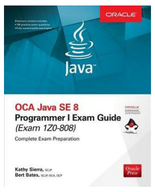

ORACLE
Java
OCA Java SE 8
Programmer I Exam Guide
(Exam 1Z0-808)
Complete Exam Preparation
Kathy Sierra
Bert Bates

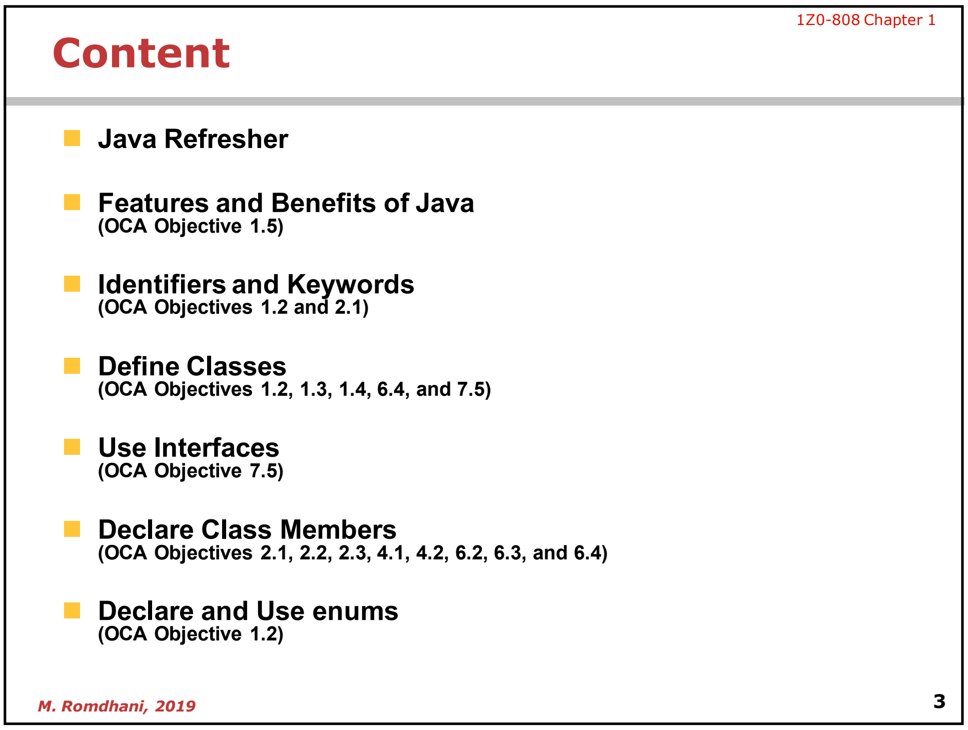
Chapter 1

Declarations and Access Control

Business Training

Content

- Java Refresher
- Features and Benefits of Java
  (OCA Objective 1.5)
- Identifiers and Keywords
  (OCA Objectives 1.2 and 2.1)
- Define Classes
  (OCA Objectives 1.2, 1.3, 1.4, 6.4, and 7.5)
- Use Interfaces
  (OCA Objective 7.5)
- Declare Class Members
  (OCA Objectives 2.1, 2.2, 2.3, 4.1, 4.2, 6.2, 6.3, and 6.4)
- Declare and Use enums
  (OCA Objective 1.2)



Content

- **Java Refresher**
- **Features and Benefits of Java**
  (OCA Objective 1.5)
- **Identifiers and Keywords**
  (OCA Objectives 1.2 and 2.1)
- **Define Classes**
  (OCA Objectives 1.2, 1.3, 1.4, 6.4, and 7.5)
- **Use Interfaces**
  (OCA Objective 7.5)
- **Declare Class Members**
  (OCA Objectives 2.1, 2.2, 2.3, 4.1, 4.2, 6.2, 6.3, and 6.4)
- **Declare and Use enums**
  (OCA Objective 1.2)

Java Refresher

Java is an OO language

- A Java program is mostly a collection of *objects* talking to other objects by invoking each other's *methods*. Every object is of a certain *type*, and that type is defined by a *class* or an *interface*. Most Java programs use a collection of objects of many different types.
- Following is a list of a few useful terms for this object-oriented (OO) language:
  - **Class** A template that describes the kinds of state and behavior that objects of its type support.
  - **Object** At runtime, when the Java Virtual Machine (JVM) encounters the newkeyword, it will use the appropriate class to make an object that is an instance of that class. That object will have its own *state* and access to all of the behaviors defined by its class.
    - **State (instance variables)** Each object (instance of a class) will have its own unique set of instance variables as defined in the class. Collectively, the values assigned to an object's instance variables make up the object's *state*.
    - **Behavior (methods)** When a programmer creates a class, she creates methods for that class. Methods are where the class's logic is stored and where the real work gets done. They are where algorithms get executed and data gets manipulated.

