1 **Abstract Classes and Interfaces**

CST141

2 **Late Binding (Page 1)**

- Programmers should create systems (applications) that are easily extensible
  - Capable of being *extended*—easy to add to later
- Superclasses are designed as more general:
  - Able to process *existing* as well as *new* subclasses
  - Classes that are added later should not require modification to the general part of the program (the superclass)

3 **Late Binding (Page 2)**

- Late binding—a method from one class is not tied to method that calls it from another class until run-time (when it is instantiated)
  - Also called dynamic binding
  - The opposite of early binding in which the methods of the two classes are *compiled* together
- Late binding makes it possible to add new classes to the hierarchy even after the base class compiles

5 **Late Binding (Page 3)**

- Consider the Shape class example:
  - Shape has:
    - An attribute point where the shape starts to draw
    - A method printIt() that "positions" a shape when drawn by calling a method named position()
  - Classes Circle and Rectangle both extend Shape
    - Circle has attribute radius; Rectangle has attributes length and width
    - Circle and radius have individual methods named draw() that "draw" the shapes, both of which are *called* by the printIt() method of class Shape

7 **Late Binding (Page 4)**

- Consider the Shape class example (*con*):
  - With early binding, if *new* class Triangle is created after Shape is compiled, method draw() of either Circle or Rectangle will have been bound previously to printIt()
  - With late binding (essentially the *equivalent of polymorphism*), method draw() of Triangle (or Circle or Rectangle) correctly binds to printIt() at *run-time*
  - Java uses late binding exclusively

9 **The Keyword abstract (Page 1)**

- Classes that are declared to be abstract cannot be instantiated …
  - No objects may be created from it
- This is true for a superclass that only has the function of *supporting subclasses* …
  - Such classes are called abstract superclasses

10 **The Keyword abstract (Page 2)**

- Example:

  public abstract class Shape extends Object
- Classes that *may* instantiate objects are called concrete classes

- E.g. the Circle, Rectangle and Triangle classes

11 **Declaring abstract Methods (Page 1)**

- A method may be declared in a superclass declaration as abstract
- As such the abstract method only may exist in an abstract class (or an interface)

12 **Declaring abstract Methods (Page 2)**

- The declaration is only a *reference* since:
  - It contains *no statements*
  - Requires implementation of the abstract method in all of its subclasses
    - So that required subclass methods are not forgotten
  - Any call to the local abstract method is *overridden* because it will be handled by methods of same name in the subclasses (uses redirection)
    - In fact this is the *only way* that a superclass can *call methods of its direct subclass*

13 **Declaring abstract Methods (Page 3)**

- Format:
  public abstract *type*|void *methodName*([*parameterList*]);
  - The *parameterList* must match in number of variables and type the implemented method
  - Methods that are abstract may be *overloaded*
- Example:
  public abstract void draw();
  - Note the placement of the semicolon (;) at end of the method header (signature)

18 **The Keyword final**

- Used to indicate that the value of an identifier *may not change* after it has been declared and initialized
  - Often used for defining a constant
- Example:
  double final CREDITS = 7;

19 **Declaring a Class as final**

- If a class is declared to be final, it must be the bottom class in an inheritance hierarchy
  - It may *not have any subclasses*
- Example:
  public final class Circle extends Shape

71 **Downcasting and Polymorphic Behavior (Page 1)**

- Casting a superclass reference to a subclass reference
- Technique makes it possible to reference a subclass method from an object instantiated from its superclass
- Accomplished by casting the superclass object (superclass is the type) to the subclass type (subclass is the constructor)

72 **Downcasting and Polymorphic Behavior (Page 2)**

- Format:
  SuperClassName object = new *SubClassConstructor*([*args*]);

- Possible only because the subclass is *derived* (extends from) from the superclass

73 **Downcasting and Polymorphic Behavior (Page 3)**

Example:

```
Student s = new SuffolkResident("Sally", "Walters", "Z", 7);
…
JOptionPane.showMessageDialog(null, s.getTuition() )
```

