Midterm Review

Adapted by Fangzhen Lin for COMP3021 from Y. Danial Liang's PowerPoints for Introduction to Java Programming, Comprehensive Version, 9/E, Pearson, 2013.

Our journey to Java expert

- Object and classes
- Java treatment of String
- Exceptions
- Inheritance
- GUI
- Abstract class and Interface
- Event driven programming

Class and Object

Constructors

```
Circle() {
}
```

Constructors are a special kind of methods that are invoked to construct objects.

```
Circle(double newRadius) {
  radius = newRadius;
}
```

Constructors, cont.

A constructor with no parameters is referred to as a *no-arg constructor*.

· Constructors must have the same name as the class itself.

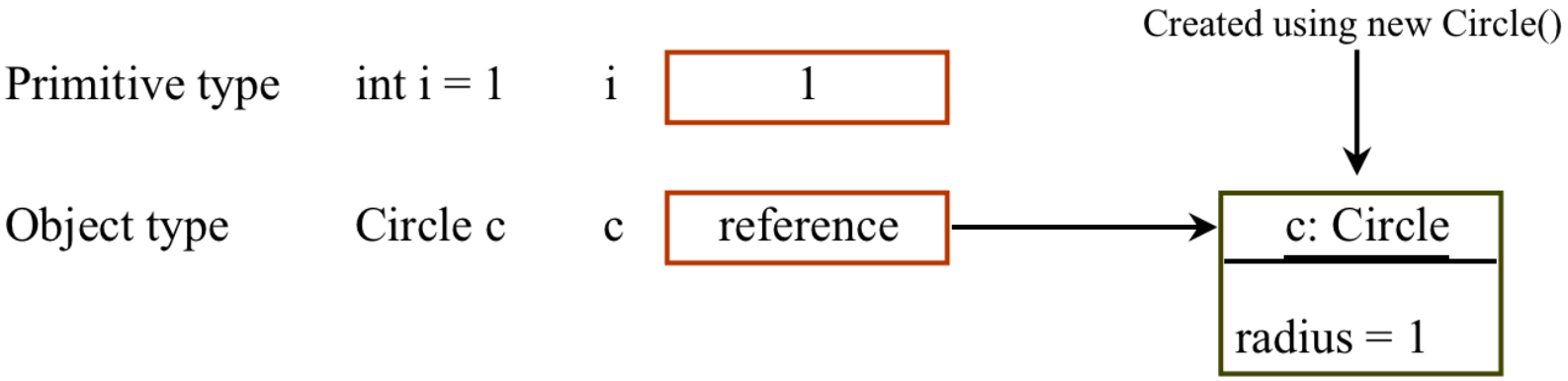
· Constructors do not have a return type—not even void.

· Constructors are invoked using the new operator when an object is created. Constructors play the role of initializing objects.

Default Constructor

A class may be declared without constructors. In this case, a no-arg constructor with an empty body is implicitly declared in the class. This constructor, called *a default constructor*, is provided automatically *only if no constructors are explicitly declared in the class*.

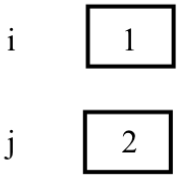
Differences between Variables of Primitive Data Types and Object Types



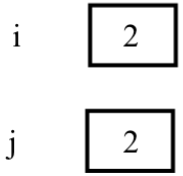
Copying Variables of Primitive Data Types and Object Types

Primitive type assignment i = j

Before:



After:

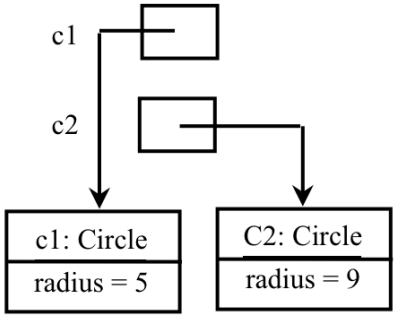


Duplicating content

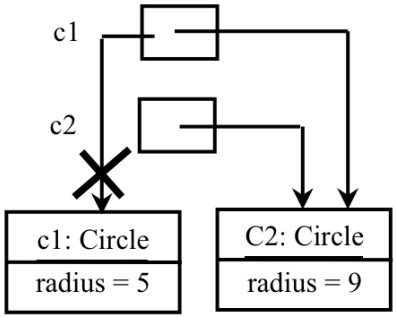
Creating an alias

Object type assignment c1 = c2

Before:



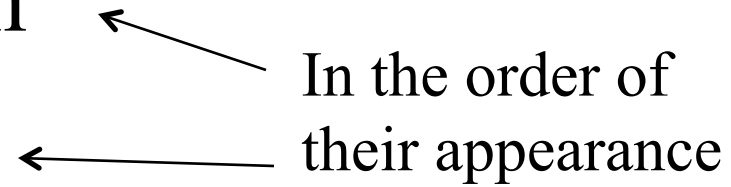
After:



Initialization of Instance Fields

- Several ways:

1. Explicit initialization
2. Initializer
   (or initialization) block
3. Constructors



Initialization of Instance Fields

- Explicit initialization: initialization at declaration.

```
private double salary = 0.0;
private String name = “”;
```

Initialization of Instance Fields

- Initializer (or initialization) block:
  - Class declaration can contain arbitrary blocks of codes.

```
class Employee
{ …
   private int id;
   private static int nextId=1;

   // object initializer block
   {
      id = nextId;
      nextId++;
   }
   …
}
```

Initialization of Instance Fields

- Initialization by constructors

```
class Employee
{
        …
        private int id;
        private static int nextId=1;

        Employee(){
           id = nextId;
           nextId++;
        }
}
```

Initialization of Instance Fields

- What happens when a constructor is called
  - All data fields initialized to their default value (`0`, `false`, `null`)
  - Field initializers and instance initializer blocks are executed according to their order of appearance
  - Body of the constructor is executed after the body of its superclass's constructor
    - Note that a constructor might call another constructor at line 1.

Initialization of Static Fields

- When a class is loaded into memory
  - All static data fields initialized to their default value (`0`, `false`, `null`)
  - Static field initializers and static initializer blocks are executed in the order of their appearance
  - Note that static fields are initialized once when its parent class is loaded.

Object Initialization Summary

- Static then instance
- Parent then child
- For each class
  - Field
  - Initialization block
  - Constructor

```
class Employee
{ …
    private int id;
    private static int nextId=1;

  // object initializer block
  {
     id = nextId;
       nextId++;
  }

  static {
    //something else
  }

  public static void main(String s[]){
     System.out.println(“”);
  }
}
```

Java Strings

Constructing Strings

String newString = new String(stringLiteral);

String message = new String("Welcome to Java");

Since strings are used frequently, Java provides a shorthand initializer for creating a string:

String message = "Welcome to Java";

Strings Are Immutable

A String object is immutable; its contents cannot be changed. Does the following code change the contents (“Java”) of the string object?

```
String s = "Java";
s = "HTML";
```

Interned Strings

Since strings are immutable and are frequently used, to improve efficiency and save memory, the JVM uses a unique instance for string literals with the same character sequence. Such an instance is called *interned*. For example, the following statements:

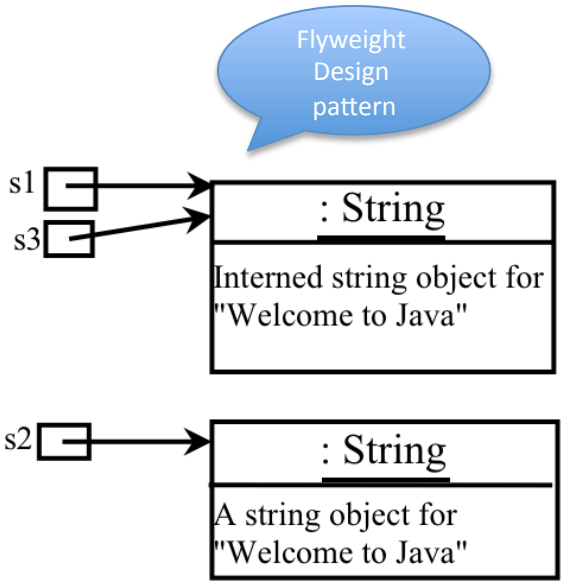
Examples

```
String s1 = "Welcome to Java";

String s2 = new String("Welcome to Java");

String s3 = "Welcome to Java";

System.out.println("s1 == s2 is " + (s1 == s2));
System.out.println("s1 == s3 is " + (s1 == s3));
```



