Lecture 3 — Java II

Patrick Lam & Jeff Zarnett
p.lam@ece.uwaterloo.ca & jzarnett@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

November 14, 2015

Refresher: Object-oriented Programming

Fundamental idea: use “objects” to encapsulate data.

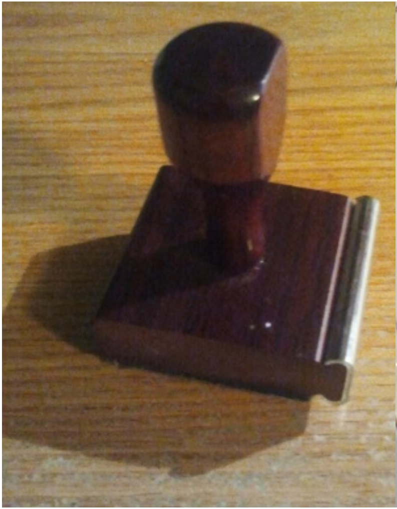
Instantiate new objects with the new keyword.

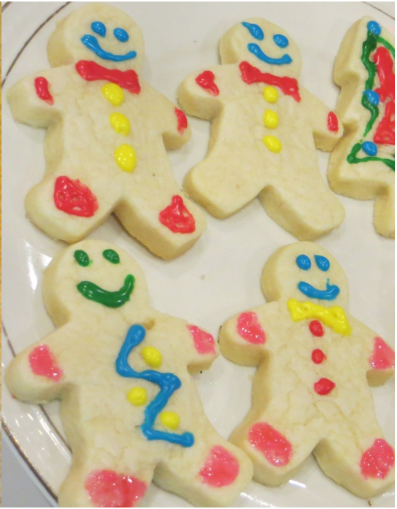
```
class[1] Point { int x, y; }
```

```
Point p = new Point();
```

[1] Java doesn’t have `struct`.

Classes versus Instances





Which is the class? Which is the instance?

Classes and Relations

Java, like C#, has *inheritance*.

B inherits from *A* if *B* `extends` *A* by using (at least some of) its methods and fields in addition to any of its own.

B is a “subclass” or “child” of *A*

A is a “superclass” or “parent” of *B*.

In this class we’ll use the subclass/superclass terminology.

Class Hierarchy

In Java, all classes have a single inheritance hierarchy: each class has exactly one superclass.

(Other languages like C++ allow multiple inheritance)

The keyword in Java for declaring a class as a subclass of another is `extends`.

For example: `public class Book extends Document`

Class Hierarchy

Every object in Java eventually descends from `Object`.

If you do not declare a specific superclass with the `extends` keyword, the superclass will be `Object`.

Subclassing Example

```
class Mug extends Donut { ... }
class WateringCan extends Donut { ... }
```

Conceptual Meaning of Subclassing

```
class Mug extends Donut { ... }
class WateringCan extends Donut { ... }
```

Subclassing encodes the “is-a” relationship.

Advantages of Sucblassing

Avoiding repetition: don’t re-implement code that is common.

Polymorphism – the ability of a class to look like another.

Example: method expects `Rectangle`. You can use it on a `Square` and it will work if `Square extends Rectangle`.

Subclassing Calling Semantics

On (or inside of) a subclass we can call a method of the parent class.

Book descends from `Object`, either directly or indirectly.

Given that `toString()` is defined on `Object`, we can call the `toString()` method on a Book.

Subclassing Calling Semantics

Also from within a subclass, we can use an implementation of a method from a parent class with the keyword `super`.

The most common situation where this happens is in the constructor.

Subclassing Calling Semantics

```
public class Book extends Document {

  public Book() {
    super();
    setPageWidthInCm(20);
    setPageHeightInCm(30);
  }

  ...
}
public class Document implements Readable

  public Document() {
    setRead(false);
    setTitle(‘‘Untitled’’);
    setAuthor(Environment.getUserID());
  }

  ...
}
```

`static`

When you see the keyword `static`, it is a modifier that means there is only one copy of the thing.

`main()` in Java is declared as `static`;
  declare variables as `static` to reference them in `main`.

Let’s examine the semantics of this keyword.

Static Variable

The variable is common to all instances of a class.

Assume you have an integer `i` in class A and there are two instances of the class called A_1 and A_2.

The value of `i` is the same and shared between A_1 and A_2.

A common variable can keep a running total of the number of times a method has been called.

Static Method

The method is shared between all instances of the class.

Cannot access instance variables/methods of the class
only methods and variables that share the static modifier.

Static Class

The modifier cannot be applied to a class.

There are a number of useful Java classes which offer their functionality only in `static` methods, such as the `Math` class.

Using `Math` you can perform such operations as square root.

