Exam Objective - Section 7 - Fundamentals

Source - *SCJP Sun Certified Programmer for Java Platform, SE 6 Study Guide* by **Richard F. Raposa**

Chapter 1 - Fundamentals

1.1 Writing Java Classes

- Java source files
  - Java classes are defined in a text file of extension **`.java`**
  - If the class is declared public then the filename must match the public class name
  - Java permits *multiple class* definitions in a single .java source file
    - BUT a `.java` file can contain only 1 top-level class of **public** access
    - other classes must have **default** access (aka **package-level** access)
      - (i.e default access is specified *implicitly* only ==> when neither public, protected nor private are specified)
      - private and protected classes are NOT allowed

- Compiled java code
  - referred to as ***bytecode***
  - has the **.class** file extension
  - the name of the bytecode file matches the name of the class
    - i.e. *`ClassName`*`.class`
  - if a single source file has multiple classes then compilation will result in the output of multiple `.class` files

1.2 Packages

- A **package** is a *grouping* of classes, interfaces, enumerations and annotated types (aka "classes and interfaces")
  - the grouping is based on their *relationship* and *usage*

- Two key benefits of using packages
  1. organizes Java programs by grouping *related* classes and interfaces
  2. creates a **namespace** for your classes and interfaces ==> avoids naming conflicts

1.2.1 The *package* Keyword

- must be the *first* line of code in a source file (excluding comments)

- puts a class or interface into a package
  - e.g. The below code puts the Employee class into the `com.sybex.payroll` package

```
package com.sybex.payroll;
public class Employee { ... }
```

- Two important side-effects of placing a class in a package
  1. The **fully qualified name** (FQN) of a class/interface changes when it is in a package
     - The package name becomes a prefix for the class name

        - e.g. (from above) the FQN of `Employee` is `com.sybex.payroll.Employee`
        - Other classes need to refer to the Employee class by its FQN
            - except other classes in the same package as Employee.
    2. The **compiled bytecode** file must appear in a ***directory structure*** that matches the package name
        - the directory structure can be created manually or through the **-d** flag of the **javac** compiler
            - (see later section *Package Directory Structure* for -d flag)
        - e.g. (from above) `Employee.class` file ***must appear*** in the directory `\com\sybex\payroll`

- Unnamed package
    - If a class or interface is not specifically declared to be a part of a package (using the package keyword) then it is a part of the **unnamed package**
    - Such classes or interfaces
        - cannot be imported into a source file
        - are only written for simple classes that are not used in a production application

1.2.2 The *import* Keyword

- allows the programmer to refer to a class/interface in a source file *without* using its FQN
    - instead the compiler searches your list of imports to determine the FQNs of classes/interfaces used in the source file
    - NOTE: the import statement has nothing to do with loading of classes. It is strictly used for shortening the use of FQNs
        - the compiler removes all import statement and replaces class names with their FQNs
        - importing a single class or a whole package *does not effect the overall size* of the compiled class.

- used to
    - import a single class
        - when the fully qualified name of the class is specified
        - e.g.

```
import com.sybex.payroll.Employee; // import the fully qualified class name
import java.io.File;                // import the File IO class from java.io
```

    - import an entire package
        - when the **wildcard** (*****) character is specified
        - e.g.

```
import com.sybex.payroll.*; // import the entire payroll package
import java.io.*;           // import the entire IO java package
```

- the import statement(s) must appear AFTER the package statement and BEFORE the class definition statement in the source file

- importing tells the compiler you are not going to use the FQN for the imported classes and packages
    - the compiler searches the list of imported classes and packages to match short class names to their FQN
    -

- **java.lang**
    - the compiler automatically imports the **java.lang** package (meaning the *public* classes/interfaces of java.lang) into every source file

- BEST PRACTICE
    - not to import entire packages since this can cause naming ambiguity
        - i.e. two packages have classes of the same name
    - in this case the naming ambiguity can be overcome by specifying the FQN of those particular classes when having to declare variables or member data
    - (this is why packages are referred to as **namespaces**)
    - naming convention for packages is to use the reverse company domain name as the root of the package hierarchy
        - e.g. com.hsa

