*David M. Keil, Framingham State University*

CSCI 252 Computer Science II Using Java

# 3. Java class design

1. Data abstraction and Java classes
2. Encapsulation
3. Code reuse and class debugging
4. Exception handling



## Inquiry

- How may we represent *things, people*, *places*, and *events*?
- What Java classes do we know about?
- How does a software developer step back from the details of data items?
- What is *abstraction*?

## Topic objective

Define and test Java classes, explaining object-oriented design concepts.



## 1. Data abstraction and Java classes

- What's an *object*?
- How are related data items associated in Java?
- What Java classes have we seen?
- Is database technology relevant to object-oriented programming?

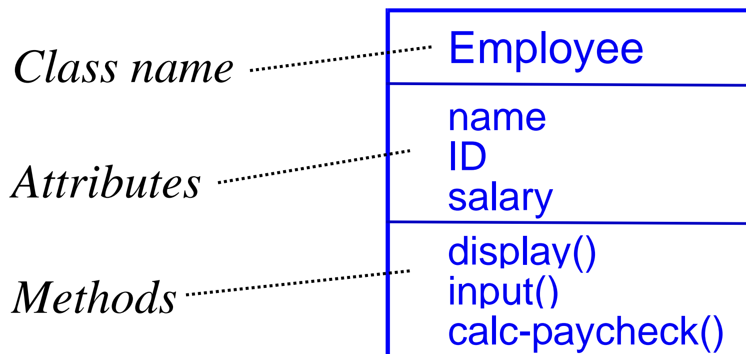
## Subtopic objectives

3.1a Describe Java data abstraction**

3.1b Define a Java class**



## UML class diagrams




- Classes model *state* and *behavior*

## Why program with classes?

- We may model the *state* and *behavior* of what we want to represent: e.g., persons, events, collections, displayed objects
- Classes let us model the *interactions* found in the environments we work with
- Class libraries may be *reused* conveniently
- Bookseller examples: *books, customers, transactions* (*challenge*: define classes for them)



## Data abstraction

- *Defn:* the creation of named data types
- Some data types, whose instances have components, are called *compound*
- *Objects* are defined by their *attributes* and *operations*
- Objects are *instances of classes*
- Encapsulation separates classes' *interfaces* from their *implementation*

## Defining data types

- Enumerated types are a simple example
- Keyword *enum* enables definition of a named integer type whose possible values are specified by constant identifiers
- *Example:*

```
public enum Seasons
   {Spring,Summer,Fall,Winter};
Seasons this_season =
   Seasons.Winter;
```



## A class associates attributes

- *Class design guidelines:*
  - Use data members that are *attributes* of an object whose values may persist
  - Use *local variables* for values that area only of use during execution of a method
- *Example:* a value like *paycheckAmount* that is computed from *wageRate* and *hours*, should be a local variable

## Object-oriented design

- Any concept is a candidate for a class: persons, things, places, transactions
- Relationships among classes include
  - *containment* (an address object is part of a customer object)
  - *inheritance* (scrollers and dialogs are two kinds of views)
- A class implements an abstraction; it may be instantiated by one or more objects



## Objects and classes

An *object* is a compound data item whose attributes (data members or instance fields) may be of types chosen by the programmer

- Example: *Class (data type)*

```
public class Location
{ public int x, y; };
```

- Usage: *Reference to instance of class*

```
Location loc = new Location;
loc.x = 5;
loc.y = 10;
```

*loc* x 5 y 10

## Classes

- Programmer may define a class name and use it to declare instances (objects)
- Member data items (also called "instance variables") exist in memory only when we declare instances of a class
- We may use object name, dot, and member name to refer to a *public* class member:

```
out.print(loc.x);
```



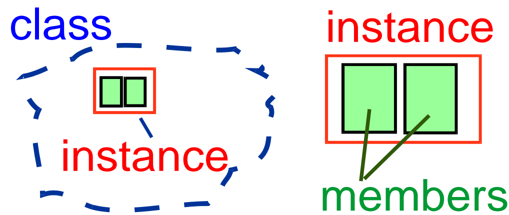
## Classes associate values

- An object implements the database concept of *record* or *tuple*
- *Example:*

```
public enum Team { RedSox,Yankees };
public class Game
{
   public Team home_team, visitor_team;
   public int home_score, visitor_score;
};
```

- *Game* associates teams with each other and with integers (scores)
- Instance: `Game g = new Game();`

## Classes, instances, members




- A *class* has 0 or more instances and occupies no memory
- A *data member* (aka "instance variable", "field") is part of an *instance*; occupies space in memory



## Using a class without methods

```
public class Employee {
  public String name;
  public int hours;
};

public static void main(String[] args)
{
  Employee emp = new Employee();
  emp.name = "Dale";
  emp.hours = 35;
  out.print(emp.name + " worked "
      + emp.hours + " hours.");
}
```