Java is an OO language

- **Identifiers and Keywords** All the Java components we just talked about—classes, variables, and methods—need names. In Java, these names are called *identifiers*, and, as you might expect, there are rules for what constitutes a legal Java identifier.
- **Inheritance** Central to Java and other OO languages is the concept of *inheritance*, which allows code defined in one class or interface to be reused in other classes. In Java, you can define a general (more abstract) *superclass* and then extend it with more specific *subclasses*.
- **Interfaces** A powerful companion to inheritance is the use of interfaces. Interfaces are *usually*like a 100 percent abstract superclass that defines the methods a subclass must support, but not *how* they must be supported.

Features anf Benefits of Java
(OCA Objective 1.5)

Benefits of Java

- Object oriented
- Encapsulation
- Memory management
- Huge library
- Secure by design
- Write once, run anywhere (cross-platform execution)
- Strongly typed
- Multithreaded
- Distributed computing

Identifiers and Keywords
(OCA Objective 1.5)

1Z0-808 Chapter 1

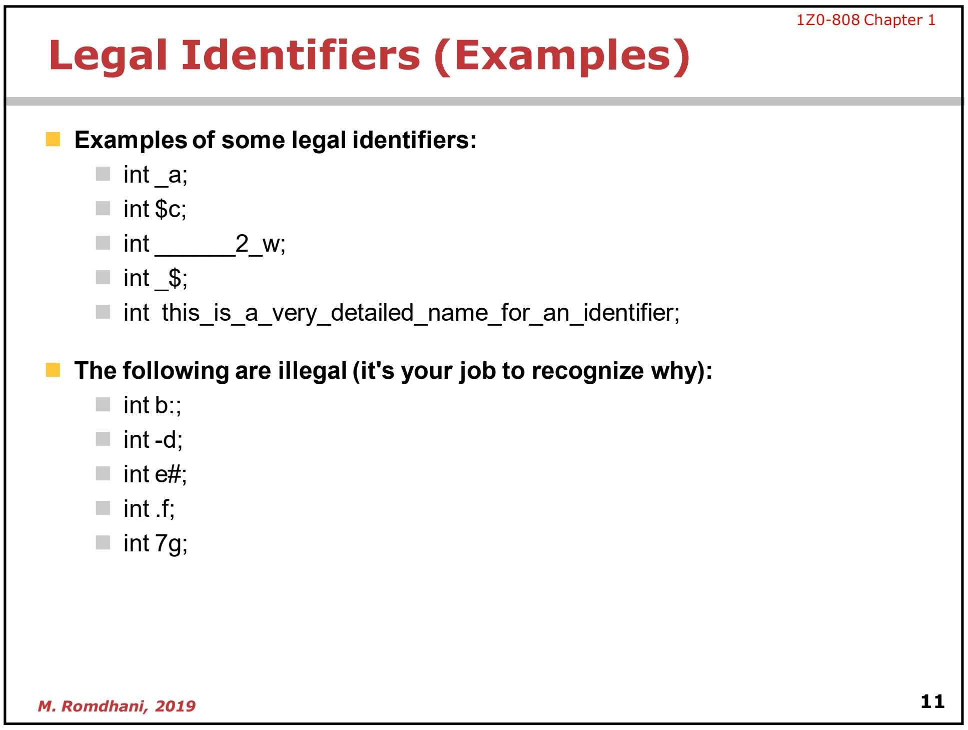
Legal Idenfiers

- **Identifiers must start with a letter, a currency character ($), or a connecting character such as the underscore (_).**
  - Identifiers cannot start with a number !
  - After the first character, identifiers can contain any combination of letters, currency characters, connecting characters, or numbers.
  - In practice, there is no limit to the number of characters an identifier can contain.
- **Identifiers in Java are case-sensitive: foo and Foo are two different identifiers.**
- **You can't use a Java keyword as an identifier.**
  - Java 8 Keywords

abstract	boolean	break	byte	case	catch
char	class	const	continue	default	do
double	else	extends	final	finally	float
for	goto	if	implements	import	instanceof
int	interface	long	native	new	package
private	protected	public	return	short	static
strictfp	super	switch	synchronized	this	throw
throws	transient	try	void	volatile	while
assert	enum				

