Java + OOP
CSC207 Winter 2018
Why OOP?

- **Modularity**: code can be written and maintained separately, and easily passed around the system.

- **Information-hiding**: internal representation hidden from the outside world.

- **Code re-use**: others can implement/test/debug complex code, which you can then use in your own code.

- **Pluggability and debugging ease**: “If a bolt breaks, you replace it, not the entire machine.”

- All together: SOLID design principles (more later, and again and again… and in A2)

https://docs.oracle.com/javase/tutorial/java/concepts/object.html
Object-Oriented Programming

- What you have seen so far...
  - Java
  - Classes
  - Static
  - Casting
  - etc.

- This week...
  - Inheritance
  - Overloading
  - Overriding
  - Shadowing
  - Polymorphism
  - Autoboxing
Inheritance Hierarchy

- All classes form a tree called the inheritance hierarchy, with Object at the root.

- Class Object does not have a parent. All other Java classes have one parent.

- If a class has no parent declared, it is a child of class Object.

- A parent class can have multiple child classes.

- Class Object guarantees that every class inherits methods toString, equals, and others.
Inheritance

• Inheritance allows one class to inherit the data and methods of another class.

• In a subclass, `super` refers to the part of the object defined by the parent class.

• Use `super."attribute"` to refer to an attribute (data member or method) in the parent class.

• Use `super("arguments")` to call a constructor defined in the parent class.
Multi-part objects

• Suppose class *Child* extends class *Parent*. (IS-A relationship)

• An instance of Child has:
  • a *Child* part, with all the data members and methods of Child
  • a *Parent* part, with all the data members and methods of Parent
  • a *Grandparent* part, … etc., all the way up to *Object*.

• An instance of *Child* can be used anywhere that a *Parent* is legal.

• But not the other way around.
To the demo...

Inheritance
Method Overloading

- Methods with the **same name** but **different parameters**
  - e.g. four versions of Math.abs():
    - double abs(double d), float abs(float f), int abs(int i), long abs(long lng)
  - Constructors are often overloaded
Back to the demo...

Method Overloading
Constructor Overloading
Shadowing and Overriding

- Suppose class A and its subclass AChild each have an instance variable x and an instance method m.
- A’s m is overridden by AChild’s m.
  - This is often a good idea. We often want to specialize behaviour in a subclass.
- A’s x is shadowed by AChild’s x.
  - This is confusing and rarely a good idea.
- If a method must not be overridden in a descendant, declare it final.
Name Lookup

• A subclass can reuse a name already used for an inherited data member or method.

• Example:
  
  • class Person could have a data member motto and so could class Student. Or they could both have a method with the signature sing().

  • When we construct
    
    x = new Student();
    
    the object has a Student part and a Person part.

  • If we say x.motto or x.sing(), we need to know which one we’ll get!

• In other words, we need to know how Java will look up the name motto or sing inside a Student object.
Name Lookup Rules

• Calling a method: `expression.method(arguments)`
  
  • Java looks for method in the most specific, or bottom-most part of the object referred to by `expression`.
  
  • If it’s not defined there, Java looks “upward” until it’s found (else it’s an error).

• Referencing an instance variable: `expression.variable`
  
  • Java determines the type of `expression`, and looks in that box.
  
  • If it’s not defined there, Java looks “upward” until it’s found (else it’s an error).
Back to the demo...

Shadowing
Casting for the compiler

• If we could run this code, Java would find the `charAt` method in `o`, since it refers to a `String` object:

```
Object o = new String("hello");
char c = o.charAt(1);
```

• But the code won’t compile because the compiler cannot be sure it will find the `charAt` method in `o`.

  • Remember: the compiler doesn’t run the code. It can only look at the type of `o`.

• So we need to cast `o` as a `String`:

```
char c = ((String) o).charAt(1);
```
Polymorphism

- **Definition**: the ability of one thing to have multiple forms (i.e. inheritance, overloading, etc.), or one form to apply to several things (i.e. interfaces)

- **Example**: if `Student` and `Instructor` both extend `Person`

  ```java
  Person p;
p = new Person("Lindsey"); // OK
p = new Student("Jaisie"); // OK
p = new Instructor("Paul"); // OK
  ```

https://docs.oracle.com/javase/tutorial/java/IandI/polymorphism.html
Back to the demo...

Polymorphism
Autoboxing/Unboxing

• The automatic conversion Java makes between primitive types and their respective object wrappers

• It makes code clean!

• e.g. `Integer i = 5;`

• **Autoboxing**: primitive -> wrapper

• **Unboxing**: wrapper -> primitive

• Good for when we need object versions of things (e.g. in generics)

<table>
<thead>
<tr>
<th>Primitive Type</th>
<th>Wrapper Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>Boolean</td>
</tr>
<tr>
<td>byte</td>
<td>Byte</td>
</tr>
<tr>
<td>char</td>
<td>Character</td>
</tr>
<tr>
<td>float</td>
<td>Float</td>
</tr>
<tr>
<td>int</td>
<td>Integer</td>
</tr>
<tr>
<td>long</td>
<td>Long</td>
</tr>
<tr>
<td>short</td>
<td>Short</td>
</tr>
<tr>
<td>double</td>
<td>Double</td>
</tr>
</tbody>
</table>

[https://docs.oracle.com/javase/tutorial/java/data/autoboxing.html](https://docs.oracle.com/javase/tutorial/java/data/autoboxing.html)