1 **Creating and Instantiating Java Classes and Objects**

CST141

2 **Java Classes**

- Java programming is *always* objected-oriented (everything is a class) …
  - Classes are the individual pieces in programs
  - Classes consist of pieces called methods
- Building blocks from which applications are developed (*reusable* software)
  - Classes and methods which you write
  - Java class libraries, and classes and methods developed by others

3 **Object-Oriented Programming**

- Classes are programmed representations of entities in the real world
- In OOP (object-oriented programming) each object has its own:
  - Attributes (the data, e.g. instance variables)—defines the state of an object
  - Behaviors (the methods)—defines the *actions* of the object

4 **OOP Programming and Thinking**

- A class is a *blueprint*:
  - Consider that a factory produces "heaters" from the "Heater" blueprint (class)
  - A button on outside of the factory has the keyword new
  - Press the button, out comes another "heater"

5 **Objects and Classes**

- Classes
  - Represent all objects of a kind, e.g. "Heater"
  - A model for creating "heater" objects
- Objects
  - Represent "things" from the real world, or from some problem domain (Example: "The heater currently set to a temperature of 150 degrees")

6 **Characteristics of Objects**

- Many instances (the objects themselves) can be created from a *single class*
- An object has attributes, values stored in its instance variables (data fields)
  - The class defines what instance variables an object has (*all objects* instantiated from the same class have exactly the *same* variables)
  - However each object stores its *own set of values* for the instance variables (called the state of the object)

7 **Classes (Page 1)**

- In Java the unit of programming is the class
  - Every Java application contains at least one programmer-defined class
- Each Java class is written and saved in a *separate* ".java" file (there are exceptions)
- Objects are instantiated from classes and work together to build an application

8 **Classes** **(Page 2)**

- By *convention* each word in a class name should begin with an uppercase letter, e.g.
  - MyHeater
- When the program file is saved, the class name must be the *same* as the filename, e.g.
  - MyHeater.java

9 **Classes** **(Page 3)**

- The layout of each class file includes:
  - The outer wrapping made up of a class header which names the class
  - The inner body of the class enclosed in left and right braces {always used in pairs} provides the class' functionality

10 **Classes** **(Page 4)**

- Format:

```
public class ClassName
{
   instanceVariables
   constructors
   methods
}
```

  - The words public and class are keywords

11 **Classes** **(Page 5)**

- Example:

```
public class Heater
{
   private int temperature;

   public Heater()
   {
      temperature = 15;
   }
   …
}
```

12 **Classes** **(Page 6)**

- Example (alternative brace placement):

```
public class Heater {

   private int temperature;

   public Heater() {
```

```
        temperature = 15;
    }
    ...
}
```

14 **Starting a New Project in BlueJ**

- A BlueJ project is a *folder* that contains all the files that make up the application
- From Project menu, select New Project…
- Enter a Folder name: (where project will be stored) and click <Create> button

16 **Creating a New Class in BlueJ**

- Click <New Class…> button
- In the "Create New Class" dialog window:
  - Enter Class Name: for new class (starts with uppercase letter)
  - Keep Class radio button checked
  - Click the <OK> button
- Double-click the class icon to reach the "Code Editor"

21 **Instance Variables (Page 1)**

- Instance variables store the values for an object (e.g. its attributes/characteristics)
- Each object has its own set of instance variables (data fields) no matter how many objects are instantiated from a class
- The specific values assigned to the instance variables define the state of an object

22 **Instance Variables (Page 2)**

- Format:

```
public class ClassName
{
    private type variableName;
    ...
```

- Example:

```
public class Heater
{
    private int temperature;
    private int min;
    private int max;
    ...
```

23 **Access Modifiers (Page 1)**

- Access modifiers control whether or not class members (the class, variables or methods) can be accessed from other classes
  - Also called visibility modifiers
- There is no restriction accessing members from inside the class

24 **Access Modifiers (Page 2)**

- The modifier public specifies the member is accessible by any other class in the same package (folder)
  - The default if no access modifier is specified
- The modifier private specifies the member is accessible only within its own class
- The modifier protected specifies the member is accessible only from one of its own subclasses (Chapter 11)

25 **Access Modifiers (Page 3)**

- Example of an instance variable being declared as private:

```
private int temperature;
```

