SCJP 5.0 (CX-310-055) - CONTENTS

Revision #	*Date*	*Changes*
0.1	22nd Apr, 2006	Initial Version.

Disclaimer: This document is my notes taken from the SCJP 5.0 Study Guide by Kathy Sierra and Bert Bates. It builds over the 2 minute drill in the book. Though you are free to refer to my notes but this notes is not a substitute for the book itself. It should only be used for revision purposes.

Declarations & Access control

Identifiers (Objective 1.3)

- Identifiers can begin with a **letter**, an **underscore**, or a **currency** character.

  Examples of legal and illegal identifiers follow, first 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;
```

- After the first character, identifiers can also include digits.
- Identifiers can be of any length.
- JavaBeans methods must be named using camelCase, and depending on the method's purpose, must start with **`set`**, **`get`**, **`is`**, **`add`**, or **`remove`**.

  Examples of valid JavaBean method signatures are

```
public void setMyValue(int v)
public int getMyValue()
public boolean isMyStatus()
public void addMyListener(MyListener m)
public void removeMyListener(MyListener m)
```

  Examples of *invalid* JavaBean method signatures are

```
void setCustomerName(String s) // must be public
public void modifyMyValue(int v) // can't use 'modify'
public void addXListener(MyListener m) // listener type mismatch
```

TABLE 1-1 Complete List of Java Keywords (`assert` added in 1.4, `enum` added in 1.5)

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				

Which are valid declarations? (Choose all that apply.)

A. `int $x;`
B. `int 123;`
C. `int _123;`
D. `int #dim;`
E. `int %percent;`
F. `int *divide;`
G. `int central_sales_region_Summer_2005_gross_sales;`

Answer:

- **A, C,** and **G** are legal identifiers.
- **B** is incorrect because an identifier can't start with a digit.

D, E, and **F** are incorrect because identifiers must start with $, _, or a letter. (Objective 1.3)

Declaration Rules (Objective 1.1)

- A source code file can have **only one `public` class**.
- If the source file contains a `public` class, the filename must match the `public` class name.
- A file can have **only one `package` statement**, but multiple `import`s.
- The `package` statement (if any) must be the first (non-comment) line in a source file.
- The `import` statements (if any) must come after the `package` and before the class declaration.
- If there is no `package` statement, `import` statements must be the first (noncomment) statements in the source file.
- `package` and `import` statements apply to all classes in the file.
- A file can have **more than one nonpublic class.**
- **Files with no `public` classes have no naming restrictions.**

Class Access Modifiers (Objective 1.1)

- There are three **access modifiers:** `public`, `protected`, and `private`.
- There are four **access levels**: public, protected, default, and private.
- **Classes can have only `public` or default access.**
- A class with default access can be seen only by classes within the same package.
- A class with `public` access can be seen by all classes from all packages.
- Class visibility revolves around whether code in one class can
- Create an instance of another class
- Extend (or subclass), another class
- Access methods and variables of another class

Class Modifiers (Nonaccess) (Objective 1.2)

- **Classes can also be modified with `final`, `abstract`, or `strictfp`.**
  For the exam, you need to know only that **`strictfp`** is a keyword and can be used to modify a class or a method, but never a variable. Marking **a class as `strictfp`** means that any method code in the class will conform to the IEEE 754 standard rules for floating points. Without that modifier, floating points used in the methods might behave in a platform-dependent way. If you don't declare a class as `strictfp`, you can still get `strictfp` behavior on a method-by-method basis, by declaring **a method as `strictfp`**.

- **A class cannot be both `final` and `abstract`.**
- **A `final` class cannot be subclassed.**
- An `abstract` class cannot be instantiated.
- A single `abstract` method in a class means the whole class must be abstract.
- An `abstract` class can have both `abstract` and nonabstract methods.
- The first concrete class to extend an `abstract` class must implement all of its `abstract` methods.

TABLE 1-2 Determining Access to Class Members

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 non-subclass class outside the package	Yes	No	No	No

Which declare a compilable `abstract` class? (Choose all that apply.)

A. `public abstract class Canine { public Bark speak(); }`
B. `public abstract class Canine { public Bark speak() { } }`
C. `public class Canine { public abstract Bark speak(); }`
D. `public class Canine abstract { public abstract Bark speak(); }`

Answer:

- **B** is correct. `abstract` classes don't have to have any `abstract` methods.
- **A** is incorrect because `abstract` methods must be marked as such. **C** is incorrect because you can't have an `abstract` method unless the class is `abstract`. **D** is incorrect because the keyword `abstract` must come before the classname. (Objective 1.1)

Interface Implementation (Objective 1.2)

- Interfaces are contracts for what a class can do, but they say nothing about the way in which the class must do it.
- Interfaces can be implemented by any class, from any inheritance tree.
- **An interface is like a 100-percent `abstract` class, and is implicitly abstract whether you type the `abstract` modifier in the declaration or not.**
  The following is a legal interface declaration:

```
public abstract interface Rollable { }
```

- An interface can have only abstract methods, no concrete methods allowed.
- **Interface methods are by default `public` and `abstract`—explicit declaration of these modifiers is optional.**
  For example, the following five method declarations, if declared within their own interfaces, are legal and identical!

```
void bounce();
public void bounce();
abstract void bounce();
public abstract void bounce();
abstract public void bounce();
```

The following interface method declarations won't compile:

```
final void bounce(); // final and abstract can never be used
// together, and abstract is implied
static void bounce(); // interfaces define instance methods
private void bounce(); // interface methods are always public
protected void bounce(); // (same as above)
```

- **Interfaces can have constants, which are always implicitly `public`, `static`, and `final`.**

- **Interface constant declarations of `public`, `static`, and `final` are optional in any combination.**
  For example, the following are all identical and legal:

```
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

Example:
interface Foo {
       int BAR = 42;
       void go();
}
class Zap implements Foo {
       public void go() {
              BAR = 27; // ERROR! Cannot change a constant
       }
}
```

- A legal nonabstract implementing class has the following properties:
  - It provides concrete implementations for the interface's methods.
  - It must follow all legal override rules for the methods it implements.
  - **It must not declare any new checked exceptions for an implementation method. (unchecked exceptions are fine).**
  - **It must not declare any checked exceptions that are broader than the exceptions declared in the interface method.**
  - **It may declare runtime exceptions on any interface method implementation regardless of the interface declaration.**
  - It must maintain the exact signature (allowing for covariant returns) and return type of the methods it implements (**but does not have to declare the exceptions of the interface).**
- **A class implementing an interface can itself be `abstract`.**
- **An `abstract` implementing class does not have to implement the interface methods (but the first concrete subclass must).**
- A class can extend only one class (no multiple inheritance), but it can implement many interfaces.
- **Interfaces can extend one or more other interfaces.**
- Interfaces cannot extend a class, or implement a class or interface.

Given the following,

