1 **Objected-Oriented and Inheritance**

CST141

2 **OOP**

- Object-Oriented Programming is characterized by three features:
  - Encapsulation
  - Inheritance
    - New classes created from (extends) existing classes by absorbing (*inheriting*) their attributes/properties (instance variables) and behaviors (methods)
  - Polymorphism
    - The ability of an object to take on many forms; a common use occurs in OOP when a parent class reference is used to refer to a child class object

4 **Code Duplication without Inheritance**

- Classes that operate on similar entities often have many identical elements
  - Makes maintenance difficult/more work
  - Introduces danger of bugs through incorrect maintenance
- Code duplication also can carry over introducing problems to the driver classes

5 **Inheritance**

- When a new class is created, it *inherits* the instance variables and methods of any previously defined superclass
- This subclass gets its *initial* features from the direct superclass
- An indirect superclass is inherited from *two or more* levels above in a class hierarchy

6 **The Subclass**

- The subclass is usually *larger* than its superclass …
  - Because it *adds* instance variables and methods of its own to those of the superclass
  - Also it is possible to define *additions* to, or *replacements* for, inherited superclass features
- It is more *specific* than the superclass …
  - Therefore it has a smaller number of situations in which it can be used

7 **The Superclass**

- Each superclass exists at the top of a *hierarchical* relationship with its subclasses
- A superclass may have *several* direct subclasses which inherit its features
- A subclass *to one* superclass may be a superclass *to other* subclasses

10 **"Has a …" vs. "Is a …"**

- These two phrases that express the nature of *relationships* and class *attributes* between superclasses and subclasses in inheritance:
  - A class to its own attributes ("has a")
  - A subclass to the superclass from which it inherits additional attributes ("is a")

11 **"Has a …" Relationships**

- "Has a" relationship expresses the attributes (instance variables) *within* the class (called composition)
- *A class "has a(n)" attribute*, i.e.
  - HourlySalaryCheck "has an" HoursWorked, "has a" Pay Rate

- CommunityMember "has a" First Name, "has a" Last Name, "has an" Address, etc.

12 **"Is a ..." Relationships**

- "Is a" relationship expresses inheritance
- *Subclass "is a" superclass*, i.e.
  - AnnualSalaryCheck "is a" PayrollCheck
    - And has all the attributes of a PayrollCheck, i.e. if PayrollCheck "has a" Check Number attribute, AnnualSalaryCheck does also
  - Teacher "is a" Faculty
    - And has all the attributes of a Faculty member, i.e. if Faculty "has a" Rank attribute, Teacher does also

13 **Class Libraries (Page 1)**

- New classes inherit features from an organization's *own* class library
- When developing a new class:
  - First try to find a place for it in the *existing* inheritance hierarchy
  - Only if it does not fit into the current class library structure should it be the beginning of a *new* inheritance hierarchy segment

14 **Class Libraries (Page 2)**

- Java API uses inheritance to build its vast library collection of classes

23 **The Keyword extends (Page 1)**

- Declares that this class is a *direct* subclass of the superclass that is named following the keyword extends
- The class *inherits* all public and protected members (instance variables and methods) of the superclass
- A class may extend (inherit) directly only from one class (its direct superclass)

24 **The Keyword extends (Page 2)**

- Format:

  public class *SubClassName* extends *DirectSuperClassName* {
- Examples:

```
public class HourlySalaryCheck extends PayrollCheck {
public class Faculty extends Employee {
```

25 **Superclass Constructor Call (Page 1)**

- Subclass constructors always must contain a call to "super" (to its direct superclass constructor method), or ...
- If none is written, the compiler inserts one (an *implicit call* without parameters)
  - Works only if superclass has a constructor *without parameters*

26 **Superclass Constructor Call (Page 2)**

- Must be the first statement in the body of the subclass constructor
- Example:

```
public AnnualSalaryCheck(int checkNumber, int employeeID, double annualSalary)
{
    super(checkNumber, employeeID);
    setAnnualSalary(annualSalary);
}
```

27 **Calling Superclass Methods**

- The public members of a superclass are callable from the subclass
- Format:
  [super.]*superclassMethod*(parameters)
  - Keyword super is not required (and is not standard usage) unless overriding superclass methods
- Examples:
  super.toString()
  super.getEmployeeID;
  getEmployeeID;