- Example of a method being declared as public:

```
public void warmer()
{
    temperature +=5;
}
```

26 **The Constructor Method (Page 1)**

- A special method that *initializes* the instance variables within the class
- It has the *same name* as the class
- Constructor executes whenever application *instantiates* an object from the class
- Execution guarantees that instance variables always will be in a *consistent state*

27 **The Constructor Method (Page 2)**

- Format:

```
public ConstructorName( [parameter1, parameter2, ...] )
  //Name is the same as the class name
```

  - Can take parameters but *never* returns a value
  - Never specify a type, not even void
- Example 1:

```
public Heater()
{
   temperature = 15;
}
```

28 **The Constructor Method (Page 3)**

- Example 2:

```
public Heater(int initMin, int initMax)
{
   temperature = 15;
   min = initMin;
   max = initMax;
   increment = 5;
}
```

29 **Encapsulation**

- Encapsulation is achieved by making instance variables (data fields) private
  - Also called "information hiding"
- Only the class' own public methods may directly inspect or manipulate its data fields
- Protects the data and makes the class easier to maintain since the functionality is managed in just one place

30 **Using *set* and *get* Methods (Page 1)**

- Instance variables which are private may not be manipulated from other classes
- Often public methods are provided inside a class to allow private instance variables to be *updated* and/or *retrieved*

31 **Using *set* and *get* Methods (Page 2)**

- *Set* methods (also called *setter* or *mutator* methods) change (update) an instance variable value:

```
public void warmer()
{
   temperature += 5;
}
```

32 **Using *set* and *get* Methods (Page 3)**

- *Get* methods (also called *getter*, *accessor* or *query* methods) retrieve (return) a copy of the value:

```
public int getTemperature()
{
   return temperature;
}
```

-

33 **Naming *get* and *set* Methods**

- Conventionally a method that *updates* an instance variable uses the word *set* and the variable name, e.g.
  - If the instance variable is temperature, the method name would be setTemperature()
- Methods that *retrieve* a variable's value use the word *get* and the variable name, e.g.
  - If the instance variable is temperature, the method name would be getTemperature()

51 **Methods Types (Page 1)**

- Methods that *return a value* have a type other than void
  - The method's type must be the same as the type of the return value

34 **UML Diagrams (Page 1)**

- Unified Modeling Language (UML) notation is a standardized method for representing class structure

- The notation is called a UML class diagram or simply a class diagram

35 **UML Diagrams (Page 2)**

- Instance variable (data field) format:
  instanceVariable: *type*
- Instance variable example:
  temperature: int

36 **UML Diagrams (Page 3)**

- Constructor format:
  ConstructorName([*parameterName1*: *type1*, ...])
  – *ConstructorName* is the same as the class name
- Constructor examples:
  Heater()
  Heater(initMin: int, initMax: int)

37 **UML Diagrams (Page 4)**

- Method format:
  methodName([*parameterName1*: *type1*, ...]): *returnType*
- Method examples:
  warmer(): void
  getTemperature(): int
  setIncrement(newIncrement: int): void

38 **UML Diagrams (Page 5)**

- Access modifiers:
  – A plus sign (+) in front of a member means that its access is public
  – A minus sign (–) in front of a member means that its access is private
- Access modifier examples:
  – temperature: int
  + Heater()
  + warmer(): void

40 **Instantiating an Object (Page 1)**

- Instantiate means to create an object (*create an instance* from the class)
- The keyword new instantiates the object

41 **Instantiating an Object (Page 2)**

- Format:
  ClassName objectVariable = new *ConstructorName*([*parameters*]);
- Example 1:
  Heater heater1 = new Heater();
  – The *ClassName* and *ConstructorName*() are the same

42 **Instantiating an Object (Page 3)**

- Format:
  ClassName objectVariable = new *ConstructorName*([*parameters*]);
- Example 2:
  Heater heater2 = new Heater(initMin, initMax);
  – In this example parameter values are being passed to the constructor method

43 **Instantiating an Object (Page 4)**

- To instantiate a new object in BlueJ:
  – Right-click class name and the constructor statement from the shortcut menu, e.g. new Heater() or new Heater (int, int)
  – Enter a name for new object (or accept given default name) and then click <OK> button