1.2.3 Package Directory Structure

- The use of packages requires that the **bytecode** (.class) files must appear in a directory structure that matches its package name
    - If this is not adhered to the compiler / JVM will be unable to find your class(es)
    - e.g. the bytecode for the class com.sybex.payroll.Employee must appear in the directory \com\sybex\payroll
    - the \com directory can appear ANYWHERE in your file system
        - therefore you must either compile your source files
            - specifying this location
            - or directly from the directory containing \com

- It is common to place source files under a **\src** directory and place your bytecode files under a **\build** directory
    - e.g. your development root is c:\projects\dev.
    - e.g. your source file root directory, ***root_source_directory***, is then c:\projects\dev\src.
    - e.g. your compiled root directory, ***build_directory***, is the c:\projects\dev\build

- Use javac's **-d** option to specify the location of the **build** directory
    - `javac -d` ***`build_directory src_file`***
        - e.g. `javac -d c:\projects\dev\build .\com\sybex\payroll\Employee.java`

        - (this would be run from the c:\projects\dev\src directory)
    - This has two effects
        1. The appropriate directory structure that matches the package names of the classes will be *automatically* created in ***build_directory***
            - e.g. (from above) c:\projects\dev\build\com\sybex\payroll
        2. The compiled code will be *placed* in the directory specified by the -d option (the ***build_directory***)
            - e.g. (from above) c:\projects\dev\build\com\sybex\payroll\Employee.class

- NOTE:
    - When compiling your classes you must
        - specify your ***build_directory*** as the option to your **-d** flag
        - specify the relative path to your .java file from the current working directory
        - e.g.1
            - cd c:\projects\dev\src
            - javac -d c:\projects\dev\build .\com\sybex\payroll\Employee.java
        - e.g.2
            - cd c:\projects\dev\src\com\sybex
            - javac -d c:\projects\dev\build .\payroll\Employee.java

1.2.4 The *CLASSPATH* Environment Variable

- The ***classpath*** refers to the path on your file system where your .class files are saved
    - it plays a key part in ***compiling*** and ***running*** java applications
    - it allows Java to run and compile programs independent of the current working directory

- The ***classpath*** is defined by the **CLASSPATH** environment variable
    - which specifies the directories and JAR files where you want the **compiler** and **JVM** to *search* for bytecode
    - multiple directories and JAR files can be specified
    - if you have a package then your **classpath** must point to the root of your package and NOT within directories of your package
    - e.g. if you have a class with a FQN of com.sybex.payroll.Employee then the compiler/JVM will look for \com\sybex\payroll\Employee.class in locations specified in your CLASSPATH

- To specify your CLASSPATH
    - in Windows, do the following:
        - `set CLASSPATH="c:\Documents and Settings\`*`username`*`\projects\build";c:\projects\build;c:\tomcat\lib\servlet.jar;.;`
        - NOTE:
            - use a **semi-colon** '**;**' to delimit multiple entries
            - use a **period** '**.**' to notate the current working directory - by default Windows does not search the current working directory

        - use **double-quotes ""** if spaces are present in a pathname
    - in Unix/Linux do the following
        - `setenv CLASSPATH /usr/build/:/projects/dev/build:/tomcat/lib/servlet.jar`
        - NOTE:
            - use a **colon** '**:**' to delimit multiple entries
    - DO NOT include part of your package pathname in your CLASSPATH
        - the compiler and JVM will automatically look for the package directory hierarchy!!

1.3 Running Java Applications

- If using Sun's JDK then **java.exe** contained in the **\bin** directory of the JDK is the executable used to run Java applications on your system
    - your PATH environment variable must point to **JDK_DIR\bin**

- The entry point of a Java program is the **main()** method
    - It can be defined in any class
    - It will be run if **`java YourClassName`** is called on the command line (or via a shortcut)
    - The signature of main() must look exaclty like the below in order to be *successfully* executed as a Java Application:

```
public static void main(String[] args)
```

        - where
            - public - so the JVM has access to it
            - static - so the JVM does not have to instantiate an object of the containing class
        - with the following exceptions
            - the keywords **public** and **static** can be in *reverse order* (i.e. static public void main(...))
            - the **[]** after **String** can be placed after the args parameter (i.e. static public void main(String args[])
            - args can have a different parameter name (e.g. arguments, instead of args)
            - and String[] args can be replaced by the ellipses '**String... args**' to specify variable length String array
        - NOTE: if the main method does not conform to this rule it will compile successfully, but will NOT *run* successfully

