CPSC 210

Personal Access Token: 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

Sept. 6 - B1 Program Structure

- Course Outline
    - Instructor: Paul Carter
    - Why I think CS is great...?
        - Creative
        - Challenging
        - Far-reaching impact
        - Raises interesting moral, ethical and social questions
        - For *everyone*, may be approaching from different perspectives
    - Interactive Lectures
        - Part of every lecture will be dedicated to providing you with an opportunity to gain hands-on experience
        - We encourage you to really engage with these exercises - the more you learn in class, the less you have to do on your own time outside of class
        - To facilitate this format, we ask you to spend time preparing for class by reviewing short, online videos
    - Object-Oriented Design vs. Java
        - Focus of this course is object-oriented design (OOD)
        - Vehicle for illustrating OOD concepts is a programming language called Java
        - Will not be teaching Java the same way that were introduced to the BSL/ISL/ASL teaching languages in CPSC 110
        - Will be taking an immersive approach
    - Course Resources

- Links on Canvas
    - Course Outline
    - edX
    - PrarieLearn
    - Piazza
- Grading Scheme
    - Lecture Tickets - 3%
    - Term Project - 15%
    - Labs - 7%
    - In-term Exams ($\times$ 2) - 15% + 20%
    - Final Exam - 40%
- Lecture Ticket
    - On PrarieLearn
    - Always **a week ahead** to work
    - Available points
        - Number of attempts
    - Open-book
- Weekly Format
    - Every week you will attend one lab session
    - *Lab 1* complete it by Friday's lecture
    - Starting next week, the attendance at the registered lab section is required
- Term project
    - An opportunity to get creative and design an application that is of interest to you
    - An opportunity to develop an application that you may want to showcase for future employers
    - More about this next week
- Academic Integrity
    - Respect the hard work of your fellow students and don't engage in academic misconduct
    - Collaboration Policy
        - Content in course outline

- Ask a member of the course staff if there is anything that is not understood
- All the code submitted must be own

- Contract Cheating
  - This refers to having someone else help you complete graded work (labs / term project / exams)
  - Can be dealt with severely, expect that the staff will request the highest penalty possible

- CWL credentials
  - **Never** share CWL with **anyone**
  - Do not share your password with someone claiming to be a tutor

- Prerequisites
  - One of CPSC 107 or CPSC 110

- B1 - Package Relationship Diagrams
  - Example: Space Invaders
    - Data Definitions (CPSC 110)
      - Canvas
      - Position
      - Size
      - Hit?
      - Speed
      - Missile
        - Position
        - Size
        - Color
        - Speed...
      - Space Shuttle
      - Invader
      - Game (itself)
    - How do files/packages relate with each other? (Focus on import statements)

- Pkg 1. model
    - Invader
    - Missile
    - SIGame
    - Tank
- Pkg 2. ui
    - GamePanel
        - → Invader
        - → Missile
        - → Tank
        - → SIGame
    - ScorePanel
        - → SIGame
    - SpaceInvaders
        - ```
          import ca.ubc.cpsc210.spaceinvaders.model.SIGame
          ```
        - → SIGame
- ...

Sept. 8 - B2: Methods and Method Calls

- CPSC 110 Recall
    - ```
      (substring "JavaWorld" 0 4)
      (fn_name arg1 arg2 ... argn)
      ```
    - Java
        - ```
          method_name(arg1, arg2, ..., argn)
          ```
        - this.
          ```

        - var.
        - Argument section could be empty
- ...

Sept. 11 - B3. Classes, Objects, and Variables

- Science Co-op Info Sessions
    - sciencecoop.ubc.ca/prospective/infosessions
    - Sept 19 @ 5-6 pm (DMP 101)
    - Sept 23 @ 9-10 am (Zoom)
- Primitive Types
    - 8 primitive types
    - byte, short, int, long, float, double, boolean, char
        - "=": gets
        - ```
          int count = 4;
          ```
    - All other types are reference types - they are used to **reference** an object
        - ```
          Person p = new Person();
          ```
        - $p\square \rightarrow \circ(PersonObject)$
- Tank class
    - fields (7), constructor (1), methods (5)
    - ```
      Tank myTank;
      myTank = new Tank(100);
      myTank.faceLeft();
      myTank.move();
      ```
    - Field in a class can be accessed by all methods within the class
    - myTank = new Tank(100); - $myTank\square \rightarrow (x : 100, direction : 1)$
    - myTank.faceLeft(); - $myTank\square \rightarrow (x : 100, direction : -1)$
        - myTank is an "implicit parameter"
      ```

    - myTank.move(); - $myTank\square \rightarrow (x : 98, direction : -1)$
- ArrayList
    - ArrayList$< E >$ is part of Java library.