45 **Calling Object Methods (Page 1)**

- Object methods are called by naming the method preceded by the object name using dot (.) notation
- Format:
  objectName.methodName([*parameters*]);
- Example:
  heater1.warmer();

46 **Calling Object Methods (Page 2)**

- To call a method in BlueJ, right-click the object and select method name from the shortcut menu

47 **Return Values (Page 1)**

- Information (a single value) returned by the method when it concludes executing
- The return value is the method's "result"
- The keyword return *outputs* (sends it back) the value from the called method

48 **Return Values (Page 2)**

- Format:
  return *expression*;
  – The *expression* may be a value, variable, calculation, etc.
- Examples:
  return 50;
  return temperature;
  return hoursWorked * payRate;
  return "Gross Pay: " + grossPay;

49 **Return Values (Page 3)**

- The method header indicates whether the method will return a value by specifying the *method's type*:
  – The type precedes the method name, e.g.
  public int getTemperature()

– Or the keyword void indicates that the method does not return a value, e.g.
public void warmer()

50 **Return Values in BlueJ**

- In BlueJ methods that return values are called in the same way as methods that do not
  – Right-click the object and select the method name from the shortcut menu
- The returned value is displayed in a dialog window

52 **Methods Types (Page 2)**

- Format:
  public *type methodName*([*parameterList*])
- Example:

```
public int getTemperature()
{
   …
}
```

53 **Methods Types (Page 3)**

- If a value is not returned, the type is void

```
public void warmer()
{
   …
}
```

54 **State of an Object**

- The set of all values assigned to instance variables for each individual object is called its state
- In BlueJ the state of an object is viewed in "Object Inspector" window by either:
  – Right-clicking the object and selecting the Inspect command
  – Double-clicking the object

55 **The Classic set Method (Page 1)**

- The "classic" set method *updates* the value of an instance variable:
  – Always has a *parameter*
  – Always is type void
  – Always *assigns* the parameter as the new value for the instance variable, e.g.
  temperature = newTemperature;
  – Sometimes validates the value of the parameter using an if statement

56 **The Classic set Method (Page 2)**

- Example 1:

```
public void setTemperture(int newTemperature)
{
   temperature = newTemperature;
}
```

57 **The Classic set Method (Page 3)**

- Example 2:

```
public void setTemperture(int newTemperature)
{
   if (newTemperature > 0)
   {
      temperature = newTemperature;
   }
}
```

58 **The Classic get Method (Page 1)**

- The "classic" get method *returns* the value of an instance variables:
  - Never has a *parameter*
  - Always has only a single statement which is return of an instance variable
    return temperature;
  - Always is same type as the instance variable it returns

59 **The Classic get Method (Page 2)**

- Example:

```
public int getTemperture()
{
   return temperature;
}
```

67 **Method Calls with Parameters**

- Place the value or values inside the method name's parentheses
- Format:
  objectName.methodName([*parameters*]);
- Example:
  multiplier1.setX(10);

68 **Passing Parameters to Methods (Page 1)**

- When a method needs additional values before executing, that information is passed to it in the form of parameters
- Parameters are *variables* declared in the method's header (also called the signature)
  - Reminder: parameters also may need to be passed to a constructor method when new objects are instantiated from a class

69 **Passing Parameters to Methods (Page 2)**

- Format:
  [public] void/*type methodName*(*type parameter1*, [type *parameter1*, …])
- Examples:
  Heater(int initMin, int initMax)
  public void setX(int newX)

70 **Passing Parameters to Methods (Page 3)**

- To pass parameters to a method in BlueJ:
  - Right-click the object and select the method name from the shortcut menu
  - For each parameter to be passed, in the dialog window enter value to be passed and then click the <OK> button
  - Strings must be contained in "quotation marks"

73 **The System.out Variable**

- The out *variable* (*field*) (which is member of the System class) is commonly known as the *standard output stream*
- Employs methods that display output to the command window (called the terminal window in BlueJ) , e.g.
  System.out.println(…);

74 **The println() Method (Page 1)**

- The println() method is contained within (a member of) the System.out output field
- Prints a line of text in the terminal window

