v1.0

THE UNIVERSITY OF MELBOURNE

Programming and Software Development
COMP90041

Lecture 8

Polymorphism

Tilman Dingler
Sem 1, 2020, 12-05-2020

Slides developed by Rose Williams

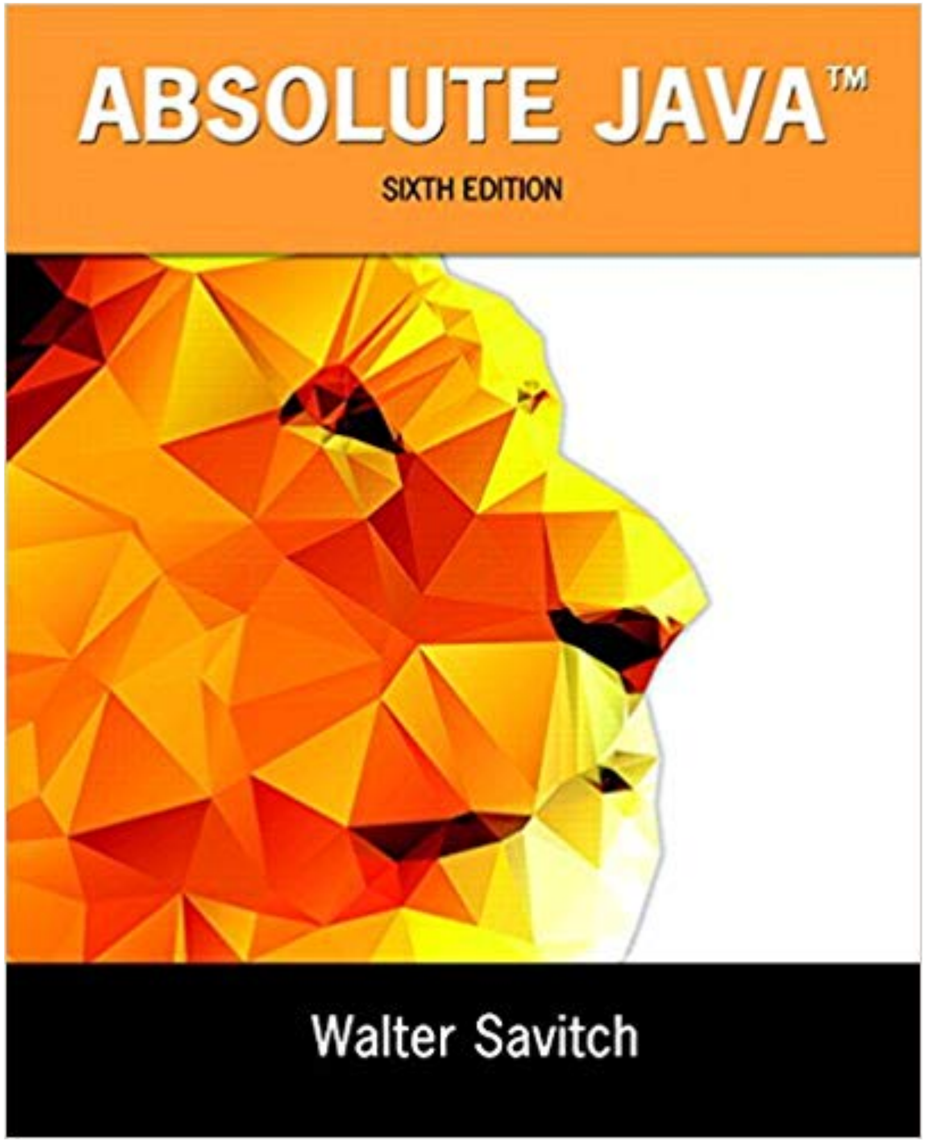
- Mini Lectures
  - Polymorphism
  - Abstract Classes
  - Handling Exceptions
- Live Lecture
  - Dungeons & Dragons

Take-Aways

1. Housekeeping
   - Assignment 3 is out. Due: 22 May
2. Project C
3. Dungeons & Dragons
4. Q&A

Live Session: Agenda

- Chapter 8



Textbook

Text on the book cover image: ABSOLUTE JAVA™, SIXTH EDITION, Walter Savitch

Polymorphism

Introduction to Polymorphism

- There are three main programming mechanisms that constitute object-oriented programming (OOP)
  - Encapsulation
  - Inheritance
  - Polymorphism
- Polymorphism is the ability to associate many meanings to one method name
  - It does this through a special mechanism known as *late binding* or *dynamic binding*

Introduction to Polymorphism

- Inheritance allows a base class to be defined, and other classes derived from it
  - Code for the base class can then be used for its own objects, as well as objects of any derived classes
- Polymorphism allows changes to be made to method definitions in the derived classes, *and have those changes apply to the software written for the base class*

Late Binding with `toString`

- If an appropriate `toString` method is defined for a class, then an object of that class can be output using `System.out.println`

```
Sale aSale = new Sale("tire gauge", 9.95);
System.out.println(aSale);
```

  - Output produced:

```
tire gauge Price and total cost = $9.95
```

- This works because of late binding

Late Binding with `toString`

- One definition of the method **`println`** takes a single argument of type **`Object`**:

```
public void println(Object theObject)
{
  System.out.println(theObject.toString());
}
```

  - In turn, It invokes the version of **`println`** that takes a **`String`** argument
- Note that the **`println`** method was defined before the **`Sale`** class existed
- Yet, because of late binding, the **`toString`** method from the **`Sale`** class is used, not the **`toString`** from the **`Object`** class

Example

- toString();

```
HourlyEmployee joe = new HourlyEmployee("Joe Worker",
                              new Date("January", 1, 2004), 50.50, 160);
Employee mike = new Employee("Mike Jordan", new Date("March", 1,
1984));

System.out.println();
System.out.println("joe's record is as follows:");
System.out.println(joe);

Sstem.out.println();
System.out.println("mike's record is as follows:");
System.out.println(mike);
```