Sept. 13 - B4: Data Flow

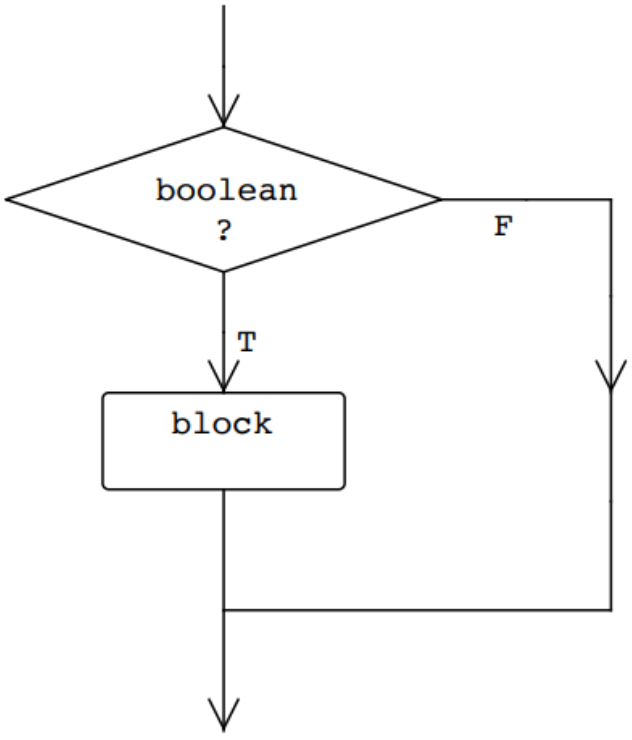
- Passing arguments to methods
    - Copy of the value returned if a method returns a value
    - ...

Sept. 15 - B5: Intra-method Control Flow (Flowcharts)

-

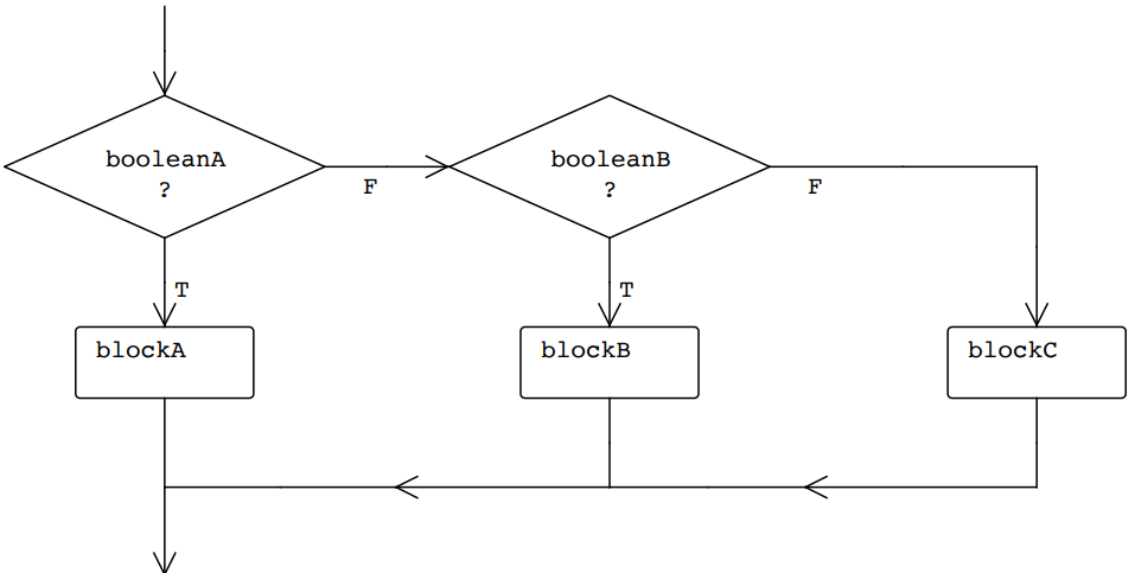
```
This flowchart models the flow of control through
an if statement:

 if (boolean) {
     // block
 }
```



```
This flowchart models the flow of control through
an if...else if...else statement:

 if (booleanA) {
     // blockA
 }
 else if (booleanB) {
     // blockB
 }
 else {
     // blockC
 }
```

