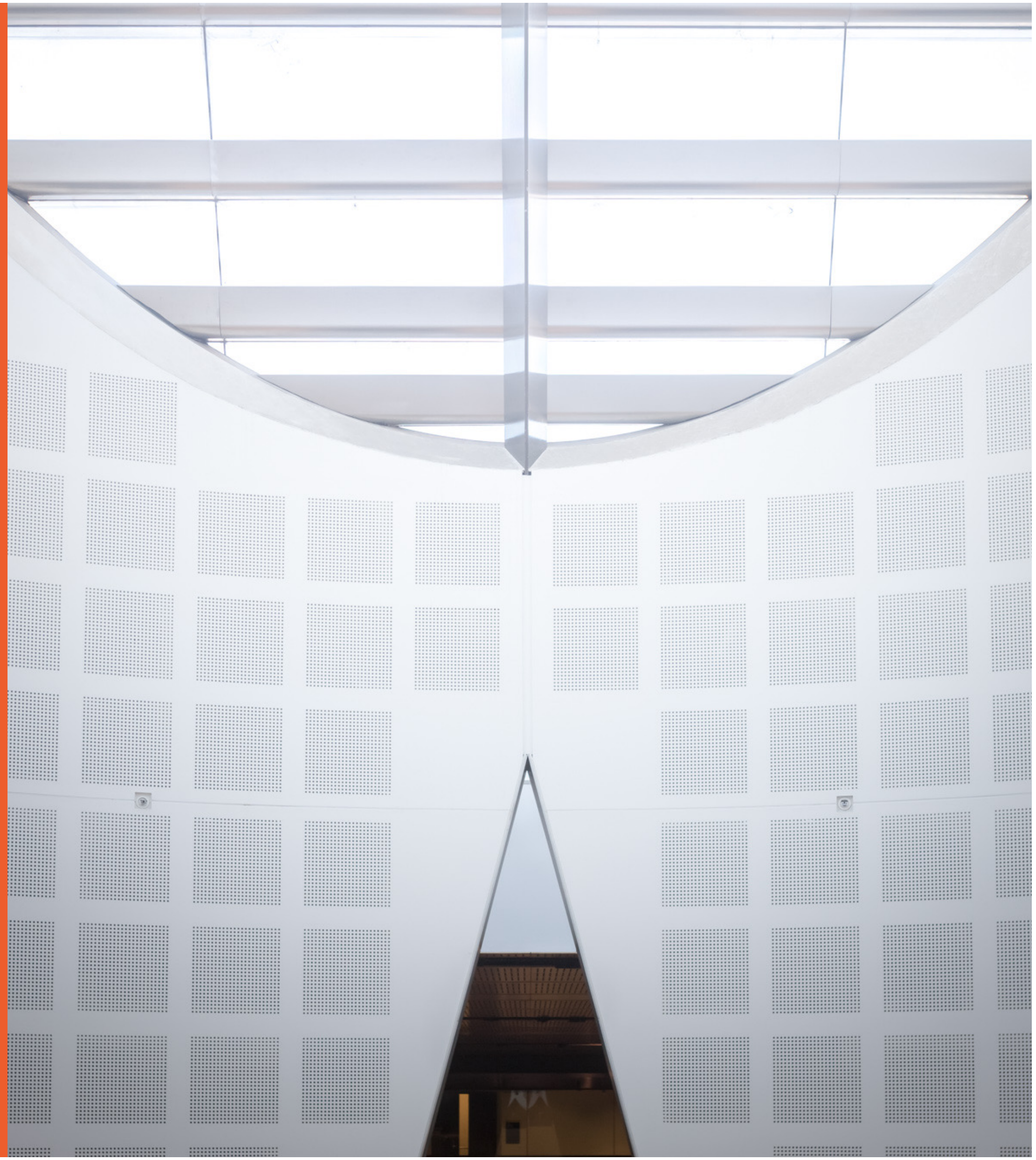
Software Design and Construction 1
SOFT2201 / COMP9201

