02: ♥♦ Java Compile-time Vs Run-time Interview Q&As

java-success.com/compile-time-vs-run-time-basics-interview-qa/

Arulkumaran Kumaraswamipillai 9/17/2014

During development and design, one needs to think in terms of **compile-time, run-time**, and **build-time**. It will also help you understand the fundamentals better. These are beginner to intermediate level questions.

Q1. What is the difference between line A & line B in the following code snippet?

```
public class ConstantFolding {
   static final  int number1 = 5;
   static final  int number2 = 6;
   static int number3 = 5;
   static int number4= 6;

   public static void main(String[ ] args) {
        int product1 = number1 * number2;            //line A
        int product2 = number3 * number4;            //line B
   }
}

```

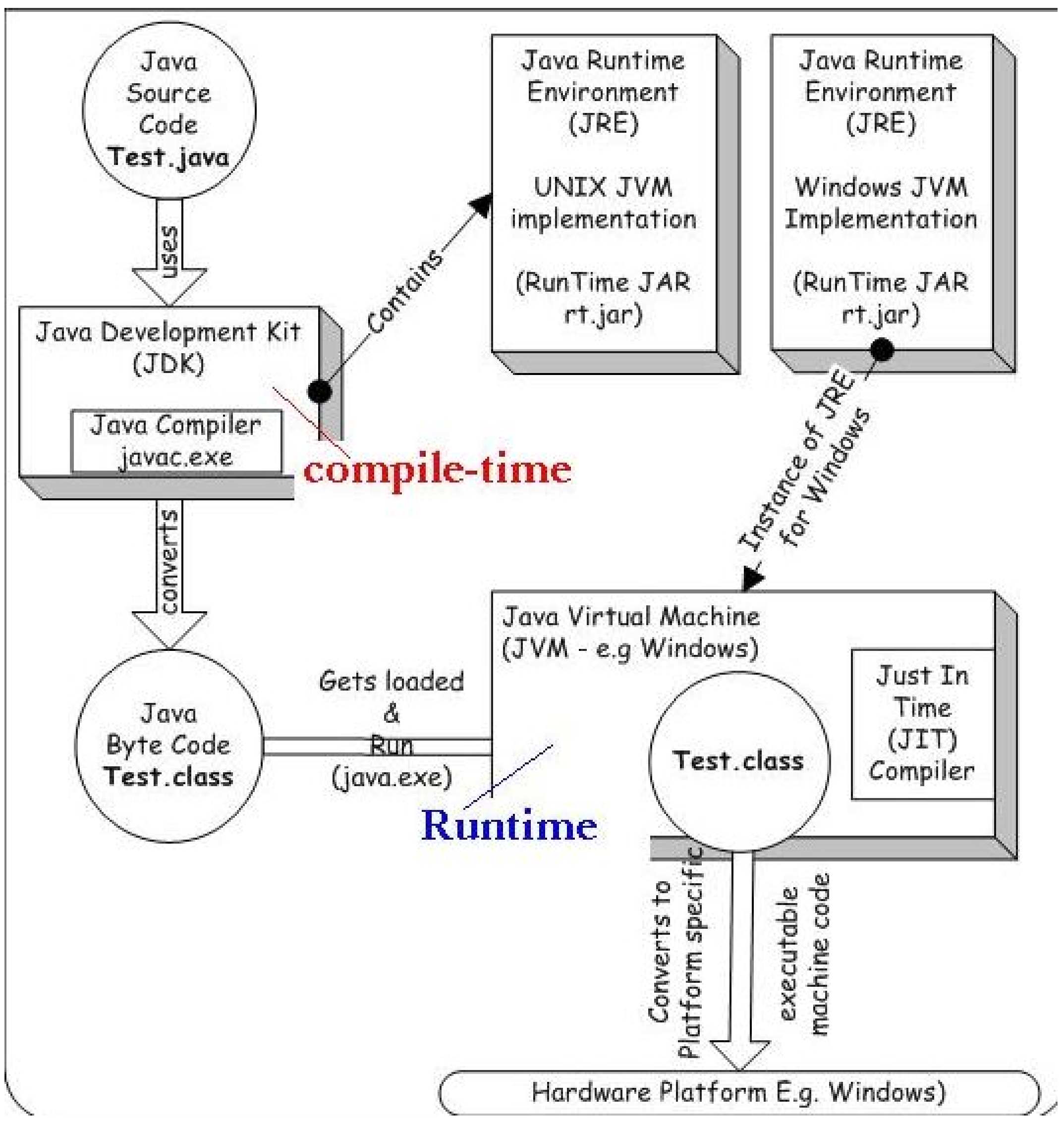
A1. Line A, evaluates the product at **compile-time**, and Line B evaluates the product at **runtime**. If you use a Java Decompiler (e.g. jd-gui), and decompile the compiled *ConstantFolding.class* file, you will see whyas shown below.

```
public class ConstantFolding
{
  static final int number1 = 5;
  static final int number2 = 6;
  static int number3 = 5;
  static int number4 = 6;

  public static void main(String[ ] args)
  {
     int product1 = 30;
     int product2 = number3 * number4;
  }
}

