Cheatsheet

Imports

`import x.x.x`

- `List<E>`: `java.util.List`
- `LinkedList<E>`: `java.util.LinkedList`
- `IllegalArgumentException`: `java.lang.IllegalArgumentException` (imported by default)
- `Comparable<E>`: `java.lang.Comparable` (imported by default)
- `Math`: `java.lang.Math`
  - `Math.PI`
  - `Math.abs()`
  - `Math.exp()`

Overriding

Must annotate with `@Override`

```
public String toString() {}
public boolean equals(Object other) {}
public int compareTo(E other) {}
```

Implementing equals

- It is reflexive: for any non-null reference value x, x.equals(x) should return true.
- It is symmetric: for any non-null reference values x and y, x.equals(y) should return true if and only if y.equals(x) returns true.
- It is transitive: for any non-null reference values x, y, and z, if x.equals(y) returns true and y.equals(z) returns true, then x.equals(z) should return true.
- It is consistent: for any non-null reference values x and y, multiple invocations of x.equals(y) consistently return true or consistently return false, provided no information used in equals comparisons on the objects is modified.
- For any non-null reference value x, x.equals(null) should return false.

```
@Override
public boolean equals(Object other) {
  if (this == other) {
    return true;
  }

  if (!(other instanceof Class)) {
    return false;
  }

  Class c = (Class) other;
  // comparisons

  return ...
}
```

Implementing compareTo

Compares this object with the specified object for order. Returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.

```
public class Car implements Comparable<Car> {
  @Override
  public int compareTo(Car other) {
    return this.speed - other.speed;
  }
}
```

Custom exceptions

```
public class CustomException extends Exception {
  public CustomException() {
    super("message");
  }
}

CustomException e = new CustomException();
e.getMessage(); // "message"
```

Take note

- All methods defined in an interface is `public abstract` by default so do not include those modifiers
- Do not use `public final` fields, just use `private` and include getters where necessary
- Ternary operators are allowed
- Try to prioritize using `equals` and `toString` for type comparison over using `instanceof`
  - From practice PE: when told to check against a list of types, add a list of supported type instances in the class and check using equality of the supported type instance and the given type

    ```
    boolean canProvideService(Service service) {
      for (Service supportedService : supportedServices) {
        // Override equals in Service
        if (service.equals(supportedService)) {
          return true;
        }
      }

      return false;
    }
    ```
- Anything that might use the String representation of an object should have `toString` be overriden (including from the superclass)
- Access exception messages with `getMessage()` on the exception
- Compile files using `javac -Xlint:rawtypes -Xlint:unchecked *.java`
- `@SuppressWarnings("unchecked")` or `@SuppressWarnings(["unchecked", "rawtypes"])`
- Check styles with

  - `java -jar ~cs2030s/bin/checkstyle.jar -c ~cs2030s/bin/cs2030_checks.xml *.java` (with Javadocs) or
  - `java -jar ~cs2030s/bin/checkstyle.jar -c ~cs2030s/bin/cs2030_nodoc_checks.xml *.java` (without Javadocs)
- Check if all files contain `@author` using `grep -lr '@author A0253334L' *.java` Append `grep -lr '@author A0253334L' | wc -l` to quickly get the number of files with the annotation (to tally)
- Add `@author A0253334L` Javadoc to all files created or edited
- Check style of files that is not `Test`, `PE`:

  ```
  java -jar ~cs2030s/bin/checkstyle.jar -c ~cs2030s/bin/cs2030_checks.xml *.java | grep -v 'Test|PE'
  ```

Connecting to NUS SoC server

```
ssh wjiahao@stu.comp.nus.edu.sg
```

SSH Hop into the PLAB server on the day itself

- Hostname: `peXXX`
- Userid: `plab0XXX`
- Password: `YYYYYYYY`

```
ssh plab0XXX@peXXX.comp.nus.edu.sg
```

Setup

General

```
mkdir .backup/
```

.vimrc

```
set tabstop=2
set shiftwidth=2
set expandtab
set nocompatible
set ruler
set number

syntax on
set autoindent
set smartindent
set background=light
set cc=80
filetype plugin indent on

let g:rainbow_active = 1
```

Adding colors

```
ln -s ~cs2030s/.vim/vim-colorschemes/colors ~/.vim/colors
```

Adding plugins

- `ln -s ~cs2030s/.vim/vim-plugins/delimitMate/ ~/.vim/pack/plugins/start`
- `ln -s ~cs2030s/.vim/vim-plugins/vim-rainbow/ ~/.vim/pack/plugins/start`
- `ln -s ~cs2030s/.vim/vim-plugins/nerdtree/ ~/.vim/pack/plugins/start`
- `ln -s ~cs2030s/.vim/vim-plugins/lightline/ ~/.vim/pack/plugins/start`

If does not work, try

`ln -s ~cs2030s/.vim/pack/plugins/start/<target plugin>/ ~/.vim/pack/plugins/start/`