- Running a java class
    - To run a java class it must contain a main method that conforms to the above rule

    - To execute the class call

`java YourClassName` # which looks for a bytecode file called *YourClassName.*class

    - Do NOT call

`java YourClassName.class` # which looks for a class named *class* in the *YourClassName* package

- The JVM requires
  - the FQN of the class to run successfully
    - the FQN *must always be specified* independent of the CLASSPATH set or the current working directory
      - **e.g.** `java com.sybex.payroll.Employee` # given that Employee contains a compliant main() method, this will run the Employee program
    - this is due to the fact that putting a class into a *package* changes the name of a class to it's FQN which must always be used on the command line

  - If you want the class to be called from *ANY* directory then the `CLASSPATH` has to be set correctly
    - to find the package directory structure and the bytecode file
    - Do this BY adding the ***build_directory*** location to your `CLASSPATH` environment variable (see section above)
      - e.g. `set CLASSPATH=c:\projects\dev\build;%CLASSPATH%`
    - OR BY setting the classpath on the command line of the java command (see section below)
      - e.g. `java -classpath c:\projects\dev\build com.sybex.payroll.Employee`

1.3.1 The -classpath Flag

- The **java** execuatble (which starts the JVM) has a **`-classpath`** (or **`-cp`** shorthand) flag that permits the classpath to be specified on the command line

- This FLAG is used to ensure that the classpath is pointing to the right directories and JAR files

- Using this flag *overrides* the $CLASSPATH environment variable

- Multiple directories and/or JAR files can be specified similar to setting the CLASSPATH environment variable
  - on Windows systems separate multiple directories/JAR files using a **semi-colon** '**;**'
  - on Unix/Linux systems separate multiple directories/JAR files using a **colon** '**:**'

- Examples
  - Windows
    - `java -cp c:\projects\dev\build;c:\tomcat\lib\servlet.jar FQN_SRCFILENAME`
    - `java -classpath c:\projects\build com.sybex.demos.TestColours`
  - Unix/Linux
    - `java -cp /projects/dev/build:/tomcat/lib/servlet.jar FQN_SRCFILENAME`
    - `java -classpath /projects/dev/build com.sybex.demos.TestColours`

1.3.2 JAR Files

- Bytecode can be stored in archived, compressed files known as JAR files
- The compiler and JVM can find bytecode files in JAR files without the need to *uncompress* them
- Are the most common way to **distribute** Java code safely
  - you can package up bytecode files independent of source files
- **`jar.exe`** can be found in the JDK/bin directory
  - to add bytecode to a JAR file run the following command from your ***build_directory***

```
jar -cvf myproject.jar .
```

    - where
      - **-c** flag specifies creating a new JAR file
      - **-v** flag tells the jar command to run in verbose mode
      - **-f** flag denotes that the next argument is the name of the JAR file
      - and '**.**' specifies to JAR up all files under the current directory and subdirectories
- If a class is located in a package then ALL directories of the package must be added to the JAR
- To make your classes inside your JAR file available to javac and/or the JVM
  - THEN include the location of your JAR file in your
    - **CLASSPATH** environment variable, or
    - **`-classpath`** command line arguement
- Typically JAR files are added to your build/lib directory and class files to your build/classes directory

1.3.3 Command-Line Arguments

- command line arguments are passed into the main method as a *single array* of String objects
  - example:
    - java com.sybex.demos.PrintGreetings hi goodbye see you later
    - 5 arguments exist in this example and they are:
      - `args[0] = "hi", args[1] = "goodbye", ..., args[4] = "later"`
  - NOTE:
    - The class/program name does not appear in the list of arguments passed to the main method
    - indexing of ALL arrays starts from 0 (ZERO)
- ALL command line arguments are treated as String objects
  - use the ***`parse`***`XXX()` methods of the **Wrapper** - classes to treat a command-line

argument as a primitive
    - e.g.
        - `int x = Integer.parseInt(args[0]);`
        - `boolean b = Boolean.parseBoolean(args[1]);`
    - NOTE: Strings do not need to be converted to **char**s, (since Strings are arrays of `char`s)
        - just use the **.charAt(***position***)** method to obtain the character at a particular position

