Day 5

COMP1006/1406
Summer 2016

M. Jason Hinek
Carleton University
today’s agenda

- assignments
  - Assignment 2 is in
  - Assignment 3 is out

- a quick look back
  - inheritance and polymorphism

- interfaces
  - the Comparable interface

- Problem solving

- assignments
last time...

inheritance allows us to reduce duplicate code by sharing code among different classes

▶ when we extend a class we inherit all public/protected attributes and methods from that class

▶ we can only have one direct parent class (except Object which has no parent)

polymorphism - having multiple forms

▶ we have seen three kinds of polymorphism
  ▶ method overloading
  ▶ method overriding
  ▶ subtype polymorphism
interfaces

let’s first review **abstract classes** in Java

```java
public abstract class Insect{
    ...
}
```

- related classes **extend** the same **abstract** class
  - similar (general) behaviour is implemented in the abstract class itself
  - different (specific) behaviour is implemented in overridden methods
- you can only **extend** one single class (abstract or not)
- abstract class can provide a contract between code and users of code
- the abstract class defines what an object **is**
  - name is usually a noun
interfaces

an **interface** in Java is **similar** to an abstract class

```java
public interface Printable{
    ...
}
```

- are valid reference data types
- cannot be instantiated
- intention is for other classes to *implement* it (using *implements*)
  - public class A extends Q implements B{...}
  - public class A implements B{...}
  - A is-a B

- provides a contract between code and users of code
interfaces
an interface in Java is different from abstract classes

```java
public interface Printable{
    ...
}
```

- classes that implement an interface can be very unrelated
- a class can implement any number of interfaces (using `implements`)
  - public class A extends Q implements B,C,D{...}
  - A is-a B and A is-a C and A is-a D (and A is-a Q)
  - no implicit interface implemented
- interfaces usually define what an object can do
  - name is usually an adjective (Comparable, Serializable)
  - can also define what you are (List, Set) but doesn’t specify the data
interfaces

```java
public interface Printable{

    int MAX_NUM_JOBS = 99; // constant

    boolean sendToPrint(String); // abstract
    boolean killPrintJob(int); // methods

}
```

- can contain abstract method declarations (without definition)
  - are public by default (you can omit this)
  - implicitly abstract (you do not write this)

- can contain constant class attributes
  - are public static final by default (can omit this)

- can have default methods and static methods (these have definitions) and enum types
Consider the `Comparable` interface

```java
public interface Comparable<Type>{
    int compareTo(Type other);
    // returns an integer X satisfying
    //    X < 0 if this is "less than" other
    //    X = 0 if this is "equal to" other
    //    X > 0 if this is "greater than" other
}
```

- declared like a `class`, using `interface` instead
- has a single method called `compareTo()`
  - by default all methods are `public abstract`
- uses generics (we’ll revisit this in more detail later)
  - allows us to treat types as parameters `<type>`
  - specifies exactly what we can compare our objects with
  - avoids the messiness we saw with `String’s equals()`
Comparable

using the Comparable interface

```java
public interface Comparable<Type>{
    int compareTo(Type other);
}
```

```java
public class MyClass implements Comparable<MyClass>{
    // ...
}```

- class that implements the Comparable interface either
  - must override (define) the method compareTo, or
  - must be abstract

```java
public interface SomeInterface extends Comparable<T>{
    // ...
}
```

- interfaces can extends any number of other interfaces
public class Student implements Comparable<Student>{
    private String name;
    private int id;

    @Override
    public static int compareTo(Student other){
        if(other == null){
            return 1;
        }
        return this.getID() - other.getID();
    }

    ...

}

Student s = new Student("cat", 12);
Student t = new Student("dog", 7);
System.out.println(s.compareTo(t));
public class Student implements Comparable<Student>{
    private String name;
    private Integer id;

    @Override
    public static int compareTo(Student other){
        if(other == null){
            return 1;
        }
        /* use built-in compareTo of other objects to help us */
        return this.getID().compareTo(other.getID());
    }

    ...
The **Arrays** class provides static methods to help work with arrays.

**toString(...)** prints an array nicely (like Python)

```java
House[] houses = new House[3]{ ... };
System.out.println(java.util.Arrays.toString(houses));
```

**sort(...)** sorts the elements in an array