```
1. interface Base {
2. boolean m1 ();
3. byte m2(short s);
4. }
```

Which code fragments will compile? (Choose all that apply.)

A.
```
interface Base2 implements Base { }
```

B.
```
abstract class Class2 extends Base {
public boolean m1() { return true; } }
```

C.
```
abstract class Class2 implements Base { }
```

D.
```
abstract class Class2 implements Base {
public boolean m1() { return (true); } }
```

E.
```
class Class2 implements Base {
boolean m1() { return false; }
byte m2(short s) { return 42; } }
```

Answer:

- **C** and **D** are correct. **C** is correct because an `abstract` class doesn't have to implement any or all of its interface's methods. **D** is correct because the method is correctly implemented.
- **A** is incorrect because interfaces don't implement anything. **B** is incorrect because classes don't extend interfaces. **E is incorrect because interface methods are implicitly `public`, so the methods being implemented must be `public`.** (Objective 1.1)

Member Access Modifiers (Objectives 1.3 and 1.4)

- Methods and instance (nonlocal) variables are known as "members."
- **Members can use all four access levels: public, protected, default, private.**
- Member access comes in two forms:
  - Code in one class can access a member of another class.
  - A subclass can inherit a member of its superclass.
- **If a class cannot be accessed, its members cannot be accessed.**
- Determine class visibility before determining member visibility.
- `public` members can be accessed by all other classes, even in other packages.
- If a superclass member is public, the subclass inherits it—regardless of package.
- Members accessed without the dot operator (.) must belong to the same class.
  - `this.` always refers to the currently executing object.
  - `this.aMethod()` is the same as just invoking `aMethod()`.
- `private` members can be accessed only by code in the same class.
- `private` members are not visible to subclasses, so private members cannot be inherited.
- Default and `protected` members differ only when subclasses are involved:
- Default members can be accessed only by classes in the same package.
- **`protected` members can be accessed by other classes in the same package, plus subclasses regardless of package.**
- **`protected` = package plus kids (kids meaning subclasses).**
- For subclasses outside the package, the `protected` member can be accessed only through inheritance; a subclass outside the package cannot access a `protected` member by using a reference to a superclass instance (in other words, inheritance is the only mechanism for a subclass outside the package to access a `protected` member of its superclass).
- **A `protected` member inherited by a subclass from another package is not accessible to any other class in the subclass package, except for the subclass' own subclasses.**

Local Variables (Objective 1.3)

- Local (method, automatic, or stack) variable declarations cannot have access modifiers.
- **`final` is the only modifier available to local variables.**
- **Local variables don't get default values, so they must be initialized before use.**

Other Modifiers—Members (Objective 1.3)

- **`final` methods cannot be overridden in a subclass.**
- `abstract` methods are declared, with a signature, a return type, and an optional throws clause, but are not implemented.
- `abstract` methods end in a semicolon—no curly braces.
- Three ways to spot a non-abstract method:
  - The method is not marked `abstract`.
  - The method has curly braces.
  - The method has code between the curly braces.
- The first nonabstract (concrete) class to extend an `abstract` class must implement all of the `abstract` class' `abstract` methods.
- **The `synchronized` modifier applies only to methods and code blocks.**

- `synchronized` methods can have any access control and can also be marked `final`.
- **`abstract` methods must be implemented by a subclass, so they must be inheritable. For that reason:**
  - **`abstract` methods cannot be `private`.**
  - **`abstract` methods cannot be `final`.**
  - **`abstract methods cannot be static.`**

```
abstract static void doStuff(); // Compiler Error!
```

- **The `native` modifier applies only to methods.** The `native` modifier indicates that a method is implemented in platform-dependent code, often in C.
- **The `strictfp` modifier applies only to classes and methods.**

Methods with var-args (Objective 1.4)

- As of Java 5, methods can declare a parameter that accepts from zero to many arguments, a so-called var-arg method.
- A var-arg parameter is declared with the syntax `type... name`; for instance:
  `doStuff(int... x) { }`
- **A var-arg method can have only one var-arg parameter.**
- **In methods with normal parameters and a var-arg, the var-arg must come last.**
  Legal:

```
void doStuff(int... x) { } // expects from 0 to many ints as parameters
void doStuff2(char c, int... x) { } // expects first a char,
                                    // then 0 to many ints
void doStuff3(Animal... animal) { } // 0 to many Animals
```

  Illegal:

```
void doStuff4(int x...) { } // bad syntax
void doStuff5(int... x, char... y) { } // too many var-args
void doStuff6(String... s, byte b) { } // var-arg must be last
```

- **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).** Here are some legal and illegal constructor declarations:

```
class Foo2 {
       // legal constructors
       Foo2() { }
       private Foo2(byte b) { }
       Foo2(int x) { }
       Foo2(int x, int... y) { }

       // illegal constructors
       void Foo2() { } // it's a method, not a constructor
       Foo() { } // not a method or a constructor
       Foo2(short s); // looks like an abstract method
       static Foo2(float f) { } // can't be static
       final Foo2(long x) { } // can't be final
       abstract Foo2(char c) { } // can't be abstract
       Foo2(int... x, int t) { } // bad var-arg syntax
}
```

Given:

```
1. class Voop {
2. public static void main(String[] args) {
3. doStuff(1);
4. doStuff(1,2);
5. }
6. // insert code here
7. }
```

Which, inserted independently at line 6, will compile? (Choose all that apply.)

A. `static void doStuff(int... doArgs) { }`
B. `static void doStuff(int[] doArgs) { }`
C. `static void doStuff(int doArgs...) { }`
D. `static void doStuff(int... doArgs, int y) { }`
E. `static void doStuff(int x, int... doArgs) { }`

Answer:

- **A** and E use valid var-args syntax.
- **B** and **C** are invalid var-arg syntax, and **D** is invalid because the var-arg must be the last of a method's arguments. (Objective 1.4)

Variable Declarations (Objective 1.3)

- **Instance variables can**
  - **Have any access control**
  - **Be marked `final` or `transient`**
- Instance variables can't be `abstract`, `synchronized`, `native`, or `strictfp`.
- It is legal to declare a local variable with the same name as an instance variable; this is called "shadowing."
- `final` variables have the following properties:
  - `final` variables cannot be reinitialized once assigned a value.
  - `final` reference variables cannot refer to a different object once the object has been assigned to the `final` variable.
  - **`final` reference variables must be initialized before the constructor completes.**
  - **There is no such thing as a `final` object. An object reference marked `final` does not mean the object itself is immutable.**
- The `transient` modifier applies only to instance 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.
- The `volatile` modifier applies only to instance 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.

TABLE 1-3 Ranges of Numeric Primitives

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

Array Declarations (Objective 1.3)

- Arrays can hold primitives or objects, but the **array itself is always an object.**
- When you declare an array, the brackets can be to the left or right of the variable name.
- It is never legal to include the size of an array in the declaration.
- An array of objects can hold any object that passes the IS-A (or instanceof) test for the declared type of the array. For example, **if Horse extends Animal, then a Horse object can go into an Animal array.**
- **The elements of array will always be initialized to default value even in method local scope.**

Which are legal declarations? (Choose all that apply.)

A. `short x [];`
B. `short [] y;`
C. `short[5] x2;`
D. `short z2 [5];`
E. `short [] z [] [];`
F. `short [] y2 = [5];`

Answer:

- **A, B,** and **E** are correct array declarations; E is a three dimensional array.
- **C, D,** and **F** are incorrect, you can't include the size of your array in a declaration unless you also instantiate the array object. F uses invalid instantiation syntax. (Objective 1.3)

Static Variables and Methods (Objective 1.4)

- They are not tied to any particular instance of a class.
- No classes instances are needed in order to use `static` members of the class.
- There is only one copy of a `static` variable / class and all instances share it.
- **`static` methods do not have direct access to non-static members.**
- **Things you can mark as `static`:**
  - **Methods**
  - **Variables**
  - **A class nested within another class, but not within a method**
  - **Initialization blocks**

- **Things you can't mark as `static`:**
  - **Constructors (makes no sense; a constructor is used only to create instances)**
  - **Classes (unless they are nested)**
  - **Interfaces**
  - **Method local inner classes**
  - **Inner class methods and instance variables**
  - **Local variables**

Enums (Objective 1.3)

- **An `enum` specifies a list of constant values that can be assigned to a particular type.**
- An `enum` is NOT a String or an int; an enum constant's type is the enum type. For example, WINTER, SPRING, SUMMER, and FALL are of the enum type Season.

```
enum CoffeeSize { BIG, HUGE, OVERWHELMING };
CoffeeSize cs = CoffeeSize.BIG;
```

- **An `enum` can be declared outside or inside a class, but NOT in a method.**

```
public class CoffeeTest1 {
        public static void main(String[] args) {
                enum CoffeeSize { BIG, HUGE, OVERWHELMING } // WRONG!
                                // Cannot declare enums in methods
                Coffee drink = new Coffee();
                drink.size = CoffeeSize.BIG;
        }
}
```

- **An `enum` declared outside a class must NOT be marked `static`, `final`, `abstract`, `protected`, or `private`. Just like non-inner classes, an outer enum can be public or default. There is no use of final or abstract as enum don't support inheritance.**
- **Enums can contain constructors, methods, variables, and constant class bodies.**

```
enum CoffeeSize {
        BIG(8), HUGE(10), OVERWHELMING(16);
        // the arguments after the enum value are "passed"
        // as values to the constructor
```

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

class Coffee {
	CoffeeSize size; // each instance of Coffee has-a CoffeeSize enum
	public static void main(String[] args) {
		Coffee drink1 = new Coffee();
		drink1.size = CoffeeSize.BIG;
		Coffee drink2 = new Coffee();
		drink2.size = CoffeeSize.OVERWHELMING;
		System.out.println(drink1.size.getOunces()); // prints 8
		System.out.println(drink2.size.getOunces()); // prints 16
	}
}
```

- **`enum` constants can send arguments to the `enum` constructor, using the syntax BIG(8), where the int literal 8 is passed to the `enum` constructor.**
- **`enum` constructors can have arguments, and can be overloaded.**
- **`enum` constructors can NEVER be invoked directly in code. They are always called automatically when an `enum` is initialized.**
- **Constant specific class body – when you need a particular constant in enum to override a method defined in enum.**

```
enum CoffeeSize {
	BIG(8),
	HUGE(10),
	OVERWHELMING(16) { // start a code block that defines
		// the "body" for this constant
		public String getLidCode() { // override the method
			// defined in CoffeeSize
			return "A";
		}
	}; // <-- the semicolon is REQUIRED when you have a body

	CoffeeSize(int ounces) {
		this.ounces = ounces;
	}

	private int ounces;

	public int getOunces() {
		return ounces;
	}

	public String getLidCode() { // this method is overridden
		// by the OVERWHELMING constant
		return "B"; // the default value we want to return for
		// CoffeeSize constants
	}
}
```

- **The semicolon at the end of an `enum` declaration is optional. These are legal:**

```
enum Foo { ONE, TWO, THREE}
enum Foo { ONE, TWO, THREE};
```

Examples:

1. Non-inner enum:

```
enum CoffeeSize { BIG, HUGE, OVERWHELMING } // cannot be private or
protected
class Coffee {
	CoffeeSize size;
```

```
}
public class CoffeeTest1 {
	public static void main(String[] args) {
		Coffee drink = new Coffee();
		drink.size = CoffeeSize.BIG; // enum outside class
	}
}
```

2. Inner enum:

```
class Coffee2 {
	enum CoffeeSize {BIG, HUGE, OVERWHELMING }
	CoffeeSize size;
}
public class CoffeeTest2 {
	public static void main(String[] args) {
		Coffee2 drink = new Coffee2();
		drink.size = Coffee2.CoffeeSize.BIG; // enclosing class
                                                     // name required
	}
}
```

Given:

```
1. enum Animals {
2.     DOG("woof"), CAT("meow"), FISH("burble");
3.     String sound;
4.     Animals(String s) { sound = s; }
5. }
6. class TestEnum {
7.     static Animals a;
8.     public static void main(String [] args) {
9.         System.out.println(a.DOG.sound + " " + a.FISH.sound);
10.    }
11. }
```

What is the result?