*Output:* `Dale worked 35 hours.`

## Creating instances of a class

```
public class Item
{
   String id;
   double price;
};

public static void main(String[] args)
{
   Item item1 = new Item(),
        item2 = new Item();
}
```



- *item1*, *item2* are *references* to (addresses of) 2 objects



## Classes normally have methods

*Class:* a compound type defined by data attributes and operations

```
public class Employee
{
 String name;
 int hours;
 public void display()
 { out.print(name + " " + hours); }
};
```

*Object:* an instance of a class

```
Employee emp1 = new Employee(),
         emp2 = new Employee();
```

## A class with methods

```
public class Employee
{
  String name;
  int hours;
  void input() {
    out.print("Enter name, hrs: ");
    name = in.next(); hours = in.nextInt();
  }
  void display()
  { out.print(name + " worked " + hours + " hrs"); };
}
public static void main(String[] args)
{
  Employee emp = new Employee();
  emp.input();
  emp.display();
}
```

*Class*

*Methods*

*Instance of class*



## Methods and objects

- A call to a method names an instance of the class, using a dot

```
Employee emp = new Employee();
emp.display();
```

- Methods of a class have access to the members of that class's instances

```
public void display()
{  out.print(name + hours); }
```

- A method that returns an object name returns a *reference* to the object, not a *copy*

## Shadowing

- A parameter or local variable *shadows* a data member if it has the same name
- *Example:* Suppose *x* is a member of the class whose method is *setX:*

```
void setX(int x)
{
   this.x = x;
}
```

- Then parameter *x* shadows data member *x*

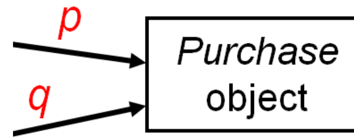


## Degenerate classes

- A *degenerate class* is one that has no instances
- *Example:* the class whose name is the same as the argument to *java* when the class file is run at the command line
- The method *main* is a member of a degenerate class

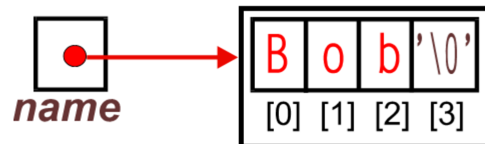
## Objects vs. object references

- When a variable is declared with a class as its type, it refers to an *object* or is a *reference*
- If *Purchase* is a class, then
  `Purchase p = new Purchase();`
  declares an *object*
- If *p* is a *Purchase, then* `Purchase q = p;` declares a *reference* to *p*
- References store *addresses* of objects, not *copies*






## References are addresses




- A *String* instance's name is a *reference* (name of an address):
  `String name = new String("Bob");`
- More than one *reference* may refer to the same *address:*
  `String p = name;`

# 2. Encapsulation

- How transparent are objects?
- What is *hidden* in a class? Why?
- What is the process of creating objects in Java?
- What is a reference?



## Subtopic objectives

3.2a Describe Java encapsulation*

3.2b Write and document a class with encapsulation*

3.2c Describe class debugging concepts

3.3d Test and debug a class**†

## Interface and implementation

- *Interface:* public method declarations, accessible from client code
- *Implementation:* private members and definitions of member methods
- A class's user needs to know only its *interface*
- A programmer writing or maintaining a class must understand its implementation
- Access specifiers: *public*, *private*



## Encapsulation hides data

```
public class Employee
{
  public void input();
  public void display();
  private String name;
  private int hours;
};
```

hidden

```
public static void
 main()
 {
   Employee emp =
    new Employee();
   emp.hours = 40;
   emp.input();
 }
```

valid  invalid

*Public* members are accessible from outside a class, *private members* are hidden

## Interface: what the client code sees

- *Methods* comprise a class's interface
- *Example:* Java documentation tells the *methods* of the *System, Scanner*, and *Math* classes, but not the *data members*
- A programmer who uses a class is called the *client*
- Client code may change even if data members (not in interface) change



## Cohesion and coupling of classes

- A guideline of software development practice is to maintain strong cohesion in a single class and weak coupling among different classes
- *Cohesion*: All attributes and methods are closely related to the concept implemented by the class
- *Coupling:* Dependencies among different classes. A class depends on another if it uses instances of the other class. Two valid dependencies are *containment* and *inheritance*

## Accessors and mutators

- An *accessor* (getter) is a method that does not modify data members
- Accessors are used to provide member data values, or values computed from them, to calling statements
- A *mutator* (setter) is a method that may modify member data
- In the *employees* class, *get_hours* is an accessor; *set_name* is a mutator



