COMPILE TIME AND RUNTIME

Compile time and runtime are two distinctly different times during the active life of a computer program. ***Compile time*** is when the program is compiled; ***runtime*** is when it executes (on either a physical or virtual computer).

Programmers use the term ***static*** to refer to anything that is created during compile time and stays fixed during the program run. They use the term ***dynamic*** to refer to things that are created and can change during execution.

Example
In application **`MyApp`** shown below, the data type of variable **`tax`** is statically set to **`double`** for the entire program run. Its value is dynamically set when the assignment statement (marked **`**`**) executes.

```
     public class MyApp
     {
        public static void main( String [] args )
        {
           double tax;
           . . .
**         tax = price * TAX_RATE;
           . . .
        }
     }
```

Compiler Tasks

The main tasks of the compiler are to:

- Check program statements for errors and report them.
- Generate the machine instructions that carry out the operations specified by the program.

To do this, the compiler must gather as much information as possible about the items (e.g. variables, classes, objects, etc.) used in the program.

Example
Consider this Java assignment statement:

```
          x = y2 + z;
```

To determine if it is correct, the compiler needs to know if `y2` is a declared variable (perhaps the programmer meant to type `y*2`). To generate correct bytecode, the compiler needs to know the data types of the variables (integer addition is a different machine instruction than floating-point addition).

Declarations

Program text that is primarily meant to provide information to the compiler is called a ***declaration***, which the compiler processes by collecting the information given.

Example
A variable declaration. It tells the compiler that the identifier `mile` refers to a `double` variable.

```
double mile;
```

Example
A class declaration. It tells the compiler that the identifier `Pixel` refers to a class and, furthermore, that every `Pixel` object has fields `x`, `y` and `hue`.

```
public class Pixel
{
   public int x, y;
   public Color hue;
}
```

Example
A method declaration. It tells the compiler that the identifier `pay` refers to a method and, furthermore, that this method returns the product of two arguments.

```
public double pay( double wage, int hours )
{
   return wage * hours;
}
```

Declarations allow the compiler to find errors in subsequent statements.

Example
Because of the information in the first line, the compiler can flag the third line as an error.

```
int x;
. . .
x = 2.5;  // ERROR
```

Declarations also allow the compiler to generate proper machine code.

Example
Because of the declarations, the compiler knows the **+** on the left must be translated to string concatenation and that on the right to integer addition.

`String a, b, c;`
`. . .`
`a = b + c;`
	`int a, b, c;`
	`. . .`
	`a = b + c;`

Executable Statements

Program statements that are primarily meant to specify operations that the computer is to carry out at run-time are called ***executable statements***, which the compiler processes by translating them to machine code.

Examples

`mpg = miles / gallons;`	An executable statement that directs the JVM to fetch the value of `miles` and that of `gallons`, divide them and store the result into `mpg`.
`System.out.println( "equals" );`	An executable statement that directs the JVM to send the string `equals` to the standard output object.

Housekeeping

Housekeeping refers to tasks that the compiler and JVM must perform to support the translation and execution of a computer program. Housekeeping is not usually explicit in the Java program; you've got to infer it from knowing how Java constructs are compiled and executed.

Example

Consider these Java statements:

```
1  double x, y;
2  y = 3.5;
3  x = 2 * y;
```

Line 1 results in this inexplicit processing: (1) the compiler does its own housekeeping by gathering information about `x` and `y`, (2) it generates housekeeping code, which (3) the JVM executes at runtime.

Line	Compile Time	Runtime
1	Compiler performs housekeeping by recording identifiers `x` and `y` as `double` variables	
	Compiler performs housekeeping by recording available addresses (say *x* and *y*) in internal memory in which to keep `x` and `y`	
	Compiler generates housekeeping code to reserve memory addresses *x* and *y*	JVM executes housekeeping code reserving memory addresses *x* and *y*
2	Compiler generates code to store 3.5 into memory address *y*	JVM executes code storing 3.5 into memory address *y*
3	Compiler generates code to fetch value from memory address *y*	JVM executes code fetching the value from memory address *y*
	Compiler generates code to multiply this value by 2	JVM executes code multiplying this value by 2
	Compiler generates code to store result into memory address *x*	JVM executes code storing the result into memory address *x*