A. `woof burble`
B. Multiple compilation errors
C. Compilation fails due to an error on line 2
D. Compilation fails due to an error on line 3
E. Compilation fails due to an error on line 4
F. Compilation fails due to an error on line 9

Answer:

- **A** is correct; **enums can have constructors and variables.**
- **B, C, D, E,** and **F** are incorrect; these lines all use correct syntax.

Given:

```
1. enum A { A }
2. class E2 {
3.     enum B { B }
4.     void C() {
5.         enum D { D }
6.     }
7. }
```

Which statements are true? (Choose all that apply.)

A. The code compiles.
B. If only line 1 is removed the code compiles.
C. If only line 3 is removed the code compiles.
D. If only line 5 is removed the code compiles.
E. If lines 1 and 3 are removed the code compiles.
F. If lines 1, 3 and 5 are removed the code compiles.

Answer:

- **D** and **F** are correct. Line 5 is the only line that will not compile, because **enums cannot be**

local to a method.

- **A, B, C** and **E** are incorrect based on the above.

Object Orientation

Encapsulation, IS-A, HAS-A (Objective 5.1)

- **Encapsulation helps hide implementation behind an interface (or API).**
- Encapsulated code has two features:
  - Instance variables are kept protected (usually with the private modifier).
  - Getter and setter methods provide access to instance variables.
- **IS-A refers to inheritance.**
- IS-A is expressed with the keyword `extends`.
- IS-A, "inherits from," and "is a subtype of" are all equivalent expressions.
- **HAS-A means an instance of one class "has a" reference to an instance of another class.**

Inheritance (Objective 5.5)

- Inheritance is a mechanism that allows a class to be a subclass of a superclass, and thereby inherit variables and methods of the superclass.
- Inheritance is a key concept that underlies IS-A, polymorphism, overriding, overloading, and casting.
- All classes (except class `Object`), are subclasses of type `Object`, and therefore they inherit `Object`'s methods.

Polymorphism (Objective 5.2)

- Polymorphism means 'many forms'.
- A reference variable is always of a single, unchangeable type, but it can refer to a subtype object.
- A single object can be referred to by reference variables of many different types— as long as they are the same type or a supertype of the object.
- **The reference variable's type (not the object's type), determines which methods can be called!**
- **Polymorphic method invocations apply only to overridden *instance* methods.**

```
class Animal {
       public void eat() {
              System.out.println("Generic Animal Eating Generically");
       }
}
class Horse extends Animal {
       public void eat() {
              System.out.println("Horse eating hay, oats, " + "and horse
              treats");
       }
       public void buck() { }
}

Animal c = new Horse();
c.buck(); // Can't invoke buck();  Animal class doesn't have that method
```

Overriding and Overloading (Objectives 1.5 and 5.4)

- Methods can be overridden or overloaded; **constructors can be overloaded but not overridden.**
- Abstract methods must be overridden by the first concrete (nonabstract) subclass.
- With respect to the method it overrides**, the overriding method**
  - **Must have the same argument list**
  - Must have the same return type, except that as of Java 5, the **return type can be a subclass—this is known as a covariant return.**

```
class Alpha {
        Alpha doStuff(char c) {
                return new Alpha();
        }
}
class Beta extends Alpha {
        Beta doStuff(char c) { // legal override in Java 1.5
                return new Beta();
        }
}
```

But with source level of 1.4 or earlier you will get a compiler error:

```
javac -source 1.4 Beta.java
```

you would get a compiler error something like this:

```
attempting to use incompatible return type
```

IMP:

Given:

```
1. class Plant {
2.     String getName() { return "plant"; }
3.     Plant getType() { return this; }
4. }
5. class Flower extends Plant {
6.     // insert code here
7. }
8. class Tulip extends Flower { }
```

Which statement(s), inserted at line 6, will compile? (Choose all that apply.)

A. `Flower getType() { return this; }`

B. `String getType() { return "this"; }`

C. `Plant getType() { return this; }`

D. `Tulip getType() { return new Tulip(); }`

Answer:

- **A, C,** and **D** are correct. **A** and **D** are examples of co-variant returns, i.e., `Flower` and `Tulip` are both subtypes of `Plant`.
- **B** is incorrect, `String` is not a subtype of `Plant`. (Objective 1.5)

- **Must not have a more restrictive access modifier**
- May have a less restrictive access modifier
- Must not throw new or broader checked exceptions
- **May throw fewer or narrower checked exceptions, or any unchecked exception.**

If a method is overridden but you use a polymorphic (supertype) reference to refer to the subtype object with the overriding method, the* compiler *assumes you're calling the supertype version of the method. If the supertype version declares a checked exception, but the overriding subtype method does not, the compiler* still *thinks you are calling a method that declares an exception:

```
class Animal {
      public void eat() throws Exception {
            // throws an Exception
      }
}
class Dog2 extends Animal {
      public void eat() { // no Exceptions }
      public static void main(String [] args) {
            Animal a = new Dog2();
            Dog2 d = new Dog2();
            d.eat(); // ok
```

```
                a.eat();  // compiler error - unreported exception
        }
}
```

This code will not compile because of the Exception declared on the Animal eat() ***method. This happens even though, at runtime, the*** `eat()` ***method used would be the Dog version, which does not declare the exception.***

Illegal Overrides Example:

```
public class Animal {
        public void eat() { }
}
```

`private void eat() { }`	Access modifier is more restrictive
`public void eat() throws IOException { }`	Declares a checked exception not defined by superclass version
`public void eat(String food) { }`	A legal overload, not an override, because the argument list changed. **Methods from a superclass can be overloaded in a subclass.**
`public String eat() { }`	Not an override because of the return type, not an overload either because there's no change in the argument list

IMP:

Given:

```
import java.io.*;
class Master {
        String doFileStuff() throws FileNotFoundException { return "a"; }
}
class Slave extends Master {
        public static void main(String[] args) {
                String s = null;
                try { s = new Slave().doFileStuff();
                } catch ( Exception x) {
                s = "b"; }
                System.out.println(s);
        }
        // insert code here
}
```

Which, inserted independently at `// insert code here`, will compile, and produce the output `b`? (Choose all that apply.)