1.4 Reference vs. Primitive Types

1.4.1 Primitive Types

- Primitive types are the 8 built-in types that represent the building blocks for Java objects
- Need to know the following for the SCJP exam
    - The 8 different primitive types
    - The # of bits that represent each type
    - The range of values for <u>some of the types</u>

Primitive	**Size** (bits)		**<u>Range</u>**
byte	8		-2^7 to (2^7 -1) : -128 to 127
short	16		-2^15 to (2^15 -1)
int	32		-2^31 to (2^32 -1)
long	64		-2^63 to (2^63 -1)
float	32		
double	64		
char	16 (unicode)		'\u0000' to '\uffff' : 0 to 65535
boolean	unspecified		*true* OR *false*

- Primitive types are allocated in memory when they are **declared**
    - *i.e.*
        - `int x;  // automatically allocates 32 bits upon declaration`
        - `double d;  // automatically allocates 64 bits upon declaration`
- The **value** of a primtive is *stored* in the memory where the **variable** is *allocated*

1.4.2 Reference Types

- Reference types are variables that are
    1. class types
    2. interface types, or
    3. array types
        1. Arrays are objects and have a reference type
            - Java implicitly defines a reference type for each possible array type
                - one for each of the **8 primitive types**
                - one for an **Object** array
            - Example

```
int [] grades;
Runnable [] threadTargets;
```

- A reference **points** to an object by storing the memory address location of the object
    - aka a reference is a **pointer**
    - however Java does not allow the programmer to access a physical memory address!
        - Objects do not have a name that can be used to access it
        - rather programmers must use a reference to gain access to an object
    - all references consume the same amount of memory (within the same JVM)
        - the size of a reference is JVM-implementation specfic

- Declaring reference types
    - specify the reference type and a variable name
    - Example

```
java.util.Date today; // Date is the reference type, today is the
variable name
```

- Initializing reference types
    - Values are assigned to references in 2 ways:
        1. a reference is assigned to *another reference* of a **compatible** type
        2. a reference is assigned to *a new object* using the **new** keyword
        - Example:
          today = new java.util.Date();

```
String newHello = new String("Hello");   // creating a new String object
String hello = "Hello World";            // creating a new String object
(using a literal, without using new)
String greeting = hello;                 // assigning one String reference
to another
```

- The **null** type
    - is a special data type in Java
        - it does not have a name
        - you cannot construct a reference of type **null**
    - rules
        - **null** *can only* be assigned to a *reference type* (any reference type)
        - **null** *cannot* be assigned to any of the *primitive types*

- References can be assigned to other references as long as their data types are compatible
    - Reference compatibility
        - A first reference is **compatible** with a second reference if the second reference is the **same type** or a **subtype**
        - A first reference is **incompatible** with a second reference if the second reference is of **another type** or a **parent type**
        - **i.e.** they are based on an **inheritance relationship**
    - Two incompatible types cannot be assigned even if an explicit cast is used, since they are inconvertible

- When two references point to the same object they are both permitted to act on the object
    - e.g. adding values to an array

1.5 Garbage Collection

- All Java Objects (i.e. not primitive types) are stored in your program's memory heap.

- Differences between Objects and References**:**
    - **References**
        - may or may not be created on the heap
        - is a **variable** that has a **name**
        - it can be used to access contents of an object
        - can be assigned to an object, another reference, passed to a method, or returned from a method
        - all references are the same size, independent of their type
    - **Objects**
        - always created on the heap
        - do not have a name
        - there is no way to access an object, other than through a reference
        - objects vary in size *depending* on their *class definition*
            - objects may contain references in their class definition
                - these references only take up the size of a reference in the objects memory allocation
        - objects **cannot** be assigned to other objects
    - It is the **object** that gets garbage collected and not the reference

- Garbage collection
    - refers to the process of automatically freeing memory on the heap
        - by deleting objects that are no longer **reachable**
    - every JVM has a garbage collector
        - however they all operate according to different algorithms that determine ***when*** objects get garbage collected
        - the garbage collector typically runs in a different thread from your main program
    - It is important to know what **IS** and **IS NOT** guaranteed by garbage collection

