More on Java

Object-Oriented Programming
Outline

- Instance variables vs. local variables
- Primitive vs. reference types
- Object references, object equality
- Objects' and variables' lifetime
- Parameters passing and return values
- Methods overloading
- this reference
- Input/Output

Readings:
- HFJ: Ch. 3, 4.
- GT: Ch. 3, 4.
Variables and types

- Two kinds of variables: *primitive* and *object reference*.
- *Primitive* variables(*) hold fundamental types of values: int, float, char…
- *Reference* variables hold *references* to objects (similar to pointers)

```java
byte a = 7;
boolean done = false;

Dog d = new Dog();
d.name = "Bruno";
d.bark();
```

(*) read the text book
Primitive data types

- Java’s primitive types:
  - Numerical: byte, int, long, float, double
  - Logical: boolean (true/false)
  - Characters: char
- Primitive data are NOT objects
- There’re corresponding wrapper classes, useful when we want to treat primitive values as objects
  - Integer, Float, …
    - Integer count = new Integer(0);
  - Provide utility functions: parseInt(), valueOf()…
Object references – controlling objects

- There is actually no such thing as an object variable.
- There's only an object reference variable.
- An object reference variable holds bits that represent a way to access an object, something like a pointer or an address (we don't really know what is inside a reference variable).
- Think of an object reference as a remote control.
Object equality

- "==" and "!=" compares references (not objects) to see if they are referring to the same object.

```java
Integer b = new Integer(10);
Integer c = new Integer(10);
Integer a = b;
```

- a==b is true
- b==c is false

- Use the `equals()` method to see if two objects are equal.

```java
Integer b = new Integer(10);
Integer c = new Integer(10);
if (b.equals(c)) { // true }
```
Object equality

Method equals()

- Pre-defined classes:
  - Ready to use

- User-created classes:
  - equals() must be defined, otherwise, it always returns \textit{false}
  - This is overriding (more on that later)

```java
Integer m1 = new Integer(10);
Integer m2 = new Integer(10);
System.out.print(m1.equals(m2));
```

```java
class MyInteger {
    private int value;

    public boolean equals(MyInteger other) {
        return (value == other.value);
    }

    ...
}
```
Object references

Dog myDog = new Dog();

Remember: References are not objects!
Object's life on the heap

- Objects are created in the heap memory
  - a constructor is automatically called to initialize it
  - the set of parameters determine which constructor to call and the initial value of the object
- Objects are manipulated via references
  - dot notation is used to access the object's instance variables and methods
- when an object is no longer used, i.e. there's no more reference to it, it will be collected and freed by Java garbage collector.
Object's life on the heap

```java
Book b = new Book();
Book c = new Book();
```
Object's life on the heap

```java
Book b = new Book();
Book c = new Book();
b = c;
```

There is no way to reach Book object 1.
It is ready to be collected.
Object's life on the heap

```java
Book b = new Book();
Book c = new Book();
b = c;
c = null;
```

Book object 1 is waiting to be disallocated.
Book object 2 is safe as b is still referring to it.
Garbage collection

- To reclaim the memory occupied by objects that are no longer in use
- Programmers don’t have to disallocate objects
- Java Virtual Machine (JVM) performs automatic garbage collection
  - Method finalize() is called by JVM, not programmers.
  - Guarantee no memory leaks
- However, there’s no guarantee when/whether an object is freed before the program terminates
  - Might not needed as memory is still available
  - Clean-up tasks must be done explicitly by other “clean-up” methods rather than finalize()
Instance variables vs. local variables

Instance variables
- belong to an **object**
- located inside the object in the heap memory
- has the same lifetime as the object

Local variables
- belong to a **method**
- located inside the method's frame in the stack memory
- has the same lifetime as the method call.

```java
class Dog {
    int size;
    String breed;
    String name;
    ...
}
```

```java
public class DogTestDrive {
    public static void main(String[] args) {
        Dog dog = new Dog();
        dog.name = "Bruno";
        dog.bark();
    }
}
```
Parameter passing & return value