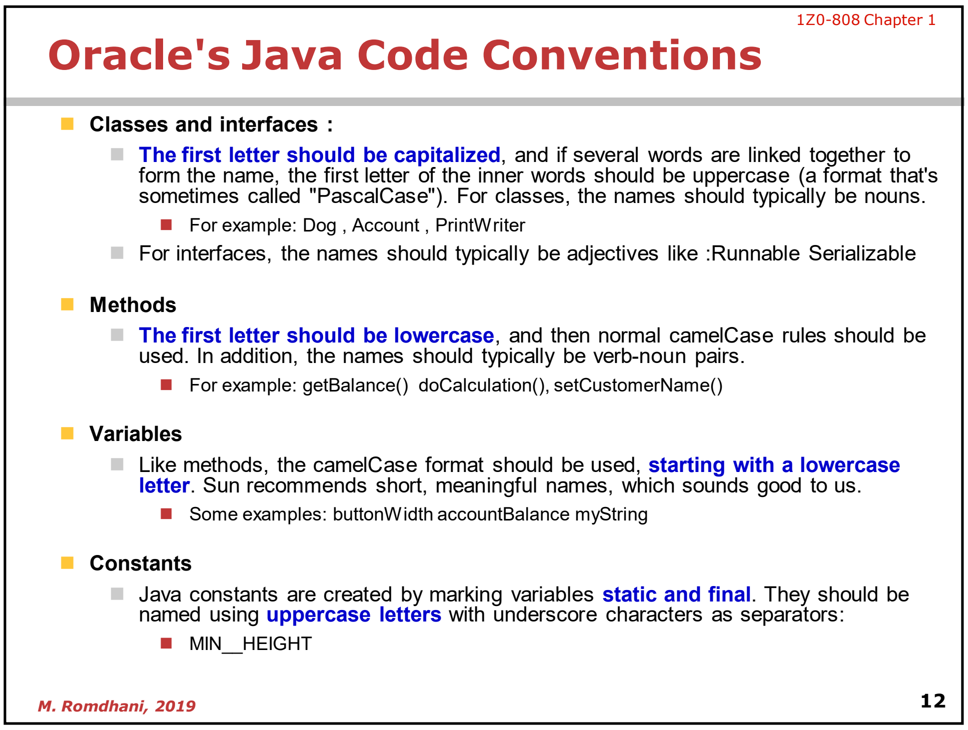
M. Romdhani, 2019

10



Legal Identifiers (Examples)

- **Examples of some legal identifiers:**
  - int _a;
  - int $c;
  - int ______2_w;
  - int _$;
  - int this_is_a_very_detailed_name_for_an_identifier;
- **The following are illegal (it's your job to recognize why):**
  - int b:;
  - int -d;
  - int e#;
  - int .f;
  - int 7g;



Oracle's Java Code Conventions

- **Classes and interfaces :**
  - **The first letter should be capitalized**, and if several words are linked together to form the name, the first letter of the inner words should be uppercase (a format that's sometimes called "PascalCase"). For classes, the names should typically be nouns.
    - For example: Dog , Account , PrintWriter
  - For interfaces, the names should typically be adjectives like :Runnable Serializable
- **Methods**
  - **The first letter should be lowercase**, and then normal camelCase rules should be used. In addition, the names should typically be verb-noun pairs.
    - For example: getBalance() doCalculation(), setCustomerName()
- **Variables**
  - Like methods, the camelCase format should be used, **starting with a lowercase letter**. Sun recommends short, meaningful names, which sounds good to us.
    - Some examples: buttonWidth accountBalance myString
- **Constants**
  - Java constants are created by marking variables **static and final**. They should be named using **uppercase letters** with underscore characters as separators:
    - MIN__HEIGHT

JavaBeans Standards

- **First, JavaBeans are Java classes that have *properties*.**
  - For our purposes, think of properties as private instance variables.
  - The methods that change a property's value are called *setter* methods, and the methods that retrieve a property's value are called *getter* methods.
- **JavaBean Property Naming Rules**
  - If the property is not a boolean, **the getter method's prefix must be *get***.
    - If the property is a boolean, the getter method's prefix is either get or is. For example, getStopped() or isStopped() are both valid JavaBeans names for a boolean property.
  - **The setter method's prefix must be *set***.
  - To complete the name of a getter or setter method, change the first letter of the property name to uppercase, and then append it to the appropriate prefix (get, is, or set).
  - Setter method signatures must be marked public, with a void return type and an argument that represents the property type.
  - Getter method signatures must be marked public, take no arguments, and have a return type that matches the argument type of the setter method for that property.