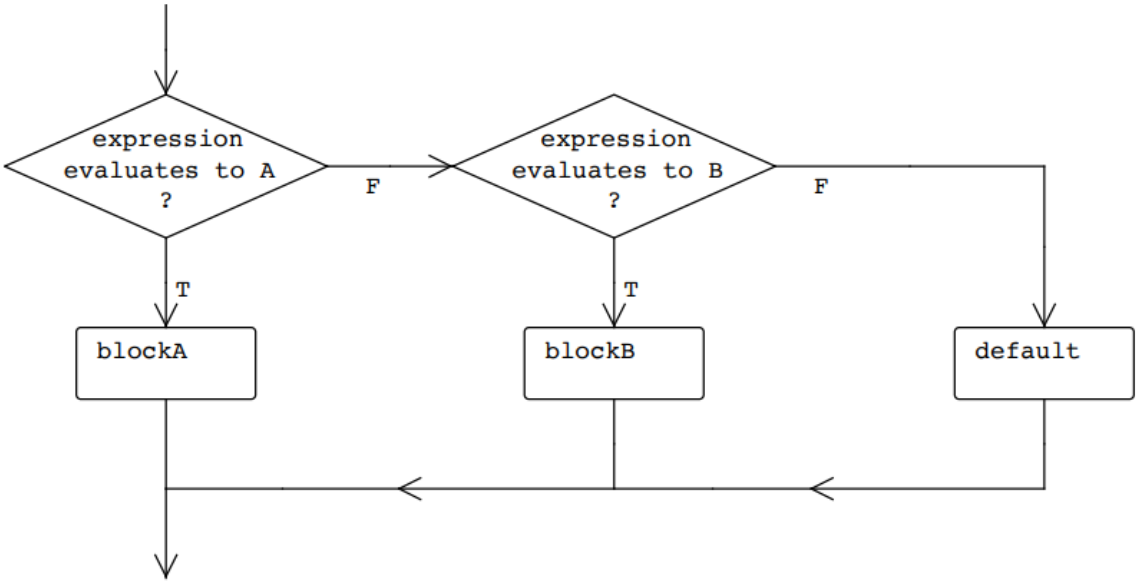


-
  - else if: optional or repeated
  - else: optional

- This flowchart models the flow of control through a switch statement:

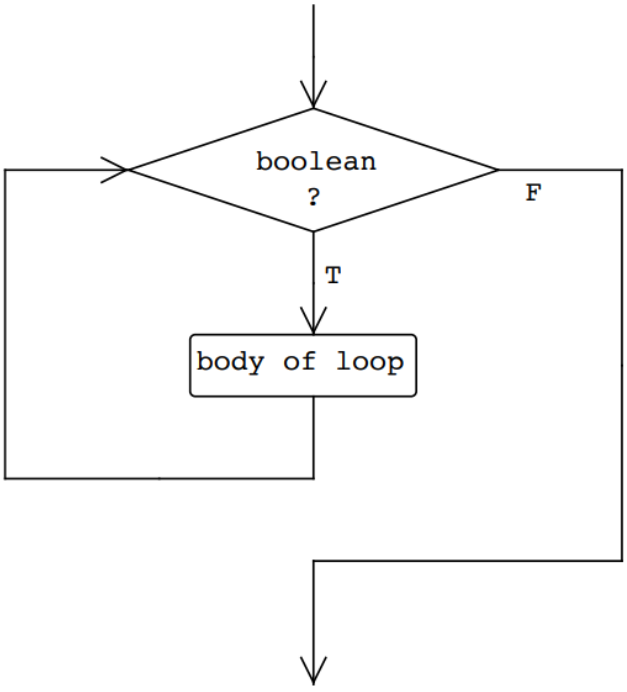
```
switch (expression) {
case A:
    // blockA
    break;
case B:
    // blockB
    break;
default:
    // default
}
```

