INTRODUCTION

Definition of a Data Structure

- A *data structure* is an arrangement of data in a computer's memory (or disk).
- Questions :
  - Give some examples of data structures you already know about from your Java course?
  - How can the arrangement of data in memory affect performance?

Definition of an Algorithm

- An *algorithm* provides a set of instructions for manipulating data in structures.
- Questions :
  - What's an example of an algorithm?
  - How can the design of an algorithm affect performance? How can it affect memory?

Data Structure or Algorithm?

- Linked List
- Sort
- Search
- Stack
- Vector

Data Structure Trade-offs

- A structure we have dealt with before: arrays
- Requirement that is enforced:
  - Arrays store data sequentially in memory.
- Advantages (i.e., when is an array efficient?)
  - Accessing by index is very fast – if you know the index of the element that you want to access
- Disadvantages
  - All elements of same type
  - Fixed size – maybe too small or too big
  - Slow insertion and deletion

Overall Costs for Structures We'll Study

Structure	Access	Search	Insert	Delete	Implement	Memory
Array	Low	High	Med	High	Low	Low
Ord. Array	Low	Med	High	High	Med	Low
Linked List	High	High	Low	Low	Med	Med
Stack	Med	High	Med	Med	Med	Med
Queue	Med	High	Med	Med	Med	Med
Bin. Tree	Med	Low	Low	Low	High	High
R-B Tree	Med	Low	Low	Low	Very High	High
234 Tree	Med	Low	Low	Low	Very High	High
Hash Table	Med	Med	Low	High	Low	High
Heap	Med	Med	Low	Low	High	High
Graph	High	High	Med	Med	Med	Med

Point that you should be getting: No 'universal' data structure!!!

Algorithms We'll Study

- Insertion/Searching/Deletion
- Iteration. Java contains data types called iterators which accomplish this.
- Sorting.
- Recursion.
  - What's the key attribute of a recursive function in Java?
  - This technique will be used to develop some of the afore mentioned algorithms.

Java.util Package

- Includes Vector, Stack, Dictionary, and Hashtable. We won't cover these particular implementations but know they are there and accessible through:
  - **import java.util.*;**
  - You may not use these in homeworks unless I explicitly say you can.
- Several other third-party libraries are available

Review of Object-Oriented Programming

- Procedural Programming Languages
  - Examples: C, Pascal, early BASIC
  - What is the main unit of abstraction?
  - Procedural languages weren't good enough in all cases because of a Poor Real World Modeling.
- Object-Oriented Languages:
  - Examples: C++, Ada, Java
  - What is the main unit of abstraction?

Idea of Objects

- A programming unit which has associated:
  - Variables (data), and
  - Methods (functions to manipulate this data).
  - This allows for better Real World Modeling

Idea of Classes (Java, C++)

- With a class we specify a blueprint for one or more objects. For example, in Java:

```
class Thermostat{
   private float currentTemp;
   private float desiredTemp;
   public void furnace_on()
       {
       // method body goes here
       }
   public void furnace_off()
       {
       // method body goes here
       }
} // end class thermostat
```

Instances of a Class

- Java creates objects using the new keyword, and stores a reference to the created instance in a variable:

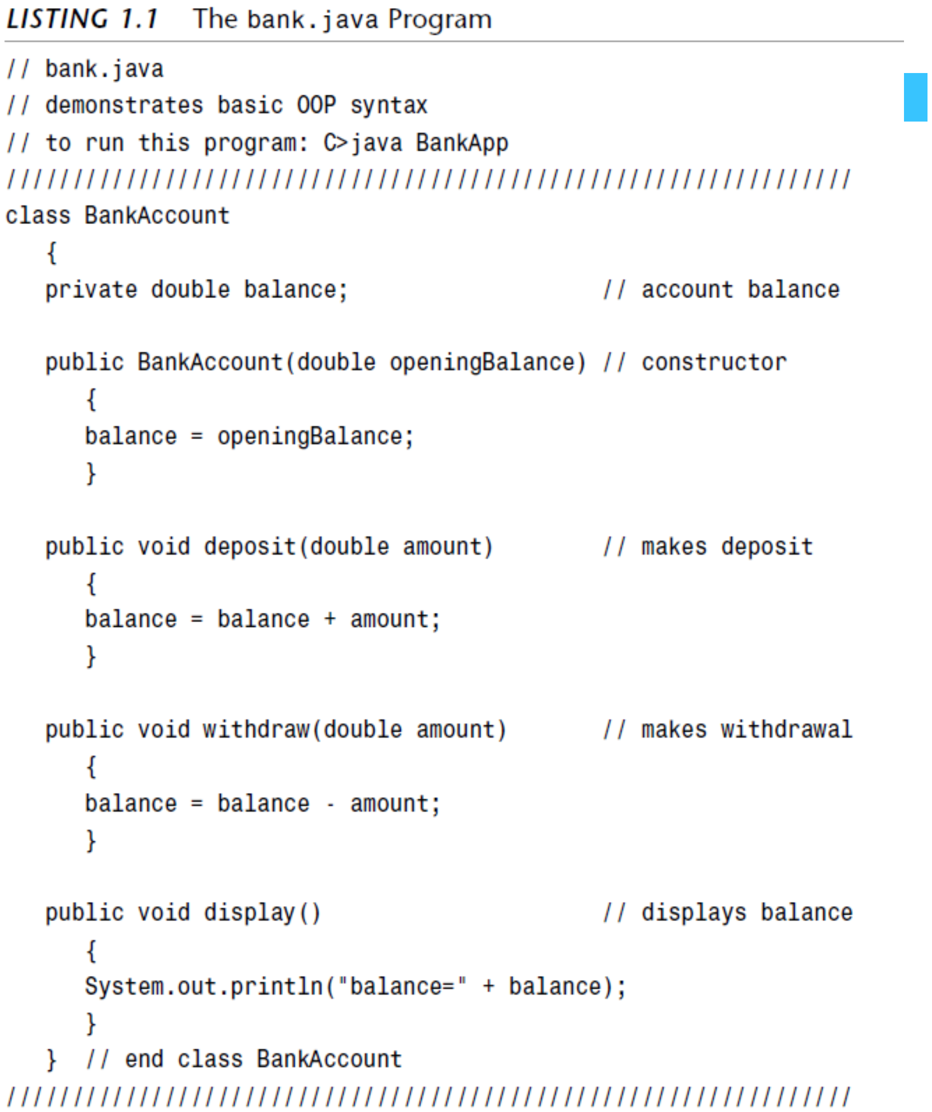
```
Thermostat therm1;    // variable therm1 declared
therm1 = new Thermostat(); // therm1 crated
Thermostat therm2 = new Thermostat(); //therm2 declared
  and created
```

- Object creation (or *instantiation*) is done through the constructor.
- Which constructor did we use here?

Invoking Methods of an Object

- Parts of the program external to the class can access its methods (unless they are not declared public)
- Dot operator:
  - `therm2.furnace_on();`
- Can I access data members similarly?
  - `therm2.currentTemp = 77;`
- What would I need to change to do so?
  - Is this change good programming practice?
  - How, ideally, should data members be accessed?

Another Example



LISTING 1.1 The `bank.java` Program

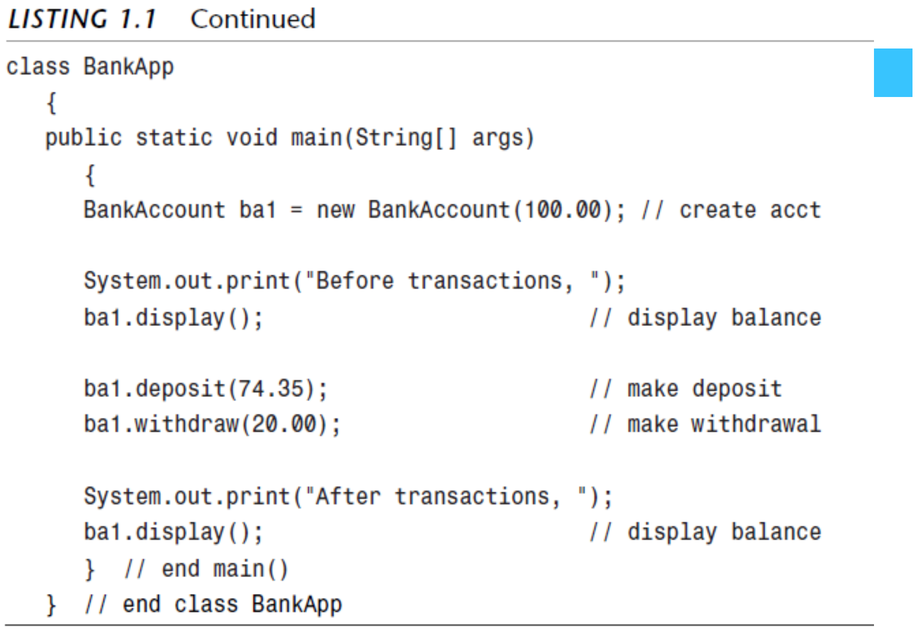
```
// bank.java
// demonstrates basic OOP syntax
// to run this program: C>java BankApp
////////////////////////////////////////////////////////////////
class BankAccount
   {
   private double balance;                   // account balance

   public BankAccount(double openingBalance) // constructor
      {
      balance = openingBalance;
      }

   public void deposit(double amount)        // makes deposit
      {
      balance = balance + amount;
      }

   public void withdraw(double amount)       // makes withdrawal
      {
      balance = balance - amount;
      }

   public void display()                     // displays balance
      {
      System.out.println("balance=" + balance);
      }
   }  // end class BankAccount
////////////////////////////////////////////////////////////////
```