OO Theory in Java

Dr. Xi Wu

School of Computer Science

THE UNIVERSITY OF SYDNEY



Copyright warning

Overview

- Types
- Encapsulation
- Inheritance
- Variable Binding & Polymorphism
- Virtual Dispatch
- Abstract Classes
- Interfaces

Types in Java

- Overview
  - Java is a strongly typed language
  - Primitive types: built-in types of the virtual machines
    - Cannot be changed by programs
    - Have same meaning across computer platforms
  - Compositional types
    - Interfaces
    - Classes

Primitive Types in Java

- Primitive types represent basic data-types
- In-built and cannot be changed in programs
- Integral types
  - `byte, short, int, long`
- Floating point number types
  - `float, double`
- Character type
  - `char`
- Boolean
  - `boolean`

Integral and Floating Types

Type	Internal	Smallest Value	Largest Value
`byte`	8 bits	-128	+127
`short`	16 bits	-32,768	+32,767
`int`	32 bits	-2,147,483,648	+2,147,483,647
`long`	64 bits	-2.3E+18	+2.3E+18
`float`	32 bits	-/+1.4e-45	-/+3.4e+38
`double`	64 bits	-/+4.9e-324	-/+1.7e+308d

Conversion between Primitive Types

- Implicit widening of integral & floating point number types
  - byte to short, int, long, float, or double
  - short to int, long, float, or double
  - char to int, long, float, or double
  - int to long, float, or double
  - long to float or double
  - float to double

- Everything else requires type casts or is not possible

Casting/Widening Matrix

From/To	byte	char	short	int	long	float	double	boolean
byte								**n/a**
char	`(byte)`		`(short)`					**n/a**
short	`(byte)`	`(char)`						**n/a**
int	`(byte)`	`(char)`	`(short)`					**n/a**
long	`(byte)`	`(char)`	`(short)`	`(int)`				**n/a**
float	`(byte)`	`(char)`	`(short)`	`(int)`	`(long)`			**n/a**
double	`(byte)`	`(char)`	`(short)`	`(int)`	`(long)`	`(float)`		**n/a**
boolean	**n/a**	**n/a**	**n/a**	**n/a**	**n/a**	**n/a**	**n/a**	

Example for Implicit Widening and Casting

```
…
byte x = 10;

/* implicit widening */
short y =  x;
long z = x;
float f = x;

/* casting */
byte p = (byte) z;
short q = (short) f;
```

Silent Failure of Types

- Example:

```
…
int i = Integer.MAX_VALUE;
System.out.println(i);
i = i + 1;
System.out.println(i);
…
```

- Integer overflow occurs for variable `i`
- Type system cannot prevent this at compile-time
  - weak notion of correctness

Primitive Types

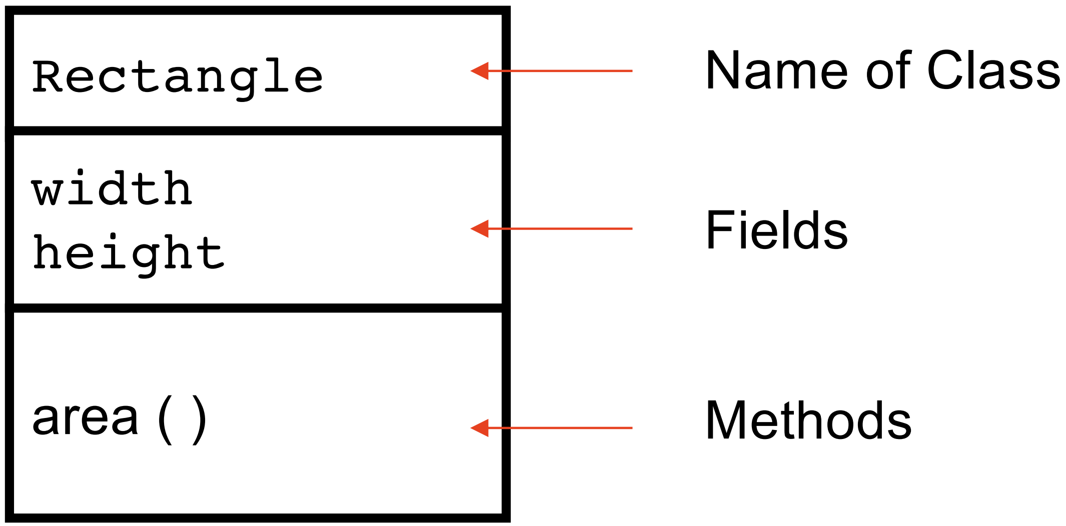
- Basic data-types
  - Behave the same on all platforms running Java's virtual machine
- Primitive Types are not classes nor interfaces
- Primitive Types are building blocks for more complex types
- Beside the storing the actual value there are very little additional overheads
  - Note that classes have extra information to keep of dynamic runtime info
- Operations on primitive data-types are very efficient