Note that switch statements work with expressions that evaluate to int, short, byte or char (and with their corresponding wrapper classes Integer, Short, Byte and Character), and with enumerations and Strings.



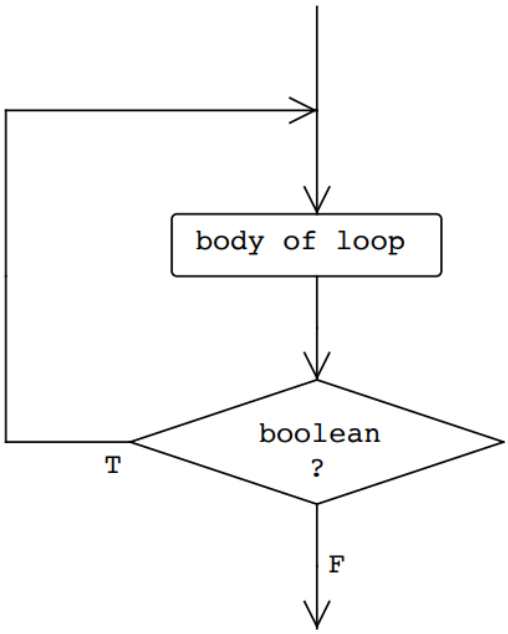
- This flowchart models the flow of control through a while loop:

```
while (boolean) {
    // body of loop
}
```



- This flowchart models the flow of control through a do-while loop:

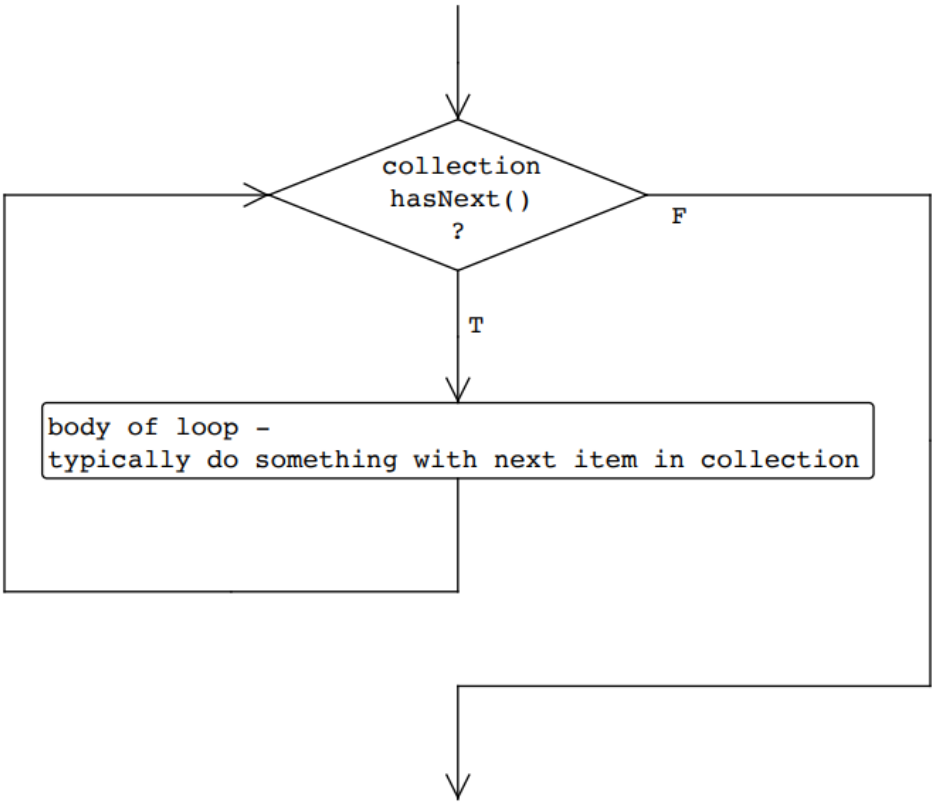
```
do {
    // body of loop
} while (boolean);
```