Define Classes

(OCA Objectives 1.2, 1.3, 1.4, 6.4, and 7.5)

Source File Declaration Rules

- There can be only one public class per source code file.
- Comments can appear at the beginning or end of any line in the source code file; they are independent of any of the positioning rules discussed here.
- If there is a public class in a file, the name of the file must match the name of the public class.
- A file can have more than one non public class.
- Files with no public classes can have a name that does not match any of the classes in the file.
- If the class is part of a package, the package statement must be the first line in the source code file, before any import statements that may be present.
- If there are import statements, they must go between the package statement (if there is one) and the class declaration.
- import and package statements apply to all classes within a source code file. In other words, there's no way to declare multiple classes in a file and have them in different packages, or use different imports.

Using the javac and java Commands

- **Compiling with javac**
  - The javac command is used to invoke Java's compiler. You can specify many options when running javac. For example, there are options to generate debugging information or compiler warnings. Here's the structural overview for javac:
  - javac [options] [source files]
- **Launching Applications with java**
  - The java command is used to invoke the Java Virtual Machine (JVM). Here's the basic structure of the command:
  - java [options] class [args]

Using public static void main(String[ ] args)

- As far as the compiler and the JVM are concerned, the **only** version of main() with superpowers is the main() with this signature:
  - public static void main(String[] args)
  - Other versions of main() with other signatures are perfectly legal, but they're treated as normal methods.
- **The following are all legal declarations for the "special" main():**

```
static public void main(String[] args)
public static void main(String... x)
static public void main(String bang_a_gong[])
```

  - *For the OCA 8 exam, the only other thing that's important for you to know is that main() can be overloaded.*

Import Statements and the Java API

- **In order to use a class that belongs to a different package, use the keyword import**
  - In this example, we import the ArrayList class

```
import java.util.ArrayList;
public class MyClass {
  public static void main(String[] args) {
    ArrayList<String> a = new ArrayList<String>();
}}
```

  - You can add a wildcard character (*) to your import statement that means, "If you see a reference to a class you're not sure of, you can look through the entire package for that class," like so:

```
import java.util.*;
public class MyClass {
  public static void main(String[] args) {
    ArrayList<String> a = new ArrayList<String>();
    TreeSet<String> t = new TreeSet<String>();
  }
}
```

    - When the compiler and the JVM see this code, they'll know to look through java.util for ArrayList and TreeSet.
    - For the exam, the last thing you'll need to remember about using import statements in your classes is that you're free to mix and match.

Static Import Statements

- In Java 5 (a long time ago), the **import** statement was enhanced to provide even greater keystroke-reduction capabilities, although some would argue that this comes at the expense of readability. This feature is known as *static imports*.
  - Before static imports:

```
public class TestStatic {
  public static void main(String[] args) {
    System.out.println(Integer.MAX_VALUE);
    System.out.println(Integer.toHexString(42));
  }
}
```

  - After static imports:

```
import static java.lang.System.out;             // 1
import static java.lang.Integer.*;              // 2
public class TestStaticImport {
  public static void main(String[] args)  {
    out.println(MAX_VALUE);                     // 3
    out.println(toHexString(42));               // 4
  }
}
```

Class Declarations and Modifiers

- **In general, modifiers fall into two categories:**
  - **Access modifiers** (public, protected, private)
  - **Nonaccess modifiers** (including strictfp, final, and abstract)
- **Access control in Java is a little tricky because there are four access controls (levels of access) but only three access modifiers (private, protected, public).**
  - The fourth access control level (called default or package access) is what you get when you don't use any of the three access modifiers.
    - In other words, every class, method, and instance variable you declare has an access control, whether you explicitly type one or not. Although all four access controls (which means all three modifiers) work for most method and variable declarations, **a class can be declared with only public or default access**; the other two access control levels don't make sense for a class, as you'll see.
    - Note: As of Java 8, the word default can ALSO be used to declare certain methods in interfaces. When used in an interface's method declaration, default has a different meaning than the default access level.

Class Access

- **Default Access**
  - A class with default access has no modifier preceding it in the declaration! A class with default access **can be seen only by classes within the same package**.
- **Public Access**
  - A class declaration with the public keyword gives **all classes from all packages access to the public class**.