Classes in Java

- Classes are types
  - Composite type consisting of fields(=state) and methods(=behavior)
- Class Abstraction
  - Separate class implementation from the use of a class
  - User of a class does not need to know the implementation / just the public methods
  - The implementor provides the implementation of the class
  - The details are encapsulated and hidden from the user
- Class creates objects (instantiate)
- Object communicates with each other via methods

Classes

– A basic class has a name, fields, and methods:



Classes (cont'd)

– The basic syntax for a class is

```
class <class name> [extends <super class> ]
{
      <field declarations>
      <method declarations>
}
```

– Example:

```
class Rectangle
{
      <field declarations>
      <method declarations>
}
```

Adding Fields: Class Rectangle with Fields

- Add *fields*

```
public class Rectangle {
  public double width, height;
}
```

- The fields (data) are also called the *instance* variables
- Fields can be
  - Primitive types (int, bool, etc.)
  - References to other class instances

Adding Methods

- To change / retrieve state of a class, methods are necessary
- Methods are declared inside the body
  - Methods are messages in an object-oriented programming sense
  - A return type must be specified
  - A list of arguments with their types must be specified
- The general form of a method declaration is:

```
type Method (argument-list) {
    Method-body;
}
```

Adding Methods to Rectangle

```
public class Rectangle {
      public double width, height;  // fields

      public void setWidth(double w) {
        width = w;
      }
      public void setHeight(double h) {
        height = h;
      }
      public double area() {
       return width * height;
      }
}
```

Method Body

Mutators

- Methods that change the state of a class object are called mutators
- Example:

```
public void setWidth(double w) {
  width = w;
}
public void setHeight(double h) {
  height = h;
}
```

- Gives clients of the class write access to the state

Accessor

- Methods that read the state of a class object are called accessor
- Example:

```
public double area() {
 return width * height;
}
```

- Gives clients of the class read access to the state

Constructors

- Special method in class whose task is to initialize fields of the class
- The name is the same name as class
- Constructors are invoked whenever an object of that class is created
- It is called constructor because it constructs values of data members.
- A constructor cannot return a value
- There can be several constructors for a class
  - Default constructor with no parameters
  - Parameterized constructors

Example: Default Constructor

- Default constructor

```
public class Rectangle {
  public double width, height;
  public Rectangle() { // default constructor
     width = 1.0;
     height = 1.0;
  }
}
```

- Java has a synthesized default constructor when not specified

Example: Parameterized Constructor

– Parameterized constructor:

```
public class Rectangle {
  public double width, height;

  // parameterized constructor
  public Rectangle(double w, double h) {
     width = w;
     height = h;
  }
}
```

Example: Multiple Constructors

- Default and parameterized constructor (overloading)

```
public class Rectangle {
  public double width, height;
  // default constructor
  public Rectangle() {
     width = 1.0;
     height = 1.0;
  }
  // parameterized constructor
  public Rectangle(double w, double h) {
     width = w;
     height = h;
  }
}
```