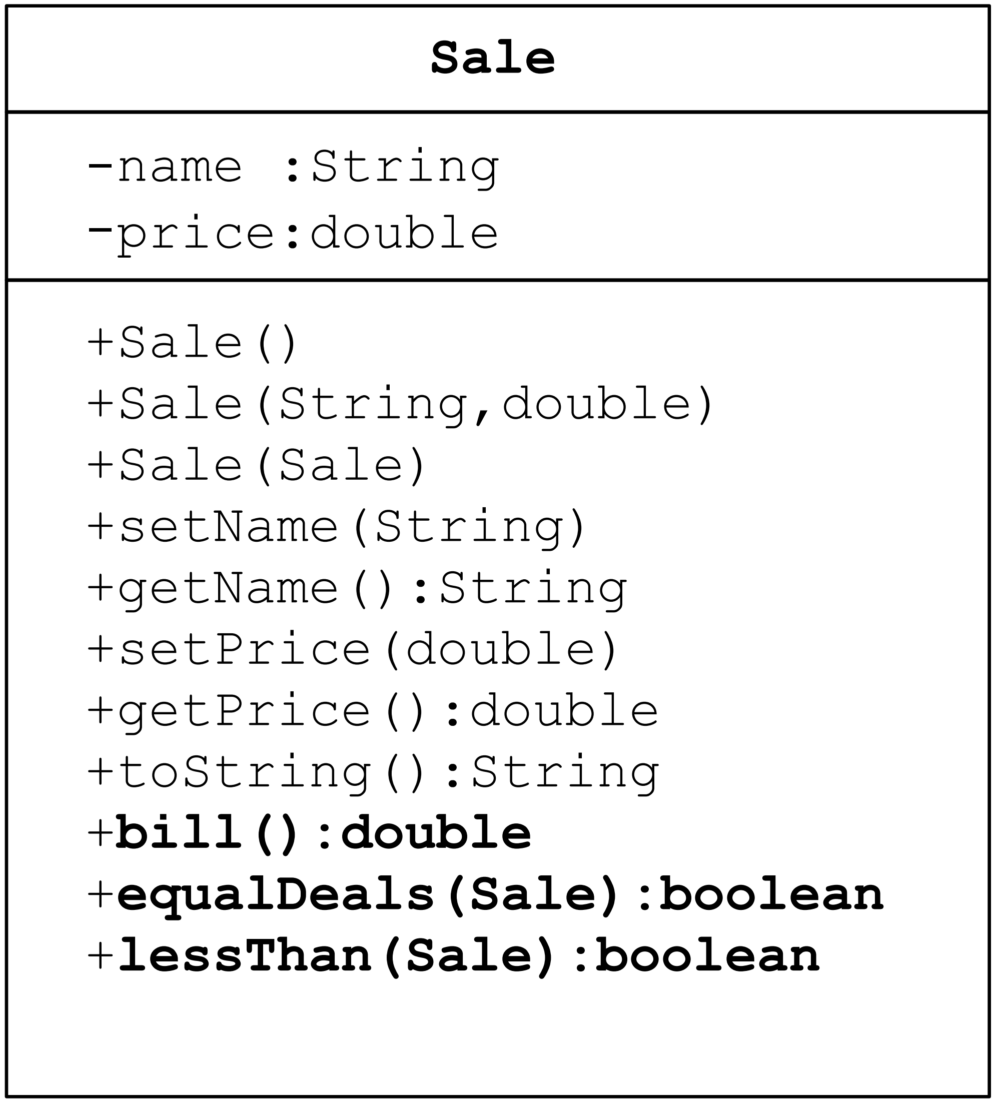
Late Binding

- The process of associating a method definition with a method invocation is called *binding*
- If the method definition is associated with its invocation when the code is compiled, that is called *early binding* or *static binding*
- If the method definition is associated with its invocation when the method is invoked (at run time), that is called *late binding* or *dynamic binding*

Late Binding

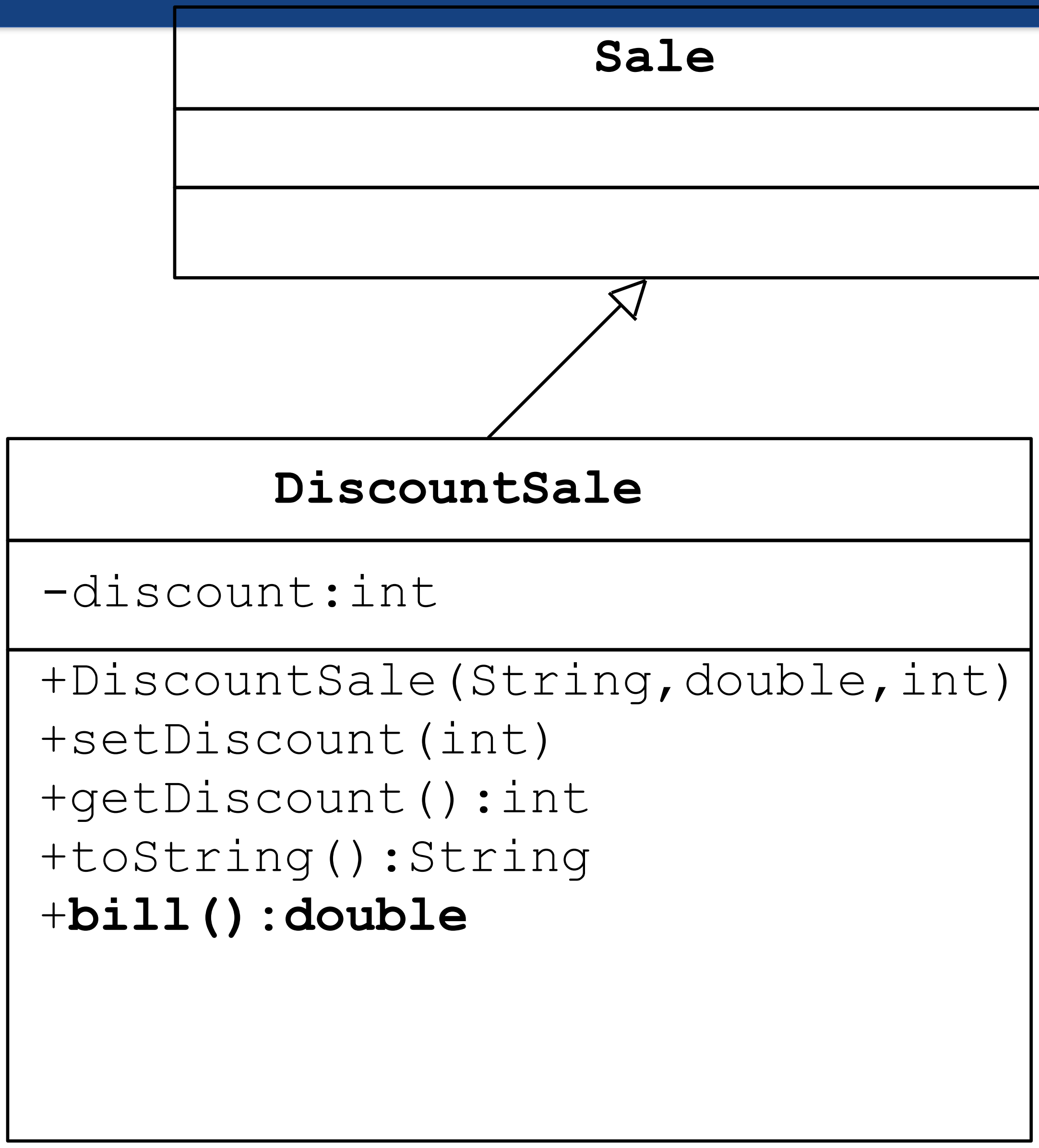
- Java uses late binding for all methods (except private, `final`, and static methods)
- Because of late binding, a method can be written in a base class to perform a task, even if portions of that task aren't yet defined
- For an example, the relationship between a base class called `Sale` and its derived class `DiscountSale` will be examined

Sale class



```
Sale
-name :String
-price:double
+Sale()
+Sale(String,double)
+Sale(Sale)
+setName(String)
+getName():String
+setPrice(double)
+getPrice():double
+toString():String
+bill():double
+equalDeals(Sale):boolean
+lessThan(Sale):boolean
```

DiscountSale class



The `Sale` and `DiscountSale` Classes

- The **`Sale`** class **`lessThan`** method
  - Note the **`bill()`** method invocations:

```
public boolean lessThan (Sale otherSale)
{
  if (otherSale == null)
  {
    System.out.println("Error: null object");
    System.exit(0);
  }
  return (bill( ) < otherSale.bill( ));
}
```

The `Sale` and `DiscountSale` Classes

- The **`Sale`** class **`bill()`** method:

```
public double bill( )
{
  return price;
}
```

- The **`DiscountSale`** class **`bill()`** method:

```
public double bill( )
{
    double discountedPrice = getPrice() * (1-discount/
      100);
    return discountedPrice + discountedPrice *
      SALES_TAX/100;
}
```

The `Sale` and `DiscountSale` Classes

- Given the following in a program:

```
   . . .
Sale simple = new sale("floor mat", 10.00);
DiscountSale discount = new
           DiscountSale("floor mat", 11.00, 10);
   . . .
if (discount.lessThan(simple))
  System.out.println("$" + discount.bill() +
                " < " + "$" + simple.bill() +
                " because late-binding works!");
   . . .
```

  - Output would be:

```
  $9.90 < $10 because late-binding works!
```

The `Sale` and `DiscountSale` Classes

- In the previous example, the **`boolean`** expression in the **`if`** statement returns **`true`**
- As the output indicates, when the **`lessThan`** method in the **`Sale`** class is executed, it knows which **`bill()`** method to invoke
  - The **`DiscountSale`** class **`bill()`** method for **`discount`**, and the **`Sale`** class **`bill()`** method for **`simple`**
- Note that when the **`Sale`** class was created and compiled, the **`DiscountSale`** class and its **`bill()`** method did not yet exist
  - These results are made possible by late-binding

Upcasting and Downcasting

- *Upcasting* is when an object of a derived class is assigned to a variable of a base class (or any ancestor class)

```
Sale saleVariable; //Base class
DiscountSale discountVariable = new
   DiscountSale("paint", 15,10); //Derived class
saleVariable = discountVariable; //Upcasting
System.out.println(saleVariable.toString());
```