display

s1 == s2 is false

s1 == s3 is true

A new object is created if you use the new operator.

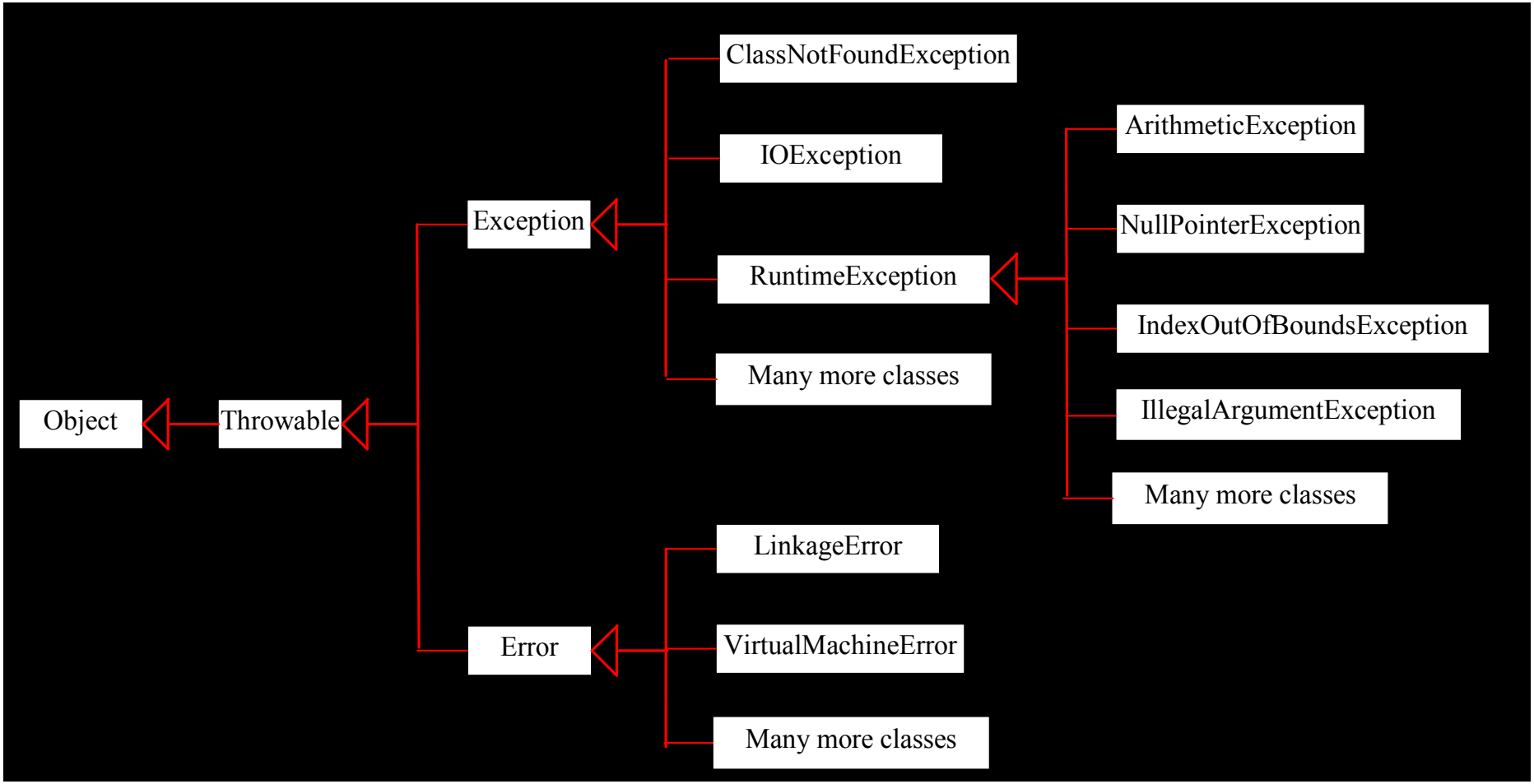
If you use the string initializer, no new object is created if the interned object is already created.