Should be preinstalled already

Vim shortcuts

- `<ctrl>+w v` - split vertically
- `<ctrl>+w s` - split horizontally
- `:wqa` - write to all and quit all (close every window and buffer)
- `:term` - launches terminal
- `:resize <number>` - resizes windows
- `:vertical resize <number>` - resizes windows
- `<ctrl>+n` - autocomplete with text in file

Code snippets

ArrayStack

```
public interface Stack<T> {
  T pop();
  void push(T t);
  int getStackSize();
}

public class ArrayStack<T> implements Stack<T> {
  private T[] arr;
  private final maxDepth;
  private int index;

  public ArrayStack(int maxDepth) {
    @SuppressWarnings("unchecked");
    T[] temp = (T[]) new Object[maxDepth];
    this.arr = temp;
    this.maxDepth = maxDepth;
  }

  public static <T> ArrayStack<T> of(T[] arr, int maxDepth) {
    ArrayStack<T> arrStack = new ArrayStack<T>(maxDepth);
    for (int i = 0; i < maxDepth; i++) {
      arrStack.push(this.arr[i]);
    }

    return arrStack;
  }
```

```
public void pushAll(ArrayStack<? extends T> other) {
  T o = other.pop();
  while (o != null) {
    this.push(o);
    o = other.pop();
  }
}

public void pushAll(ArrayStack<? super T> other) {
  T t = this.pop();
  while (t != null) {
    other.push(t);
    t = this.pop();
  }
}

@Override
public void push(T t) {
  if (this.index < this.maxDepth) {
    this.arr[this.index] = t
    this.index++;
  }
}

@Override
public T pop() {
  if (this.index == 0) {
    return null;
  }

  // Might need to move this on top and check if this.index < 0
  this.index--;
  T t = this.arr[this.index];
  this.arr[this.index] = null;
  return t;
}

@Override
public int getStackSize() {
  return this.index;
}

@Override
public String toString() {
  String contents = "";
  for (int i = 0; i < this.arr.length; i++) {
    if (this.arr[i] != null) {
      contents += this.arr[i].toString() + " ";
    }
  }
```

```
    return String.format("Stack: %s", contents);
  }
}
```

Generic Array

```
public class Array<T extends Comparable<T>> {
  private T[] array;

  public Array(int size) {
    @SuppressWarnings("unchecked")
    T[] temp = (T[]) new Comparable<?>[size];
    this.array = temp;
  }

  public void set(int index, T item) {
    this.array[index] = item;
  }

  public T get(int index) {
    return this.array[index];
  }

  public int length() {
    return this.array.length;
  }

  public T min() {
    if (this.array.length == 0) {
      return null;
    }

    T smallest = this.array[0];
    for (int i = 1; i < this.array.length; i++) {
      T current = this.get(i);

      if (current.compareTo(smallest) < 0) {
        smallest = current;
      }
    }

    return smallest;
  }

  @Override
  public String toString() {
    StringBuilder s = new StringBuilder("[ ");
    for (int i = 0; i < this.array.length; i++) {
      s.append(i + ":" + this.array[i]);
      if (i != this.array.length - 1) {
        s.append(", ");
```

```
      }
    }
    return s.append(" ]").toString();
  }
}
```

Queue

```
public class Queue<T> {
  private T[] items;

  private int first;

  private int last;

  private int maxSize;

  private int len;

  public Queue(int size) {
    this.maxSize = size;
    @SuppressWarnings("unchecked")
    T[] temp = (T[]) new Object[size];
    this.items = temp;
    this.first = -1;
    this.last = -1;
    this.len = 0;
  }

  public boolean enq(T e) {
    if (this.isFull()) {
      return false;
    }
    if (this.isEmpty()) {
      this.first = 0;
      this.last = 0;
    } else {
      this.last = (this.last + 1) % this.maxSize;
    }
    this.items[last] = e;
    this.len += 1;
    return true;
  }

  public T deq() {
    if (this.isEmpty()) {
      return null;
    }
    T item = this.items[this.first];
    this.first = (this.first + 1) % this.maxSize;
    this.len -= 1;
```

```
    return item;
  }

  boolean isFull() {
    return (this.len == this.maxSize);
  }

  boolean isEmpty() {
    return (this.len == 0);
  }

  public int length() {
    return this.len;
  }

  @Override
  public String toString() {
    String str = "[ ";
    int i = this.first;
    int count = 0;
    while (count < this.len) {
      str += this.items[i] + " ";
      i = (i + 1) % this.maxSize;
      count++;
    }
    return str + "]";
  }
}
```