This flowchart models the flow of control through a for-each loop:

```
for (Type next : collection) {
    // body of loop -
    // typically do something with next item in collection
}
```

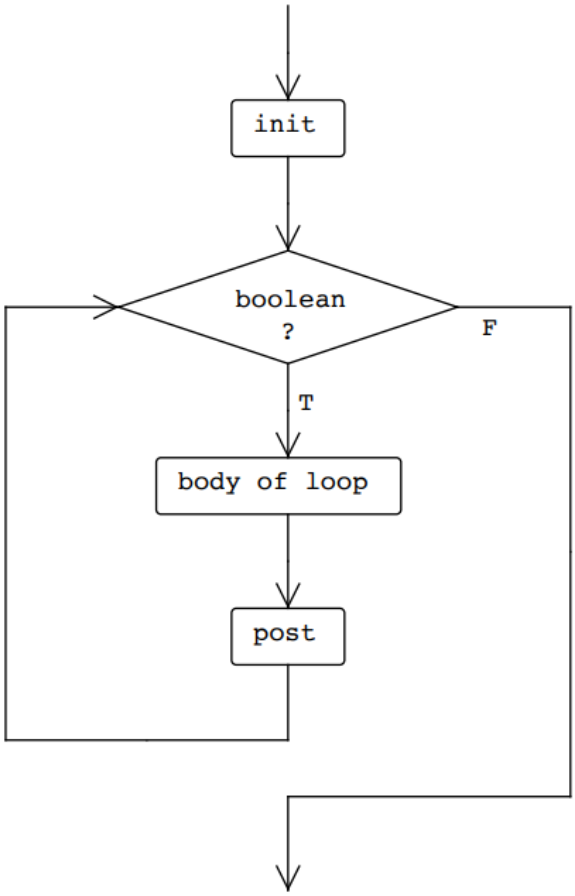
Note that collection must be a Java collection such as a List or Set (later in the course we'll see that it can also be used with an array or with a type that is Iterable).



-
  - Sequentially access elements in a list - Iteration Abstraction
  - ```
    for (String name: names) {
        // do smth
    }
    ```
    - Is there more items in the list? (Hidden boolean statements)
    ```

- This flowchart models the flow of control through a for loop:

```
for (init; boolean; post) {
    // body of loop
}
```



- ...

Sept. 18th - A1: Specifying and Using Data Abstraction

- REQUIRES: specify a condition that must be true when the call to the method is made, if the method is to provide the behaviour specified in the EFFECTS clause
- MODIFIES: specify what objects (if any) are modified as the result of the call to the method
- EFFECTS: specift *what* will happen when this method is called (not *how* it happened)
- Clauses should never refer to fields of the class for two reasons
    - Should design the specification before the implementation - so fields are not known when method specification is designed

        - Allow someone understand what the method will do without knowing anything about the method's implementation
        - ...

Sept. 20th - A2: Testing a data abstraction

- The term project
    - Phase 0 will be released by noon on Sept. 21st
    - Find the experience of designing a personal project far more enjoyable and far less stressful if you approach it incrementally:
        - *expect* that problems cannot be resolved remotely using Piazza that you *will* have attend office hours
        - a small amount of time spent each day will be far more productive than longer sessions over fewer days
    - How ambitious?
        - Develop incrementally. Start with user stories that meet the requirements for each phase of the term project. Once successfully tested and implemented them, feel free to add additional user stories - testing and implementing each one as you go.
- Testing: Overview
    - If a method has a REQUIRES clause, test only cases that satisfy that clause
        - *cannot* test this method when input is not of requirements
    - Test *boundary* cases, *typical* cases; test *all* aspects of cases
    - If a method modifies an object, test that behaviour of the method is as expected when called multiple times
- jUnit 5 Assertions
    - assertTrue(expression);
    - assertFalse(expression);
    - assertNull(expression);
    - assertEquals(expected, actual);

Sept. 22nd - A3: Implementing a Data Abstraction

- ...

Sept. 29th - A4: Types and Interfaces

- Java interfaces
    - Specifies a new type of data
    - *Implementation* of methods is not provided
    - Implementation is deferred to the classes that *implement* the interface
    - Used ArrayList, one way to implement List in Java; LinkedList is another way
    - Allows us to record the similarity between types - specifically, to specify their common behaviours
- ```
  List<E> = {ArrayList<E>, LinkedList<E>}
  ```
    - ArrayList and LinkedList are **subtypes** of List
    - List is **supertype** of ArrayList and LinkedList
    - Subtypes *must* provide an implementation for all methods specified in the supertypes
    - Cannot be instantiated
- Substitution
    - Type substitution
        - substitute a subtype for any of its super types
        - replace a super type with any of its subtypes
    - Apparent Type: left hand
        - Determines the methods that can be called on an object
        - Any type is considered to be a subtype of itself
    - Actual Type: right hand
    - Example
  ```

- ```
  List<Integer> data = new ArrayList<>();
  ArrayList<Integer> myList = new
  ArrayList();

  data = myList;
  myList = data;
  myCollection.addAll(data);
  myCollection.addAll(myList);

  public class Collection {
      void addAll(ArrayList<Integer> list) {
          // stub
      }
  }
  ```