Exception Handling

Review points

- Basic syntax of declaring/checking a generic type.
- Difference between checked/unchecked exceptions
- The exception handling mechanism
  - "Stack unwinding" sequence
  - Polymorphic matching

Exception Types



Checked Exceptions vs. Unchecked Exceptions

RuntimeException, Error and their subclasses are known as *unchecked exceptions*. All other exceptions are known as *checked exceptions*, meaning that the compiler forces the programmer to check and deal with the exceptions.

Catching Exceptions

```
try {
  statements;  // Statements that may throw exceptions
}
catch (Exception1 exVar1) {
  handler for exception1;
}
catch (Exception2 exVar2) {
  handler for exception2;
}
...
catch (ExceptionN exVar3) {
  handler for exceptionN;
}
finally {
  finalStatements;
}
```

Summary of Exception Handling

- When an exception is thrown
  1. Check local "catch" first where "throws" is executed
  2. Match exact "catch" first
  3. If no match, match "catch" of super types
     1. Match only once! Only one "catch" block is executed for multiple catches.
  4. If no match,
     - Execute the "finally block". Exception thrown in "finally block" will be the only exception passed to caller
     - Go to caller, repeat 2-4, until "main".
- When matched
  - Resume normal flow of execution if no throwing in "catch" blocks
  - Or return to caller at throwing

Declaring Exceptions

Every method must state the types of checked exceptions it might throw. This is known as *declaring exceptions*.

```
public void myMethod()
   throws IOException

public void myMethod()
   throws IOException, OtherException
```

Throwing Exceptions

When the program detects an error, the program can create an instance of an appropriate exception type and throw it. This is known as *throwing an exception*. Here is an example,

```
throw new TheException();
```

```
TheException ex = new TheException();
throw ex;
```

Rethrowing Exceptions

```
try {
   statements;
}
catch(TheException ex) {
   perform operations before exits;
   throw ex;   // throw new TheException();
}
```

Be Careful with Multiple Exceptions

- Note that the following will produce a compiling error. Why?

```
try {…}
catch (Exception e3) {…}
catch (ArithmeticException e1){…}
catch (IOException e2) {…}
```

MultipleException

Catch or Declare Checked Exceptions

Java forces you to deal with checked exceptions. If a method declares a checked exception (i.e., an exception other than Error or RuntimeException), you must invoke it in a try-catch block or declare to throw the exception in the calling method. For example, suppose that method p1 invokes method p2 and p2 may throw a checked exception (e.g., IOException), you have to write the code as shown in (a) or (b).

```
void p1() {
  try {
    p2();
  }
  catch (IOException ex) {
    ...
  }
}
```

(a)

```
void p1() throws IOException {

  p2();

}
```

(b)

Defining Custom Exception Classes

- Use the exception classes in the API whenever possible.
- Define custom exception classes if the predefined classes are not sufficient.
- Define custom exception classes by extending Exception or a subclass of Exception.

Inheritance

Are superclass's Constructor Inherited?

No. They are not inherited.