75 **The println() Method (Page 2)**

- Method println() executes a *carriage return* and *line feed* after printing
  - Equivalent of the <Enter> key
- Method print() displays the text output without the carriage return/line feed
  - Does not advance to the next line

76 **The println() Method (Page 3)**

- Format:
  System.out.println(*outputString*);
- Example:
  System.out.println("No negative values");

78 **Strings**

- Characters contained in *quotation marks* and stored in memory using Unicode coding are called strings
  "Do not enter negative value"
- May be a combination of letters, numbers and other special characters
- *Blank spaces* within strings are *not* ignored
- String variables store strings of characters

79 **Concatenation**

- The concatenation operator (+) is used to join a string (or string variable) to the value of one or more variables into a *single* string
- Format:
  "*String text*" + *expression* [+ …]
- Example:
  "Pay rate: " + payRate

- If variable payRate = 20, the string will be:
  "Pay rate: 20"

80 **Local Variables (Page 1)**

- Instance variables are one sort of variable
  - Store values through the life of an object
  - Accessible throughout the class (e.g. are global to the entire class)
- Methods can include shorter-lived variables referred to as local variables
  - They are accessible only from within the body of the method
  - They exist (persist) only as long as the method is being executed

81 **Local Variables (Page 2)**

- Example:

```
public void calculateGrossPay()
{
   double regularPay, overtimePay;

   if (hoursWorked > 40)
   {
      regularPay = …
   }
   else
   {
      …
   }
}
```

82 **Local Variables (Page 3)**

- In addition parameter variables also are local variables since they are accessible only from within the body of the method:

```
public void setX(int newX)
{
   x = newX;
}
```

98 **The Java API (Page 1)**

- The Java Programming Language API (Application Programming Interface) is a "rich" set of classes
  - Contains thousands of classes with tens of thousands of methods
  - Used by Java developers to make programming much easier—they do not have to understand the implementation (coding)
  - API elements are used simply by understanding the interface (documentation)

99 **The Java API (Page 2)**

- Part of the JDK (Java Development Kit) that is installed along with the compiler
  - Also called the "Java Class Library"
- The competent Java programmer must be able to work with the libraries:
  - Know the most important classes by name
  - Be able to find out about other classes

100 **The Java API (Page 3)**

- Not necessary to view the code for library methods or see how they are implemented
- You just need to know the class name, understand its methods and what they do, as well as their parameters and return types
- Information is available by reading on-line documentation for each class on the Internet

101 **Package Names**

- Java classes located in the API are organized into related groups called packages
- Each piece of a package name is actually a *folder* (directory) where class is located, e.g.
- For example the "Date" class is located in the java.util package (folder)
  - So its partial path is: ../java_api/java/util/Date.class

103 **The import Statement (Page 1)**

- Classes from the Java API must be imported using an import statement …
  - Except classes from the java.lang package which are *fundamental* to the development of Java programs (e.g. System, String, Math, etc.)
- Then they can be used like other classes from the current project
- The import statement(s) should be the first statement(s) in the class file

104 **The import Statement (Page 2)**

- Format:
  import *packageName.ClassName*;
- Example:
  import java.util.Date;
  - import statements come *before* class header
- Example to import an *entire package* (that is all the classes in the folder):
  import java.util.*;

105 **Bypassing the import Statement**

- Class names can be reference directly (skipping the import statement) in Java statements by fully qualifying the name
  - *Prefixing* the package name to the class name
- Format:
  packageName.ClassName
- Example:
  java.util.Date time = new java.util.Date();

106 **The Date Class (Page 1)**

- Instantiates objects that represent current (or specific) date and time
- Measured in number of *milliseconds* since 12:00 midnight January 1, 1970 GMT (Greenwich Mean Time)
- Found in the java.util package (must be imported prior to usage):
  import java.util.Date;

107 **The Date Class (Page 2)**

- The no-argument constructor for class Date instantiates in object that stores the date and time it was created:
  Date *dateObject* = new Date();
- Example:
  Date payrollDate = new Date();

108 **The Date Class (Page 3)**

- Alternate constructor takes a long integer that represents the number of milliseconds since 12:00 midnight January 1, 1970 GMT:
  Date *dateObject* = new Date(*milliseconds*);
- Example:
  Date payrollDate = new Date(200000);
  – The payrollDate will be Wed, Dec 31, 1969, 19:03:20 EST (Thur, Jan 1, 1970, 12:03:20 GMT) (adjusted to operating system time zone)

