of testing is making **black box tests**
- A black box test takes some input **A** and knows that the output is supposed to be **B**
- It assumes nothing about the internals of the program, only the specification
- To write black box tests, you come up with a set of input you think covers lots of cases and you run it and see if it works
- In the real world, black box testing can easily be done by a team that did not work on the original development

White box testing

- **White box testing** is the opposite of black box testing
  - Sometimes white box testing is called "clear box testing"
- In white box testing, you can use your knowledge of how the code works to generate tests
- Are there lots of if statements?
  - Write tests that go through all possible branches
- There are white box testing tools that can help you generate tests to exercise all branches
- Which is better, white box or black box testing?

Component testing

- Beyond unit testing is **component testing**
- Components are made up of several independent units
- The errors are likely to be from interactions between the units
  - Hopefully, the individual units have already been unit tested
- The interfaces between the units have to be tested
  - Parameter interfaces in method calls
  - Shared memory interfaces
  - Procedural interfaces in which an object implements a set of procedures
  - Message passing interfaces

System testing

- System testing is when we integrate components together in a version of the whole system
- Though similar to component testing, there are differences:
  - Older reusable components and commercial components might be integrated with new components
  - Components developed by different teams might be integrated for the first time
- Sometimes, you only see certain behavior when you get everything together
- Try testing all the use cases you expect the system to see

JUnit

JUnit

- JUnit is a popular framework for automating the unit testing of Java code
- JUnit is built into Eclipse and many other IDEs
- It is possible to run JUnit from the command line after downloading appropriate libraries
- JUnit is one of many xUnit frameworks designed to automate unit testing for many languages
- You are required to make JUnit tests for Project 4
- JUnit 5 is the latest version of JUnit, and there are small differences from previous versions

JUnit classes

- For each set of tests, create a class
- Code that must be done ahead of every test has the **`@BeforeEach`** annotation
- Each method that does a test has the **`@Test`** annotation

```
import org.junit.jupiter.api.*;
public class Testing {

        private String creature;

        @BeforeEach
        public void setUp() {
                creature = "Wombat";
        }

        @Test
        public void testWombat() {
                Assertions.assertEquals("Wombat", creature, "Wombat failure");
        }
}
```

Assertions in JUnit tests

- When you run a test, you expect to get a certain output
- You should assert that this output is what it should be
- JUnit 5 has a class called **`Assertions`** that has a number of static methods used to assert that different things are what they should be
  - Running JUnit takes care of turning assertions on
- The most common is **`assertEquals()`**, which takes the expected value, the actual value, and a message to report if they aren't equal:
  - **`assertEquals(int expected, int actual, String message)`**
  - **`assertEquals(char expected, char actual, String message)`**
  - **`assertEquals(double expected, double actual, double delta, String message)`**
  - **`assertEquals(Object expected, Object actual, String message)`**
- Another useful method in **`Assertions`**:
  - **`assertTrue(boolean condition, String message)`**

Assertion example

- We know that the **`substring()`** method on **`String`** objects works, but what if we wanted to test it?

```
import org.junit.jupiter.api.*;

public class StringTest {

	@Test
	public void testSubstring() {
		String string = "dysfunctional";
		String substring = string.substring(3,6);
		Assertions.assertEquals("fun", substring, "Substring failure!");
	}
}
```

Test driven development

- **Test driven development** (TDD) is an approach to development where testing and coding are interleaved
- Never move to the next increment of code until the current one passes its tests
- The key idea of TDD is that you write tests for the code **before** you write the code

Benefits of TDD

- By making the test first, you really understand what you're trying to implement
- Your testing has better code coverage, testing every segment of code at least once
- Regression testing happens naturally
- Debugging should be easier since you know where the problem likely is (the new code added)
- The tests are a form of documentation, showing what the code should and shouldn't do

Practice questions

- Write a doubly-linked list class containing **`String`** values that can add to and remove from the front and the back
- Convert that doubly-linked list class to a generic class holding values of type **`T`**
- Write a **`Tuna`** class that implements **`Comparable<Tuna>`**, based on a **`weight`** member variable
- Write a custom comparator to sort a list of **`double`** values in *descending* order (largest values first)
- What's the difference between black box testing and white box testing?

Upcoming

Next time…

- There is no next time!

Reminders

- **Finish Project 4**
  - **Due tonight by midnight!**
- Review chapters 7, 10-12, and , 15-21 and notes
- Look over labs, quizzes, and projects to prepare
- Final Exam:
  - Monday, April 27, 2020
  - 10:15 a.m. to 12:15 p.m.