- RULES for garbage collection
    - There is NO GUARANTEE (in Java) as to **when** an object is actually garbage collected
    - Only rule is that when an object becomes eligible for garbage collection the garbage collector must eventually free the memory
    - Java programmers *cannot explicitly* free memory
    - Java programmers can only ensure that objects that they want freed are *no longer reachable*
        - Objects remain on the heap until it is *no longer* ***reachable***
            - An object is no longer reachable when one of the two following situations occur:
                1. The object no longer has any references pointing to it
                2. All references to an object go **out-of-scope**

1.5.1 The *System.gc* method

- The **`java.lang.System`** class has a static method called **`gc()`** (aka **`System.gc()`**) that ATTEMPTS to run the garbage collector
    - the **`gc()`** method *does not guarantee anything* - it is only an **attempt** to force garbage collection
    - the java specification states that calling **`gc()`** suggests that the JVM may attempt to spend effort toward recycling unused objects

- Programmers can only ensure that their objects are ***eligible*** for garbage collection

1.5.2 The *finalize* method

- The garbage collector invokes **`finalize()`** method on an object **only once** just before it *actually removes* it from the heap

- The **`finalize()`** method is declared in the **`java.lang.Object`** class
    - any subclass can override the finalize() method to perform any necessary cleanup or dispose of system resources
    - is INVOKED on an object only **ONCE** by the garbage collector
    - is INVOKED after the object becomes eligible for garbage collection and just prior to actual garbage collection
    - it is a good idea to call **`super.finalize()`**so the parent class performs any necessary cleanup
        - when calling `super.finalize()` we need to declare that the **finalize()** method throws the **`java.lang.Throwable`** exception
            - since the parent class's `finalize()` method may throw the Throwable exception.

1.6 Call By Value

- Java simplifies the concept of passing **arguments** into methods by providing only 1 way to pass arguments: **by value**

    - which means that a **copy** of the **argument** is passed to the method's **parameter**
        - argument = the variable passed into the method call
        - parameter = the name of a variable declared in the method signature
    - both primitive and reference types are passed by value

- Method return types are also passed by value
    - which means that a **copy** of the **variable** is returned

- The values of the primitives/references cannot be modified since primitive/references are **COPIED** when passed
    - only the underlying object of references can be modified via object method invocations

- Passing Primitives vs Passing References
    - Passing **Primitives**: it is impossible for a method to alter the value of the original primitive
        - assignment operations on the parameter do not alter the value of the original argument
            - only the value of the parameter is changed
            - the value of the argument remains unchanged

    - Passing **References**: it is impossible to alter the original reference inside the method
        - however the method *can* alter the object that is being referenced by acting on the local reference
            - assignment operations on a reference do not alter the pointer of the original reference
                - only the reference of the parameter is changed
                - the reference of the argument remains unchanged
            - only method invocations on the reference alters the underlying object
                - parameter.method() will change the underlying object
                    - since the parameter reference points to the same object as the argument reference

- When a method returns (i.e. the method completes) primitives and references still hold the same value and point to the same object, respectively.

- NOTE: Strings are immutable
    - so they can never be changed when passed into a method

- Objects cannot be passed into a method or returned from a method
    - only references to objects can be passed into a method or returned from a method

1.7 Java Operators

- Order of precedence of operators is specified in the below order
  - operators are evaluated in this order
  - If operators have the same level of precedence then evaluation occurs from **left-to-right**
  - **Order of Precedence**

■

Operator-Type	**Operator-Symbol**
Post-increment, Post-decrement	expr++, expr--
Pre-increment, Pre-decrement	++expr, --expr
Unary operator	+ , -, ~, !
Multiplication, Division, Modulos	* , / , %
Addition, Subtraction	+ , -
Shift Operators	<< , >> , >>>
Relational tests	< , > , <= , >= , **instanceof**
Equality / Inequality tests	== , !=
Bitwise AND , XOR, OR	& , ^ , |
Logical AND, OR	&& , ||
Ternary	? :
Assignment operators	= , += , -= , *= , /= , %= , &= , ^= , |= , <<= , >>= , >>>=