109 **The setTime() Method of Class Date**

- The type void setTime() method sets a new number of elapsed milliseconds since 12:00 midnight January 1, 1970 GMT
- Format:
  dateObject.setTime(*milliseconds*);
- Example:
  payrollDate.setTime(200000);

110 **The getTime() Method of Class Date**

- Returns as type long number of milliseconds since 12:00 midnight January 1, 1970 GMT for the object's current date and time
- Format:
  dateObject.getTime()
- Example:
  long millisecs = payrollDate.getTime();

111 **The toString() Method of Class Date**

- Returns a type String representation of the object's date and time
- Adjusted to operating system's time zone
  – E.g. "Tue Aug 27 10:13:32 EDT 2013"
- Format:
  dateObject.toString()
- Example:

```
System.out.println( payrollDate.toString() );
```

113 **Static Members (Page 1)**

- Class components (variables and methods) available to *every* object derived from class
- Static member declaration includes the keyword static, e.g.
- Format for static method declaration:

  public static *type methodName*([*parameterList*])
  { …
- Example:

  public static double getPayRate()
  { …

114 **Static Members (Page 2)**

- *Only one* instance of the member exists which is *shared* by all objects
- A static method cannot access class instance members (methods and variables)
- May be called:
  - Using the *class* name (the norm):

    Payee.getPayRate();
  - Or an *object* name

    pay1.getPayRate();

115 **Static Members (Page 3)**

- Static class members exist as soon as the class is loaded into memory …
  - Even *before* objects of the class have been instantiated
  - In such a case, they must be *referenced by their class name*, not an object name (because no object yet exists)
- Underline static members in UML diagram

116 **Static Class Variables (Page 1)**

- Review: *Instance variables* hide their values from other objects, even if objects are instantiated from the same class
- A static class variable shares the same data (one RAM location) with all objects of the same class (class scope)
- May also be called simply called static variables or class variables

117 **Static Class Variables (Page 2)**

- For example:
  - Two objects instantiated from the Payee class can share a single static value for "payRate"
    - Although each has its own "hoursWorked"
  - Objects instantiated from a SavingsAccount class can share a single value for "interestRate"
    - Although each has its own "savingsBalance" value

118 **Declaring Static Class Variables**

- Static variables are declared by including the keyword static in the declaration
- Format:
  private static *type variableName*;
- Example:
  private static double payRate;
- Static variables should be *set* (updated in) or *get* (retrieved from) in static methods

119 **Data Fields**

- Both instance variables and static class variables are data fields:
  – Instance variables—each object has its own set of "hidden" instance values no matter how many objects are instantiated from the class
  – Static class variables—share the same data with all objects of instantiated from the class

122 **The main() Method (Page 1)**

- Every Java application must have a single method named main()
  – There can be *only one* instance of main() in the entire application
- The method may be placed in any class file, but usually is found in the application's *controlling* class

123 **The main() Method (Page 2)**

- Whenever an application executes, the Java *runtime* (JVM—Java Virtual Machine) finds main() and executes it first
- Normal execution of an application *starts* and *ends* with the main() method
  – The main() method then directly or indirectly calls all the other methods within all classes in the application

124 **The main() Method (Page 3)**

- The method always has the same header:
  public static void main(String[] args)
  { …
  – Not necessary to completely understand it at this time although …
    • It has public access (although it never is called from another method)
    • It is static and therefore never instantiated
    • It is void and never returns a value
    • The parameter args is a String[] array

126 **Constants (Page 1)**

- A constant is a programmer-named identifier whose value *cannot change* as a result of some program action
- Indicated by using the keyword final

127 **Constants (Page 2)**

- Usually is given an initial (and final) value in the declaration statement
- Format:

[public/private] final [static] *type constantName* [= *value*];

- Example:

  private final double PAY_RATE = 10;

  – *Java standard*—all constants are in all caps with underscore between words (also in UML)

128 **Constants (Page 3)**

- A constant that is an class data field may be declared as final but initialized later in one of the *constructors*
  – It may not be initialized later in any other method within the class