- Because of late binding, `toString` above uses the definition given in the `DiscountSale` class

Upcasting and Downcasting

- *Downcasting* is when a type cast is performed from a base class to a derived class (or from any ancestor class to any descendent class)
  - Downcasting has to be done very carefully
  - In many cases it doesn't make sense, or is illegal:

```
discountVariable =                //will produce
     (DiscountSale)saleVariable;//run-time error
discountVariable = saleVariable //will produce
                                //compiler error
```

  - There are times, however, when downcasting is necessary, e.g., inside the `equals` method for a class:

```
Sale otherSale = (Sale)otherObject;//downcasting
```

Pitfall: Downcasting

- It is the responsibility of the programmer to use downcasting only in situations where it makes sense
  - The compiler does not check to see if downcasting is a reasonable thing to do
- Using downcasting in a situation that does not make sense usually results in a run-time error

Recap: A Better `equals` Method for the Class `Employee`

```
public boolean equals(Object otherObject)
{
  if(otherObject == null)
    return false;
  else if(getClass( ) != otherObject.getClass( ))
    return false;
  else
  {
    Employee otherEmployee = (Employee)otherObject;
    return (name.equals(otherEmployee.name) &&
      hireDate.equals(otherEmployee.hireDate));
  }
}
```

Tip: Checking to See if Downcasting is Legitimate

- Downcasting to a specific type is only sensible if the object being cast is an instance of that type
  - This is exactly what the **`instanceof`** operator tests for:
    `object instanceof ClassName`
  - It will return true if ***`object`*** is of type ***`ClassName`***
  - In particular, it will return true if ***`object`*** is an instance of any descendent class of ***`ClassName`***

A First Look at the `clone` Method (SKIP)

- Every object inherits a method named **`clone`** from the class **`Object`**
  - The method **`clone`** has no parameters
  - It is supposed to return a deep copy of the calling object
- However, the inherited version of the method was not designed to be used as is
  - Instead, each class is expected to override it with a more appropriate version

A First Look at the `clone` Method (SKIP)

- The heading for the **`clone`** method defined in the **`Object`** class is as follows:
  `protected Object clone()`
- The heading for a **`clone`** method that overrides the **`clone`** method in the **`Object`** class can differ somewhat from the heading above
  - A change to a more permissive access, such as from protected to public, is always allowed when overriding a method definition
  - Changing the return type from **`Object`** to the type of the class being cloned is allowed because every class is a descendent class of the class **`Object`**
  - This is an example of a covariant return type

A First Look at the `clone` Method (SKIP)

- If a class has a copy constructor, the **`clone`** method for that class can use the *copy constructor* to create the copy returned by the **`clone`** method

```
public Sale clone()
{
  return new Sale(this);
}
```

and another example:

```
public DiscountSale clone()
{
  return new DiscountSale(this);
}
```

Pitfall: Sometime the `clone` Method Return Type is `Object` (SKIP)

- Prior to version 5.0, Java did not allow covariant return types
  - There were no changes whatsoever allowed in the return type of an overridden method
- Therefore, the **`clone`** method for all classes had **`Object`** as its return type
  - Since the return type of the clone method of the **`Object`** class was **`Object`**, the return type of the overriding clone method of any other class was **`Object`** also

Pitfall: Sometime the `clone` Method Return Type is `Object` (SKIP)

- Prior to Java version 5.0, the **`clone`** method for the **`Sale`** class would have looked like this:

```
public Object clone()
{
  return new Sale(this);
}
```

- Therefore, the result must always be type cast when using a **`clone`** method written for an older version of Java

```
Sale copy = (Sale)original.clone();
```

Pitfall: Sometime the `clone` Method Return Type is `Object` (SKIP)

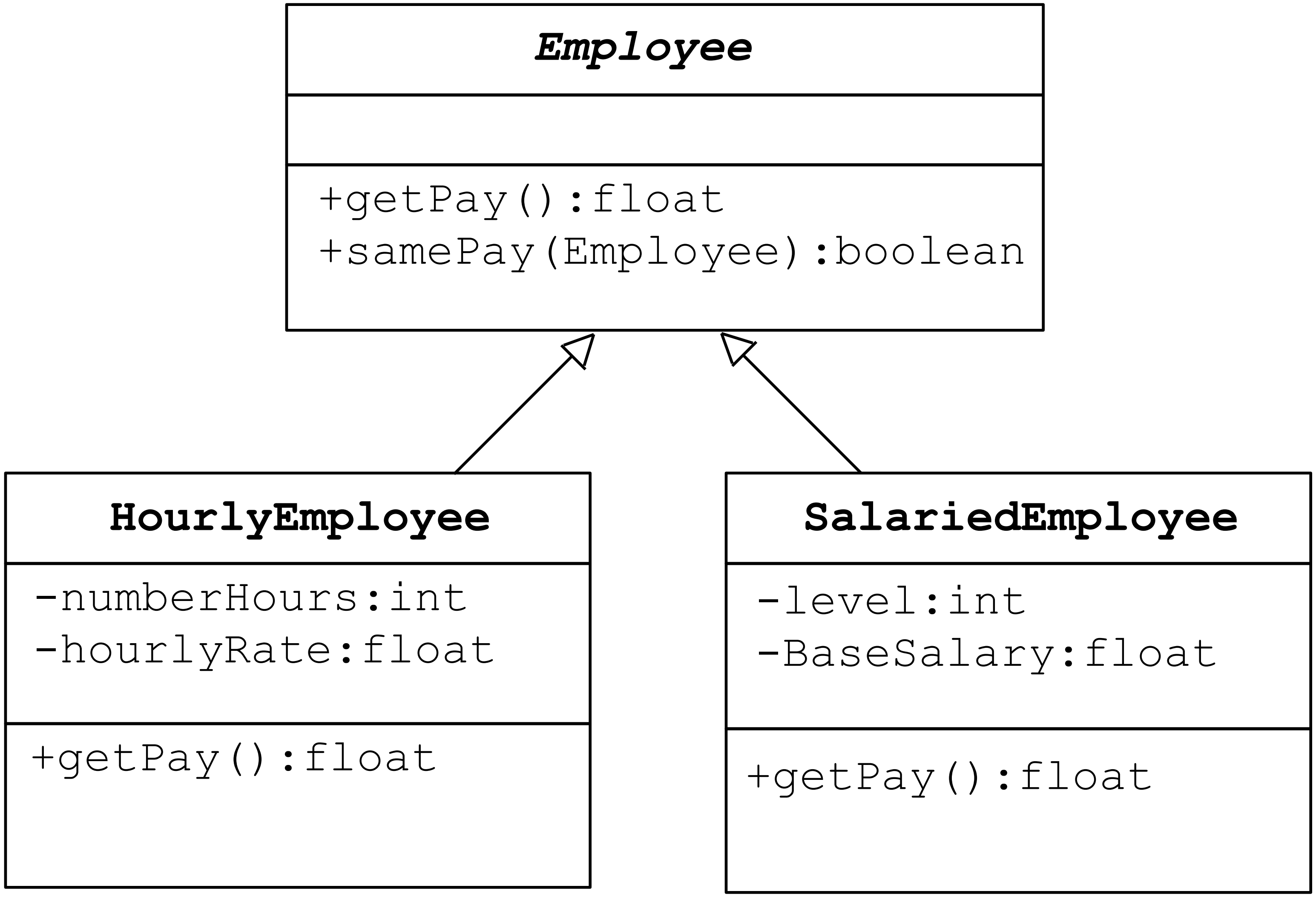
- It is still perfectly legal to use Object as the return type for a clone method, even with classes defined after Java version 5.0
  - When in doubt, it causes no harm to include the type cast
  - For example, the following is legal for the clone method of the Sale class:

    ```
    Sale copy = original.clone();
    ```
  - However, adding the following type cast produces no problems:

    ```
    Sale copy = (Sale)original.clone();
    ```

Abstract Classes

Introduction to Abstract Classes



Introduction to Abstract Classes

- In Chapter 7, the **`Employee`** base class and two of its derived classes, **`HourlyEmployee`** and **`SalariedEmployee`** were defined
- The following method is added to the **`Employee`** class
  - It compares employees to to see if they have the same pay:

```
public boolean samePay(Employee other)
{
  return(this.getPay() == other.getPay());
}
```

Introduction to Abstract Classes

- There are several problems with this method:
  - The `getPay` method is invoked in the `samePay` method
  - There are `getPay` methods in each of the derived classes
  - There is no `getPay` method in the `Employee` class, nor is there any way to define it reasonably without knowing whether the employee is hourly or salaried

Introduction to Abstract Classes

- The ideal situation would be if there were a way to
  - Postpone the definition of a `getPay` method until the type of the employee were known (i.e., in the derived classes)
  - Leave some kind of note in the `Employee` class to indicate that it was accounted for
- Surprisingly, Java allows this using abstract classes and methods

Introduction to Abstract Classes

- In order to postpone the definition of a method, Java allows an *abstract method* to be declared
  - An abstract method has a heading, but no method body
  - The body of the method is defined in the derived classes
- The class that contains an abstract method is called an *abstract class*

Abstract Method

- An abstract method is like a placeholder for a method that will be fully defined in a descendent class
- It has a complete method heading, to which has been added the modifier `abstract`
- It cannot be private
- It has no method body, and ends with a semicolon in place of its body

```
public abstract double getPay();
public abstract void doIt(int count);
```

Abstract Class

- A class that has at least one abstract method is called an *abstract class*
  - An abstract class must have the modifier **`abstract`** included in its class heading:

```
public abstract class Employee
{
  private instanceVariables;
  . . .
  public abstract double getPay();
  . . .
}
```

Abstract Class

- An abstract class can have any number of abstract and/or fully defined methods
- If a derived class of an abstract class adds to or does not define all of the abstract methods, then it is abstract also, and must add **`abstract`** to its modifier
- A class that has no abstract methods is called a ***concrete class***

Pitfall: You Cannot Create Instances of an Abstract Class

- An abstract class can only be used to derive more specialized classes
  - While it may be useful to discuss employees in general, in reality an employee must be a salaried worker or an hourly worker
- An abstract class constructor cannot be used to create an object of the abstract class
  - However, a derived class constructor will include an invocation of the abstract class constructor in the form of `super`
  - The constructor in an abstract class is only used by the constructor of its derived classes

Tip: An Abstract Class Is a Type

- Although an object of an abstract class cannot be created, it is perfectly fine to have a parameter of an abstract class type
  - This makes it possible to **plug in** an object of any of its **descendent** classes
- It is also fine to use a variable of an abstract class type, as long as it names objects of its concrete descendent classes only

THE UNIVERSITY OF MELBOURNE

Programming and Software Development
COMP90041

Exception Handling

Slides developed by Rose Williams

Introduction to Exception Handling

- Sometimes the best outcome can be when nothing unusual happens
- However, the case where exceptional things happen must also be prepared for
  - Java exception handling facilities are used when the invocation of a method may cause something exceptional to occur

Introduction to Exception Handling

- Java library software (or programmer-defined code) provides a mechanism that signals when something unusual happens
  - This is called ***throwing an exception***
- In another place in the program, the programmer must provide code that deals with the exceptional case
  - This is called ***handling the exception***

`try-throw-catch` Mechanism

- The basic way of handling exceptions in Java consists of the ***`try-throw-catch`*** trio
- The **`try`** block contains the code for the basic algorithm
  - It tells what to do when everything goes smoothly
- It is called a **`try`** block because it "tries" to execute the case where all goes as planned
  - It can also contain code that throws an exception if something unusual happens

```
try
{
   CodeThatMayThrowAnException
}
```

The try-throw-catch Trio

```
. . . // method code
try
{
  . . .
  throw new Exception(StringArgument);
  . . .
}
catch(Exception e)
{
  String message = e.getMessage();
  System.out.println(message);
  System.exit(0);
}  . . .
```

Eg. ExceptionDemo

`throw`

```
throw new
   ExceptionClassName(PossiblySomeArguments)
   ;
```

- If exception is thrown, **`try`** block stops
  - Normally, the flow of control is transferred to another portion of code known as the **`catch`** block
- The value thrown is the argument to the **`throw`** operator, and is always an object of some exception class

`throw`

- A `throw` statement is similar to a method call:
  `throw new ExceptionClassName(SomeString);`
  - In the above example, the object of class ***`ExceptionClassName`*** is created using a string as its argument
  - This object, which is an argument to the `throw` operator, is the exception object thrown
- Instead of calling a method, a `throw` statement calls a `catch` block

`catch`

- When an exception is thrown, the `catch` block begins execution
  - The `catch` block has only **one** parameter
  - The exception object thrown is plugged in for the `catch` block parameter
- The execution of the `catch` block is called *catching the exception*, or *handling the exception;* the catch block is an exception handler
  - Whenever an exception is thrown, it should ultimately be handled (or caught) by some `catch` block

`catch`

```
catch(Exception e)
{
   ExceptionHandlingCode
}
```

- A **`catch`** block looks like a method definition that has a parameter of type ***`Exception`*** class
  - It is not really a method definition

`catch`

```
    catch(Exception e) { . . . }
```

- `e` is called the **catch block parameter**
- The `catch` block parameter does two things:
  - It specifies the type of thrown exception object that the `catch` block can catch (e.g., an `Exception` class object above)
  - It provides a name (for the thrown object that is caught) on which it can operate in the `catch` block
    - Note: The identifier `e` is often used by convention, but any non-keyword identifier can be used

`try-throw-catch` Mechanism

- When a `try` block is executed, two things can happen:
  1. No exception is thrown in the `try` block
     - The code in the `try` block is executed to the end of the block
     - The `catch` block is skipped
     - The execution continues with the code placed after the `catch` block

`try-throw-catch` Mechanism

2. An exception is thrown in the `try` block and caught in the `catch` block
   - The rest of the code in the `try` block is skipped
   - Control is transferred to a following `catch` block (in simple cases)
   - The thrown object is plugged in for the `catch` block parameter
   - The code in the `catch` block is executed
   - The code that follows that `catch` block is executed (if any)

Using the `getMessage` Method

```
. . . // method code
try
{
   . . .
   throw new Exception(StringArgument);
   . . .
}
catch(Exception e)
{
   String message = e.getMessage();
   System.out.println(message);
   System.exit(0);
}  . . .
```

Using the `getMessage` Method

- Every exception has a **`String`** instance variable that contains some message
  - This string typically identifies the reason for the exception
- In the previous example, **`StringArgument`** is an argument to the **`Exception`** constructor
- This is the string used for the value of the string instance variable of exception **`e`**
  - Therefore, the method call **`e.getMessage()`** returns this string

Exception Classes

- There are more exception classes than just the single class `Exception`
  - There are more exception classes in the standard Java libraries
  - New exception classes can be defined like any other class
- All predefined exception classes have the following properties:
  - There is a constructor that takes a single argument of type `String`
  - The class has an accessor method `getMessage` that can recover the string given as an argument to the constructor when the exception object was created
- All programmer-defined classes must be derived from the class `Exception`

Exception Classes from Standard Packages

- The predefined exception class **`Exception`** is the root class for all exceptions
  - Every exception class is a descendent class of the class **`Exception`**
  - Although the **`Exception`** class can be used directly in a class or program, it is most often used to define a derived class
  - The class **`Exception`** is in the **`java.lang`** package, and so requires no **`import`** statement

Exception Classes from Standard Packages

- Numerous predefined exception classes are included in the standard packages that come with Java
  - For example:
    ```
    IOException
    NoSuchMethodException
    FileNotFoundException
    ```
  - Many exception classes must be imported in order to use them
    ```
    import java.io.IOException;
    ```

Defining Exception Classes

- A `throw` statement can throw an exception object of any exception class
- Instead of using a predefined class, exception classes can be programmer-defined
  - These can be tailored to carry the precise kinds of information needed in the `catch` block
  - A different type of exception can be defined to identify each different exceptional situation

Defining Exception Classes

- Every exception class to be defined must be a derived class of some already defined exception class
  - It can be a derived class of any exception class in the standard Java libraries, or of any programmer defined exception class
- Constructors are the most important members to define in an exception class
  - They must behave appropriately with respect to the variables and methods inherited from the base class
  - Often, there are no other members, except those inherited from the base class
- The following exception class performs these basic tasks only

A Programmer-Defined Exception Class

Display 9.3 A Programmer-Defined Exception Class

```
public class DivisionByZeroException extends Exception
{
    public DivisionByZeroException()
    {
        super("Division by Zero!");
    }

    public DivisionByZeroException(String message)
    {
        super(message);
    }
}
```

You can do more in an exception constructor, but this form is common.

super is an invocation of the constructor for the base class Exception.