1.7.1 The Assignment Operators

- Java has 12 assignment operators
  - 1 simple assignment operator: `=`
  - 11 compound assignment operators: `=+  -=  *=  /=` ... etc
- Assignment stores the result of the right-hand-side (**RHS**) expression into the variable on the left-hand-side (**LHS**)
- Assignments do not compile if the RHS cannot be converted to the data type of the LHS
  - when this occurs the RHS requires a **cast** to allow for **loss of data**
  - if a decimal value is downcast to a non-decimal value the decimal portion is truncated
- Compound Assignments

- perform the given operator first (e.g. addition or multiplication, etc)
- then the result is stored in the LHS variable
- Example:
  - `int x = 10;`
  - `int y = 9;`
  - `x *= y; // is the same as x = x * y;`
- can save the programmer from explicitly performing a required cast
  - Example:
    - `int x = 10;`
    - `float y = 9.0;`
    - `x = x * y; // compile error: a cast is required`
    - `x *= y ;  // no compile error, automatic downcast performed since compound assignment`

1.7.2 The Arithmetic Operators

- Arithmetic operators include
  - Additive operators : +, -
  - Multiplicative operators: *, /
  - Modulus operator: %
  - Increment/Decrement operators: ++, --

- Order of evaluation - the JVM ensures that evaluation occurs
  - in the order of precedence (see table above)
    - multiplicative operators have higher precedence than additive operators
  - from left-to-right - when precedence levels are at the *same level*
  - order of evaluation can be controlled using parentheses - ( ) - to group operators and operands
    - an expression within parentheses is evaluated before any external operands can be applied

- Casting
  - The sole purpose of casting to make the compiler happy about loss of precision
  - When dealing with primitives loss of precision occurs when a larger primitive type is assigned to a smaller primitive type

- Arithmetic promotion rules:
  - are applied when arithmetic operations are applied
    - this includes additive, multiplicative,
  - The smaller operand is automatically promoted to the larger operand
  - *ALL* operands smaller than **`int`** are promoted to **`int`**
    - and that the result of the operation will be an **`int`**
    - if assigning the result back to a primitive type smaller than an int the appropriate cast must be performed

- Additive Operators: **`+`** **`-`**
  - Can be evaluated on any primitive, except booleans
  - Can also be evaluated on Strings : results in string concatenation

- Arithmetic promotion rules apply
- Example
  - `short s1 = 10, s2  = 12;`
  - `boolean f = false;`
  - `short s3 = 10 + f;  // does not compile! - booleans cannot be treated as integer types`
  - `String x = s1 + s2 + "Hello";  // result x = "22Hello" - string concatenation`
  - `short sum = s1 + s2 ;  // does not compile! - loss of precision, must be cast to short due to auto-promotion to int`
  - `short result = (short) (s1+s2); // compiles - since cast to short is performed`

- Multiplicative Operators: `* / %`
  - Have a higher order of precedence than additive operators
  - Can be applied only to numeric primitive types (including **chars**)
  - Arithmetic promotion rules apply
  - We have to be careful with multiplicative operators, result types and assigned types
  - Example:
    - `int a = 26, b = 5;`
    - `double div = a / b; // result = 5.0 and not 5.2`
    - `// first step : integer division 26 / 5 - which results in 5`
    - `// second step : assignment to double type - 5 (an int) is converted to a double, resulting in 5.0`
    - `float c = 5;`
    - `double div2 = a / c ; // result = 5.2 as expected`
    - `// first step: a is promoted to float, since float is the larger primitive`
    - `// second step: floating-point arithmetic division takes place resulting in 5.2`
    - `// third step: assignment to a float from a result of type float`
  - Modulus Operator %
    - aka remainder operator
    - calculates the remainder of two numeric operands (which includes **chars**) when they are divided
    - Modulus rules
      - If the first operand is negative then the result is negative
      - floating point operands can be used as well - the result is calculated by
        - taking away the 2nd operand from the 1st until a value smaller than the 2nd operand remains
    - Example
      - `int x = 12 % 5 ;  // x = 2`
      - `int y = -17 %4 ;  // y = -1`
      - `float f = 12.4 % 3.2  // f = 2.8`
- Increment/Decrement Operators : `++ --`