A. `String doFileStuff() { return "b"; }`
B. `String doFileStuff() throws IOException { return "b"; }`
C. `String doFileStuff(int x) throws IOException { return "b"; }`
D. `String doFileStuff() throws FileNotFoundException { return "b"; }`
E. `String doFileStuff() throws NumberFormatException { return "b"; }`
F. `String doFileStuff() throws NumberFormatException, FileNotFoundException { return "b"; }`

Answer:

- **A , D, E,** and **F** are correct**. It's okay for an overriding method to throw the same exceptions, narrower exceptions, or no exceptions**. **And it's okay for the overriding method to throw any runtime exceptions.**
- **B** is incorrect, because the overriding method is trying to throw a broader exception. **C** is incorrect. This method doesn't override, so the output is `a`. (Objective 2.4)

- **`final` methods cannot be overridden.**

Given:

```
class Clidders {
	public final void flipper() { System.out.println("Clidder"); }
}
public class Clidlets extends Clidders {
	public void flipper() {
		System.out.println("Flip a Clidlet");
		super.flipper();
	}
	public static void main(String [] args) {
		new Clidlets().flipper();
	}
}
```

What is the result?

A. `Flip a Clidlet`

B. `Flip a Clidder`

C. `Flip a Clidder`
`Flip a Clidlet`

D. `Flip a Clidlet`
`Flip a Clidder`

E. Compilation fails.

Answer:

- **E** is correct. `final` methods cannot be overridden.
- **A, B, C,** and **D** are incorrect based on the above. (Objective 5.3)

IMP:

Given:

```
class Clidder {
	private final void flipper() { System.out.println("Clidder"); }
}
public class Clidlet extends Clidder {
	public final void flipper() { System.out.println("Clidlet"); }
	public static void main(String [] args) {
		new Clidlet().flipper();
	}
}
```

What is the result?

A. `Clidlet`

B. `Clidder`

C. `Clidder`
`Clidlet`

D. `Clidlet`
`Clidder`

E. Compilation fails.

Answer:

- **A** is correct. **Although a final method cannot be overridden, in this case, the method is private, and therefore hidden. The effect is that a new, accessible, method flipper is created.** Therefore, no polymorphism occurs in this example, the method invoked is simply that of the child class, and no error occurs.
- **B, C, D,** and **E** are incorrect based on the preceding. (Objective 5.3)

- **`static` methods cannot be overridden.**
- **Only inherited methods may be overridden,** and remember that private methods are not inherited.
- A subclass uses **`super.overriddenMethodName()` to call the superclass version of an overridden method.**
- Overloading means reusing amethod name, but with different arguments.
- **Overloaded methods**
  - **Must have different argument lists**

- May have different return types, if argument lists are also different
- May have different access modifiers
- May throw different exceptions

- **Methods from a superclass can be overloaded in a subclass.**
- Polymorphism applies to overriding, not to overloading
- **Object type (not the reference variable's type), determines which overridden method is used at runtime.**
- **Reference type determines which overloaded method will be used at compile time.**

```
Example1:
class Animal {
       void makeNoise() {System.out.println("generic noise"); }
}
class Dog extends Animal {
       void makeNoise() {System.out.println("bark"); }
       void playDead() { System.out.println(" roll over"); }
}
class CastTest2 {
       public static void main(String [] args) {
              Animal [] a = {new Animal(), new Dog(), new Animal() };
              for(Animal animal : a) {
                     animal.makeNoise();
                     if(animal instanceof Dog) {
                            animal.playDead(); // Compiler Error! Cannot find symbol.
                     }
              }
       }
}
```

Compiler complains as it sees the Animal class does not have a playDead() method.

```
       if(animal instanceof Dog) {
              Dog d = (Dog) animal; // casting the ref. var.
              d.playDead();
       }
```

With the above change the compiler is happy.

IMP:

Example2:

```
       class Animal { }
       class Dog extends Animal { }
       class DogTest {
              public static void main(String [] args) {
                     Animal animal = new Animal();
                     Dog d = (Dog) animal; // compiles but fails later: ClassCastException
              }
       }
```

Compiler has to trust us when we are downcasting an Animal reference to a subtype Dog reference. But since the animal reference did not actually hold the instance of Dog so the casting will raise a unchecked ClassCastException at runtime.

Given:

```
1. class Dog { }
2. class Beagle extends Dog { }
3.
4. class Kennel {
5.     public static void main(String [] arfs) {
6.            Beagle b1 = new Beagle();
7.            Dog dog1 = new Dog();
8.            Dog dog2 = b1;
9.            // insert code here
10.    }
11. }
```

Which, inserted at line 9, will compile? (Choose all that apply.)

A. `Beagle b2 = (Beagle) dog1;`
B. `Beagle b3 = (Beagle) dog2;`
C. `Beagle b4 = dog2;`
D. None of the above statements will compile

Answer:

- **A** and **B** are correct. However, at runtime, **A** will throw a `ClassCastException` because `dog1` refers to a Dog object, which can't necessarily do Beagle stuff.
- **C** and **D** are incorrect based on the preceding. (Objective 5.2).

Reference Variable Casting (Objective 5.2)

- There are two types of reference variable casting: downcasting and upcasting.
    1. **Downcasting:** If you have a reference variable that refers to a subtype object, you can assign it to a reference variable of the subtype. You **must make an explicit cast** to do this, and the result is that you can access the subtype's members with this new reference variable.
    2. **Upcasting:** You can assign a reference variable to a supertype reference variable **explicitly or implicitly**. This is an inherently safe operation because the assignment restricts the access capabilities of the new variable.

