Chapter 2

Additional Features of Java

The static modifier is applied to constant identifiers (always) and method identifiers (sometimes).

Static **means "applying to the class as a whole, rather than to an instance of the class."**

For example,

```
public static final int   MAX_VALUE = 0x7fffffff;
```

= 2^{31} - 1

There is only one copy of MAX_VALUE **instead of a separate copy for each instance of** Integer.
final means "Can be assigned to only once."

To access this constant, the class name is followed by the member-designator operator (the dot) followed by the constant identifier:

```
if (size == Integer.MAX_VALUE)
```

```
public static int parseInt (String s)
```

returns the int value corresponding to the digit characters that constitute s.

```
String line = reader.readLine();
int score = Integer.parseInt (line);
```

There is no calling object; the argument has all the method needs.

Exception Handling

A *robust* program is one that does not terminate abnormally from invalid user input.

An *exception* is an object created by an unusual condition, such as an attempt at invalid processing

Try to execute a block of code:

```
try
{
    …
} // try
```

Catch the exceptions thrown (one catch block per exception):

```
catch (Exception e)
{
    …
} // catch
```

```
public void processInput (String s)
{
  int score;

  try
  {
     score = Integer.parseInt (s);
     System.out.println (score * 2);
  } // try
  catch (NumberFormatException e)
  {
      System.out.println(e);//print  "NumberFormatException"
                            // + line#
   } // catch
  System.out.println ("Continue here whether or not  " +
                "NumberFormatException was thrown");
} // method processInput
```

Exercise: In the following method

```
public void processInput (String s) {

    …

} // method processInput
```

S is supposed to contain a last name followed by a blank followed by a first name.

The output should be:

The last name, if there is a last name, followed by a blank followed by the first three letters of the first time;

NoSuchElementException **message, if either name is missing;**

StringIndexOutOfBoundsException **message, if the first name has fewer than three letters.**

Hint: in the String **class,**

public String **substring (int start, int finish)**

returns the substring from start **to** finish-1.

```
public void processInput (String s)
{
      try {
          StringTokenizer st = new StringTokenizer (s);
          String lastName = st.nextToken( ),
                 firstName = st.nextToken( );
          System.out.println (…);
      } // try
      catch (…)
      {
      …
      } //
      catch (…)
      {
      …
      } //
}  method processInput
```

Exceptions can be explicitly thrown:

```
if (size == Integer.MAX_VALUE)
    throw new ArithmeticException ("size too large");
```

If an exception is not caught when it is thrown, it is "propagated" back to the calling method. For example, suppose the input is "1Z"

```
    try
    {
        String s = reader.readLine();
        process (s);
    } // try
    catch (NumberFormatException e)
    {
        System.out.println ("oooops" + e);
    } // catch


public void process (String s)
{
    int age = Integer.parseInt (s);
    …
} // method process
```

Run-time exceptions, such as NumberFormatException **and** NullPointerException**, are automatically propagated. Other exceptions, especially** IOException**, are called checked exceptions. They can be propagated only if a** throws **clause appears right after the method heading.**

```
public int findMedian (int [ ] a) throws IOException
{
    BufferedReader reader = new BufferedReader
                (new InputStreamReader (System.in));

    System.out.println (reader.readLine());

    …
```

Cultural note: It is considered bad form for the main **method to throw any exception – because then the exception will not be caught by your program.**

File Output

```
PrintWriter fileWriter = new PrintWriter
    (new BufferedWriter (new FileWriter ("maze.out")));

fileWriter.println ("Here is the grid:");
```

The output does not immediately go to the file, but is saved in a *buffer* – a temporary storage area in memory. Output is transferred from the buffer to the file when the buffer is full (and fileWriter.println **is called) or when** fileWriter.close() **is called.**

The last method called by a file object should be close().

File input

```
BufferedReader  fileReader = new BufferedReader
        (new FileReader (fileName));
```

The file name is usually read in from the keyboard.