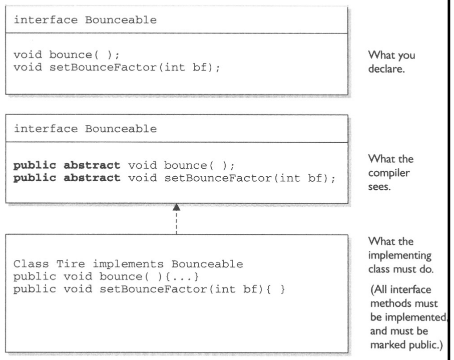
Other (Nonaccess) Class Modifiers

- **You can modify a class declaration using the keyword final, abstract, or strictfp.**
    - Marking a class as **strictfp** means that any method code in the class will conform to the IEEE 754 standard rules for floating points.
  - **Final Classes**
    - When used in a class declaration, the final keyword means the class can't be subclassed. In other words, no other class can ever extend (inherit from) a final class, and any attempts to do so will give you a compiler error.
  - **Abstract Classes**
    - An abstract class can never be instantiated, Its sole purpose is to be extended (subclassed).
- **You can't mark a class as both abstract and final. They have nearly opposite meanings.**

Declare Interfaces

Declaring an Interface

- **Think of an interface as a 100-percent abstract class. These rules are strict:**
  - **All interface methods are implicitly public and abstract.** You do not need to actually type the public or abstract modifiers in the method declaration, but the method is still always public and abstract.
  - **All variables defined in an interface must be public, static, and final—**in other words, interfaces can declare only constants, not instance variables.
  - **Interface methods must not be static**.
  - Because interface methods are abstract, they cannot be marked final, strictfp, or native.
  - An interface can extend **one or more other interfaces.**
  - An interface cannot extend anything but another interface.
  - **An interface cannot implement another interface or class.**
  - Interface types can be used polymorphically.



```
interface Bounceable

void bounce( );
void setBounceFactor(int bf);
```

```
interface Bounceable

public abstract void bounce( );
public abstract void setBounceFactor(int bf);
```

```
Class Tire implements Bounceable
public void bounce( ){...}
public void setBounceFactor(int bf){ }
```

What you declare.
What the compiler sees.
What the implementing class must do.
(All interface methods must be implemented, and must be marked public.)

Declaring Interface Constants

- **You need to remember one key rule for interface constants. They must always be public static final**
  - Because interface constants are defined in an interface, they don't have to be declared as public, static, or final. They must be public, static, and final, **but you don't have to actually declare them that way**.
    - Example :

```
interface Foo {
        int BAR = 42;
        void go();
}
```

- **For example, the following interface definitions that define constants are correct and arre all identical.**

```
public int x = 1;          // Looks non-static and non-final,
                           // but isn't!
int x = 1;                 // Looks default, non-final,
                           // non-static, but isn't!
static int x = 1;        // Doesn't show final or public
final int x = 1;         // Doesn't show static or public
public static int x = 1;       // Doesn't show final
public final int x = 1;        // Doesn't show static
static final int x = 1         // Doesn't show public
public static final int x = 1;  // what you get implicitly
```

Declaring default Interface Methods

- As of Java 8, interfaces can include inheritable* methods with concrete implementations. For now we'll just cover the simple declaration rules:
  - default methods are declared by using the **default** keyword.
    The defaultkeyword can be used only with interface method signatures, not class method signatures.
  - default methods are **public by definition**, and **the public modifier is optional**.
  - default methods **cannot be marked as private, protected, static, final, or abstract**.
  - default methods must have a concrete method body.
- **Here are some examples of legal and illegal default methods:**

```
interface TestDefault {
  default int m1(){return 1;} // legal
  public default void m2(){;} // legal
  static default void m3(){;} // illegal: default cannot be marked static
  default void m4();          // illegal: default must have a method body
}
```

Declaring static Interface Methods

- As of Java 8, interfaces can include **static** methods with concrete implementations.
  - static interface methods are declared by using the static keyword.
  - static interface methods are public by default, and the public modifier is optional.
  - static interface methods cannot be marked as private, protected, final, or abstract.
  - static interface methods must have a concrete method body.
  - When invoking a static interface method, the method's type (interface name) MUST be included in the invocation.
- **Here are some examples of legal and illegal static interface methods and their use:**