Tip: An Exception Class Can Carry a Message of Any Type: int Message

- An exception class constructor can be defined that takes an argument of another type
  - It would stores its value in an instance variable
  - It would need to define accessor methods for this instance variable

An Exception Class with an `int` Message

Display 9.5 An Exception Class with an `int` Message

```
public class BadNumberException extends Exception
{
    private int badNumber;

    public BadNumberException(int number)
    {
        super("BadNumberException");
        badNumber = number;
    }

    public BadNumberException()
    {
        super("BadNumberException");
    }

    public BadNumberException(String message)
    {
        super(message);
    }

    public int getBadNumber()
    {
        return badNumber;
    }
}
```

Exception Object Characteristics

- The two most important things about an exception object are its **type** (i.e., exception class) and the **message** it carries
  - The message is sent along with the exception object as an instance variable
  - This message can be recovered with the accessor method `getMessage`, so that the catch block can use the message

Programmer-Defined Exception Class Guidelines

- Must be a derived class of an already existing exception class
- At least two constructors should be defined, sometimes more
- The exception class should allow for the fact that the method `getMessage` is inherited

Preserve `getMessage`

- For all predefined exception classes, **`getMessage`** returns the string that is passed to its constructor as an argument
  - Or it will return a default string if no argument is used with the constructor
- This behavior must be preserved in all programmer-defined exception class, two constructors must be included:
  - A constructor that takes a string argument and begins with a call to **`super`**, which takes the string argument
  - A no-argument constructor that includes a call to **`super`** with a default string as the argument

Multiple `catch` Blocks

- A `try` block can potentially throw any number of exception values, and they can be of differing types
  - In any one execution of a `try` block, at most one exception can be thrown (since a throw statement ends the execution of the `try` block)
  - However, different types of exception values can be thrown on different executions of the `try` block

Multiple `catch` Blocks

- Each `catch` block can only catch values of the exception class type given in the `catch` block heading
- Different types of exceptions can be caught by placing more than one `catch` block after a `try` block
  - Any number of `catch` blocks can be included, but they must be placed in the correct order

Pitfall: Catch the More Specific Exception First

- When catching multiple exceptions, the order of the `catch` blocks is important
  - When an exception is thrown in a `try` block, the `catch` blocks are examined in order
  - The first one that matches the type of the exception thrown is the one that is executed

Pitfall: Catch the More Specific Exception First

```
catch (Exception e)
{ . . . }
catch (NegativeNumberException e)
{ . . . }
```

- Because a `NegativeNumberException` is a type of `Exception`, all `NegativeNumberExceptions` will be caught by the first `catch` block before ever reaching the second block
  - The catch block for `NegativeNumberException` will never be used!
- For the correct ordering, simply reverse the two blocks

Throwing an Exception in a Method

- Sometimes it makes sense to throw an exception in a method, but not catch it in the same method
  - Some programs that use a method should just end if an exception is thrown, and other programs should do something else
  - In such cases, the program using the method should enclose the method invocation in a `try` block, and catch the exception in a `catch` block that follows
- In this case, the method itself would not include `try` and `catch` blocks
  - However, it would have to include a ***`throws`*** *clause*

Declaring Exceptions in a `throws` Clause

- If a method can throw an exception but does not catch it, it must provide a warning in the heading
  - This warning is called a ***`throws`*** *clause*
  - The process of including an exception class in a throws clause is called *declaring the exception*

    ```
    throws AnException  //throws clause
    ```

  - The following states that an invocation of **`aMethod`** could throw **`AnException`**

    ```
    public void aMethod() throws AnException
    ```

- If a method throws an exception and does not catch it, then the method invocation ends immediately

Declaring Multiple Exceptions in a `throws` Clause

- If a method can throw more than one type of exception, then separate the exception types by commas

```
public void aMethod() throws
   AnException, AnotherException
```

Defining Exceptions in a Method

- Here is an example of a method from which the exception originates:

```
public void someMethod()
                         throws SomeException
{
   . . .
   throw new
              SomeException(SomeArgument);
   . . .
}
```

Handling Exceptions in another Method

- When **someMethod** is used by an **otherMethod**, the **otherMethod** must then deal with the exception:

```
public void otherMethod()
{
   try
   {
      someMethod();
      . . .
   }
   catch (SomeException e)
   {
      CodeToHandleException
   }
   . . .
}
```

The Catch or Declare Rule: Two ways

- Two ways of handling exceptions thrown in a method:
    1. The code that can throw an exception is placed within a `try` block, and the possible exception is caught in a `catch` block within the same method
    2. The possible exception can be declared at the start of the method definition by placing the exception class name in a `throws` clause

The Catch or Declare Rule: An Exception must be handled somewhere

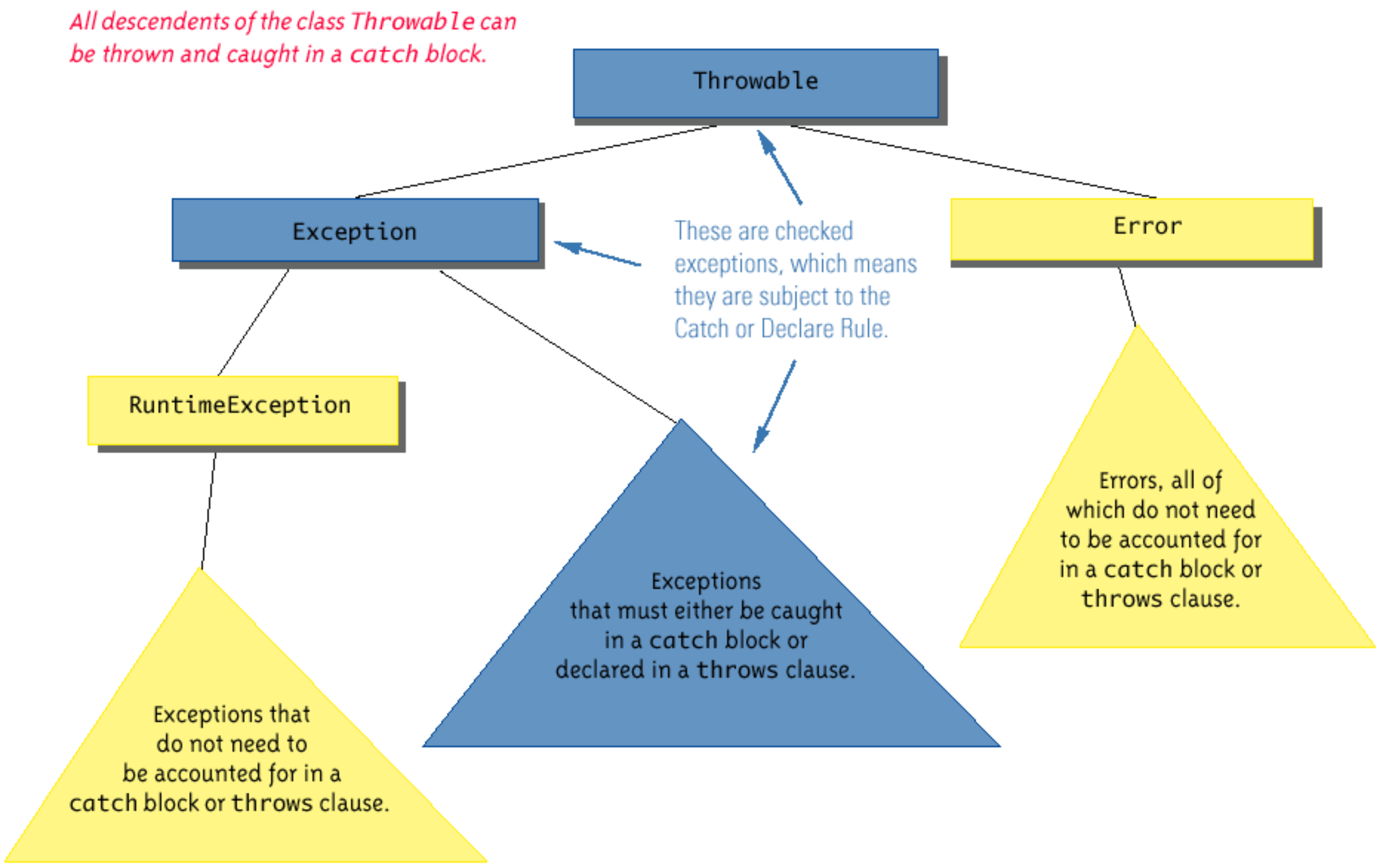
- The first technique handles an exception in a `catch` block
- The second technique is a way to shift the exception handling responsibility to the method that invoked the exception throwing method
- The invoking method must handle the exception, unless it too uses the same technique to "pass the buck"
- Ultimately, every exception that is thrown should eventually be caught by a `catch` block in some method that does not just declare the exception class in a `throws` clause