LISTING 1.1 Continued

```
class BankApp
   {
   public static void main(String[] args)
      {
      BankAccount ba1 = new BankAccount(100.00); // create acct

      System.out.print("Before transactions, ");
      ba1.display();                         // display balance

      ba1.deposit(74.35);                    // make deposit
      ba1.withdraw(20.00);                   // make withdrawal

      System.out.print("After transactions, ");
      ba1.display();                         // display balance
      }  // end main()
   }  // end class BankApp
```

- Here's the output from this program:
  - Before transactions, balance=100
  - After transactions, balance=154.35
- Look at the output.
- Let's go over why this is generated.

Inheritance

- Creation of one class, called the base class
- Creation of another class, called the derived class
  - Has all features of the base, plus some additional features.
- Example:
  - A base class **Animal** may have associated methods **eat()** and **run()** and variable **name**, and a derived class **Dog** can inherit from **Animal**, gaining these methods plus a new method **bark()**.
  - If **name** is private, does **Dog** have the attribute?
  - How do we enforce **Dog** to also have attribute **name**?

Polymorphism

- Idea: Treat objects of different classes in the same way.
- What's the requirement?
  - Same way of calling different methods for different classes. These classes should be derived from the same superclass.

- Let's return to an example with **Animal** and **Dog**, and throw in another derived class **Cat**.

Review of some Java Concepts

- Difference between a value and a reference:

  ```
  int intVar;
  BankAccount bc1;
  ```

- Which is a value and which is a reference?
  - `intVar` is a value and `bc1` is a reference
- How did the interpreter know to allocate them differently?

- What does the address stored in `bc1` contain right now?
- What must I do before I use `bc1`?

Java Assignments

- □ What must be noted about the following code snippet:

```java
int intVar1 = 45;
int intVar2 = 90;

BankAccount bc1= new BankAccount(72.45);
BankAccount bc2 = bc1;

bc2.withdraw(30.00);   // Does this modify object bc1?

intVar1 = intVar2;
intVar1 = 33;  // Does this modify intVar2?
```

Java Garbage Collection

- When is the memory allocated to an object reclaimed in Java?
  - When the object has no reference to it (garbage collection)

- Code like this would leak memory in C++, but does not in Java because of the garbage collector:

```
while (true) {
     Integer tmp = new Integer();
          …
}
```

Passing by Value vs. Reference

- □ Same idea:

```
void method1() {
   float num = 4;
   BankAccount ba1 = new BankAccount(350.00);
   method2(num);
   method3(ba1);
}
void method2(float f) { … }
void method3(BankAccount acct) { … }
```

- □ If I change **f** in **method2**, does that affect **num**?
- □ If I change **acct** in **method3**, does that affect object **ba1**?

