KING FAHD UNIVERSITY OF PETROLEUM & MINERALS
Information and Computer Science Department

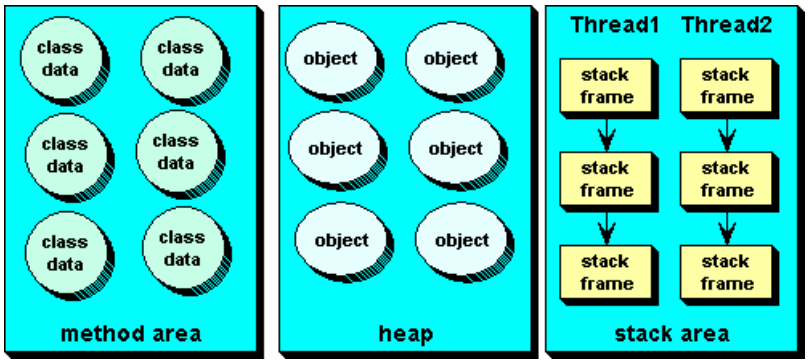
ICS-201 Introduction to Computer Science
Lab 03: Java Virtual Machines and Java Packages

Objectives: In this lab, the following topics will be covered

1. Java Virtual Machine
2. Java Packages
3. Examples & Exercises for practice

1. Java Virtual Machine:

JVM is a component of the Java system that interpretes and executes the instructions in our class files. Each instance of the JVM has one method area, one heap, and one or more stacks - one for each thread. When JVM loads a class file, it puts its information in the method area. As the program runs, all objects instantiated are stored in the heap. The stack area is used to store activation records as a program runs.



Class Loading:

Loading means reading the class file for a type, parsing it to get its information, and storing the information in the method area. For each type it loads, the JVM must store the following kinds of information in the method area:

- The fully qualified name of the type
- The fully qualified name of the type's direct superclass or if the type is an interface, a list of its direct super interfaces .
- Whether the type is a class or an interface
- The type's modifiers (public, abstract, final, etc)
- Constant pool for the type: constants and symbolic references.
- Field info: name, type and modifiers of variables (not constants)
- Method info: name, return type, number & types of parameters, modifiers, bytecodes, size of stack frame and exception table.

The end of the loading process is the creation of an instance of java.lang.Class for the loaded type. The purpose is to give access to some of the information captured in the method area for the type, to the programer.

`TestCircleClass.java`:

The following program gives information regarding the class `Circle` [Refer to Lab 2].

```
import java.lang.reflect.Method;

public class TestCircleClass
{
   public static void main(String[] args)
   {
      Circle myCircle = new Circle(15.75);

      Class circleClassInfo = myCircle.getClass();
      System.out.println("Class Name is: " + circleClassInfo.getName());
      System.out.println("Parent is: " + circleClassInfo.getSuperclass());
      System.out.println("Methods are: ");
      Method[] methods = circleClassInfo.getMethods();
      for(int i = 0; i < methods.length; i++)
         System.out.println(methods[i]);
   }
}
```

Exercises:

1. Compile and execute the above program and observe its output. Now study the methods available in the class `Class` (from the java documentation). Some of the methods available are:

   ```
   public String getName()
   public Class getSuperClass()
   public boolean isInterface()
   public Class[] getInterfaces()
   public Method[] getMethods()
   public Field[] getFields()
   public Constructor[] getConstructors()
   ```

   Use these (and other) methods to get more information regarding the class `Circle` in your program `TestCircleClass.java`

2. Can you find similar information (as found for the class `Circle`) for the class `TestCircleClass`? Write a program that finds all the methods, fields, the name of the superclass, etc of the class `TestCircleClass`.

Linking: Verification, Preparation and Resolution:

The next process handled by the class loader is Linking. This involves three sub-processes: Verification, Preparation and Resolution. Verification is the process of ensuring that binary representation of a class is structurally correct. Example of some of the things that are checked at verification are:

- Every method is provided with a structurally correct signature.
- Every instruction obeys the type discipline of the Java language
- Every branch instruction branches to the start not middle of another instruction.

In the Preparation phase, the Java virtual machine allocates memory for the class (i.e static) variables and sets them to default initial values. Note that class variables are not initialized to their proper initial values until the initialization phase - no java code is executed until initialization. The default values for the various types are shown below:

Type	Initial Value
int	0
long	0L
short	(short) 0
char	'\u0000'
byte	(byte) 0
boolean	false
reference	null
float	0.0f
double	0.0d

Resolution is the process of replacing symbolic names for types, fields and methods used by a loaded type with their actual references. Symbolic references are resolved into a direct references by searching through the method area to locate the referenced entity.