The Catch or Declare Rule: Mixed usage

- Two techniques can be **mixed**
  - Some exceptions may be caught, and others may be declared in a `throws` clause
- However, these techniques must be used consistently with a given exception
  - If an exception is not declared, then it must be handled within the method
  - If an exception is declared, then the responsibility for handling it is shifted to some other calling method
  - Note that if a method definition encloses an invocation of a second method, and the second method can throw an exception and does not catch it, then the first method must catch or declare it

Hierarchy of Throwable Objects

Display 9.10 **Hierarchy of Throwable Objects**



All descendents of the class Throwable can be thrown and caught in a catch block.
Throwable
Exception
Error
These are checked exceptions, which means they are subject to the Catch or Declare Rule.
RuntimeException
Exceptions that do not need to be accounted for in a catch block or throws clause.
Exceptions that must either be caught in a catch block or declared in a throws clause.
Errors, all of which do not need to be accounted for in a catch block or throws clause.

Checked and Unchecked Exceptions

- Exceptions that are subject to the catch or declare rule are called *checked* exceptions
  - The compiler checks to see if they are accounted for with either a catch block or a throws clause
  - The classes `Throwable`, `Exception`, and all descendants of the class `Exception` are checked exceptions
- All other exceptions are *unchecked* exceptions -- **must be corrected**.
- The class `Error` and all its descendant classes are called *error classes*
  - Error classes are *not* subject to the Catch or Declare Rule

The `throws` Clause in Derived Classes

- When a method in a derived class is overridden, it should have the same exception classes listed in its **`throws`** clause that it had in the base class
  - Or it should have a subset of them
- A derived class may not add any exceptions to the **`throws`** clause
  - But it can delete some

What Happens If an Exception is Never Caught?

- If every method up to and including the main method simply includes a `throws` clause for an exception, that exception may be thrown but never caught
  - In a GUI program (i.e., a program with a windowing interface), nothing happens - but the user may be left in an unexplained situation, and the program may be no longer be reliable
  - In non-GUI programs, this causes the program to terminate with an error message giving the name of the exception class
- Every well-written program should eventually catch every exception by a `catch` block in some method

When to Use Exceptions

- Exceptions should be reserved for situations where a method encounters *an unusual or unexpected case that cannot be handled easily in some other way*
- How exceptions are handled depends on how a method is called.

Event Driven Programming

- Exception handling is an example of a programming methodology known as *event-driven programming*
- When using event-driven programming, objects are defined so that they send events to other objects that handle the events
  - An event is an object also
  - Sending an event is called *firing an event*

Event Driven Programming

- In exception handling, the event objects are the exception objects
  - They are fired (thrown) by an object when the object invokes a method that throws the exception
  - An exception event is sent to a `catch` block, where it is handled

Pitfall: Nested `try-catch` Blocks

- It is possible to place a `try` block and its following catch blocks inside a larger `try` block, or inside a larger `catch` block
  - If a set of `try-catch` blocks are placed inside a larger `catch` block, **different names** must be used for the `catch` block parameters in the inner and outer blocks, just like any other set of nested blocks
  - If a set of `try-catch` blocks are placed inside a larger `try` block, and an exception is thrown in the **inner** `try` block that is **not caught**, then the exception is thrown to the **outer** `try` block for processing, and may be caught in one of its `catch` blocks

The `finally` Block

- The `finally` block contains code to be executed whether or not an exception is thrown in a `try` block
  - If it is used, a `finally` block is placed after a `try` block and its following `catch` blocks

```
try
{   .  .  .   }
catch(ExceptionClass1 e)
{   .  .  .   }
   .  .  .
catch(ExceptionClassN e)
{   .  .  .   }
finally
{
   CodeToBeExecutedInAllCases
}
```

The `finally` Block

- If the `try-catch-finally` blocks are inside a method definition, there are three possibilities when the code is run:
  1. The `try` block runs to the end, no exception is thrown, and the finally block is executed
  2. An exception is thrown in the `try` block, caught in one of the `catch` blocks, and the `finally` block is executed
  3. An exception is thrown in the `try` block, there is no matching `catch` block in the method, the `finally` block is executed, and then the method invocation ends and the exception object is thrown to the enclosing method

The `AssertionError` Class

- When a program contains an assertion check, and the assertion check fails, an object of the class **`AssertionError`** is thrown
  - This causes the program to end with an error message
- The class **`AssertionError`** is derived from the class **`Error`**, and therefore is an unchecked Throwable
  - In order to prevent the program from ending, it could be handled, but this is not required

Exception Handling with the `Scanner` Class

- The `nextInt` method of the `Scanner` class can be used to read `int` values from the keyboard
- However, if a user enters something other than a well-formed `int` value, an `InputMismatchException` will be thrown
  - Unless this exception is caught, the program will end with an error message
  - If the exception is caught, the `catch` block can give code for some alternative action, such as asking the user to reenter the input

The `InputMismatchException`

- The **`InputMismatchException`** is in the standard Java package **`java.util`**
  - A program that refers to it must use an **`import`** statement, such as the following:
    `import java.util.InputMismatchException;`
- It is a descendent class of **`RuntimeException`**
  - Therefore, it is an unchecked exception and does not have to be caught in a **`catch`** block or declared in a **`throws`** clause
  - However, catching it in a **`catch`** block is allowed, and can sometimes be useful

Tip: Exception Controlled Loops

- Sometimes it is better to simply loop through an action again when an exception is thrown, as follows:

```
boolean done = false;
while (! done)
{
   try
   {
      CodeThatMayThrowAnException
      done = true;
   }
   catch (SomeExceptionClass e)
   {
      SomeMoreCode
   }
}
```

An Exception Controlled Loop (Part 1 of 3)

Display 9.11 **An Exception Controlled Loop**

```
import java.util.Scanner;
import java.util.InputMismatchException;

public class InputMismatchExceptionDemo
{
    public static void main(String[] args)
    {
        Scanner keyboard = new Scanner(System.in);
        int number = 0; //to keep compiler happy
        boolean done = false;
```

(continued)

An Exception Controlled Loop (Part 2 of 3)

Display 9.11 An Exception Controlled Loop

```
            while (! done)
            {
                try
                {
                    System.out.println("Enter a whole number:");
                    number = keyboard.nextInt();
                    done = true;
                 }
                 catch(InputMismatchException e)
                 {
                     keyboard.nextLine();
                     System.out.println("Not a correctly written whole number.");
                     System.out.println("Try again.");
                 }
            }

            System.out.println("You entered " + number);
        }
    }
```

If **nextInt** *throws an exception, the* **try** *block ends and so the* **boolean** *variable* **done** *is not set to* **true**.