Implementing an Interface (Objective 1.2)

- When you implement an interface, you are fulfilling its contract.
- You implement an interface by properly and concretely overriding all of the methods defined by the interface.
- A single class can implement many interfaces.

```
class Foo { } // OK
class Bar implements Foo { } // No! Can't implement a class
interface Baz { } // OK
interface Fi { } // OK
interface Fee implements Baz { } // No! Interface can't implement an interface
interface Zee implements Foo { } // No! Interface can't implement a class
interface Zoo extends Foo { } // No! Interface can't extend a class
interface Boo extends Fi { } // OK. Interface can extend an interface
class Toon extends Foo, Button { } // No! Class can't extend multiple classes
class Zoom implements Fi, Fee { } // OK.class can implement multiple interfaces
interface Vroom extends Fi, Fee { } // OK.interface can extend
                                    // multiple interfaces
class Yow extends Foo implements Fi { } // OK. Class can do both
                                        // (extends must be 1st)
```

Return Types (Objective 1.5)

- Overloaded methods can change return types; overridden methods cannot, except in the case of covariant returns.
- Object reference return types can accept `null` as a return value.
- An array is a legal return type, both to declare and return as a value.
- For methods with primitive return types, any value that can be implicitly converted to the return type can be returned.
- Nothing can be returned from a `void`, but you can return nothing. You're allowed to simply say `return`, in any method with a `void` return type, to bust out of a method early. But you can't return nothing from a method with a non-void return type.
- **Methods with an object reference return type, can return a subtype.**
- Methods with an interface return type, can return any implementer.

Constructors and Instantiation (Objectives 1.6 and 5.4)

- You cannot create a new object without invoking a constructor.
- **Each superclass in an object's inheritance tree will have a constructor called.**
- Every class, even an abstract class, has at least one constructor.
- Constructors must have the same name as the class.
- **Constructors don't have a return type. If the code you're looking at has a return type, it's a method with the same name as the class, and a constructor.**
- Typical constructor execution occurs as follows:

  - The constructor calls its superclass constructor, which calls its superclass constructor, and so on all the way up to the Object constructor.
  - The Object constructor executes and then returns to the calling constructor, which runs to completion and then returns to its calling constructor, and so on back down to the completion of the constructor of the actual instance being created.
- Constructors can use any access modifier (even `private`!).
- The compiler will create a default constructor if you don't create any constructors in your class.
- The default constructor is a no-arg constructor with a no-arg call to `super()`.
- **The first statement of every constructor must be a call to either `this()`, (an overloaded constructor), or `super()`.**
- The compiler will add a call to `super()` if you do not, unless you have already put in a call to `this()`.
- Instance members are accessible only after the super constructor runs.
- Abstract classes have constructors that are called when a concrete subclass is instantiated.
- **Interfaces do not have constructors.**
- **If your superclass does not have a no-arg constructor, you must create a constructor and insert a call to `super()` with arguments matching those of the superclass constructor.**

```
class Clothing {
       Clothing(String s) { }
}
class TShirt extends Clothing { }

Compiler gives error as follows:

Clothing.java:4: cannot resolve symbol
symbol : constructor Clothing ()
location: class Clothing
class TShirt extends Clothing { }
^
```

To fix this error put a constructor as follows or similar:

```
TShirt() {
       super("some string");
}
```

Given:

```
class Top {
       public Top(String s) { System.out.print("B"); }
}
public class Bottom2 extends Top {
       public Bottom2(String s) { System.out.print("D"); }
       public static void main(String [] args) {
              new Bottom2("C");
              System.out.println(" ");
       }
}
```

What is the result?

A. `BD`
B. `DB`
C. `BDC`
D. `DBC`
E. Compilation fails.

Answer:

✹ **E** is correct. The implied `super()` call in Bottom2's constructor cannot be satisfied because there isn't a no-arg constructor in Top. A default, no-arg constructor is generated by the compiler only if the class has no constructor defined explicitly.

✹ **A, B, C,** and **D** are incorrect based on the above. (Objective 1.6)

- **Constructors are never inherited, thus they cannot be overridden.**
- A constructor can be directly invoked only by another constructor (using a call to `super()` or `this()`).
- Issues with calls to `this()`:
  - May appear only as the first statement in a constructor.
  - The argument list determines which overloaded constructor is called.
- Constructors can call constructors can call constructors, and so on, but sooner or later one of them better call `super()` or the stack will explode.

```
class A {
	A() {
		this("foo");
	}

	A(String s) {
		this();
	}
}

% java A
Exception in thread "main" java.lang.StackOverflowError
```

- **Calls to `this()` and `super()` cannot be in the same constructor. You can have one or the other, but never both.**

Statics (Objective 1.3)

- Use `static` methods to implement behaviors that are not affected by the state of any instances.
- Use `static` variables to hold data that is class specific as opposed to instance specific—there will be only one copy of a `static` variable.
- All `static` members belong to the class, not to any instance.
- A `static` method can't access an instance variable directly.

```
class Foo {
	int x = 3;
	float y = 4.3f;
	public static void main (String [] args) {
		for (int z = x; z < ++x; z--, y = y + z) {
		// complicated looping and branching code
	}
}

Compiler Error: static memebers x and y cannot be accessed in the static
main() method.
```

- Use the dot operator to access `static` members, but remember that using a reference variable with the dot operator is really a syntax trick, and the compiler will substitute the class name for the reference variable, for instance:

```
d.doStuff();
```

becomes:

```
Dog.doStuff();
```

- **`static` methods can't be overridden, but they can be redefined.**

```
class Animal {
	static void doStuff() {
		System.out.print("a ");
	}
}
class Dog extends Animal {
	static void dostuff() { // it's a redefinition, not an override
		System.out.print("d ");
	}
	public static void main(String [] args) {
	Animal [] a = {new Animal(), new Dog(), new Animal()};
	for(int x = 0; x < a.length; x++)
```

```
                a[x].doStuff(); // invoke the static method
        }
}
```

Running this code produces the output:

```
a a a
```

Remember, the syntax `a[x].doStuff()` is just a shortcut (the syntax trick)…the compiler is going to substitute something like **`Animal.doStuff()`** instead. So static method doStuff was inherited but not overridden so polymorphism does not work here. The only way to call the Dog's static method doStuff is to use Dog.doStuff() or a dog reference type which will evaluate to Dog.doStuff() eventually.

Coupling and Cohesion (Objective 5.1)

- **Coupling refers to the degree to which one class knows about or uses members of another class.**
- **Loose coupling is the desirable state of having classes that are well encapsulated, minimize references to each other, and limit the breadth of API usage. So good encapsulation leads to loose coupling too. When a class A can access a public instance member of another class B then class B is not well encapsulated and class A is tightly coupled with class B.**

Ideally, all interactions between objects in an OO system should use the APIs, in other words, the contracts, of the objects' respective classes. Theoretically, if all of the classes in an application have well-designed APIs, then it should be possible for all interclass interactions to use those APIs exclusively.