- These operators increase or decrease the value of the operand by 1
- Can only be applied to numeric operands
- The result is the same data type as the operand
- Come in two flavours - **postfix** and **prefix**
  - **postfix**
    - the result of the increment/decrement is calculated AFTER the current operation is evaluated
  - **prefix**
    - the result of the increment/decrement is calculated BEFORE the current operation is evaluated
  - Example:

```
int x = 5;
int y = x++;  // y = 5, x = 6
int z = ++x;  // z = 7, x = 7
int a = y-- * 3 / --y ; // multiplication expression occurs first
                        // since the decrement on y is post-decrement
                        //           the multiplication (5 * 3 = 15) is performed before the decrement
                        //           the post-decremented is performed : y = 4
                        // division expression is evaluated next
                        // since the decrement on y is pre-decrement
                        //           the pre-decrement is performed before the division : y = 3
                        //           the division (15 / 3 = 5) is performed after the decrement
```

1.7.3 The Relational Operators

- 4 relational (comparison) operators exist : `<  <=  >=  >`
- Can only be performed on numeric primitive types (including **`char`**s)
- The result type is always a **`boolean`**
- Arithmetic promotion applies : smaller operands are promoted to larger operand before the comparison takes place
- Since the result is always a boolean the result of a relational comparison cannot be assigned to a non-boolean type
  - In Java the `boolean` primitive type is not compatible with the `int` primitive type
    - the `boolean` type can never be treated as an `int` and vice-versa!
- Examples:
  - int x = 10, y = 20, z = 10;
  - System.out.println(x < y); // true is printed
  - System.out.println(x <= y); // true is printed
  - System.out.println(x > z); // false is printed
  - System.out.println(x >= z); // true is printed

- int comp = x < y ; // compile error! - since ints and booleans are not compatible

1.7.4 The *instanceof* Operator

- Compares a reference to a class data type, or an interface data type
  - the result is always a boolean
  - the result is **true** iff the reference is
    - the exact type of the stated class or interface name
    - a subtype of the stated class of interface name
  - The underlying object of the reference is always compared and not the type of the reference variable
  - NOTE: we will look at this later more in-depth

- The syntax is
  - *referenceVariable* `instanceof` *ClassOrInterfaceName*
  - NOTE: `instanceof` is all in lowercase

- Main Usages:
  - When you cast a reference to a possible subclass type it is best to check that the underlying object is an instance of the cast type
    - this is done to avoid a **`ClassCastException`**

- Example:
  - `Object x = new String("20 May 2010");`
  - `Object y = new Date();`
  - `System.out.println(x instanceof Date); // false is printed since x refers to a String object`
  - `System.out.println(y instanceof Date); // true is printed since y refers to a Date object`

1.7.5 Bitwise and Logical Operators

- Bitwise Operators : **`& ^ |`**
  - only when operating on numeric integer operands
    - valid only on **`byte char short int long`** primitive types
      - the result is computed on the binary representation of the operands
    - ==> invalid on **`float`** and **`double`** primitive types
  - when operating on **`boolean`** operands they act as logical operators (see below)
  - perform bitwise AND, XOR and OR functions, respectively, on numeric operands using their binary representation
    - AND : is 1 IFF both bits are 1
    - XOR : is 1 IFF both bits are different
    - OR : is 0 IFF both bits are 0
  - Arithmetic promotion rules apply to all valid numeric operands
  - NOTE: order of precedence for bitwise operators is the **`&`** then **`^`** then **`|`**
- Logical Operators: **`& ^ | && ||`**
  - **`& ^ |`** become logical operators when dealing with **`boolean`** operands or expressions

- perform logical AND, XOR and OR functions, respectively, on boolean operands and expressions
  - AND : is true IFF both operands are true
  - XOR : is true IFF both operands are different
  - OR : is false IFF both operands are false
- `&& ||` are **short-circuit** operators
  - short-circuit operators will stop evaluating the expression if it can determine the result from the first part of the expression
    - i.e. from the part before **`&&`** or **`||`**
    - this is useful if you want to take care before evaluating the second part of an expression by using **`&&`**
      - e.g. when dividing we can check that the denominator is not 0 (zero)
        - Example: `(c != 0) && (a/c > 1)`
      - e.g. check for a null before de-referencing an object reference
        - Example: `(`objRef `!= null) && (`objRef`.method())`
    - the non-short-circuit operators can be *dangerous* because the second half of the expression will evaluate irrespective of the first half result