(continued)

An Exception Controlled Loop (Part 3 of 3)

Display 9.11 **An Exception Controlled Loop**

Sample Dialogue

```
Enter a whole number:
forty two
Not a correctly written whole number.
Try again.
Enter a whole number:
fortytwo
Not a correctly written whole number.
Try again.
Enter a whole number:
42
You entered 42
```

`ArrayIndexOutOfBoundsException`

- An `ArrayIndexOutOfBoundsException` is thrown whenever a program attempts to use an array index that is out of bounds
  - This normally causes the program to end
- Like all other descendents of the class `RuntimeException`, it is an unchecked exception
  - There is no requirement to handle it
- When this exception is thrown, it is an indication that the program contains an error
  - Instead of attempting to handle the exception, the program should simply be fixed

THE UNIVERSITY OF MELBOURNE

Programming and Software Development
COMP90041

Interfaces

Slides developed by Rose Williams

Interfaces

- An *interface* is something like an extreme case of an abstract class
  - However, *an interface is not a class*
  - *It is a type that can be satisfied by any class that implements the interface*
- The syntax for defining an interface is similar to that of defining a class
  - Except the word **`interface`** is used in place of **`class`**
- An interface specifies a set of methods that any class that implements the interface must have
  - It contains **method headings** and **constant definitions** only
  - It contains no instance variables nor any complete method definitions

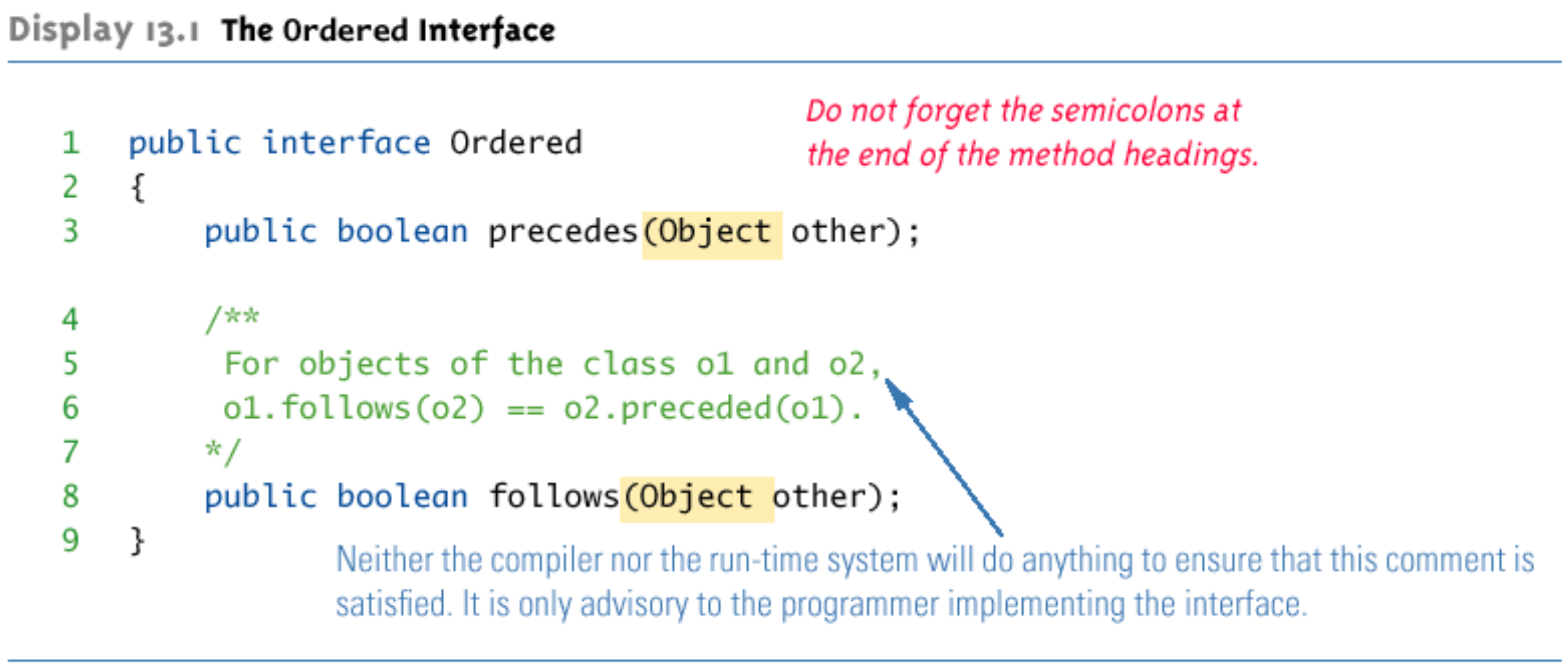
Interfaces

- An interface serves a function similar to a base class, though it is not a base class
  - Some languages allow one class to be derived from two or more different base classes
  - This *multiple inheritance* is not allowed in Java
  - Instead, Java's way of approximating multiple inheritance is through interfaces

Interfaces

- An interface and all of its method headings should be declared public
  - They cannot be given private, protected, or package access
- When a class implements an interface, it must make all the methods in the interface public
- Because an interface is a type, a method may be written with a parameter of an interface type
  - That parameter will accept as an argument any class that implements the interface

The `Ordered` Interface



Display 13.1 The Ordered Interface

Do not forget the semicolons at the end of the method headings.

```
public interface Ordered
{
    public boolean precedes(Object other);

    /**
     For objects of the class o1 and o2,
     o1.follows(o2) == o2.preceded(o1).
    */
    public boolean follows(Object other);
}
```

Neither the compiler nor the run-time system will do anything to ensure that this comment is satisfied. It is only advisory to the programmer implementing the interface.

Interfaces

- To *implement an interface*, a concrete class must do two things:
    1. It must include the phrase
       `implements Interface_Name`
       at the start of the class definition
        - If more than one interface is implemented, each is listed, separated by commas
    2. The class must implement **all** the method headings listed in the definition(s) of the interface(s)
- Note the use of `Object` as the parameter type in the following examples

Implementation of an Interface

Display 13.2 Implementation of an Interface

```
public class OrderedHourlyEmployee
          extends HourlyEmployee implements Ordered
{
    public boolean precedes(Object other)
    {
        if (other == null)
            return false;
        else if (!(other instanceof OrderedHourlyEmployee))
            return false;
        else
        {
            OrderedHourlyEmployee otherOrderedHourlyEmployee =
                                 (OrderedHourlyEmployee)other;
            return (getPay() < otherOrderedHourlyEmployee.getPay());
        }
    }
```

Although `getClass` *works better than* `instanceof` *for defining* `equals`, `instanceof` *works better in this case. However, either will do for the points being made here.*

Implementation of an Interface

```
    public boolean follows(Object other)
    {
        if (other == null)
            return false;
        else if (!(other instanceof OrderedHourlyEmployee))
            return false;
        else
        {
            OrderedHourlyEmployee otherOrderedHourlyEmployee =
                                  (OrderedHourlyEmployee)other;
            return (otherOrderedHourlyEmployee.precedes(this));
        }
    }
}
```

Abstract Classes Implementing Interfaces

- Abstract classes may implement one or more interfaces
  - Any method headings given in the interface that are not given definitions are made into abstract methods
- A concrete class must give definitions for all the method headings given in the abstract class *and the interface*

An Abstract Class Implementing an Interface

Display 13.3 An Abstract Class Implementing an Interface

```
public abstract class MyAbstractClass implements Ordered
{
    int number;
    char grade;

    public boolean precedes(Object other)
    {
        if (other == null)
            return false;
        else if (!(other instanceof HourlyEmployee))
            return false;
        else
        {
            MyAbstractClass otherOfMyAbstractClass =
                                            (MyAbstractClass)other;
            return (this.number < otherOfMyAbstractClass.number);
        }
    }

    public abstract boolean follows(Object other);

}
```

Derived Interfaces

- Like classes, an interface may be derived from a base interface
  - This is called *extending* the interface
  - The derived interface must include the phrase
    `extends` ***`BaseInterfaceName`***
- A concrete class that implements a derived interface must have definitions for any methods in the derived interface as well as any methods in the base interface

Extending an Interface

Display 13.4 Extending an Interface

```
public interface ShowablyOrdered extends Ordered
{
    /**
      Outputs an object of the class that precedes the calling object.
    */
    public void showOneWhoPrecedes();
}
```

Neither the compiler nor the run-time system will do anything to ensure that this comment is satisfied.