Let's look at an obvious example of tight coupling, which has been enabled by poor encapsulation:

```
class DoTaxes {
       float rate;
       float doColorado() {
              SalesTaxRates str = new SalesTaxRates();
              rate = str.salesRate; // ouch
                           // this should be a method call:
                           // rate = str.getSalesRate("CO");
              // do stuff with rate
       }
}
class SalesTaxRates {
       public float salesRate; // should be private
       public float adjustedSalesRate; // should be private
       public float getSalesRate(String region) {
              salesRate = new DoTaxes().doColorado(); // ouch again!
              // do region-based calculations
              return adjustedSalesRate;
       }
}
```

- Tight coupling is the undesirable state of having classes that break the rules of loose coupling.
- **Cohesion refers to the degree in which a class has a single, well-defined role or responsibility.**
- High cohesion is the desirable state of a class whose members support a single, well-focused role or responsibility.
- Low cohesion is the undesirable state of a class whose members support multiple, unfocused roles or responsibilities.

Select the two statements that best indicate a situation with low coupling. (Choose two.)

A. The attributes of the class are all `private`.
B. The class refers to a small number of other objects.
C. The object contains only a small number of variables.
D. The object is referred to using an anonymous variable, not directly.
E. The reference variable is declared for an interface type, not a class. The interface provides a
small number of methods.
F. It is unlikely that changes made to one class will require any changes in another.
Answer:

- **E** and **F** are correct. Only having access to a small number of methods implies limited coupling. **If the access is via a reference of interface type, it may be argued that there is even less opportunity for coupling as the class type itself is not visible. Stating that changes in one part of a program are unlikely to cause consequences in another part is really the essence of low coupling.** There is no such thing as an anonymous variable. Referring to only a small number of other objects might imply low coupling, but if each object has many methods, and all are used, then coupling is high. Variables (attributes) in a class should usually be private, but this describes encapsulation, rather than low coupling. Of course**, good encapsulation tends to reduce coupling as a consequence**.
- **A, B, C** and **D** are incorrect based on the preceding treatise. (Objective 5.1)

Which statement(s) are true? (Choose all that apply.)
A. Cohesion is the OO principle most closely associated with hiding implementation details.
B. Cohesion is the OO principle most closely associated with making sure that classes know about other classes only through their APIs.
C. Cohesion is the OO principle most closely associated with making sure that a class is designed with a single, well-focused purpose.
D. Cohesion is the OO principle most closely associated with allowing a single object to be seen as having many types.
Answer:

- Answer **C** is correct.
- **A** refers to encapsulation, **B** refers to coupling, and **D** refers to polymorphism. (Objective 5.1)

Assignments

Stack and Heap

- Local variables (method variables) live on the stack.
- Objects and their instance variables live on the heap.

Literals and Primitive Casting (Objective 1.3)

- Integer literals can be decimal, octal (e.g. `013`), or hexadecimal (e.g. `0x3d`).
- **Literals for `long`s end in `L` or `l`.**
- **Float literals end in `F` or `f`, `double` literals end in a digit or `D` or `d`.**
- The `boolean` literals are `true` and `false`.
- Literals for `char`s are a single character inside single quotes: `'d'`.

Scope (Objectives 1.3 and 7.6)

- Scope refers to the lifetime of a variable.
- There are four basic scopes:
  - Static variables live basically as long as their class lives.
  - Instance variables live as long as their object lives.
  - Local variables live as long as their method is on the stack; however, if their method invokes another method, they are temporarily unavailable.

- Block variables (e.g.., in a `for` or an `if`) live until the block completes.

Given:

```
class Scoop {
       static int thrower() throws Exception { return 42; }
       public static void main(String [] args) {
              try {
                     int x = thrower();
              } catch (Exception e) {
                     x++;
              } finally {
                     System.out.println("x = " + ++x);
              }
       }
}
```

What is the result?

A. `x = 42`
B. `x = 43`
C. `x = 44`
D. Compilation fails.
E. The code runs with no output.

Answer:

- **D** is correct, the variable `x` is only in scope within the `try` code block, it's not in scope in the `catch` or `finally` blocks.
- **A, B, C,** and **E** is are incorrect based on the above. (Objective 1.3)

Basic Assignments (Objectives 1.3 and 7.6)

- Literal integers are implicitly ints.
- Integer expressions always result in an `int`-sized result, never smaller.
- Floating-point numbers are implicitly doubles (64 bits).
- Narrowing a primitive truncates the *high order* bits.
- Compound assignments (e.g. +=), perform an automatic cast.
- A reference variable holds the bits that are used to refer to an object.
- Reference variables can refer to subclasses of the declared type but not to superclasses.
- When creating a new object, e.g., `Button b = new Button();`, three things happen:
  - Make a reference variable named `b`, of type Button
  - Create a new Button object
  - Assign the Button object to the reference variable `b`

Using a Variable or Array Element That Is Uninitialized and Unassigned (Objectives 1.3 and 7.6)

- When an array of objects is instantiated, objects within the array are not instantiated automatically, but all the references get the default value of null.
- **When an array of primitives is instantiated, elements get default values.**
- **Instance variables are always initialized with a default value.**
- Local/automatic/method variables are never given a default value. If you attempt to use one before initializing it, you'll get a compiler error.

Passing Variables into Methods (Objective 7.3)

- Methods can take primitives and/or object references as arguments.
- Method arguments are always copies.
- Method arguments are never actual objects (they can be references to objects).
- A primitive argument is an unattached copy of the original primitive.
- A reference argument is another copy of a reference to the original object.
- Shadowing occurs when two variables with different scopes share the same name. This leads to hard-to-find bugs, and hard-to-answer exam questions.