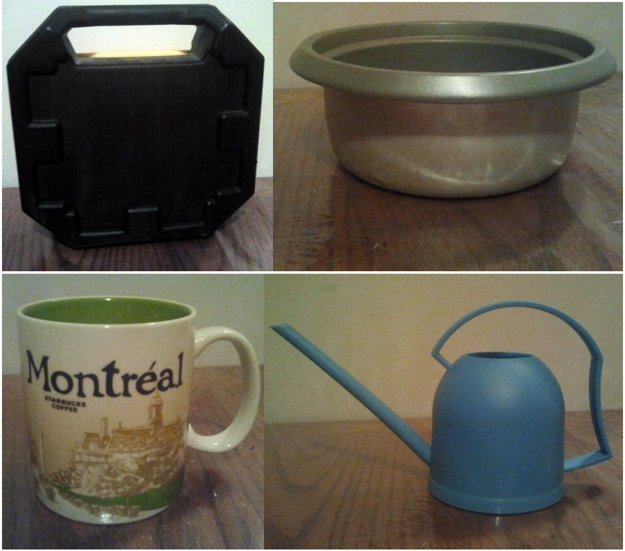
It doesn't make logical sense to instantiate `new Math()` to perform such an operation.

Interfaces

One of these things is not like the others:

Interfaces

All but one of these things implement the `HasHandle` interface.



Using Interfaces

```
interface HasHandle {
    void pickup();
}

class Donut implements HasHandle {
    void pickup() { ... }
    ...
}
```

Interfaces as Contracts

An `interface` is a way of specifying in Java what is effectively a contract.

The interface specifies some number of methods which any class that wants to implement this interface must have.

Interfaces as Contracts

```
public interface Drawable {
  void draw();
}
```

Any class which implements `Drawable` must contain an implementation of the method `draw()`.

Interface Semantics

Methods that are declared in an interface are always public.
Other modifiers are not allowed.

Interfaces can extend other interfaces, but may never contain implementation.

An interface may declare some constants.

Accordingly, an interface can never be instantiated.

A class can implement as many interfaces as desired.

Why Interfaces?

Interact with an object without knowing what it is underneath.

Example: `Comparable` allows the JRE to sort objects.

For a built-in sort, the object must implement `Comparable`.
(otherwise Java has no idea how to order them).

`Comparable`: only one method (`CompareTo`) is required.

Why Interfaces?

If the objects we are looking at represent soccer players.

Sort them by their uniform numbers ascending.

Implement that logic in `CompareTo`.

Then we can use something like `Collections.sort()`.

The JRE sorts without having to know anything in advance about the soccer players.

Abstract Classes

Halfway between an interface and a regular class is the abstract class.

Declared using the `abstract` keyword.

May (but does not have to) contain abstract methods.

However, any class that contains methods marked `abstract` must also be marked as `abstract`.

Cannot be instantiated, but may be subclassed.

An abstract class can have as much or as little implementation as desired.

Abstract Classes

To declare an abstract method, write the method signature, but with the keyword `abstract` attached:

```
public abstract void draw(int parameter);.
```

Abstract classes may have static fields and methods.
  e.g. `AbstractClass.method()`.

Abstract Classes vs. Interfaces

There are similarities between abstract classes & interfaces.

When should each of these be used?

Use Abstract Class When

- To share code among several closely related classes.
- Classes that extend your abstract class have many common methods or fields, or require access modifiers other than public (such as protected and private).
- To declare non-static or non-final fields.

Use Interface When

- Unrelated classes would implement your interface.
- You want to specify the behaviour of a particular data type, but not concerned about who implements its behaviour.
- You want to take advantage of multiple inheritance of type (i.e. implement multiple interfaces on one class)

Why Not Both?

An interface can be partly implemented by an abstract class.

The concrete class that extends the abstract class is responsible for implementing any of the methods of the interface that its superclass does not.

Why Not Both?

Imagine interface K has 4 methods.

There is an abstract class L declared as `implements K`.

Suppose L implements 3 of the 4 methods of K.

When a subclass P of L is declared, then P must implement that method that L did not (or else be abstract itself).

Visibility Modifiers

There are four options for visibility (accessibility) of something (a variable, method, class, etc) declared outside of a function.

- `public`
- `private`
- `protected`
- *(No Modifier)* Don't use this.

Visibility Modifiers: Why Bother?

Could one in theory just declare everything `public`?

Probably, but it is very poor programming practice.

Java convention: encapsulation and information hiding: the internal state of objects should not be visible to the world.

Visibility Modifiers: Why Bother?

In the real world anything that is `public` will be accessed by other programmers.

They may come to depend on a particular implementation detail (a problem that Microsoft has in abundance).

Consider carefully if a method should be accessible from outside the given class.

If the answer is no, then `private` is the right answer for the modifier, or `protected` if it may be used in a subclass.

`final`

Another keyword: `final`. This can be applied to three things, with slightly different meanings:

- Field
- Method
- Class

We cannot apply `final` to an interface, because we will obviously have to implement it somehow.

Similarly, we cannot make an abstract class final, because it is meant to be subclassed.

Worked Example: Shapes

Done on the board to facilitate understanding.

(See notes for full details)

Modifier Summary

Modifier	Interface	Class	Nested Class	Field	Method
`public`	Accessible from any class.				
`private`	Accessible only from this class.				
`protected`	Accessible only from this class and its subclasses.				
(No modifier)	Accessible from any class within the same package. Don't use this.				
`abstract`	N/A	Contains at least one `abstract` method; cannot be instantiated.		N/A	Its implementation is not defined; only signature & return type declared.
`final`	N/A	Cannot be subclassed.		Its value cannot be changed.	It cannot be overridden by a subclass.
`static`	N/A	N/A	Not an inner class.	Exactly one instance exists for all objects of the class.	Exactly one instance exists for all objects of the class.