Inheritance in Java

CSC207 Fall 2016

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 automatically 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.

Constructors and inheritance

If the first step of a constructor is `super(«arguments»)`, the appropriate constructor in the parent class is called.

- Otherwise, the no-argument constructor in the parent is called.

Net effect on order if, say, A is parent of B is parent of C?

Which constructor should do what? Good practise:

- Initialize your own variables.
- Count on ancestors to take care of theirs.

private variable in parent class cant be accessed in the child 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 = 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

For a method call: `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).

For 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 `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 can be confusing. Use with caution.

If a method must not be overridden in a descendant, declare it `final`.

always declare final

final method cannot be overriden

boxing : primitive > object
unboxing: object > primitive
casting: primitive > primitive
object > object

Casting for the compiler

casitng is basically telling the compiler that an Object A is more specific than an object, and grants the methods of a more specific class B

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

usually its String here

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

o has properties of Object and point at a string
1. however cannot directly use String's method

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);
```

cast Object > primitive

Javadoc

Like a Python docstring, but more structured, and placed above the method.

```
/**
 * Replace a square wheel of diagonal diag with a round wheel of
 * diameter diam. If either dimension is negative, use a wooden tire.
 * @param diag  Size of the square wheel.
 * @param diam  Size of the round wheel.
 * @throws PiException  If pi is not 22/7 today.
 */
public void squareToRound(double diag, double diam) { ... }
```

Javadoc is written for classes, member variables, and member methods.

This is where the Java API documentation comes from!

In Eclipse: Project → Generate Javadoc

Java naming conventions

The Java Language Specification recommends these conventions

Generally: Use `camelCase` not `pothole_case`.

Class name: A noun phrase starting with a capital.

Method name: A verb phrase starting with lower case.

Instance variable: A noun phrase starting with lower case.

Local variable or parameter: ditto, but acronyms and abbreviations are more okay.

Constant: all uppercase, pothole.

E.g., MAX_ENROLMENT

Direct initialization of instance variables

You can initialize instance variables inside constructor(s).

An alternative: initialize in the same statement where they are declared.

Limitations:

- Can only refer to variables that have been initialized in previous lines.
- Can only use a single expression to compute the initial value.

What happens when

1. Allocate memory for the new object.
2. Initialize the instance variables to their default values:

   0 for ints, `false` for booleans, etc., and `null` for class types.

3. Call the appropriate constructor in the parent class.

   The one called on the first line, if the first line is `super(`*arguments*`)`, else the no-arg constructor.

4. Execute any direct initializations in the order in which they occur.
5. Execute the rest of the constructor.

Abstract classes and interfaces

A class may define methods without giving a body. In that case:

- Each of those methods must be declared `abstract`.
- The class must be declared `abstract` too.
- The class can't be instantiated.

A child class may implement some or all of the inherited abstract methods.

abstract class is used if want to have a default implementation, which cant be done with interfaces

- If not all, it must be declared `abstract`.
- If all, it's not abstract and so can be instantiated.

If a class is completely abstract, we may choose instead to declare it to be an `interface`.

interface is where to declare any methods that a class MUST have

keep in mind that class can only extend one class but may interfaces; have to choose class wisely

Interfaces

cannot store data

An interface is (usually) a class with no implementation.

It has just the method signatures and return types.

It guarantees capabilities.

Example: `java.util.List`

"To be a List, here are the methods you must support."

A class can be declared to `implement` an interface.

This means it defines a body for every method.

A class can implement 0, 1 or many interfaces, but a class may extend only 0 or 1 classes.

An interface may extend another interface.

Generics: naming conventions

The Java Language Specification recommends these conventions for the names of type variables:

- very short, preferably a single character
- but evocative
- all uppercase to distinguish them from class and interface names

Specific suggestions:

Maps:	K, V
Exceptions:	X
Nothing particular:	T (or S, T, U or T1, T2, T3 for several)