Creating Instances of a Class

- Objects are created dynamically using the *new* keyword.
- The new operator creates a new object on the heap
  - The object will be selected as a candidate if not further used
  - The object becomes a candidate for automatic garbage collection
  - Garbage collection kicks in periodically and releases candidates
  - developers don't need to delete their own objects
- Example:

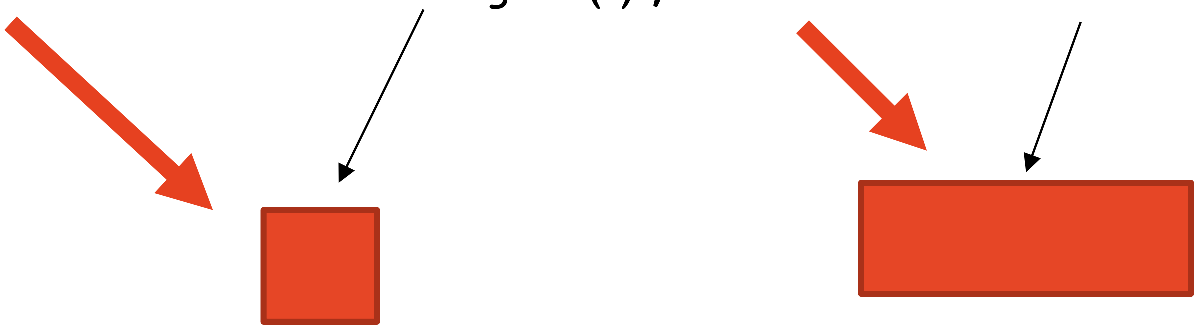
```
<Class> X = new <Class> (<args>);
```

  - Define a reference variable X of type class that points to a new instance

Example: Creating Instances of a Class

- rec1 and rec2 refer to Rectangle objects

```
rec1 = new Rectangle();   rec2 = new Rectangle(10,1);
```



- First instance calls default constructor
- Second instance calls parameterized constructor

Copy Constructor

- Java has no copy constructor
- We can simulate copy constructor by having a constructor with objects of same type as argument
- All classes are derived from class Object
- Object has the method clone()

Example: Copy Constructor

– Copy constructor

```
public class Rectangle {
  public double width, height;
  // copy constructor
  public Rectangle(Rectangle o) {
     width = o.width;
     height = o.height;
  }
}
```

Accessing Fields and Methods

- Access is done via the . operator
- Access to field and methods of an instance in the same style

```
<object> . <field>
<object> . method(<args>)
```

- Access modifier is important
  - public/protected/private
  - Controls whether access is permitted at client or member level

Example: Accessing fields / methods

- Using object methods:

```
Rectangle rec = new Rectangle();

rec.width = 1.0;
rec.height = 100.0;
double area = rec.area();
```

send ‘message’ area to an instance of type Rectangle which rec is referring to

Access Modifiers

- Access modifiers provide encapsulation for object-oriented programming
- Protect state from client
- Access modifiers work on different levels:
  - Class level
  - Member level
  - Client level

Access Modifier: public

- A class that is labeled as public is accessible to all other classes inside and outside a package

```
package com.foo.goo;

public class test1 {
   public void test();
}
```

```
package com.foo.hoo;
import com.foo.goo;

public class test2 {
   public void test(){
        test1 x = new test1();
   }
}
```

Access Modifier: public

- A class marked as public is accessible to all classes with the import statement
- There is no need for import statement if a public class is used in the same package

Access Modifier: default

- Default access is a class definition with no specified access qualifier
- Classes marked with default access are visible only in package but not outside of package
- Note that protected and private classes are not available for packages

Member access modifiers for methods and fields

- Public
  - Accessible for client, class, and sub-classes
- Protected
  - Accessible for class, and sub-classes
- Private
  - Accessible for class itself only

Example: Member access modifiers

- Example: public

```
public class Rectangle {
  public double width, height;
}

// client can access state
Rectangle x = new Rectangle();
x.width = 100;
x.height = 100;
```