28 **Method Overriding (Page 1)**

- To modify the implementation of an inherited method in a subclass
- Example:

```
public String toString()
{
   return super.toString()
      + "\nHours worked: " + getHoursWorked()
      + "\nPay rate: " + getPayRate()
      + "\nGross pay: " + getGrossPay();
}
```

29 **Method Overriding (Page 2)**

- Superclass method must be public (accessible)
  - A private superclass method cannot be overridden
- Methods that are static can be inherited but not overridden
  - To access a "hidden" (because a method of the same name exists in the subclass) static method in a superclass, use the class name, e.g. *SuperclassName.staticMethodName*()

30 **The @Override Annotation**

- Placing @Override before a subclass method denotes that the method *must* override the method in the superclass
- Format:
  @Overrides
  public *type subclassMethodName*()
  { …
- Example:

```
@Overrides
public String toString()
{ …
```

37 **The DecimalFormat Class (Page 1)**

- Class used to create objects used to *format numbers* for output
- Stored in the java.text package
  import java.text.DecimalFormat;
- Format:
  DecimalFormat *objectName* = new DecimalFormat("*formatString*");

- *formatString* argument is a String of characters that specify *how* numbers will be formatted

38 **The DecimalFormat Class (Page 2)**

Example 1:

```
DecimalFormat commaFormat = new DecimalFormat("#,##0");
```

The String argument "#,##0" specifies that the number will display:

- With *commas* at the thousands, millions, etc.
  - Only if number is *1000 or greater*; otherwise printing of leading zeros and commas are from the 10's position to the left are suppressed
- Rounded to the nearest *integer*

39 **The DecimalFormat Class (Page 3)**

Example 2:

```
DecimalFormat twoDecimals = new DecimalFormat("0.00");
```

The String argument "0.00" specifies that the number will display:

- *At least one* digit to the left of the decimal point
- *Exactly two digits* (rounded) to the right of the decimal point

40 **The DecimalFormat Class (Page 4)**

The functionality of Examples 1 and 2 can be combined to add *commas* to the *two decimals* rounded:

```
DecimalFormat grossPayFormat = new DecimalFormat("#,##0.00");
```

A *floating dollar sign* could be inserted prior to the rest of the format string:

```
DecimalFormat grossPayFormat = new DecimalFormat("$#,##0.00");
```

41 **The format Method**

Formats a numeric value according to the DecimalFormat object's *format string*

Takes one variable/value (either float or double) as its *single* argument

Format:

*decimalFormatObject*.format(*float*| *double*);

Example:

```
JOptionPane.showMessageDialog(null, grossPayFormat.format(grossPay));
```

52 **Extendible Classes (Page 1)**

Software is extendible when it can be easily updated and *reused* to do something that the original author never imagined

Extendibility is enhanced by:

- *Loose coupling*—few connections
- *Class cohesion* —classes with one single, well defined entity
- *Responsibility-driven design* in which classes are responsible for manipulating their own data

53 **Extendible Classes (Page 2)**

When developing a new class, look to find a place where it can extend another class in the *existing* inheritance hierarchy

Sometime superclasses in an inheritance hierarchy only serve to support subclasses

- Such superclasses are called abstract classes (never have objects instantiated from them)

73 **The Class Object (Page 1)**

- The superclass of *all classes* (either direct or indirect) is Object from the Java API …
  - If a class definition does not explicitly extend another class, it extends Object directly
- The following two class headers effectively are identical:

```
public class PayrollCheck {
public class PayrollCheck extends Object
 {
```

74 **The Class Object (Page 2)**

- As a result all classes inherit eleven (11) public methods from Object including:
  - toString(), equals() and hashCode()
- Additionally classes from the Java API use inheritance extensively and extend also from class Object (directly or indirectly)

77 **The toString() Method of Class Object**

- Method toString() is a member of class Object that returns a String representation of an object
- All classes inherit the toString() method either directly or indirectly from Object
  - May be called or overridden
- Returns the class name of which the object is an instance and a hash code (start address where the object is stored in memory in hexadecimal), e.g.
  - HourlySalaryCheck@15037e5