129 **Constants (Page 4)**

- Example:

```
public class Payee
{
    …
    private final double PAY_RATE;

    public Payee()
    {
        …
        PAY_RATE = 10;
    }
```

132 **Immutable Objects**

- An object whose contents may not be changed once it is instantiated and initialized in the constructor
  – All data fields are private
  – There are no set (mutator) for any data field
  – No get (accessor) can return a reference to a data field that is mutable

134 **Scope (Page 1)**

- The scope of a local variable is the block in which it is declared:
  – Instance variables and static class variables have class scope even though declared with the access modifier private
    • Accessible from any method in the class
  – Local variables (including parameters) have method scope
    • Accessible only in the method in which declared

135 **Scope (Page 2)**

- Scope may limited further by subordinate blocks with methods, e.g.
  – An if or loop (for or while) block
  – Any set of braces within the method

136 **Life Time**

- The lifetime (or simply life) of a variable is the time of execution (the time is exists) within the block in which it is declared:
  - For a static class variable as soon as the class is loaded into memory
  - For an instance variable as soon as the object is instantiated and as long as it is in memory
  - Within a method only as long as the method still is executing
  - Or within lesser blocks as well

137 **The this Reference (Page 1)**

- Every object has a reference to itself in the keyword this
- Reference to instance members (variables and methods) with the prefix this, e.g.
  this.hoursWorked
  this.getHoursWorked()
  – Refers to the private instance variable of the object …
  – Not the local variable within the method

138 **The this Reference (Page 2)**

- For example (although the convention in this case is to *not* use the this reference):

```
public void printPayee()
{
   System.out.println("Hours worked: " + this.hoursWorked );
   System.out.println("Pay rate: " + this.payRate );
   System.out.println("Gross pay: " + this.getGrossPay() );
}
```

139 **The this Reference and Parameters (Page 1)**

- It is common to define *parameter* variable names that are the same as *instance variable* names in the set methods
- In this case the local parameter name takes precedence over the instance variable name
  – The instance variable becomes *hidden*
- Prefix this reference to the instance variable name to access to it

140 **The this Reference and Parameters (Page 2)**

- For example, in the following set method:

```
public class Payee
{
   private double hoursWorked;
   private double payRate;

   …

   public setPayRate(double payRate)
   {
```

```
        this.payRate = payRate;
    }
```

142 **Class Variables and Parameters (Page 1)**

- If a *parameter* variable name is the same as a static *class variable* name in a set method or constructor …
- Prefix the class name to the static variable name to access to it, e.g. *ClassName.staticVariable*

143 **Class Variables and Parameters (Page 2)**

- For example if payRate is static:

```
public class Payee
{
    private double hoursWorked;
    private static double payRate;

    …

    public static payRate(double payRate)
    {
      Payee.payRate = payRate;
    }
```

144 **Using this to Invoke Constructor (Page 1)**

- Use of this by itself in one constructor refers to another constructor in the class
  – A good habit when there is more than one constructor in the class
- Must be the first statement in a constructor before any other statement

145 **Using this to Invoke Constructor (Page 2)**

- Example:

```
public class Payee
{
    private double hoursWorked;
    private double payRate;
    public Payee()
    {
        this(0, 0);
    }
     public Payee(double hoursWorked, double payRate)
    {
        setHoursWorked(hoursWorked);
        setPayRate(payRate);
    }
```

146 **Responsibility Driven Design (Page 1)**

- "Responsibility drive design" is the concept that each object is responsible for its own data (instance variable values)
- All manipulation of instance variables is handing exclusively in *set* and *get* methods

147 **Responsibility Driven Design (Page 2)**

- To enhance this design, the object-oriented standard in Java is to have the constructor *not directly* update instance variables
  - Instead constructors initialize instance variables by calling and passing their parameters to the *set* methods

148 **Responsibility Driven Design (Page 3)**

- Example:

```
public Payee(double hoursWorked, double payRate)
{
   setHoursWorked(hoursWorked);
   setPayRate(payRate);
}

public setHoursWorked(double hoursWorked)
{
   this.hoursWorked = hoursWorked;
}

public setPayRate(double payRate)
{
   this.payRate = payrate;
}
```