Inheritance in Java

CSC207 Winter 2018
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.

• 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.
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
    
    \[
    x = \text{new } \text{Student}();
    \]
    
    the object has a Student part and a Person part.
  • If we say \( x.\text{motto} \) or \( x.\text{sing}() \), 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).
Shadowing and Overriding

- Suppose class $A$ and its subclass $A$Child each have an instance variable $x$ and an instance method $m$.

- $A$’s $m$ is **overridden** by $A$child’s $m$.
  - This is often a good idea. We often want to specialize behaviour in a subclass.

- $A$’s $x$ is **shadowed** by $A$child’s $x$.
  - This is confusing and rarely a good idea.

- If a method must not be overridden in a descendant, declare it **final**.
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:

  ```java
  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`:

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