- Calls getTuition() method of class SuffolkResident (not that of Student)

77 **Interfaces**

Contains abstract method definitions needed by several classes and perhaps within several class hierarchies

- An alternate to declaring them in a superclass

If a method is declared in an interface, all classes that "implement" the interface *must* declare a method with the same signature

78 **The Keyword interface**

Used to *declare* an interface (replaces the keyword class in the header signature)

- As with a class name, the name of the interface must be identical to the "*.java" filename

Example:

```
public interface Tuition
{
    public abstract int getTuition();
}
```

- Filename for the above must be "Tuition.java"

79 **Implementing Interfaces**

Interfaces are *not inherited* in subclasses but rather they are *implemented*

Classes may implement *several* interfaces …

- Sort of like *multiple* inheritance …
- Unlike subclasses which may inherit (extend) from *only one* superclass

80 **The Keyword implements**

Used to implement an interface

Format:

public class *ClassName* [extends *SuperClassName*] implements *InterfaceName1*[, *InterfaceName2*, …]

{ …

Example:

```
public class SuffolkResident extends Student implements Tuition
{ …
```

81 **Declaring Constants in Interfaces (Page 1)**

Besides method references, the only other elements that may be declared in interfaces are *constants*

The constants can be accessed by *all classes* in which the interface is implemented

The constant identifier must be:

- Declared as final and may additionally be declared as static (they are static by

default)
- *Assigned a value* which may not change

82 **Declaring Constants in Interfaces (Page 2)**

Format:

[public] [static] [final] *type CONSTANT_NAME = value*;

Example:

```
public interface Tuition
{
    static final int PT_TUITION = 105;
    static final int FT_TUITION = 1175;
}
```

83 **Interface Programming Practice (Page 1)**

According to the "Java Language Specification", in standard practice within an interface:
- Methods are declared without the keywords public and abstract because these specifications are redundant
- Constants are declared without the keywords public, static and final because they also are redundant

84 **Interface Programming Practice (Page 2)**

Example:

```
public interface Tuition
{
    int getTuition();

    int PT_TUITION = 105;
    int FT_TUITION = 1175;
}
```

96 **Abstract Classes and Interfaces (Page 1)**

A Java abstract class is a class which contains one or more abstract methods which must be implemented by the subclasses
- May contain concrete methods
- Begins with the keyword "abstract" followed by the class definition
- Useful in situations when some general methods should be implemented in super class and specialization behavior should be implemented by subclasses
- Can contain public, private and protected members
- Can have instance variables (interfaces cannot)

97 **Abstract Classes and Interfaces (Page 2)**

A Java interface may contain only method declarations and constants and does not contain their implementation.
- Classes which implement the interface must provide the method definition for all the methods present
- Begins with the keyword "interface"
- Useful in a situation when all its properties need to be implemented by subclasses
- Can only have public members

- All constants in an interface are by default public static final

98 **Abstract Classes and Interfaces (Page 3)**

- An interface is also used in situations when a class needs to extend another class apart from the abstract class
  - In such situations it is not possible to have multiple inheritance of classes
  - An interface on the other hand can be used when it is required to implement one or more interfaces
  - Abstract classes do not support multiple inheritance whereas an interface supports "multiple inheritance"

99 **Abstract Classes and Interfaces (Page 4)**

- Interfaces are slow as it requires extra indirection to find corresponding methods in the actual class; abstract classes are fast
- Interfaces are often used to describe the peripheral abilities of a class, not its central identity
  - E.g. Class "Automobile" might implement the interface "Recyclable" which could apply to many otherwise totally unrelated objects

100 **Abstract Classes and Interfaces (Page 5)**

- There is no difference between a fully abstract class (all methods declared as abstract and all fields are public static final) and an interface
- Neither abstract classes nor interfaces can be instantiated

101 **Abstract Classes and Interfaces (Page 6)**

- When to use which:
  - If the various objects are all "of-a-kind" and share a common state and behavior, then tend towards a common base (abstract) class
  - If all they share is a set of method signatures, then tend towards an interface