## Constructors initialize members

```
public class Employee
{
  public Employee()
   { name = new String();
     hours = 0; }
  public Employee(String nm,int hrs);
   { name = new String(nm);
     hours = hrs; }
  public void input();
  public void display();
  private String name;
  private int hours;
};
```

default constructor

constructor with parameters

## Constructors

- Take name of class; initialize data members
- Are called with *new* when instance is declared
- Have no return value or type
- May take parameters
- May be *overloaded*; i.e., there may be one constructor for each set of parameters the programmer desires to be able to initialize instances with



## Default constructors

- A default constructor assigns *zero values* (0, 0.0, '\0', "", etc.) to fields of a new object
- A class with no declared constructor automatically gets a default constructor
- Whereas methods are define with type (e.g., void if necessary), constructors have no type
- An attempt to define a constructor as *void* and to use it produces a puzzling compiler error
- Use this to call the default constructor from another constructor

# Methods that return objects

- Methods that construct objects may return *references* to objects that will disappear when the method terminates
- To return a usable object, use *clone*
- *Example*, to return an *Address:*

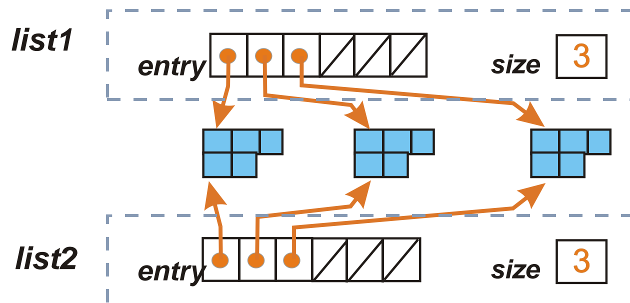
```
return (Address)address.clone();
```



# Shallow copying

**(of an object containing an array of pointers)**

```
public static void main()
{
  Phonebook list1,list2;
  list1.input();
  list2 = list1;
}
```



## Inner classes

- A class defined inside a method definition, within another class, is called an *inner class*
- It is only accessible inside that "outer" class or (if the inner class is declared within a method) a method
- Methods of an inner class may access variables of the surrounding class if they are declared *final*



## 3. Code reuse and class debugging

- Are class names used only to declare instances?

## Subtopic objectives

3.3 Locate a fault in a multi-method class†



## Code reuse

- Essential design consideration for large programming projects
- *Definition*: Code reuse is writing code so as to be useful in more than one context
- *Examples:*
  - Writing methods that solve a broad variety of instances of a common problem
  - Writing classes that are usable by a wide variety of client code

## The *Object* class

- All classes *inherit* from the *Object* class the methods *clone, equals, toString* (see topic 5)
- These methods are to be *redefined* by client code to override default methods
- Parameters and return types may be *Object*
- Thus *print* accepts *Object* parameters and as a result displays the value returned by *toString*

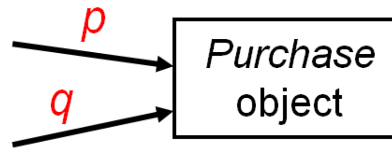


## Comparisons using *equals* method

- Default value of *equals*: comparison of addresses; not useful
- Class implementations should write *equals* with *object* parameter; cast that parameter to the class, then make comparisons of field values
- May use *instanceof* operator, with object as left parameter and class name as right parameter; it returns *false* if object is null

## Comparing objects

- == returns *true* iff the two operands refer to the same object – not necessarily whenever the two objects have the same attribute values
- Strings should be compared using *equals()* or *compareTo()* methods of the *String* class, not ==, >, <
- Correct example, given objects *x, y*:
  `if (x.equals(y))` ...






## The *toString* method

- *Recommendation:* write a *toString* method for all classes
- Default value returned by *toString*: internal numeric information that should be overridden by class designer
- If a *toString* method exists for a class, then for object *p*, *println(p)* will display the object by implicitly calling *p.toString*

## Class invariants

- *Definition:* an assertion about the state of any instance that holds during the existence of the instance
- Example: *a* minutes field always have a value in (0, 60]
- Invariants may only be enforced using encapsulation, because enabling client code to change field values would expose them to invariant violations



## Implicit and explicit parameters

- Every method call has an implicit parameter: the object that calls the method, e.g., *System.out* in **`System.out.println("Hello");`**
- Explicit parameters are those listed in parentheses
- The identifier *this* in a method definition is a reference to the implicit-parameter object

## Class relationship: dependence

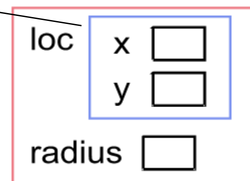
- A class *C* is *dependent* on another *D* if *C* has a method that declares an instance of *D* as a local variable



## Nested objects: containment

```
class Location { int x,y; };
class Circle
{
  Location loc;
  int radius;
};
```

*Nested object*




- A class that has a member that is an instance of another class is a *container*

## Class debugging and testing

- A well-designed class
  - uses data hiding
  - has methods that validate parameters
  - enforces all class invariants
  - handles all exceptions
- To test a class, write a driver program that declares an instance and calls the class's methods



# 4. Exception handling

- How does a well-written program handle runtime errors, such as
  - divide by zero,
  - array boundary violations, and
  - file read errors?