A (concrete) class that implements the `ShowablyOrdered` interface must have a definition for the method `showOneWhoPrecedes` and also have definitions for the methods `precedes` and `follows` given in the `Ordered` interface.

Pitfall: Interface Semantics Are Not Enforced

- When a class implements an interface, the compiler and run-time system check the syntax of the interface and its implementation
  - However, neither checks that the body of an interface is consistent with its intended meaning
- Required semantics for an interface are normally added to the documentation for an interface
  - It then becomes the responsibility of each programmer implementing the interface to follow the semantics
- If the method body does not satisfy the specified semantics, then software written for classes that implement the interface may not work correctly

The `Comparable` Interface

- Chapter 6 discussed the Selection Sort algorithm, and examined a method for sorting a partially filled array of type **`double`** into increasing order
- This code could be modified to sort into decreasing order, or to sort integers or strings instead
  - Each of these methods would be essentially the same, but making each modification would be a nuisance
  - The only difference would be the types of values being sorted, and the definition of the ordering
- Using the **`Comparable`** interface could provide a single sorting method that covers all these cases

The `Comparable` Interface

- The `Comparable` interface is in the `java.lang` package, and so is automatically available to any program
- It has only the following method heading that must be implemented:

```
public int compareTo(Object other);
```

- It is the programmer's responsibility to follow the semantics of the `Comparable` interface when implementing it

The `Comparable` Interface Semantics

- The method **`compareTo`** must return
  - A negative number if the calling object "comes before" the parameter other
  - A zero if the calling object "equals" the parameter other
  - A positive number if the calling object "comes after" the parameter other
- If the parameter **`other`** is not of the same type as the class being defined, then a **`ClassCastException`** should be thrown

The `Comparable` Interface Semantics

- Almost any reasonable notion of "comes before" is acceptable
  - In particular, all of the standard less-than relations on numbers and lexicographic ordering on strings are suitable
- The relationship "comes after" is just the reverse of "comes before"

Using the `Comparable` Interface

- The following example reworks the `SelectionSort` class from Chapter 6
- The new version, `GeneralizedSelectionSort`, includes a method that can sort any partially filled array *whose base type implements the `Comparable` interface*
  - It contains appropriate `indexOfSmallest` and `interchange` methods as well
- Note: Both the `Double` and `String` classes implement the `Comparable` interface
  - Interfaces apply to classes only
  - A primitive type (e.g., `double`) cannot implement an interface

`GeneralizedSelectionSort` class: `sort` Method

Display 13.5 Sorting Method for Array of Comparable *(Part 1 of 2)*

```
public class GeneralizedSelectionSort
{
    /**
     Precondition: numberUsed <= a.length;
                   The first numberUsed indexed variables have values.
     Action: Sorts a so that a[0, a[1], ... , a[numberUsed - 1] are in
     increasing order by the compareTo method.
    */
    public static void sort(Comparable[] a, int numberUsed)
    {
        int index, indexOfNextSmallest;
        for (index = 0; index < numberUsed - 1; index++)
        {//Place the correct value in a[index]:
            indexOfNextSmallest = indexOfSmallest(index, a, numberUsed);
            interchange(index,indexOfNextSmallest, a);
             //a[0], a[1],..., a[index] are correctly ordered and these are
             //the smallest of the original array elements. The remaining
             //positions contain the rest of the original array elements.
        }
    }
```

GeneralizedSelectionSort class: sort Method

Display 13.5 Sorting Method for Array of Comparable *(Part 1 of 2)* (continued)

```java
    /**
     Returns the index of the smallest value among
     a[startIndex], a[startIndex+1], ... a[numberUsed - 1]
    */
    private static int indexOfSmallest(int startIndex,
                                       Comparable[] a, int numberUsed)
    {
        Comparable min = a[startIndex];
        int indexOfMin = startIndex;
        int index;
        for (index = startIndex + 1; index < numberUsed; index++)
            if (a[index].compareTo(min) < 0)//if a[index] is less than min
            {
                min = a[index];
                indexOfMin = index;
                //min is smallest of a[startIndex] through a[index]
            }
        return indexOfMin;
    }
```

`GeneralizedSelectionSort` class: `interchange` Method

Display 13.5 **Sorting Method for Array of Comparable *(Part 2 of 2)***

```
    /**
     Precondition: i and j are legal indices for the array a.
     Postcondition: Values of a[i] and a[j] have been interchanged.
    */
    private static void interchange(int i, int j, Comparable[] a)
    {
        Comparable temp;
        temp = a[i];
        a[i] = a[j];
        a[j] = temp; //original value of a[i]
    }

}
```

Sorting Arrays of Comparable

Display 13.6 **Sorting Arrays of Comparable *(Part 1 of 2)***

```
/**
 Demonstrates sorting arrays for classes that
 implement the Comparable interface.
*/
public class ComparableDemo
{
    public static void main(String[] args)
    {
        Double[] d = new Double[10];
        for (int i = 0; i < d.length; i++)
            d[i] = new Double(d.length - i);

        System.out.println("Before sorting:");
        int i;
        for (i = 0; i < d.length; i++)
            System.out.print(d[i].doubleValue() + ", ");
        System.out.println();

        GeneralizedSelectionSort.sort(d, d.length);

        System.out.println("After sorting:");
        for (i = 0; i < d.length; i++)
            System.out.print(d[i].doubleValue() + ", ");
        System.out.println();
```

The classes Double and String do implement the Comparable interface.

Sorting Arrays of `Comparable`

Display 13.6 Sorting Arrays of Comparable *(Part 2 of 2)*

```
        String[] a = new String[10];
        a[0] = "dog";
        a[1] = "cat";
        a[2] = "cornish game hen";
        int numberUsed = 3;

        System.out.println("Before sorting:");
        for (i = 0; i < numberUsed; i++)
            System.out.print(a[i] + ", ");
        System.out.println();

        GeneralizedSelectionSort.sort(a, numberUsed);
```

Sorting Arrays of `Comparable`

Display 13.6 Sorting Arrays of Comparable *(Part 2 of 2)* (continued)

```
        System.out.println("After sorting:");
        for (i = 0; i < numberUsed; i++)
            System.out.print(a[i] + ", ");
        System.out.println();
    }
}
```

SAMPLE DIALOGUE

```
Before Sorting
10.0, 9.0, 8.0, 7.0, 6.0, 5.0, 4.0, 3.0, 2.0, 1.0,
After sorting:
1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0,
Before sorting;
dog, cat, cornish game hen,
After sorting:
cat, cornish game hen, dog,
```

Defined Constants in Interfaces

- An interface can contain defined constants in addition to or instead of method headings
  - Any variables defined in an interface must be **public**, **static**, and **final**
  - Because this is understood, Java allows these modifiers to be **omitted**
- Any class that implements the interface has access to these defined constants

Pitfall: Inconsistent Interfaces

- In Java, a class can have only one base class
  - This prevents any inconsistencies arising from different definitions having the same method heading
- In addition, a class may implement any number of interfaces
  - Since interfaces do not have method bodies, the above problem cannot arise
  - However, there are other types of inconsistencies that can arise

Pitfall: Inconsistent Interfaces

- When a class implements two interfaces:
  - One type of inconsistency will occur if the interfaces have **constants** with the same name, but with different values
  - Another type of inconsistency will occur if the interfaces contain **methods** with the same name but different return types
- If a class definition implements two inconsistent interfaces, then that is an error, and the class definition is **illegal**

- Mini Lectures
  - Polymorphism
  - Abstract Classes
  - Handling Exceptions
- Live Lecture
  - Dungeons & Dragons

Take-Aways

Please fill in this microblog.

http://go.unimelb.edu.au/q49r