  - ✓: data = myList; myCollection.addAll(myList);
  - ×: myList = data; myCollection.addAll(data);

- ```
  data = myCollection.makeList();
  myList = myCollection.makeList();

  public class Collection {
      List<Integer> makeList() {
          return new ArrayList<Integer>(); //
  stub
      }
  }
  ```

  - ✓: data = myCollection.makeList();
  - ×: myList = myCollection.makeList();

Oct. 6th - A5: Multiple Interfaces

- Multiple Interfaces
  - Tri-mentoring
    - Mentor-Mentee

        - Industry - Senior
        - Senior - Junior
    - Model participants (industry-senior-junior) have some behaviours independent of their roles
    - Distinguish between Mentors and Mentees
    -
```java
public interface Mentor {
    void addMentee(Mentee m);
    void removeMentee(Mentee m);
    List<Mentee> getMentees();
}

public interface Mentee {
    // stub
}

public class IndustryRep implements Mentor {
    // stub
}

public class Senior implements Mentor, Mentee {
    // stub
}

public class Junior implements Mentee {
    // stub
}
```

        - Industry Rep, Seniors → Mentor
        - Senior, Junior → Mentee

Oct. 11th - A6: Extends and Overriding - Inheritance

- Inheritance
    - A class can be declared to extend exactly one other class
    - When ClassB extends ClassA
        - ClassB becomes a subtype of ClassA

        - ClassA becomes a super type of ClassB
        - ```
          Clock clk = new AlarmClock();
          ```
- Type substitution
    - Remains the same as Interface

Oct. 12th - A7: Abstract Classes & Overloading Methods

- Abstract class
    - defines a new type of data (just like an interafce or class)
    - can optionally *not* provide an implementation for one or more of its methods
    - A class that extends an abstract class
        - must provide an implementation for all of the abstract methods
        - or declare it abstract
- Type substitution
    - can substitute any subtype for a super type
- Method overloading
    - Two methods with the same name but (necessarily) *different* parameter lists
    - ```
      public boolean contains(Point point);
      public boolean contains(int x, int y);
      ```

Oct. 13th - C1: Exception Handling

- Exceptions
    - Exception class
        - An exception is an instance of that class or one of its subclasses
        - Can be thrown by a method
        - Two types
            - checked
            - unchecked
    - Exception handling

- Flow of program !!!
- ...