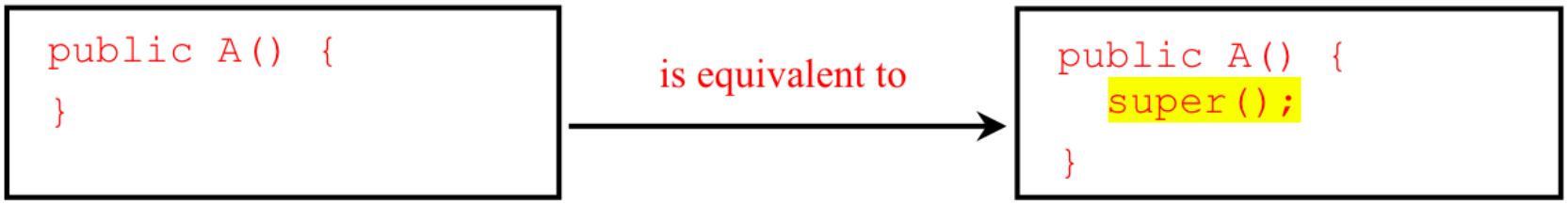
They are invoked explicitly or implicitly.

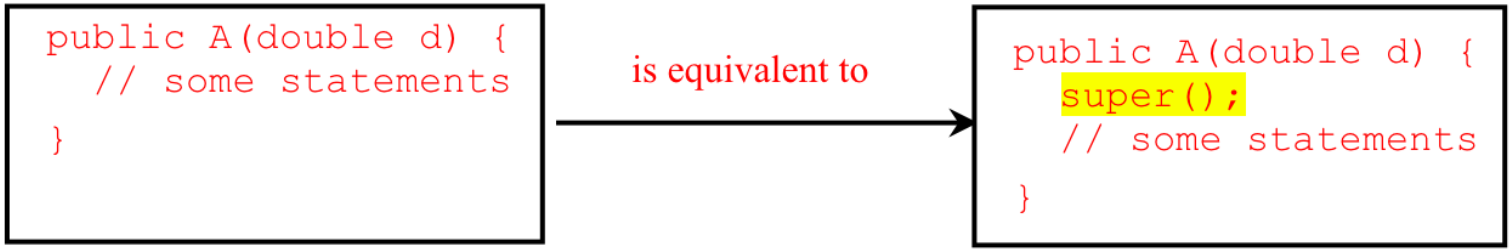
Explicitly using the super keyword.

A constructor is used to construct an instance of a class. Unlike properties and methods, a superclass's constructors are not inherited in the subclass. They can only be invoked from the subclasses' constructors, using the keyword super. *If the keyword super is not explicitly used, the superclass's no-arg constructor is automatically invoked.*

Superclass's Constructor Is Always Invoked

A constructor may invoke an overloaded constructor or its superclass's constructor. If none of them is invoked explicitly, the compiler puts super() as the first statement in the constructor. For example,





Using the Keyword `super`

The keyword `super` refers to the superclass of the class in which `super` appears. This keyword can be used in two ways:

- To call a superclass constructor
- To call a superclass method

CAUTION

You must use the keyword super to call the superclass constructor. Invoking a superclass constructor's name in a subclass causes a syntax error. Java requires that the statement that uses the keyword super appear first in the constructor.

Overriding Methods in the Superclass

A subclass inherits methods from a superclass. Sometimes it is necessary for the subclass to modify the implementation of a method defined in the superclass. This is referred to as *method overriding*.

```
public class Circle extends GeometricObject {
  // Other methods are omitted


  /** Override the toString method defined in GeometricObject */
  public String toString() {
    return super.toString() + "\nradius is " + radius;
  }
}
```

Polymorphism, Dynamic Binding and Generic Programming

```
public class PolymorphismDemo {
  public static void main(String[] args) {
    m(new GraduateStudent());
    m(new Student());
    m(new Person());
    m(new Object());
  }

  public static void m(Object x) {
    System.out.println(x.toString());
  }
}

class GraduateStudent extends Student {
}

class Student extends Person {
  public String toString() {
    return "Student";
  }
}

class Person extends Object {
  public String toString() {
    return "Person";
  }
}
```

Method m takes a parameter of the Object type. You can invoke it with any object.

An object of a subtype can be used wherever its supertype value is required. This feature is known as *polymorphism*.