Declaring static Interface Methods

```
interface StaticIface {
  static int m1(){ return 42; }      // legal
  public static void m2(){ ; }       // legal
  // final static void m3(){ ; }     // illegal: final not allowed
  // abstract static void m4(){ ; }  // illegal: abstract not allowed
  // static void m5();               // illegal: needs a method body
}

public class TestSIF implements StaticIface {
  public static void main(String[] args) {
    System.out.println(StaticIface.m1());   // legal: m1()'s type
                                            // must be included
    new TestSIF().go();
    // System.out.println(m1());     // illegal: reference to interface
                                     // is  required
  }
  void go() {
    System.out.println(StaticIface.m1());  // also legal from an instance
  }
}
```

- which produces this output:

```
42
42
```

Declare Class Members
(OCA Objectives 2.1, 2.2, 2.3, 4.1, 4.2, 6.2, 6.3, and 6.4)

Access Modifiers

- **Members can use all four:**
    - public , protected , default, private
  - Public Members
    - When a method or variable member is declared public, it means all other classes, regardless of the package they belong to, can access the member (assuming the class itself is visible).
  - Private Members
    - Members marked private can't be accessed by code in any class other than the class in which the private member was declared.
  - Protected and Default Members
    - The protected and default access control levels are almost identical, but with one critical difference. **A default member may be accessed only if the class accessing the member belongs to the same package, whereas a protected member can be accessed (through inheritance) by a subclass even if the subclass is in a different package**. **Local Variables and Access Modifiers**
    - Can access modifiers be applied to local variables? NO!
- **Local Variables and Access Modifiers**
  - Can access modifiers be applied to local variables? NO!

Access Modifiers

- This table shws all the combinations of access and visibility

Visibility	Public	Protected	*Default*	Private
From the same class	Yes	Yes	Yes	Yes
From any class in the same package	Yes	Yes	Yes	No
From a subclass in the same package	Yes	Yes	Yes	No
From a subclass outside the same package	Yes	Yes, *through inheritance*	No	No
From any nonsubclass class outside the package	Yes	No	No	No

Nonaccess Member Modifiers

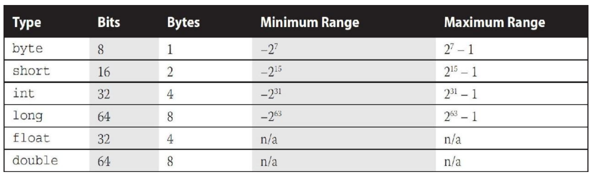
- **Final Methods**
  - The final keyword prevents a method from being overridden in a subclass, and is often used to enforce the API functionality of a method.
  - Final Arguments : A final argument must keep the same value that the parameter had when it was passed into the method.
- **Abstract Methods**
  - You mark a method abstract when you want to force subclasses to provide the implementation.
- **Synchronized Methods**
  - The synchronized keyword indicates that a method can be accessed by only one thread at a time.
- **Native Methods**
  - The native modifier indicates that a method is implemented in platform-depen dent code, often in C.
- **Strictfp Methods**
  - strictfp forces floating points to adhere to the IEEE 754 standard
- **Methods with Variable Argument Lists (var-args)**
  - Java 5 allows you to create methods that can take a variable number of arguments.
  - **The var-arg must be the last parameter in the method's signature, and you can have only one var-arg in a method.**

Constructor Declarations

- **Every class has a constructor, although if you don't create one explicitly, the compiler will build one for you.**
- **They must have the same name as the class in which they are declared**
- **Constructor declarations can however have all of the normal access modifiers, and they can take arguments (including var-args), just like methods.**
- **Constructors can't be marked static (they are after all associated with object instantiation), they can't be marked final or abstract (because they can't be overridden).**

Variable Declarations

- **There are two types of variables in Java:**
  - **Primitives**
    - A primitive can be one of eight types: **char, boolean, byte, short, int, long, double, or float**.
    - Ranges of Numero values



Type	Bits	Bytes	Minimum Range	Maximum Range
`byte`	8	1	-2^{7}	$2^{7}-1$
`short`	16	2	-2^{15}	$2^{15}-1$
`int`	32	4	-2^{31}	$2^{31}-1$
`long`	64	8	-2^{63}	$2^{63}-1$
`float`	32	4	n/a	n/a
`double`	64	8	n/a	n/a

  - **Reference variables**
    - A reference variable is used to refer to (or access) an object. A reference variable is declared to be of a specific type and that type can never be changed. Instance Variables
- **Instance variables are the fields that belong to each unique object. For the exam, you need to know that instance variables**
    - Can use any of the four access levels (which means they can be marked with any of the three access modifiers)
    - Can be marked final / Can be marked transient /Cannot be marked abstract
    - Cannot be marked synchronized / Cannot be marked strictfp
    - Cannot be marked native
    - Cannot be marked static, because then they'd become class variables.