Oct. 16th - C2: Exception Handling

- Exception Hierarchies
    - ```
      try {
          // ...
      } catch () {
          // ...
      }
      ```

- Checked vs. Unchecked Exceptions
    - RuntimeException and any of its subclasses are *unchecked exceptions*
    - Not required to handle *unchecked* exceptions
- Unchecked exceptions
    - Address commonly occuring bugs in code
    - **NullPointerException** - invoke a class member on a reference of null
- Class invariants
    - A property that is always true about every instance of that class
    - e.g. an invariant for our Account class would be: **balance >= 0**
    - Invariants can be checked using Java *assert* statements

Oct. 18th - C3: Testing Methods that throw Exceptions

- Exceptions - Testing
    - Test case where exception is *not* expected
      ```

```java
@Test
void testDepositSuccess() {
    try {
        double bal = myAccount.deposit(120.0);
        // assert call to deposit had expected
effect, as usual
    } catch (NegativeAmountException e) {
        fail("Caught NegativeAmountException");
    }
}
```

- Test case where exception *is* expected

```java
@Test
void testDepositExceptionExpected() {
    try {
        double bal = myAccount.deposit(-5.0);
        fail("Exception not thrown");
    } catch (NegativeAmountException e) {
        // expected
    }
    // check balance was not modified
}
```

- ...

Oct. 23rd - Extracting Sequence Diagrams from Code

- Sequence Diagrams
    - A sequence diagram models inter-method control flow (as does a call graph) in addition to modelling data
        - Does this by identifying objects on which methods are called

Oct 25th - C6/C7 Working With Collections

- Maps
    - key-value pairs represented using a binary search tree in CPSC 110
        - Key in collection is unique

        - Key is used to look-up the corresponding value
    - Java library includes Map<K, V> interface has two *type parameters*
        - K - represents the type of key
        - V - represents the type for the value
    - Cannot use for loop to iterate through the collection, but can use .keySet(), to retrieve the set of keys
- Java Collections
    - List: maintaining order
    - Set: no duplicate items in the collection, order not usually maintained
    - HashSet: quickly determining if an item belongs to a collection(CPSC 221: some conditions apply)
    - Maps: represent collections of key-value pairs

Oct 27th - C8: Implementing Bi-directional Relationships and Sequence Diagrams

-
```java
public class Recipe {
    public void addIngredient(Ingredient ingredient) {
        if (!this.ingredients.contains(ingredient)) {
            ingredients.add(ingredient);
            ingredient.addRecipe(this);
        }
    }
}

public class Ingredient {
    public void addRecipe(Recipe recipe) {
        if (!this.recipes.contains(recipe)) {
            recipes.add(recipe);
            recipe.addIngredient(this);
        }
    }
}
```

    - Order matters, ingredients.add(ingredient); must be before ingredient.addRecipe(this);, or ingredient won't be added to ingredients.

Nov. 1st - D1: Coupling and Cohesion

- Cohesion: Single Responsibility Principle
    - Each claass should have one, clearly defined purpose
- Coupling
    - A measure of the degree to which one part of the system depends on other parts of the system. Shows up when a change in one class results in another class not compiling(so such cases show up in an obvious way)
    - For example, you have multiple classes that depend on *classA*. Changing the signature of a method will affect the implementation of another method.

Nov. 2nd - D2: Liskov Substitution Principle

- LSP
    - Precondition **cannot** be narrower: widening means "accepting everything that is accepted in the superclass **then** accepting more values"
        - Narrowing: $(1, 3, 5) \rightarrow (1, 3) \vee (1, 3, 7, 9)$
    - Postcondition **cannot** be wider: narrowing means producing a value that is not in the scope of the superclass return
        - Widening: $(1, 3, 5, 7, 9) \rightarrow (1, 3, 5, 7, 9, 11) \vee (1, 5, 13)$
    - Any test that is designed based on the specification of a method must pass when run against an overriden versin of that method in a subtype
    -
```
class MyClass {
    // stub
}

class MySubClass extends MyClass{
    // stub
}

MyClass mc = new MyClass();
// Code block 1 works
// If we change
MyClass mc = new MySubClass();
// Code block 1 SHOULD still work
```

```
14
15  /** Code block 1:
16  assert(preconditionSuper());
17  mc.someMethod();
18  assert(postconditionSuper());
19  */
```

Nov. 8th - D3: Refactoring

- Refactoring
    - Goal: improve the design of the code without changing the functionality
        - Reducing repetitive code using helper methods
        - Reducing coupling
        - ...?