When the method m(Object x) is executed, the argument x's toString method is invoked. x may be an instance of GraduateStudent, Student, Person, or Object. Classes GraduateStudent, Student, Person, and Object have their own implementation of the toString method. Which implementation is used will be determined dynamically by the Java Virtual Machine at runtime. This capability is known as *dynamic binding*.

Accessibility Summary

Modifier on members in a class	Accessed from the same class	Accessed from the same package	Accessed from a subclass	Accessed from a different package
public	✓	✓	✓	✓
protected	✓	✓	✓	–
default	✓	✓	–	–
private	✓	–	–	–

A Subclass Cannot Weaken the Accessibility

A subclass may override a protected method in its superclass and change its visibility to public. However, a subclass cannot weaken the accessibility of a method defined in the superclass. For example, if a method is defined as public in the superclass, it must be defined as public in the subclass.

Abstract class and Interface

What's an abstract class?

- It is a special kind of class solely used for sharing/reusing data/behavior for subclasses
- Difference from the classes you have seen so far
  - Cannot be "newed".
  - Can own "abstract methods".

- Analogy:
  - Special molds used to make other molds not to make cakes

What's an abstract method?

- We need "abstract classes" for reusing data/ behavior for subclasses.
- Often we only say what a method looks like and don't provide any definition
  - No need → subclass will override
  - Impossible → depends on what subclass does
- Such methods are "abstract" methods with the `abstract` modifier.

What's an interface?

- The single most important concept in the Java language
- Think of interface as a special type of abstract class
  - Add additional roles/personality to a class
  - To satisfy the need of multiple inheritance but no black diamond.
  - Key for building massive scale Java libraries

Define an Interface

To distinguish an interface from a class, Java uses the following syntax to declare an interface:

```
public interface InterfaceName {
  constant declarations;
  method signatures;
}
```

Example:

```
public interface Edible {
  /** Describe how to eat */
  public abstract String howToEat();
}
```

Interfaces vs. Abstract Classes

In an interface, the data must be constants; an abstract class can have all types of data.

Each method in an interface has only a signature without implementation; an abstract class can have concrete methods.

	Variables	Constructors	Methods
Abstract class	No restrictions	Constructors are invoked by subclasses through constructor chaining. An abstract class cannot be instantiated using the new operator.	No restrictions.
Interface	All variables must be public static final	No constructors. An interface cannot be instantiated using the new operator.	All methods must be public abstract instance methods

GUI Basics

- You will not need to remember APIs.
- How to compose Swing widgets is not covered in the midterm.