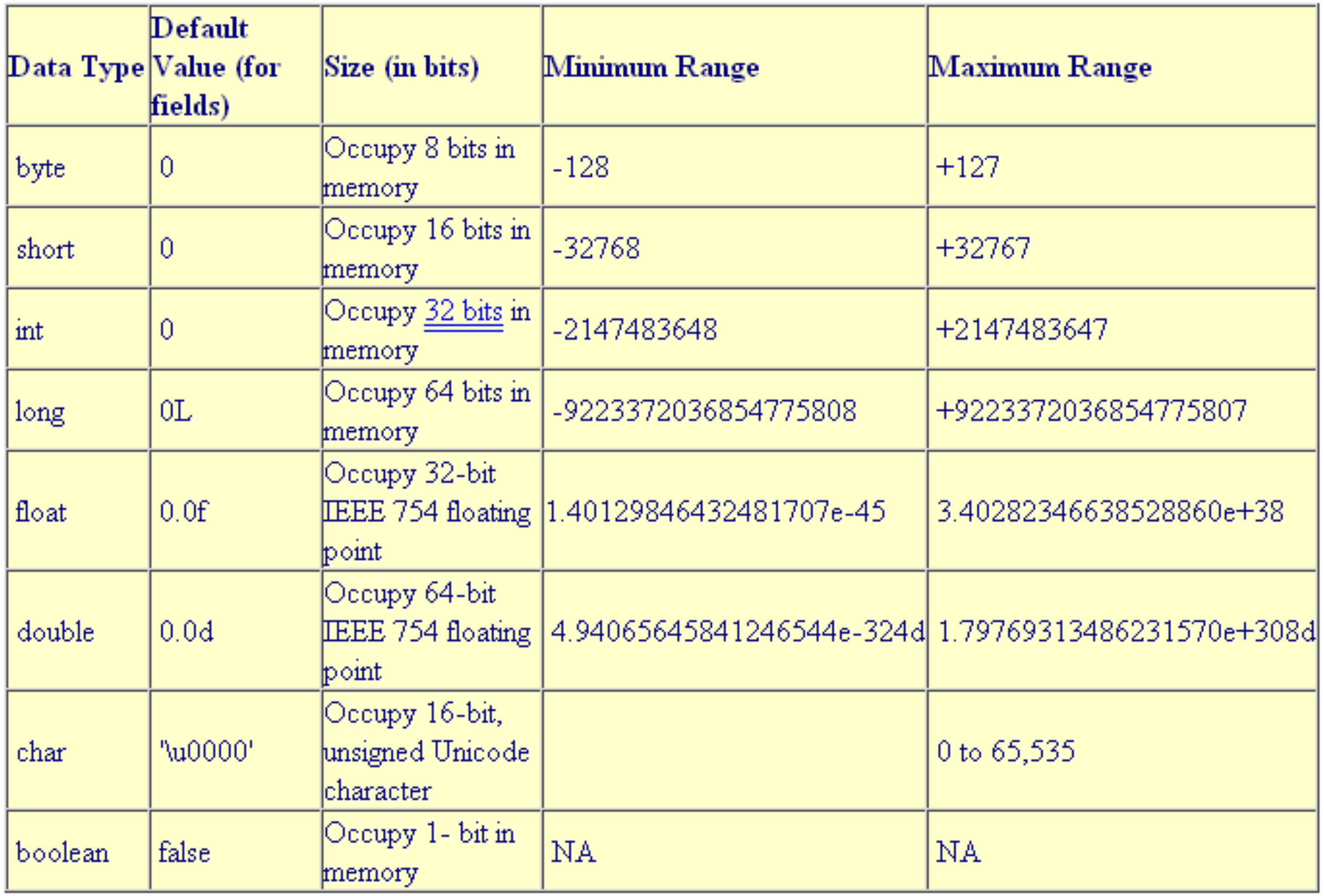
== vs. equals()

```
CarPart cp1 = new CarPart(“fender”);
CarPart cp2 = cp1;
// What’s the difference between this:
if (cp1 == cp2)
   System.out.println(“Same”);

// And this:
if (cp1.equals(cp2))
   System.out.println(“Same”);
```

- Does “Same” print twice, once, or not at all?

Primitive Sizes and Value Ranges



Data Type	Default Value (for fields)	Size (in bits)	Minimum Range	Maximum Range
byte	0	Occupy 8 bits in memory	-128	+127
short	0	Occupy 16 bits in memory	-32768	+32767
int	0	Occupy 32 bits in memory	-2147483648	+2147483647
long	0L	Occupy 64 bits in memory	-9223372036854775808	+9223372036854775807
float	0.0f	Occupy 32-bit IEEE 754 floating point	1.40129846432481707e-45	3.40282346638528860e+38
double	0.0d	Occupy 64-bit IEEE 754 floating point	4.94065645841246544e-324d	1.79769313486231570e+308d
char	'\u0000'	Occupy 16-bit, unsigned Unicode character		0 to 65,535
boolean	false	Occupy 1- bit in memory	NA	NA

Screen Output

- System.out is an output stream which corresponds to standard output, which is the screen by default:

```
int var = 33;
// What do these three statements print?
System.out.print(var);
System.out.println(var);
System.out.println(“The answer is “ +
 var);
```

Keyboard Input - 1

- Package: java.util.Scanner
- Read a string:

```
Scanner input = new Scanner(System.in);
String s1 = input.next();
String s2 = input.nextLine()
```

- Read a character:

```
char c = s1.charAt(0);
```

- Read an integer:

```
int i = input.nextInt();
```

- Read a float:

```
double d = input.nextDouble();
```

Keyboard Input - 2

- Package: java.io.*
- Read a string:

```
InputStreamReader isr = new InputStreamReader(System.in);
BufferedReader br = new BufferedReader(isr);
String s = br.readLine();
```

- Read a character:

```
char c = s.charAt(0);
```

- Read an integer:

```
int i = Integer.parseInt(s);
```

- Read a float:

```
double d = Double.parseDouble(s);
```