Nov. 10th - D4: Composite Pattern

- A design pattern - re-usable approach to solving a design problem
    - NOT a re-usable library / re-usable piece of code
    - Has to be adapted to the particular design problem under consideratinon (and that is what makes this topic a little harder)
- *Design Patterns: Elements of Reusable Object-Oriented Software* - Gamma, Helm, Johnson, Vlissides
- *Head First Design Patterns* - Freeman, Robson, Bates & Sierra
- Composite Pattern
    - Intent: Compose objects into tree-like structures to represent whole-part hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.
    - Arbitrary-arity tree
        - Composite node: arbitrarily many children
        - Leaf node: no children
- ```
  1  class Component {
  2      operation();
  3  }
  ```
  ```

```
4
5  class Leaf extends Component {
6      operation();
7  }
8
9  class Composite extends Component {
10     private List<Component> components;
11     operation();
12     add();
13     remove();
14     getChild();
15     // stub
16 }
```

- Each node ONLY has 1 parent
- In general
    - Identify composite, leaf
- Lecture Ticket analysis
    - Branches: Branches + Leaves

Nov. 17th - D4: Composite Pattern (Continued)

- Steps to identify a composite pattern
    - Work out what the "Composite" is (can have children)
    - Work out what the "Leaf" is (cannot have children)
    - Both *extends* Component

Nov. 22nd - D5: Observer Pattern

- Intent: to define a **one-to-many dependency** between objects so that when **one object changes** state, all its **dependents** are notified and updated automatically
- Roles
    - *Subject*: this object changes state → dependents to be notified
    - *Observer*: dependents that are to be notified when *Subject* changes state
- Example

- Design an application that acts as a kiosk where users can vote for their favourite colour. As the number of votes changes, we want to update a pie chart and a panel that shows the number of votes case for the favourite colour.
    - *Subject*: Kiosk
    - *Observer*: Pie chart, Panel

```
public class Subject {
    public void addObserver(Observer o);
    public void removeObserver(Observer o);
    public void notifyObservers();
}

public class ConcreteSubject extends Subject {
    public void changeState() {
        notifyObservers();
    }
}

public interface Observer {
    public void update();
}

public class ConcreteObserver implements Observer {
    @Override
    public void update() {
        // stub
    }
}
```

Nov. 24th - D5: Observer

Nov. 29th - D6: Singleton

```
public class Fields {
    // Class Variable: exists as soon as the class is created
    static int classVar = 0;
```

```
4
5       // Instance Variable: only exists after an instance
    of the object of
6       //                    this class is constructed
7       int instanceVar = 10;
8   }
```

- Intent: Ensure a class only has one instance, and provide a global point of access to it.
- Singleton Pattern
    - ```
      1  public class UniqueClass {
      2      private static UniqueClass singleton;
      3
      4      private UniqueClass() {
      5          // stub
      6      }
      7
      8      public static UniqueClass getInstance() {
      9          if (singleton == null) {
      10             singleton =  new UniqueClass();
      11         }
      12         return singleton;
      13     }
      14 }
      ```
    - Private constructor? No other classes should call the constructor
    - Only the Class can call the constructor
    - A field must be static for a static method to use it.

Dec 1st - The Iterator Pattern

- Intent: provide a way to access the elements of an aggregate object sequentially without exposing its underlying implementation
      ```

- ```java
  for (Item nextItem : collection) {
      // do smth with nextItem
  }

  Iterator<Item> itr = collection.iterator();
  while (itr.hasNext()) {
      Item nextItem = itr.next();
      // do smth with nextItem
  }
  ```

    - Collection are of example, ArrayList, LinkedList, HashSet

- ```java
  class Collection {
      public void iterator() {
          // must have
      }
  }

  class Iterator {
      public boolean hasNext() {
          // must have
      }

      public Item next() {
          // must have
      }
  }
  ```

- Pattern application
    - What collection to iterate over? - Iterable
    - What type of data is in that collection? - Iterable< Item >, Iterator< Item >
    - Implement iterator()

Dec. 4th - D8: Own Iterator

- Inner class has access to the fields from the outer class