Event-driven programming

Review points

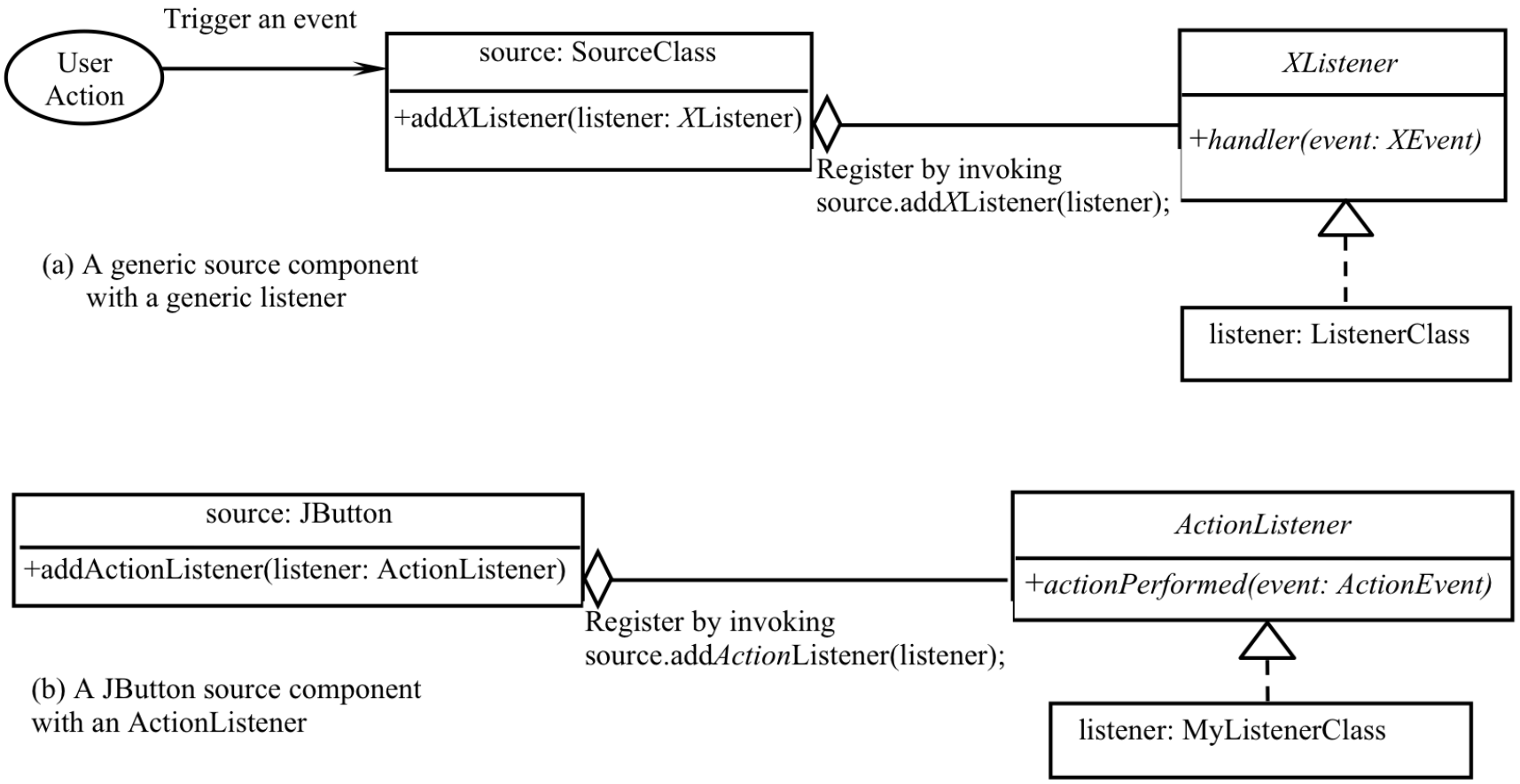
- Know how to handle events for Swing widgets, if APIs are given.
- Understand the basic syntax and rules of inner classes
- Understand the basic syntax of anonymous inner classes

Procedural vs. Event-Driven Programming

- *Procedural programming* is executed in procedural order.（Also called imperative)

- In event-driven programming, code is executed upon activation of events.

The Delegation Model



The Delegation Model: Example

```
JButton jbt = new JButton("OK");

ActionListener listener = new OKListener();

jbt.addActionListener(listener);
```

Inner Classes, cont.

```
public class Test {
  ...
}

public class A {
  ...
}
```

(a)

```
public class Test {
  ...

  // Inner class
  public class A {
    ...
  }
}
```

(b)

```
// OuterClass.java: inner class demo
public class OuterClass {
  private int data;

  /** A method in the outer class */
  public void m() {
    // Do something
  }

  // An inner class
  class InnerClass {
    /** A method in the inner class */
    public void mi() {
      // Directly reference data and method
      // defined in its outer class
      data++;
      m();
    }
  }
}
```

(c)

Inner Classes (cont.)

- Inner classes can make programs simple and concise.
- An inner class supports the work of its containing outer class and is compiled into a class named *OuterClassName* $*InnerClassName*.class. For example, the inner class InnerClass in OuterClass is compiled into *OuterClass* *$InnerClass*.class.

Anonymous Inner Classes

Inner class listeners can be shortened using anonymous inner classes. An *anonymous inner class* is an inner class without a name. It combines declaring an inner class and creating an instance of the class in one step. An anonymous inner class is declared as follows:

```
new SuperClassName/InterfaceName() {
  // Implement or override methods in superclass or interface
  // Other methods if necessary
}
```

Good luck with your midterm