```java
java.util.Arrays.sort(houses);
System.out.println(java.util.Arrays.toString(houses));
```
let’s take a break…
for 3 minutes
Problem Solving

George Pólya

- How to Solve it

- Terminology: data, unknown, condition
Problem Solving

Using the given **data** to find the **unknown** such that the **condition** is satisfied.

- the data is the information you have.
- the unknown is the information you want.
- the condition is the constraints on the problem.
  These are rules (often implicit) that must be followed.

Alternatively...

Using the given **data** to achieve a **goal** such that the **condition** is satisfied.

Using the given **data** to create an **algorithm/program** that achieves a **goal** such that the **constraints** are satisfied.
Problem Solving

The four phases of problem solving

1. Understand the problem.
   - identify the data/unknown/condition

2. Devise a plan.
   - choose a technique/heuristic/approach
   - start over if needed

3. Carry out the plan.
   - execute your plan
   - check each step
   - start over if needed

4. Look back.
   - reflect on what you did
   - start over if needed
Problem Solving

General strategies

- Related problems
  - transform the problem into one you already know how to solve

- Abstraction
  - remove details that are not relevant to the problem

- Divide and Conquer
  - break the problem into (smaller) sub-problems

- Backward Chaining
  - start from the solution and work backwards
Problem Solving

General strategies (from Think Like a Programmer, V. A. Spraul)

- Always have a plan
- Restate the problem
- Break the problem down
- Start with what you know
- Reduce the problem
- Look for analogies
- Experiment
- Don’t get frustrated!
Problem Solving

General strategies (from Think Like a Programmer, V. A. Spraul)

- Always have a plan
  - Aimless wandering wastes time.
  - Without a plan, you are hoping for a lucky break.
  - Plans give you intermediate goals.
  - Plans can change.

- Restate the problem
  - Check out the problem from every angle before starting.
  - We may find the goal is not what we thought.
  - Use restatement to confirm understanding.

- Break the problem down
  - Divide the problem into steps or phases.
  - Difficulty for each phase can be an order of magnitude lower.
  - Sometimes the sub-problems are hidden.
Problem Solving

General strategies (from Think Like a Programmer, V. A. Spraul)

- **Start with what you know**
  - Fully investigate a problem with the skills you have first.
  - Build confidence and momentum towards your goal.
  - You may learn more about the problem this way.

- **Reduce the problem**
  - Reduce scope by adding or removing constraints.
  - Work on a simpler problem that isn’t easily divided.
  - Pinpoint where remaining difficulties lie.

- **Look for analogies**
  - Look for similarities to problems you’ve already solved.
  - Recognizing analogies improves speed and skill.
  - You need to build up a store of prior problems before you can find analogies.
Problem Solving

General strategies (from Think Like a Programmer, V. A. Spraul)

- **Experiment**
  - Try things and observe the results (this is not guessing!).
  - Trial-and-error is a valid approach to problem solving (not to be confused with guessing)
  - Make small test programs.

- **Don’t get frustrated**
  - Everything will seem to take longer and be harder!
  - Avoiding frustration is a decision you make.
  - Go back to the plan, work on a different problem, or take a break.
Problem Solving

General strategies (from Think Like a Programmer, V. A. Spraul)

- Experiment
  - Try things and observe the results (this is not guessing!).
  - Trial-and-error is a valid approach to problem solving (not to be confused with guessing)
  - Make small test programs.

- Don’t get frustrated
  - Everything will seem to take longer and be harder!
  - Avoiding frustration is a decision you make.
  - Go back to the plan, work on a different problem, or take a break.
reflection...

Questions to ask yourself about assignment 1

- did I understand the questions?
  - what was given?
  - what was needed to be done?
  - what were the constraints?

- did I test my code?
  - did I verify any given example code?
  - did I generate test cases to extensively test my code

- how did I try to get help if I didn’t understand the questions?

- did I give myself enough time to complete the assignment?
let’s take a break... for 3 minutes
some review slides
(in progress)
abstract methods
- a method declared without a definition
- public abstract int foo(String[] in);
- forces the class to be abstract as well
- cannot be final

abstract classes
- cannot be instantiated
- may or may not contain abstract methods
- are valid reference types and can be subclassed
- cannot be final

concrete classes
- all methods (declared or inherited) must be defined
- can be instantiated (all objects other than arrays are instantiations of concrete classes)
- is a valid data reference type
final...

- **final attributes**
  - value cannot be changed once it is defined
  - must be defined in constructor or initialization block
  - primitive data types, strings and immutable data types are **constants**

- **final methods**
  - cannot be overridden
  - cannot be **abstract**

- **final classes**
  - cannot be extended
  - cannot be **abstract**
access modifiers...

<table>
<thead>
<tr>
<th>modifier</th>
<th>class</th>
<th>package</th>
<th>subclass</th>
<th>world</th>
</tr>
</thead>
<tbody>
<tr>
<td>public</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>protected</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>none (default)</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>private</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
</tbody>
</table>

- everything is accessible from within the class
- a class in the same package has access to everything except private members
- a subclass has access to public and protected members
- everyone else only has access to public members
arrays...

an **array** is a container that store a collection of items of the same type

```java
int[] intArray; // variable declaration

intArray = new intArray[12]; // allocation of memory in heap for array

intArray[0] = 13; //
... // population of the array with data
```

When you declare an array variable you can also initialize it using `{...}`. This only works when you declare the variable.

```java
/* array declaration, allocation and initialization */
int[] intArray = {1,3,5,7,9};

/* all of these are equivalent */
String[] words = {"cat", "dog", "eel"};
String[] words = new String[] {"cat", "dog", "eel"};
String[] words = new String[3] {"cat", "dog", "eel"};
```