- Java allows only **pass-by-value**
  - That means **pass-by-copy**
  - Argument’s content is copied to the parameter

```java
class Dog {
  ...
  void bark(int numOfBarks) {
    while (numOfBarks > 0) {
      System.out.println("ruff");
      numOfBarks--;
    }
  }
}
Dog d = new Dog();
d.bark(3);
```

A method uses *parameters*. A caller passed *arguments*
Parameter passing & return value

- A parameter is effectively a **local variable** that is initialized with the value of the corresponding argument.

```java
class Dog {
    ...
    void bark(int numOfBarks) {
        while (numOfBarks > 0) {
            System.out.println("ruff");
            numOfBarks--;
        }
    }
    
    Dog d = new Dog();
    d.bark(3);
}
```

Something like
```
int numOfBarks = 3;
```

happens at this point.
The return value is copied to the stack, then to the variable that get assigned (dogSize in this example)

class Dog {
    ...
    int getSize() {
        return size;
    }
}

int dogSize = dog.getSize();

these types must match

the value is copied to
Parameter passing & return value

Two kinds of parameters:

- **Primitive types**
  - parameter’s value is copied
  - parameters can be constants, e.g. 10, “abc”…

- **Object references**
  - the reference's value is copied, NOT the referred object.
Example

```java
class Date {
    int year, month, day;
    public Date(int y, int m, int d) {
        year = y; month = m; day = d;
    }
    public void copyTo(Date d) {
        d.year = year;
        d.month = month;
        d.day = day;
    }
    public Date copy() {
        return new Date(day, month, year);
    }
    ...
}
```

- `y, m, d` are of primitive data type. They'll take the values of the passed parameters.
- `d` is a reference. `d` will take the values of the passed parameter, which is an object location.
- Return a reference to the newly created Date object. Again, it's a value, not the object.
Example

```java
... 
int thisYear = 2010;
Date d1 = new Date(thisYear, 9, 26);

class Date {
    int year, month, day;
    public Date(int y, int m, int d) {
        year = y; month = m; day = d;
    }
    public void copyTo(Date d) {
        d.year = year;
        d.month = month;
        d.day = day;
    }
    public Date copy() {
        return new Date(day, month, year);
    }
    ...
}
```
Example

```java
class Date {
    int year, month, day;
    public Date(int y, int m, int d) {
        year = y; month = m; day = d;
    }
    public void copyTo(Date d) {
        d.year = year;
        d.month = month;
        d.day = day;
    }
    public Date copy() {
        return new Date(day, month, year);
    }
}

Date d1 = new Date(thisYear, 9, 26);
Date d2 = new Date(2000, 1, 1);
d1.copyTo(d2);
```

Example

class Date {
    int year, month, day;
    public Date(int y, int m, int d) {
        year = y; month = m; day = d;
    }
    public void copyTo(Date d) {
        d.year = year;
        d.month = month;
        d.day = day;
    }
    public Date copy() {
        return new Date(day, month, year);
    }
    ...}

Date d2 = new Date(2000, 1, 1);
Date d3 = d2.copy();

Date temp =
    new Date(d2.day, d2.month, d2.year);
d3 = temp;
Method overloading

- Methods of the same class can have the same name but different parameter lists.

```java
class Dog {
    ...
    void bark() {
        System.out.println("Ruff! Ruff!");
    }
    void bark(int numOfBarks) {
        while (numOfBarks > 0) {
            System.out.println("ruff");
            numOfBarks--;
        }
    }
}

Dog d = new Dog();
d.bark();
d.bark(3);
```
Do you still remember?