```

Constant folding is an optimization technique used by the Java compiler. Since final variables cannot change, they can be optimized. Java Decompiler and javap command are handy tool for inspecting the compiled (i.e. byte code) code.

Q2 Can you think of other scenarios other than code optimization, where inspecting a compiled code is useful?
A2Generics in Java are compile-time constructs, and it is very handy to inspect a compiled class file to understand and troubleshoot generics.

Q3.Does this happen during compile-time, runtime, or both?
A3.


Java Source Code Test.java
uses
Java Development Kit (JDK)
Java Compiler javac.exe
Contains
Java Runtime Environment (JRE) UNIX JVM implementation (RunTime JAR rt.jar)
Java Runtime Environment (JRE) Windows JVM Implementation (RunTime JAR rt.jar)
compile-time
Instance of JRE for Windows
converts
Java Byte Code Test.class
Gets loaded & Run (java.exe)
Runtime
Java Virtual Machine (JVM - e.g Windows)
Test.class
Just In Time (JIT) Compiler
Converts to Platform specific
executable machine code
Hardware Platform E.g. Windows)

compile-time Vs run-time

Method overloading: This happens at compile-time. This is also called compile-time polymorphism because the compiler must decide how to select which method to run based on the data types of the arguments.

```
public class {
    public static void evaluate(String param1);  // method #1
    public static void evaluate(int param1);     // method #2
}
```

If the compiler were to compile the statement:

```
evaluate(“My Test Argument passed to param1”);
```

it could see that the argument was a string literal, and generate byte code that called method #1.

Method overriding: This happens at runtime. This is also called runtime polymorphism because the compiler does not and cannot know which method to call. Instead, the JVM must make the determination while the code is running.

```
public class A {
  public int compute(int input) {          //method #3
      return 3 * input;
  }
}

public class B extends A {
  @Override
  public int compute(int input) {          //method #4
      return 4 * input;
  }
}

```

The method compute(..) in subclass "B" overrides the method compute(..) in super class "A". If the compiler has to compile the following method,

```
public int evaluate(A reference, int arg2)  {
    int result = reference.compute(arg2);
}
```

The compiler would not know whether the input argument 'reference' is of type "A" or type "B". This must be determined during runtime whether to call method #3 or method #4 depending on what type of object (i.e. instance of Class A or instance of Class B) is assigned to input variable "reference".

Generics (aka type checking): This happens at compile-time. The compiler checks for the type correctness of the program and translates or rewrites the code that uses generics into non-generic code that can be executed in the current JVM. This technique is known as "type erasure". In other words, the compiler erases all generic type information contained within the angle brackets to achieve backward compatibility with JRE 1.4.0 or earlier editions.

```
List<String> myList = new ArrayList<String>(10);
```

after compilation becomes:

```
List myList = new ArrayList(10);
```

Annotations: You can have either run-time or compile-time annotations.

```
public class B extends A {
  @Override
   public int compute(int input){      //method #4
       return 4 * input;
   }
}
```

@Override is a simple compile-time annotation to catch little mistakes like typing tostring( ) instead of toString( ) in a subclass. User defined annotations can be processed at compile-time using the Annotation Processing Tool (APT) that comes with Java 5. In Java 6, this is included as part of the compiler itself.

```
public class MyTest{
   @Test
    public void testEmptyness( ){
        org.junit.Assert.assertTrue(getList( ).isEmpty( ));
    }

    private List getList( ){
       //implemenation goes here
    }
}
```

@Test is an annotation that JUnit framework uses at runtime with the help of reflection to determine which method(s) to execute within a test class.

```
@Test (timeout=100)
public void testTimeout( ) {
    while(true);   //infinite loop
}
```

The above test fails if it takes more than 100ms to execute at runtime.

```
@Test (expected=IndexOutOfBoundsException.class)
public void testOutOfBounds( ) {
      new ArrayList<Object>( ).get(1);
}
```

The above code fails if it does not throw IndexOutOfBoundsException or if it throws a different exception at runtime. User defined annotations can be processed at runtime using the new AnnotatedElement and "Annotation" element interfaces added to the Java reflection API.