79 **Advantages of Inheritance (so far)**

- Avoiding code duplication
- Code reuse
- Easier maintenance
- Extendibility

80 **Subtyping (Page 1)**

- Types defined by a subclass definition actually are subtypes of their superclass
- If HourlySalaryCheck and AnnualSalaryCheck classes are extensions of class PayrollCheck:
  - The superclass object:

```
PayrollCheck pay;
```

  - Can be instantiated by calling its subtype constructor:

```
pay = new AnnualSalaryCheck();
```

  - Or in a single statement:

```
PayrollCheck pay = new AnnualSalaryCheck();
```

81 **Subtyping (Page 2)**

- Now the *subtyped object* can differentiate between getGrossPay() methods of HourlySalaryCheck and AnnualSalaryCheck classes when called:

```
JOptionPane.showMessageDialog( null, pay.getGrossPay() );
```

  - This is an example of polymorphism (meaning "many shapes" or "many forms")
  - In this case the method behavior changes based upon which constructor was used to instantiate it

84 **The ArrayList Class (Page 1)**

- Used to create a list of items in a *flexible-sized* collection
- The capacity of an ArrayList object is initialized to start at ten (10) elements but *grows* as items are added to it

85 **The ArrayList Class (Page 2)**

- The class has a whole series of methods of its own which can be used to automatically manipulate objects instantiated from it
- Found in the java.util package of the Java API library:
  import java.util.ArrayList;

86 **The ArrayList Class (Page 3)**

- Format to declare an ArrayList object:
  ArrayList<*type*> *objectName*;
- Example:
  private ArrayList<String> departmentList;
  - ArrayList is a generic class requiring a subtype specified as a parameter
  - Enclosed in <chevrons>, e.g. <angle brackets>
  - The example data field above departmentList is called an "ArrayList of Strings"

87 **Instantiating ArrayList Objects (Page 1)**

- Similar syntax to that which is used when instantiating objects …
  - Includes the second type parameter enclosed in <chevrons>
- Format:
  *objectName* = new ArrayList<*type*>();
- Example:
  departmentList = new ArrayList<String>();

88 **Instantiating ArrayList Objects (Page 2)**

- Format to *declare* and *instantiate* the object in a single statement:
  ArrayList<*type*> *objectName* = new ArrayList<*type*>();
- Example:
  ArrayList<String> departmentList = new ArrayList<String>();

89 **The add() Method for ArrayList**

- *Appends* this element (object) to the end of the ArrayList collection
- Format:
  *arrayListObject*.add(*object*);
  - The *object* represents the value added as a new element to the ArrayList collection
- Example:
  departmentList.add("COMPUTER");

90 **The size() Method for ArrayList**

- *Returns* an int which is the number of elements in the ArrayList collection
- Format:
  *arrayListObject*.size()
- Example:
  JOptionPane.showMessageDialog( null, departmentList.size() );

91 **The get() Method for ArrayList**

- *Retrieves* an individual element from the specified position in ArrayList collection
- Format:

  *arrayListObject*.get(*index*)
  - The *index* is an int between zero (0) and one less than the number of items in the ArrayList
- Example:

  JOptionPane.showMessageDialog(null, departmentList.get(index) );
  - The element is *not removed* from the collection

92 **The indexOf() Method for ArrayList (Page 1)**

- Searches for first occurrence of the object argument in the ArrayList collection—tests for an *equal to* (==) condition
- The method returns either :
  - An int which is the index representing its position in the ArrayList collection
  - Or -1 if the search criteria value is not found

93 **The indexOf() Method for ArrayList (Page 2)**

- Format:

  *arrayListObject*.indexOf(*object*)
- Example:

  index = departmentList.indexOf("COMPUTER");

94 **The remove() Method for ArrayList (Page 1)**

- *Deletes* an individual element from the specified position in ArrayList collection
- All elements after the deleted item move up one element to fill the gap

95 **The remove() Method for ArrayList (Page 2)**

- Format:

  *arrayListObject*.remove(*index*);
  - The *index* is an integer between zero (0) and one less than the number of elements in the ArrayList
- Example:

  departmentList.remove(index);