CS2030S PE1
AY25/26 Sem 1

12 marks design, 3 marks style, 5 marks correctness

General Tips

- **Compile** with `javac -Xlint:unchecked -Xlint:rawtypes *.java`
- **Student number** on top of every file:

```
/**
 * @author STUDENT_NUMBER
 */
```

- **Increment coding**: check if your program compiles after every increment
- Avoid implementing accessors and mutators
- **Check style**: `java -jar /opt/course/cs2030s/bin/checkstyle.jar -c /opt/course/cs2030s/bin/cs2030_checks.xml *.java`
- Remember to **bash submit** *both* tasks

Setup

- **.vimrc**: uncomment `set mouse+=a` , `set hidden` , `set wildmenu` , `set showcmd`
- If accidentally edited `part2.md` ,
  1. Remove `Secret.class` and `part2.secret` files
  2. Copy these files from `pristine` into home directory

Basic Commands

Unix:

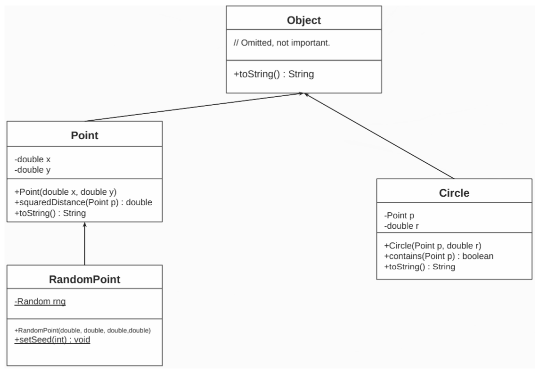
- `ls` with flags `-a` `-l`
- `cd <dirname>`
- `cp <scr> <dst>`
- `mv <scr> <dst>`
- `rm <filename>`
- `rm -r <dirname>`
- `~` home, `.` current, `..` parent
- `javac *.java`
- `java ...`
- `logout`
- If `Ctrl+z` , then `Ctrl+c`

Vim:

- `dd` delete line `dw` delete word
  `d<num>` delete next num lines
  `d$` delete from cursor to end of line
- `yy` yank (copy) line `<num>yy` yank num lines
  `y0` yank from cursor to start of line
- `p` paste `u` undo `gg=G` auto-indent
- `/<text>` find text `n` find next, `N` find previous
- `:%s/<old>/<new>/gc` replace
- `Ctrl+n` autocomplete
- `:vsp <filename>` split screen
  `:e <filename>` open file
- `:term` launch terminal
- `:!javac *.java` `:!java ...`
  `jshell` , then `/open ...java, ...` , finally `/exit`

Design (15 mins)

- **OOP Design**
  1. What are the **nouns**? Good candidates for new classes
  2. For each class, what are its **attributes / properties**?
  3. Classes related via **IS-A** or **HAS-A** relationship?
  4. For each class, what are their **responsibilities**?
  5. How do the objects of each class **interact**?
     Class A TELLS Class B something to do ...
  6. What are some **behaviour that changes** depending on specific type of Objects? E.g. `toString()`
     Use INHERITANCE and POLYMORPHISM to make adding new classes easier!
- **Class diagram**


Object
// Omitted, not important.
+toString() : String
Point
-double x
-double y
+Point(double x, double y)
+squaredDistance(Point p) : double
+toString() : String
Circle
-Point p
-double r
+Circle(Point p, double r)
+contains(Point p) : boolean
+toString() : String
RandomPoint
-Random rng
+RandomPoint(double, double, double,double)
+setSeed(int) : void

Composition, Inheritance

- **Composition**: HAS-A relationship
- **Inheritance**: IS-A relationship (use `extends`)
- **Information hiding**: always prioritise using `private` access modifier for all class fields
- **Tell, Don't Ask**: TELL an object what to do, instead of ASKING the internal fields and perform the computation on the object's behalf
  Avoid getters and setters

Overriding

- Subclass defines instance method with the same method descriptor as superclass
- Must use `@Override` annotation
- Override `toString()` : customise string representation

```java
@Override
public String toString() {
  // Use string formatting
  return String.format("%s %s", s1, s2);
  // %s string  %d decimal  %f float
  // %.1f 1 d.p. float  %c char
}
```