LinkedList

Core idea is to implement a LinkedList that accounts for the start and end of the list

```
class Node<T> {
  private T data;
  private Node<T> next;

  public Node(T data) {
    this.data = data;
  }

  public Node<T> getNext() {
    return this.next;
  }

  public void setNext(Node<T> next) {
    this.next = next;
  }

  public T getData() {
    return this.data;
```

```
  }
}

class LinkedList<T> {
  private Node<T> head;
  private int length;

  public void add(T data) {
    Node<T> temp = new Node<>(data);
    if (this.head == null) {
      this.head = temp;
    } else {
      Node<T> x = this.head;
      while(x.getNext() != null) {
        x = x.getNext();
      }
      x.setNext(temp);
    }

    length++;
  }

  public void add(int position, T data) {
    if (position > this.length + 1) {
      return;
    }

    if (position == 1) {
      Node<T> temp = this.head;
      this.head = new Node<T>(data);
      this.head.setNext(temp);
      return;
    }

    Node<T> temp = this.head;
    Node<T> prev = new Node<T>(null);
    while(position - 1 > 0) {
      prev = temp;
      temp = temp.next;
      position--;
    }

    prev.setNext(new Node<T>(data));
    prev.getNext().setNext(temp);
  }

  public void remove(T key) {
    // Dummy node
    Node<T> prev = new Node<>(null);
    prev.next = this.head;
```

```
    Node<T> next = this.head.getNext();
    Node<T> temp = this.head;

    boolean exists = false;
    if (this.head.getData() == key) {
      this.head = this.head.getNext();
      exists = true;
    }

    while (temp.getNext() != null) {
      if (String.valueOf(temp.getData()).equals(String.valueOf(key))) {
        prev.next = next;
        exists = true;
        break;
      }

      prev = temp;
      temp = temp.getNext();
      next = temp.getNext();
    }

    if (!exists && String.valueOf(temp.getData()).equals(String.valueOf(key))) {
      prev.next = null;
      exists = true;
    }

    if (exists) {
      length--;
    }
  }

  public boolean isEmpty() {
    return this.head == null;
  }

  public int length() {
    return this.length;
  }

  public void clear() {
    this.head = null;
    this.length = 0;
  }
}
```

Functional lab

```
public interface Action<T> {
  void call(T t);
}
```

```java
public interface Actionable<T> {
  void act(Action<? super T> action);
}

public interface Applicable<T> {
  <R> Probably<R> apply(
      Probably<? extends Immutator<? extends R, ? super T>> p);
}

public interface Immutator<R, P> {
  R invoke(P param);
}

public interface Immutatorable<T> {
  <R> Immutatorable<R> transform(Immutator<? extends R, ? super T> immutator);
}

public class Improbable<T> implements Immutator<Probably<T>, T> {
  @Override
  public Probably<T> invoke(T param) {
    return Probably.just(param);
  }
}

public class IsModEq implements Immutator<Boolean, Integer> {
  private int div;
  private int check;

  public IsModEq(int div, int check) {
    this.div = div;
    this.check = check;
  }

  @Override
  public Boolean invoke(Integer val) {
    return (val % this.div) == this.check;
  }
}

public class Print implements Action<Object> {
  @Override
  public void call(Object obj) {
    System.out.println(obj);
  }
}

class Probably<T> implements
    Actionable<T>,
    Immutatorable<T>,
    Applicable<T> {
  private final T value;
```

```java
private static final Probably<?> NONE = new Probably<>(null);

private Probably(T value) {
  this.value = value;
}

public static <T> Probably<T> none() {
  @SuppressWarnings("unchecked")
  Probably<T> res = (Probably<T>) NONE;
  return res;
}

public static <T> Probably<T> just(T value) {
  if (value == null) {
    return none();
  }
  return (Probably<T>) new Probably<>(value);
}

public Probably<T> check(Immutator<Boolean, ? super T> immutator) {
  if (this.value == null) {
    return Probably.none();
  }

  if (immutator.invoke(this.value).equals(Boolean.TRUE)) {
    return this;
  }

  return Probably.none();
}

@Override
public void act(Action<? super T> action) {
  if (this.value != null) {
    action.call(this.value);
  }
}

@Override
public <R> Probably<R> transform(
    Immutator<? extends R, ? super T> immutator) {
  if (this.value == null) {
    return Probably.none();
  }

  return Probably.just(immutator.invoke(this.value));
}

@Override
public <R> Probably<R> apply(
```

```
      Probably<? extends Immutator<? extends R, ? super T>> p) {
    if (this.value == null || p.value == null) {
      return Probably.none();
    }

    return Probably.just(p.value.invoke(this.value));
  }

  @Override
  public boolean equals(Object obj) {
    if (obj == this) {
      return true;
    }
    if (obj instanceof Probably<?>) {
      Probably<?> some = (Probably<?>) obj;
      if (this.value == some.value) {
        return true;
      }
      if (this.value == null || some.value == null) {
        return false;
      }
      return this.value.equals(some.value);
    }
    return false;
  }

  @Override
  public String toString() {
    if (this.value == null) {
      return "<>";
    } else {
      return "<" + this.value.toString() + ">";
    }
  }
}
```