Instance variables/methods belong to an object. Thus, when accessing them, you MUST specify which object they belong to.

```java
dot notation (.) and the object reference

public class DogTestDrive {
    public static void main(String[] args) {
        Dog d = new Dog();
        d.name = "Bruno";
        d.bark();
    }
}
```

access 'name' of the Dog

call its bark() method
class Dog {
    int size;
    String breed;
    String name;
    void bark() {
        if (size > 14) {
            System.out.println("Ruff! Ruff!");
        } else {
            System.out.println("Yip! Yip!");
        }
    }
    void getBigger() {
        size += 5;
    }
}

Which object does size belong to?
the object that owns the current method – bark() or getBigger()
dog1.bark(); //this dog's size get compared
dog2.getBigger(); //this dog's size get increased

where is the object reference and dot notation?
The **this** reference

class Dog {
    int size;
    String breed;
    String name;

    void bark() {
        if (this.size > 14)
            System.out.println("Ruff! Ruff!");
        else
            System.out.println("Yip! Yip!");
    }

    void getBigger() {
        this.size += 5;
    }
}

dog1.bark(); //this dog's size get compared
dog2.getBigger(); //this dog's size get increased
The **this** reference

- **this**: the object reference referring to the current object – the owner of the current method

- **usage of this**:
  - explicit reference to object’s attributes and methods
    - often omitted
  - parameter passing and return value
  - calling constructor from inside constructor
The **this** reference

class MyInteger {
    private int value;
    public boolean greaterThan (MyInteger other) {
        return (this.value > other.value);
    }
    public boolean lessThan (MyInteger other) {
        return (other.greaterThan(this));
    }
    public MyInteger increment() {
        value++;
        return this;
    }
}

MyInteger counter = new MyIncrement();
counter.increment().increment(); // increased by 2
Input / output

- Details:
  - Head First Java. Ch. 14

- In this slide:
  - standard input / output stream
  - simple input / output
  - simple text file input / output
Standard I/O

- Three stream objects automatically created when a Java program begins executing:
  - System.out: standard output stream object
    - normally enables a program to output data to the console
  - System.err: standard error stream object
    - normally enables a program to output error messages to the console
  - System.in: standard input stream object
    - normally enables a program to input bytes from the keyboard

- Redirect at command line (input and output stream only):
  C:\> type input.dat | java AJavaProgram > output.dat
Standard output and error streams

- System.out and System.err can be used directly
  - System.out.println("Hello, world!");
  - System.err.println("Invalid day of month!");
Standard input

- System.in
  - An InputStream object
  - must be wrapped before use
- Scanner: wrapper that supports input of primitive types and character strings
  - next(): get the next word separated by white spaces
  - nextInt(), nextDouble(),…: get the next data item
  - hasNext(), hasNextInt(), hasNextDouble(),…: check if there are data left to be read
Standard input. Example

```java
// import the wrapper class
import java.util.Scanner;
...
// create Scanner to get input from keyboard
Scanner input = new Scanner(System.in);

// read a word
String s = sc.next();

// read an integer
int i = sc.nextInt();

// read a series of big integers
while (sc.hasNextLong()) {
    long aLong = sc.nextLong();
}
```
Input from a text file. Example

```java
import java.util.Scanner;
import java.io.FileInputStream;
import java.io.IOException;

public static void main(String args[]) {
    try {
        // create Scanner to get input from a file stream
        Scanner sc = new Scanner(new FileInputStream("test.dat"));

        String s = sc.next(); // read a word
        int i = sc.nextInt(); // read an integer
        while (sc.hasNextLong()) { // read a series of big integers
            long aLong = sc.nextLong();
        }

        sc.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

To deal with errors such as file-not-found

Import required classes

Open and close the text file

Import required classes

To deal with errors such as file-not-found
Write to a text file. Example

```java
import java.io.PrintWriter;
import java.io.FileWriter;
import java.io.IOException;
...
public static void main(String args[]) {
    int i = 1; long l = 10;
    try {
        // create a printwriter to write output to a file stream
        PrintWriter out = new PrintWriter(new FileWriter("test.data"));

        // write to file
        out.println("Hello "+i+" "+l);

        out.close();
    } catch(IOException e) {
        e.printStackTrace();
    }
}
...
Command-line parameters

//CmdLineParas.java: read all command-line parameters
public class CmdLineParas {
    public static void main(String[] args) {
        //display the parameter list
        for (int i=0; i<args.length; i++)
            System.out.println(args[i]);
    }
}

C:\>java CmdLineParas hello world
hello
world