# Subtopic objectives

3.4a Explain exception handling*

3.4b Use exceptions†



# File input/output

- Input uses *FileReader* and *Scanner* classes:
```
FileReader fr = new FileReader("x.txt");
Scanner in = new Scanner(fr);
```
- Output uses *PrintWriter:*
```
PrintWriter out = new PrintWriter("y.txt");
```
- If input or output file cannot be opened, a *FileNotFoundException* object will be thrown
- Methods that open files may be declared with *throws FileNotFoundException* after method header -- this will terminate method
- *JFileChooser* dialog enables user to navigate directory to choose file

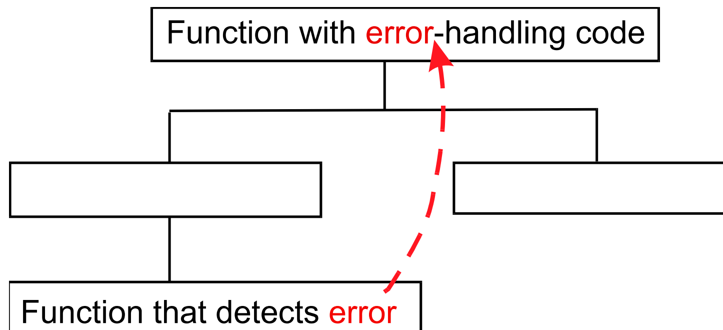
## File exceptions

- Methods that open files should have **`throws FileNotFoundException`** at end of header; called method should handle exception
- *Example:*

```
try {
   FileReader fr = new FileReader("x.txt")
   Scanner in = new Scanner(fr);
   return Integer.ParseInt(in.next());
   }
catch(IOException exc) { … }
catch(NumberFormatException exc) { … }
catch(FileNotFoundException exc) { … }
catch(NoSuchElementException exc) { … }
```



## Why exception handling?




- With Java exceptions, error information can travel directly, across method boundaries, to where it is handled

## Exception handling

- *Purpose:* to communicate information about error situations to code that handles error
- *Examples:* Array, file-not-found errors
- The calling method, not the method that detects exception, should handle it
- A *catch* response to an exception may be to throw another to its method's caller method
- With throw of exception, method terminates and control proceeds where exception is caught



## Java *try, throw, catch*

- *try:* marks block where exception might occur
- *throw:* on error, passes control directly to *catch*, throws a data item
- *catch:* marks block where a certain kind of exception is handled; catches an item of a particular type
- Program may define an exception class for objects that are to be thrown and caught
- Uncaught exceptions terminate program

## Throwing and catching exceptions

[div2.cpp]

Java?

```
Public class math_errors
{
  char message[80];
  math_errors(String msg) {message = msg; };
};

void main()
{
  out.print("Enter two integers: ";
  int a = in.nextInt(), b = in.nextInt();
  try { out.println(quotient(a,b)); }
  catch (math_errors error)
  { out.println(error.get_message() + " undefined."; }
}

float quotient(int a,int b)
{
  if (b == 0) throw math_errors("Div by 0");
  return (float)a / b;
}
```

*Sample I/O:*

```
Enter two integers: 2 0
a / b = Div by 0 undefined
```



## Guidelines for using exceptions

- Throw object; catch by class
- Exceptions may be re-thrown
- Relinquish all resources through destructors; *throw* causes destructor call
- If you can resolve a problem in current scope, do so rather than throw exception
- Every non-runtime (checked) exception thrown is caught and handled

## The *finally* clause

- Use *finally* clause after *try* block to spefcify code that should execute whether an exception is thrown or not
- *Example:*

```
try {
}
finally {
  in.close();
}
```



## Conventions with exceptions

- A method throws an exception if a precondition is violated
- Commonly used exception types:
  *NullPointerException*
  *ArrayIndexOutOfBoundsException*
  *IllegalStateException*
  *IllegalArgumentException*
  *NoSuchElementException*

## References

Cay Horstmann. *Big Java,* 3rd ed. Wiley, 2008, Ch. 3.

D. Keil. Defining and using methods. Classroom handout.

D. Keil. Defining a class. Classroom handout.

S. Reges and M. Stepp. *Building Java Programs*. Pearson, 2014.