What can go wrong? If fileName **is not the name of a file, an** IOException **object will be thrown. But then another string should be read in from the keyboard and the statement from the previous slide should be repeated.**

```
boolean filesOK = false;
while (!filesOK)
{
    try   // to read the file name
    {
        // read in file name
        while (true)
        {
            try // to read one line from the file
            {
                // read in and process one line
            }
            catch (//some RuntimeException) { … }
        } // while true
        filesOK = true;
    } // try
    catch (IOException e) { … }
} // while !filesOK
```

A method is *correct* if it satisfies its specification.

Your confidence in the correctness of your methods can be increased by testing, which can reveal the presence but not absence of errors.

For a single method, you can often easily create tests on the fly. For testing all of the methods in a class, a driver is used.

A *driver* is a program created to systematically test a class's methods in concert.

The input for the tests comes from a file, so the input is not manually re-entered for each run of the driver. The input will not be in a fixed order, so the methods can be tested in concert: m1 **followed by** m2**;** m2 **followed by** m1**, …. Also, the output will go to a file.**

```
boolean filesOK = false;
while (!filesOK)
{
    try   // to read the file name
    {
        // read in file name
        while (true)
        {
            try // to read one line from the file
            {
                // read in and process one line
                testMethod (line);
            } // inner try
            catch (//some RuntimeException) { … }
        } // while true
        filesOK = true;
    } // outer try
    catch (IOException e) { … }
} // while !filesOK
```

The call testMethod(line) **will tokenize** line **and send the appropriate message. For example, if** line **has**

e1 makesMoreThan e2

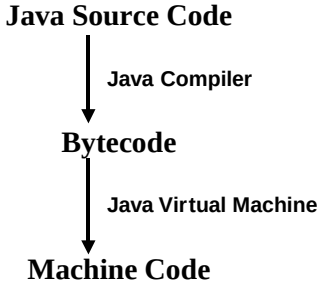
Then testMethod **will call**

fileWriter.println (e1.makesMoreThan (e2));

The Java Virtual Machine

also known as

the Java Run-Time Environment



The Java virtual machine handles all run-time aspects of your program. For example:

1. **Pre-initialization of fields, just prior to a constructor call (so constant fields cannot be assigned a value after they are declared). This ensures that all fields get initialized.**

2. **Garbage collection: De-allocation of space for inaccessible objects**

   The virtual machine handles allocation by implementing the new **Operator.**

   What about de-allocation?

Suppose we have a local variable

```
double[] d = new double [100000];
```

at the end of the execution of the method, the space for the reference d **is deallocated. But what about the space for the object (100000 doubles)?**

Space for that object can be deallocated provided there are no "live" references to the object.

The Java virtual machine keeps track of the number of live references to an object.

Visibility modifiers:

1. **public – accessible in any class**
2. **protected – accessible in any class within the same package, or in any subclass**
3. **default – accessible in any class within the same package**
4. **private – accessible only within the class**

The Object **class – the superclass of all classes – has an equals method:**

```
public boolean equals (Object obj)
{
    return this == obj;
} // method equals
```

The pre-declared variable this **references the calling object.**

Because this method compares references, not objects, we should override the above version for any class with an equals **method.**

For example, let's define an equals **method for the** FullTimeEmployee **class:**

```
public boolean equals (Object obj)
{
    // IF obj DOES NOT REFERENCE A
    // FullTimeEmployee OBJECT
    //         return false;

    // return name equals (obj's name) &&
    //           gpa == obj's gpa;
} // method equals
```

```
public boolean equals (Object obj)
{
        if (!(obj instanceof FullTimeEmployee))
            return false;
        FullTimeEmployee full = (FullTimeEmployee)obj;
        return name.equals (full.name) &&
                grossPay == full.grossPay;
} // method equals
```

Exercise: Define an equals **method in the** HourlyEmployee **class. Two** HourlyEmployee **objects are equal if they have the same name, hours worked and pay rate.**