Variable Declarations

- **Local (Automatic/Stack/Method) Variables**
  - Local variables are variables declared within a method.
- **Array Declarations**
  - In Java, arrays are objects that store multiple variables of the same type, or variables that are all subclasses of the same type.
  - **Declaring an Array of Primitives**
    - int[] key; // Square brackets before name (recommended)
    - int key []; // Square brackets after name (legal but less readable)
  - Declaring an Array of Object References
    - Thread[] threads; // Recommended
    - Thread threads []; // Legal but less readable
- **Final Variables**
  - Declaring a variable with the final keyword makes it impossible to reinitialize that variable once it has been initialized with an explicit value.
    - Burn this in: there are no final objects, only final references
- **Transient Variables**
  - If you mark an instance variable as transient, you're telling the JVM to skip (ignore) this variable when you attempt to serialize the object containing it.

Variable Declarations

- **Volatile Variables**
  - The volatile modifier tells the JVM that a thread accessing the variable must always reconcile its own private copy of the variable with the master copy in memory.
- **Static Variables and Methods**
  - The static modifier is used to create variables and methods that will exist independently of any instances created for the class. Things you can mark as static:
    - Methods
    - Variables
    - A class nested within another class, but not within a method
    - Initialization blocks

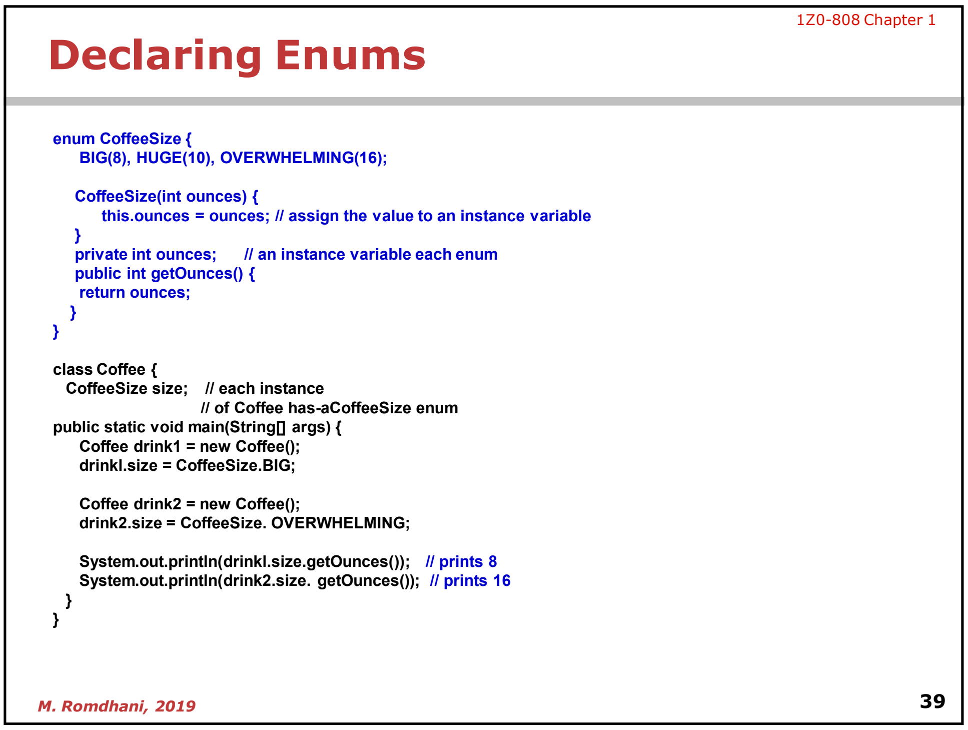
Variable Declarations

- **Comparison of modifiers on variables vs. methods**

Local Variables	Variables (nonlocal)	Methods
final	final	final
	public	public
	protected	protected
	private	private
	static	static
	transient	
	volatile	
		abstract
		synchronized
		strictfp
		native

Declaring Enums
(OCA Objective 1.2)

- **The basic components of an enum are its constants**
  - enum Seasons { Automn, Winter, Spring, Summer};
- **Enums can be declared as their own separate class, or as a class member, however they must not be declared within a method !**
- **So what gets created when you make an enum?**
  - The most important thing to remember is that enums are not Strings or ints! Each of the enumerated *Seasons* types are actually instances of *Seasons* .
  - Think of an enum as a kind of class
- **Declaring Constructors, Methods, and Variables in an Enum**
  - You can add constructors, instance variables, methods, and something really strange known as a *constant specific class body*.