Class Initialization and Instantiation

Initialization of a class consists of two steps:

- Initializing its direct superclass (if any and if not already initialized)
- Executing its own initialization statements

The above imply that, the first class that gets initialized is Object. Note that static final variables are not treated as class variables but as constants and are assigned their values at compilation.

After a class is loaded, linked, and initialized, it is ready for use. Its static fields and static methods can be used and it can be instantiated. When a new class instance is created, memory is allocated for all its instance variables in the heap. Memory is also allocated recursively for all the instance variables declared in its super class and all classes up is inheritance hierarchy. All instance variables in the new object and those of its superclasses are then initialized to their default values. The constructor invoked in the instantiation is then processed. Finally, the reference to the newly created object is returned as the result.

`Instantiation.java`

The following program demonstrates the order of class instantiation.

```
class GrandFather{
   int grandy = 70;
   public GrandFather(int grandy){
      this.grandy = grandy;
      System.out.println("Grandy: "+grandy);
   }
}

class Father extends GrandFather{
   int father = 40;
   public Father(int grandy, int father){
      super(grandy);
      this.father = father;
      System.out.println("Grandy: "+grandy+" Father: "+father);
   }
}

class Son extends Father{
   int son = 10;
   public Son(int grandy, int father, int son){
      super(grandy, father);
      this.son = son;
      System.out.println("Grandy: "+grandy+" Father: "+father+" Son: "+son);
   }
}

public class Instantiation{
   public static void main(String[] args){
      Son s = new Son(65, 35, 5);
   }
}
```

Exercises:

3. When in the above program you comment out the shaded lines (the calls to the super-constructors) and compile the program, the following errors result:

```
--------------------Configuration: JDK version 1.3 <Default>--------------------
D:\Workarea\java\LabsStuff\cs\Instantiation.java:11: cannot resolve symbol
symbol  : constructor GrandFather  ()
location: class GrandFather
public Father(int grandy, int father) {
^
D:\Workarea\java\LabsStuff\cs\Instantiation.java:19: cannot resolve symbol
symbol  : constructor Father  ()
location: class Father
public Son(int grandy, int father, int son) {
^
2 errors
```

.

Can you explain why these errors occurred? How can you fix these errors (without deleting anything from the program)?

2. Java Packages:

A Java package is a collection of sub-packages and/or classes (compilation units). The classes in a package may or may not be related by inheritance. A package is used to group similar and interdependent classes together. The Java **API** is composed of multiple packages. The **import** statement is used to assert that a particular program will use classes from a particular package. A programmer can define a package and add classes to it. The **package** statement is used to specify that all classes defined in a file belong to a particular package. The syntax of the **package** statement is:

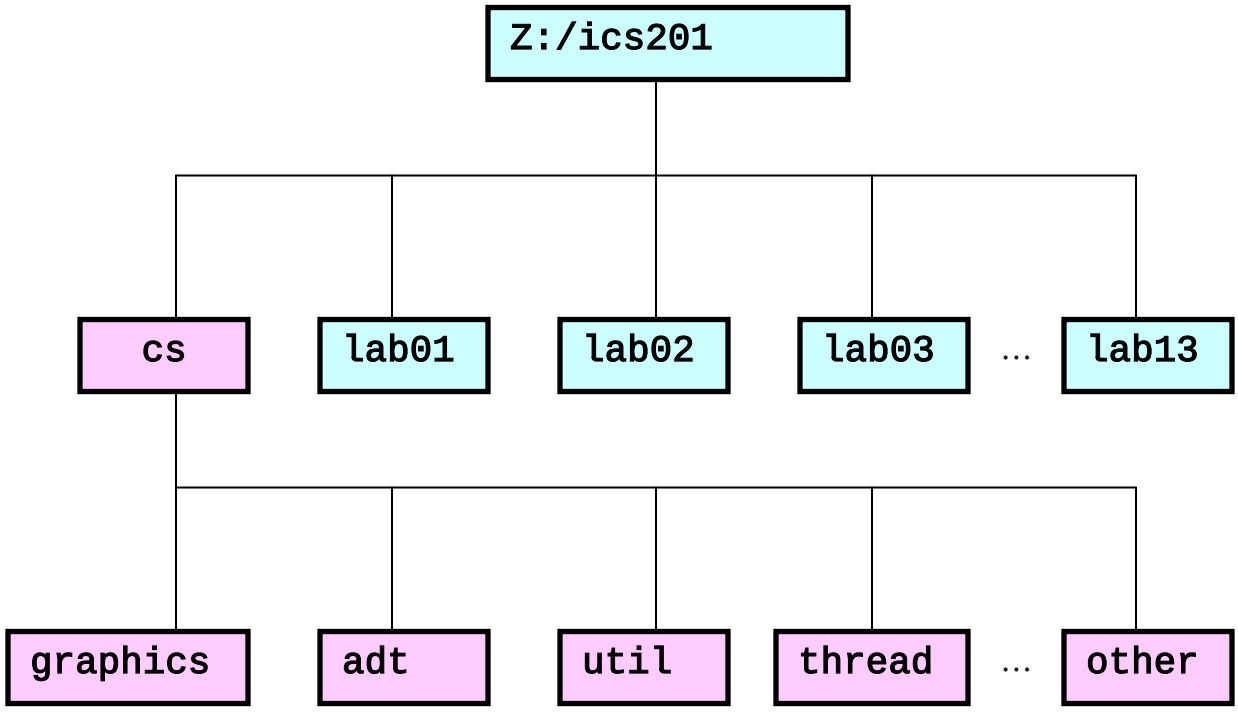
```
package package-name;
```

The **package** statement must be located at the top of a file, and there can be only one package statement per file.

Before we delve into the details of package-making and using, let us talk a bit about setting the classpath.

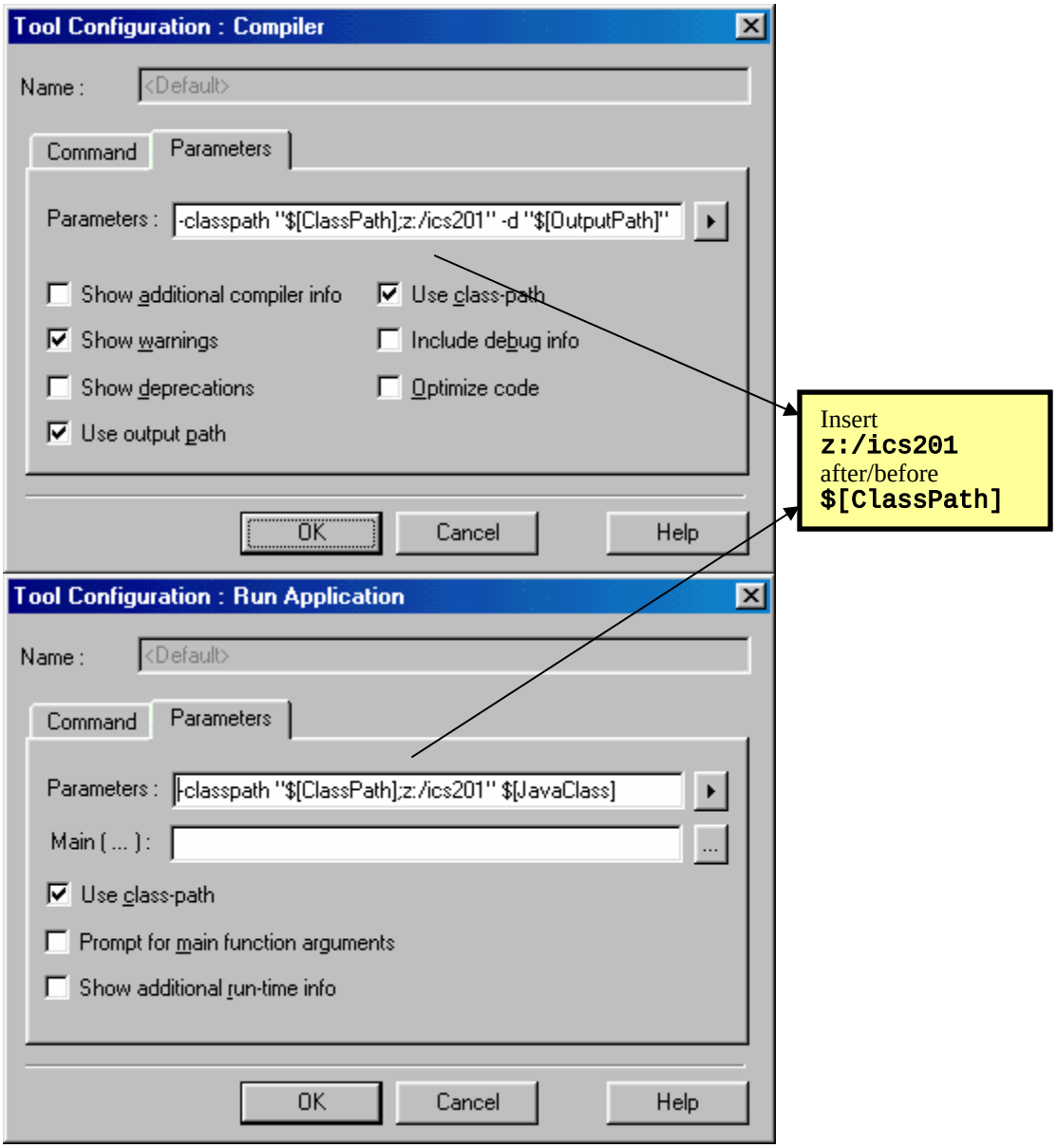
Directory Structure:

In this course we shall be using the following directory structure:



Setting the classpath (JCreator Only)

JCreator reads the package statement from the java file and automatically creates the directory structure. Make sure you save your `.java` source files into the folder `z:/ics201`. This is important because JCreator will automatically save all your class files in the relevant package. Java tools (compiler, interpreter, etc) automatically detect the location of the standard Java packages. However, for user-defined packages, the user must specify their locations. **We shall compile and execute programs that use user-defined packages from within JCreator**. To do this, we need to modify the parameter configuration of both the **Compiler** and **Run Application** as shown below:



Example: The following defines a **`Point`** class, defined in package **`cs.graphics`**.

```
package cs.graphics;

public class Point {
    private double x, y;

    public Point(double x, double y) { this.x=x; this.y=y; }
    public Point() { this(0.0, 0.0); }

    public double distanceToOrigin() { return Math.sqrt((x*x) + (y*y)); }
    public double distanceToPoint(Point p) {
        return Math.sqrt(((x-p.x)*(x-p.x))+((y-p.y)*(y-p.y)));
    }

    public void moveTo(double x, double y) { this.x=x; this.y=y; }

    public void translate(double dx, double dy) { x += dx; y += dy; }

    public double getX() { return x; }
    public double getY() { return y; }

    public String toString() { return "("+x+", "+y+")"; }

    public boolean equals(Point p) { return this.x==p.x && this.y==p.y; }
}
```

Example: The following defines a **`ConsoleReader`** class, in package **`cs.util`**.

```
package cs.util;

import java.io.BufferedReader;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.io.IOException;

public class ConsoleReader {
  private BufferedReader reader;

  public ConsoleReader(InputStream inStream) {
    reader = new BufferedReader(new InputStreamReader(inStream));
  }

  public int readInt() {
    String inputString = readLine();
    int n = Integer.parseInt(inputString);
    return n;
  }

  public double readDouble() {
    String inputString = readLine();
    double x = Double.parseDouble(inputString);
    return x;
  }

  public String readLine() {
    String inputLine = "";
    try {
      inputLine = reader.readLine();
    } catch(IOException e) { System.out.println(e); System.exit(1); }
    return inputLine;
  }
}
```

Example: The following class `TestPoint.java` shows how user-defined classes may be imported.

```
package lab03;

import cs.graphics.Point;
import cs.util.ConsoleReader;

public class TestPoint {
  static ConsoleReader stdin = new ConsoleReader(System.in);

  public static void main(String[] args) throws java.io.IOException{
    double x,y;

    System.out.println("Enter first point: ");

    x=stdin.readDouble();
    y=stdin.readDouble();

    Point p1 = new Point(x,y);

    System.out.println("Enter second point: ");

    x=stdin.readDouble();
    y=stdin.readDouble();

    Point p2 = new Point(x,y);

    System.out.println("Distance of "+p1+ " to origin = "+p1.distanceToOrigin() );
    System.out.println("Distance of "+p2+ " to origin = "+p2.distanceToOrigin() );
    System.out.println("Distance between "+p1+" & "+p2+ " = "+p1.distanceToPoint(p2));
  }
}
```

Exercise:

Q. 4: Suppose that in the class `TestPoint` you need to use another class called `Point` defined in the `java.awt` package. In other words if you add

```
import java.awt.Point;
```

to class `TestPoint.java` what kind of error messages will you get? How will you resolve those error messages (without deleting anything from the file)