  Every class should have its `toString()` method
- Override `equals(Object)` : compare objects semantically (based on field values)
  Use `String::equals(Object)` to compare strings instead of `==`
- Override `int compareTo(T)` : compare this object with the specified object → returns < 0 if less than, 0 if equal, > 0 if greater than

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

Polymorphism

- Use **abstract class / interface** when behaviour must be shared but implemented differently by subclasses
- Subclasses **override** methods to provide specific behaviour
  E.g. `obj.toString()` : override `Point::toString()` and `Circle::toString()` for customised output
- Avoid using `instanceof` for type checks → this is contrary to the notion of polymorphism

Exception

Unchecked exceptions <: `RuntimeException` :

```
class IllegalCallException extends
    RuntimeException {
  public IllegalCallException(String message
      ) {
    super(message);
  }
}

class NoCallerId {
  public void callBack(int duration) throws
      IllegalCallException {
    throw new IllegalCallException(this);
  }
}

class Main {
  public static void main(String[] args) {
    NoCallerId c = NoCallerId();
    c.callBack(5); // may crash program if
        uncaught
  }
}
```

Checked exceptions <: `Exception` :

```
class IllegalCircleException extends
    Exception {
  public IllegalCircleException(String
      message) {
    super(message);
  }
}

class Circle {
  private Point c;
  private double r;

  public Circle(Point c, double r) throws
      IllegalCircleException {
    if (r < 0) {
      throw new IllegalCircleException("
          radius cannot be negative.");
    }
    this.c = c;
    this.r = r;
  }

  @Override
  public String toString() { ... }
}

class Main {
  public static void main(String[] args) {
    try {
      c = new Circle(point, radius);
    } catch (IllegalCircleException e) {
      System.err.println("Illegal arguement:
          " + e.getMessage());
    }
  }
}
```

Generics

- Pair class

```
class Pair<S,T> {
  private S first;
  private T second;

  public Pair(S first, T second) {
    this.first = first;
    this.second = second;
  }

  public S getFirst {
    return this.first;
  }

  public T getSecond {
    return this.second;
  }
}
```

- `equals(Object)` method for `Pair`

  1. Check if `obj` is a `Pair` :

     `obj instanceof Pair<?, ?>`

  2. If yes, typecast `obj` to `Pair<?, ?>`
     (no need to suppress warning)

  3. Compare fields using fields' `equals(Object)` method

- Create **generic array**:

  1. Cannot instantiate generic array `new T[]` directly, so we instantiate `Object[]` then cast it to `T[]`

  2. Must include comment to explain why type safe

  3. Suppress warning cannot apply to assignment, only declaration. Should be used within the method only (most limited scope)

  4. If array contains `Pair` objects, we cannot use `new Object[size]` ; use `new Pair<?, ?>[size]` instead.

```
class Seq<T> {
  private T[] array;

  public Seq(int size) {
    /**
     * Only way to put object into array is
     * through set(), and we only put
     * object of type T inside.
     * Safe to cast Object[] to T[].
     */
    @SuppressWarnings("unchecked")
    T[] a = (T[]) new Object[size];
    this.array = a;
  }

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

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

- To compare objects:

```
class Seq<T extends Comparable<T>> {
  // :
  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;
  }
}
```

- **Wildcards**: allow subtyping relationship (covariance/contravariance)

  ◦ Unbounded: `A<?>`

    Upper bounded: `A<? extends T>`

    Lower bounded: `A<? super T>`

  ◦ **Raw types** are banned; instead, use wildcards

```
new Comparable[10];      // avoid this
new Comparable<?>[10];   // good
```

```
a instanceof A<String>   // doesn't work
    since type argument 'String' is not
    available during run-time due to
    erasure
a instanceof A           // avoid this
a instanceof A<?>        // good
```

    1 raw type = 1 mark deducted!

  ◦ **Producer Extends, Consumer Super** (PECS)

    Producer: method produces generic `T`

    Consumer: method takes in generic `T` as input