Example: Member access modifiers

– Example: protected

```
public class Rectangle {
  protected double width, height;
}

// client cannot access state
Rectangle x = new Rectangle();
x.width = 100;  // access error!
x.height = 100; // access error!
```

Example: Member access modifiers

- Example: private

```
public class Rectangle {
  private double width, height;
  Rectangle(double w, double h) {
     width = w; // access is ok
     height = h;  // access is ok
  }
}
public class Square extends Rectangle {
  Square(double w) {
     super.Rectangle(w,w);
  }
  void setWidth(double w) {
    super.width = w;  // access error!
    super.height = w;  // access error!
  }
}
```

Encapsulation

- Wrap data/state and methods into a class as a single unit
- Multiple instances can be generated
- Protect state via setter (mutator)/getter (accessor) methods

```
class Rectangle {
  double length; double width;
  double getLength() { return length; }
  double getWidth() { return width; }
  double area() { return length * width; }
}
```

Sub-Classes

- Classes permit inheritance
  - Methods and fields from a super-class can be inherited
- Ideal for software-engineering
  - Reuse of code
- Java uses “extends” keyword for extending from existing class
- Single-inheritance paradigm
  - Class hierarchy can be a tree most
- If no super-class is specified, default class Object becomes super-class

Inheritance

- Sub-class inherits from super-class: methods, variables, …
- Reuse structure and behavior from super-class
- Single inheritance for classes

```
class Rectangle extends Object {
  double length; double width;
  double area() { return length * width; }
}

class ColouredRectangle extends Rectangle {
  int colour;
  int colour() { return colour; }
}
```

Variable Binding, Polymorphism

- Object (on heap) has single type when created
  - Type cannot change throughout its lifetime
- Reference variables point to null or an object
- Type of object and type of reference variable may differ
  - e.g., Shape x = new Rectangle(4,2);
- Understand the difference
  - Runtime type vs compile-time type
- Polymorphism:
  - Reference variable may reference differently typed object
  - Must be a sub-type of

Virtual Dispatch

- Methods in Java permit a late binding
- Reference variable and its type does not tell which method is really invoked
- The type of reference variable and class instance may differ
- Class variables may override methods of super classes
- The method invoked is determined by the type of the class instance
- Binding is of great importance to understand OO

Example: Virtual Dispatch

- Example:

```
public class Shape{ // extends Object
  double area() { }
}

public class Rectangle extends Shape {
  double area() { }
}
…
Shape X = new Shape();
Shape Y = new Rectangle();

double a1 = X.area() // invokes area of Shape
double a2 = Y.area() // invokes area of Rectangle
```

Abstract Classes

- Method implementations are deferred to sub-classes
- Requires own key-word abstract for class/method
- No instance of an abstract class can be generated

```
abstract class Shape extends Object {
  public abstract double area();
}

public class Rectangle extends Shape {
  double width; double length;
  double area() { return width * length; }
}
```

Interfaces

- Java has no multi-inheritance
  - Interface is a way-out (introduction of multi-inheritance via the back-door)
- Interfaces is a class contract that ensures that a class implements a set of methods.
- Interfaces can inherit from other interfaces
- Ensures that a class has a certain set of behavior
- Interfaces are specified so that they form a directed acyclic graph
- Methods declared in an interface are always public and abstract
- Variables are permitted if they are static and final only

Example: Interface

```
// definition of interface
public interface A {
  int foo(int x);
}

// class X implements interface A
class X implements A {
  int foo(int x) {
    return x;
  }
}
```

Example: Interface

- Inheritance in interfaces

```
// definition of interface
public interface A {
  int foo(int x);
}

public interface B extends A{
  int hoo(int x);
}
```

- Interface B has methods foo() and hoo()

What are we going to learn next week?

- UML Modeling and Case Studies
  - Use Case Diagrams
  - Class Diagrams
  - Interaction Diagrams