1.7.6 Conditional Operator

- denoted by **`? :`** character sequence
- aka ternary operator because it uses 3 operands
- it is a condensed if-else sequence
- Syntax:
  - **`boolean_expression ? true_expression : false_expression ;`**
    - where
      - *boolean_expression* must evaluate to true or false
        - if true then the *true_expression* is evaluated
        - if false then the *false_expression* is evaluated
      - *true_expression* and *false_expression* must return a value (of any type)
- Example:
  - `double d = 0.36;`
  - `System.out.println( d > 0 && d < 1 ? d*= 100 : "not a percent");`

1.7.7 Equality Operator

- Is denoted by `==` `!=`
  - and are known as the equality operators

- Can be used in 3 different scenarios
  1. the 2 operands are numerical primitive types
  2. the 2 operands are boolean types
  3. the 2 operands are reference or null types

- Cannot mix the different types
  - a compile time error occurs : "incomparable types"
  - e.g. cannot compare a boolean with a reference or an int

- The 2 operands must be compatible
  - if one type is larger than the first then the second is promoted before the comparison

- When comparing 2 non-boolean primitives true is returned if their values are the equal after arithmetic promotion takes place
  - Example
    - `int myInt = 32;`
    - `float myFloat = 32.0f;`
    - `System.out.println( myInt == myFloat ); // true is printed since myInt and myFloat are promoted to floats that represent 32.0`

- When comparing 2 references types with the equality operators it is important to know that
  - the comparison DOES NOT check for object equality
  - it only checks to see if the references point to the SAME object
    - i.e. returns **`true`** IFF the operands point to the same object
  - Example:
    - `String s1 = "Hello";`
    - `String s2 = "Hello";`
    - `String x3 = new String("Goodbye");`
    - `String x4 = new String("Goodbye");`
    - `System.out.println( s1 == s2 ); // prints true since JVM creates only 1 string object and re-uses it`
    - `System.out.println( x3 == x4 ); // prints false since two independent objects have been created`

1.8 Equality of Objects

- In Java the programmer gets to decide if 2 objects are considered equal!
  - This is done by overriding the **`equals(Object obj)`** method from `java.lang.Object`
  - This method has the following signature:
    - `public boolean equals(Object object)`

- java.lang.Object provides a default implementation of the equals() method
  - it does NOT check object equality, ONLY reference equality
    - i.e. `this == object`

- General rule of thumb is to provide an **`equals()`** method in your classes where you want to define what it means for two objects of that class to be equal

- The **`equals()`** method should look like the following, based on a class called **`MyObject`**
  - `public boolean equals(Object obj) // note that the parameter is of type Object and not MyObject`

```
{
    if( this == obj )
    {
        return true ; // since we are comparing with the same
        object
    }

    if( ! obj instanceof MyObject )
    {
        return false; // since comparing different object types
    }

    MyObject myObj = (MyObject) obj; // cast to MyObject class type
    in order to perform member comparisons

    // perform member data comparisons
    if( this.primitiveDataMember1 == myObj.primtiveDataMember1 &&
        this.objectDataMember1.equals( myObj.objectDataMember1 ) &&
        ... other data member comparisons as appropriate ...
      )
    {
        return true;
    }

    return false;
}
```

- Since the **`equals()`** object is defined in java.lang.Object we can invoke a call to equals() on any object
  - and pass in any other object
  - When comparing objects of different types the result will always be false

- The **`hashCode()`** method
  - This method should also be considered for overriding when overriding the **`equals()`** method
  - It has the following signature
    - `public int hashCode()`
  - It is used by hash table data structures to determine ordering of data elements and data element equality
  - **`hashCode()`** and **`equals()`** methods are related in the sense that 2 objects that are equal should return the same hash code

END-OF-CHAPTER : Notes complete for Chapter 1.

ANSWERS TO CHAPTER QUIZ
1. D - since class Leaf is not declared before class Plant
2. E - since classpath option should not contain '=' character and classpath should point to the root of the package directory

3. E - since the compilation command specifies a build directory and the class is in package a.b.c
4. A - eventhough equals() signature does not override equals from java.lang.Object
5. A
6. C
7. B, D
8.