Declaring Enums

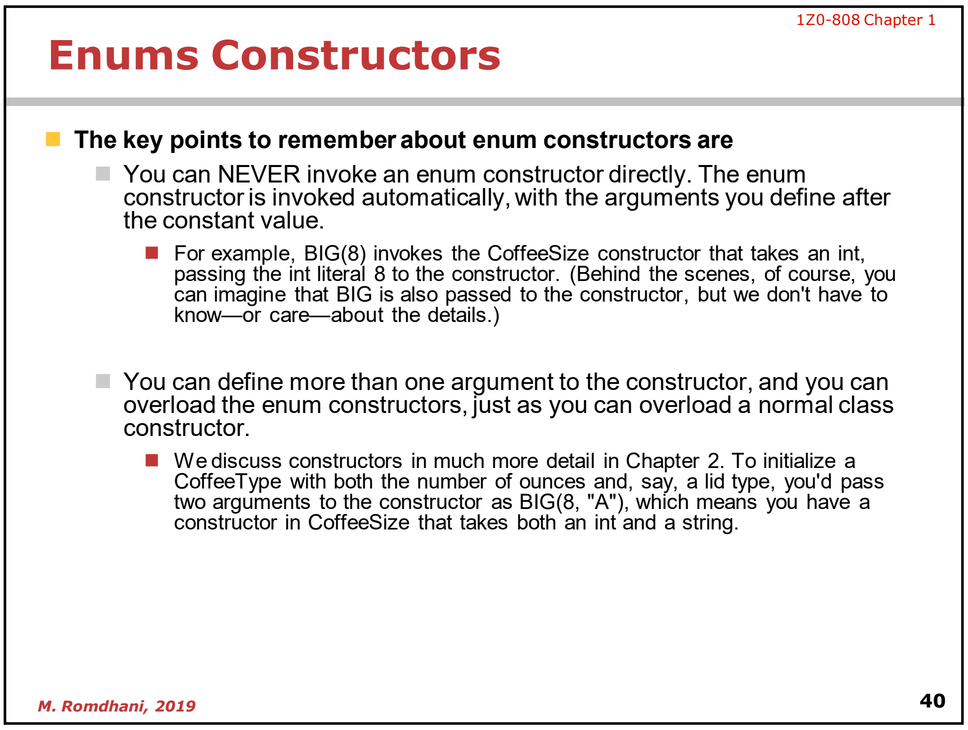
```
enum CoffeeSize {
    BIG(8), HUGE(10), OVERWHELMING(16);

   CoffeeSize(int ounces) {
       this.ounces = ounces; // assign the value to an instance variable
   }
   private int ounces;      // an instance variable each enum
   public int getOunces() {
    return ounces;
   }
}

class Coffee {
  CoffeeSize size;    // each instance
                      // of Coffee has-aCoffeeSize enum
public static void main(String[] args) {
    Coffee drink1 = new Coffee();
    drinkl.size = CoffeeSize.BIG;

    Coffee drink2 = new Coffee();
    drink2.size = CoffeeSize. OVERWHELMING;

    System.out.println(drinkl.size.getOunces());   // prints 8
    System.out.println(drink2.size. getOunces());  // prints 16
  }
}
```



Enums Constructors

- **The key points to remember about enum constructors are**
  - You can NEVER invoke an enum constructor directly. The enum constructor is invoked automatically, with the arguments you define after the constant value.
    - For example, BIG(8) invokes the CoffeeSize constructor that takes an int, passing the int literal 8 to the constructor. (Behind the scenes, of course, you can imagine that BIG is also passed to the constructor, but we don't have to know—or care—about the details.)
  - You can define more than one argument to the constructor, and you can overload the enum constructors, just as you can overload a normal class constructor.
    - We discuss constructors in much more detail in Chapter 2. To initialize a CoffeeType with both the number of ounces and, say, a lid type, you'd pass two arguments to the constructor as BIG(8, "A"), which means you have a constructor in CoffeeSize that takes both an int and a string.