Exceptions: You can have either runtime or compile-time exceptions.

RuntimeException is also known as the unchecked exception indicating not required to be checked by the compiler. RuntimeException is the superclass of those exceptions that can be thrown during the execution of a program within the JVM. A method is not required to declare in its throws clause any subclasses of RuntimeException that might be thrown during the execution of a method but not caught.

Example: *NullPointerException*, *ArrayIndexOutOfBoundsException*, etc

Checked exceptions are verified by the compiler at compile-time that a program contains handlers like throws clause or try{} catch{} blocks for handling the checked exceptions, by analyzing which checked exceptions can result from execution of a method or constructor.

Aspect Oriented Programming (AOP): Aspects can be weaved at compile-time, post-compile time, load-time or runtime.

- **Compile-time:** weaving is the simplest approach. When you have the source code for an application, the AOP compiler (e.g. ajc – AspectJ Compiler) will compile from source and produce woven class files as output. The invocation of the weaver is integral to the AOP compilation process. The aspects themselves may be in source or binary form. If the aspects are required for the affected classes to compile, then you must weave at compile-time. ?

- **Post-compile:** weaving is also sometimes called binary weaving, and is used to weave existing class files and JAR files. As with compile-time weaving, the aspects used for weaving may be in source or binary form, and may themselves be woven by aspects.?
- **Load-time:** weaving is simply binary weaving deferred until the point that a class loader loads a class file and defines the class to the JVM. To support this, one or more “weaving class loaders”, either provided explicitly by the run-time environment or enabled through a “weaving agent” are required.
- **Runtime:** weaving of classes that have already been loaded to the JVM.

Inheritance – happens at compile-time, hence is static.
Delegation or composition – happens at run-time, hence is dynamic and more flexible.

Q. Have you heard the term “**composition should be favored over inheritance**“? If yes, what do you understand by this phrase?
A. Inheritance is a polymorphic tool and is not a code reuse tool. Some developers tend to use inheritance for code reuse when there is no polymorphic relationship. The guide is that inheritance should be only used when a subclass ‘is a’ super class.

- Don’t use inheritance just to get code reuse. If there is no ‘is a’ relationship then use composition for code reuse. Overuse of implementation inheritance (uses the “extends” key word) can break all the subclasses, if the super class is modified. This is due to **tight coupling** occurring between the parent and the child classes happening at **compile time**.
- Do not use inheritance just to get polymorphism. If there is no ‘is a’ relationship and all you want is polymorphism then use interface inheritance with composition, which gives you code reuse and **runtime** flexibility.

This is the reason why the GoF (Gang of Four) design patterns favor composition over inheritance. The interviewer will be looking for the key terms — “**coupling**“, “**static versus dynamic**” and “**happens at compile-time vs runtime**” in your answers.The runtime flexibility is achieved in composition as the classes can be composed **dynamically** at runtime either conditionally based on an outcome or unconditionally. Whereas an inheritance is **static**.

Q4. Can you differentiate compile-time inheritance and runtime inheritance with examples and specify which Java supports?
A4.

The term “inheritance” refers to a situation where behaviors and attributes are passed on from one object to another. The Java programming language natively only supports compile-time inheritance through subclassing as shown below with the keyword “extends”.

```
public class Parent {
   public String saySomething( ) {
        return “Parent is called”;
   }
}

```

```
public class Child extends Parent {
    @Override
    public String saySomething( ) {
        return super.saySomething( ) +  ", Child is called";
    }
}
```

A call to saySomething( ) method on the class "Child" will return "Parent is called, Child is called" because the Child class inherits "Parent is called" from the class Parent. The keyword "super" is used to call the method on the "Parent" class. Runtime inheritance refers to the ability to construct the parent/child hierarchy at runtime. Java does not natively support runtime inheritance, but there is an alternative concept known as "delegation" or "composition", which refers to constructing a hierarchy of object instances at runtime. This allows you to simulate runtime inheritance. In Java, delegation is typically achieved as shown below:

```
public class Parent {
    public String saySomething( ) {
        return "Parent is called";
    }
}
```

```
public class Child  {
    public String saySomething( ) {
        return new Parent( ).saySomething( ) +  ", Child is called";
    }
}
```

The Child class delegates the call to the Parent class. Composition can be achieved as follows:

```
public class Child  {
    private Parent parent = null;

    public Child( ){
        this.parent = new Parent( );
    }

    public String saySomething( ) {
        return this.parent.saySomething( ) +  ", Child is called";
    }
}
```