Array Declaration, Construction, and Initialization (Obj. 1.3)

- Arrays can hold primitives or objects, **but the array itself is always an object.**
- When you declare an array, the brackets can be left or right of the name.
- It is never legal to include the size of an array in the declaration.
- You must include the size of an array when you construct it (using `new`) unless you are creating an anonymous array.
- Elements in an array of objects are not automatically created, although primitive array elements are given default values.
- You'll get a NullPointerException if you try to use an array element in an object array, if that element does not refer to a real object.
- Arrays are indexed beginning with zero.
- **An ArrayIndexOutOfBoundsException occurs if you use a bad index value.**
- Arrays have a `length` variable whose value is the number of array elements.
- The last index you can access is always one less than the length of the array.
- Multidimensional arrays are just arrays of arrays.
- The dimensions in a multidimensional array can have different lengths.
- **An array of primitives can accept any value that can be promoted implicitly to the array's declared type;. e.g., a `byte` variable can go in an `int` array.**
- **An array of objects can hold any object that passes the IS-A (or instanceof) test for the declared type of the array**. For example, if Horse extends Animal, then a Horse object can go into an Animal array.
- If you assign an array to a previously declared array reference, the array you're assigning must be the same dimension as the reference you're assigning it to.
- **You can assign an array of one type to a previously declared array reference of one of its supertypes.** For example, a Honda array can be assigned to an array declared as type Car (assuming Honda extends Car).

Given:

```
1. class Zippy {
2.     String[] x;
3.     int[] a [] = {{1,2}, {1}};
4.     Object c = new long[4];
5.     Object[] d = x;
6. }
```

What is the result?

A. Compilation succeeds.
B. Compilation fails due only to an error on line 3.
C. Compilation fails due only to an error on line 4.
D. Compilation fails due only to an error on line 5.
E. Compilation fails due to errors on lines 3 and 5.
F. Compilation fails due to errors on lines 3, 4, and 5.

Answer:

- **A** is correct, all of these array declarations are legal. **Lines 4 and 5 demonstrate that arrays can be cast.**
- **B, C, D, E,** and **F** are incorrect because this code compiles. (Objective 1.3)

Given:

```
1. class Dims {
2.     public static void main(String[] args) {
3.            int[][] a = {{1,2,}, {3,4}};
4.            int[] b = (int[]) a[1];
5.            Object o1 = a;
6.            int[][] a2 = (int[][]) o1;
7.            int[] b2 = (int[]) o1;
8.            System.out.println(b[1]);
9.     }
   }
```

What is the result?
A. `2`
B. `4`
C. An exception is thrown at runtime
D. Compilation fails due to an error on line 4.
E. Compilation fails due to an error on line 5.
F. Compilation fails due to an error on line 6.
G. Compilation fails due to an error on line 7.
Answer:
- **C** is correct. A `ClassCastException` is thrown at line 7 because `o1` refers to an `int[][]` not an `int[]`. If line 7 was removed, the output would be 4.
- **A, B, D, E, F,** and **G** are incorrect based on the above. (Objective 1.3)

Initialization Blocks (Objectives 1.3 and 7.6)

- **Static initialization blocks run once, when the class is first loaded.**
- **Instance initialization blocks run every time a new instance is created. They run after all super-constructors and before the constructor's code has run.**
- **If multiple init blocks exist in a class, they follow the rules stated above, AND they run in the order in which they appear in the source file.**

Given:
```
class Bird {
       { System.out.print("b1 "); }
       public Bird() { System.out.print("b2 "); }
}
class Raptor extends Bird {
       static { System.out.print("r1 "); }
       public Raptor() { System.out.print("r2 "); }
       { System.out.print("r3 "); }
       static { System.out.print("r4 "); }
}
class Hawk extends Raptor {
       public static void main(String[] args) {
              System.out.print("pre ");
              new Hawk();
              System.out.println("hawk ");
       }
}
```
What is the result?
A. `pre b1 b2 r3 r2 hawk`
B. `pre b2 b1 r2 r3 hawk`
C. `pre b2 b1 r2 r3 hawk r1 r4`
D. `r1 r4 pre b1 b2 r3 r2 hawk`
E. `r1 r4 pre b2 b1 r2 r3 hawk`
F. `pre r1 r4 b1 b2 r3 r2 hawk`
G. `pre r1 r4 b2 b1 r2 r3 hawk`
H. The order of output cannot be predicted.
I. Compilation fails.
Answer:
- **D** is correct. Static init blocks are executed at class loading time, instance init blocks run right after the call to `super()` in a constructor. When multiple init blocks of a single type occur in a class, they run in order, from the top down.
- **A, B, C, E, F, G, H,** and **I** are incorrect based on the above. Note: you'll probably never see this many choices on the real exam! (Objective 1.3)

Using Wrappers (Objective 3.1)

- The wrapper classes correlate to the primitive types.
- Wrappers have two main functions:
  - To wrap primitives so that they can be handled like objects
  - To provide utility methods for primitives (usually conversions)
- The three most important method families are
  - **`xxxValue()` Takes no arguments, returns a primitive**
  - **`parseXxx()` Takes a String, returns a primitive, throws NFE (NumberFormatException).**
  - **`valueOf()` Takes a String, returns a wrapped object, throws NFE**
- Wrapper constructors can take a String or a primitive, except for Character, which can only take a `char`.
- Radix refers to bases (typically) other than 10; octal is radix = 8, hex = 16.

Given:

```
1. class Convert {
2.     public static void main(String[] args) {
3.           Long xL = new Long(456L);
4.           long x1 = Long.valueOf("123");
5.           Long x2 = Long.valueOf("123");
6.           long x3 = xL.longValue();
7.           Long x4 = xL.longValue();
8.           Long x5 = Long.parseLong("456");
9.           long x6 = Long.parseLong("123");
10.    }
11. }
```

Which will compile using Java 5, but will NOT compile using Java 1.4? (Choose all that apply.)

A. Line 4.
B. Line 5.
C. Line 6.
D. Line 7.
E. Line 8.
F. Line 9.

Answer:

- **A, D,** and **E** are correct. Because of the methods' return types, these method calls required autoboxing to compile.
- **B, C,** and **F** are incorrect based on the above. (Objective 3.1)

Boxing (Objective 3.1)

- As of Java 5, **boxing allows you to convert primitives to wrappers or to convert wrappers to primitives automatically.**
- **IMP: Using == with wrappers is tricky; wrappers with the same small values (typically lower than 127), will be ==, larger values will not be ==.**

Advanced Overloading (Objectives 1.5 and 5.4)

- Primitive widening uses the "smallest" method argument possible.
- Used individually, boxing and var-args are compatible with overloading.
- **You CANNOT widen from one wrapper type to another. (IS-A fails.) So a Short cannot be widened to a Long or Integer but it can be widened to Object/Number.**
- You CANNOT widen and then box. (An `int` can't become a Long.)
- **You can box and then widen. (An `int` can become an Object, via an Integer.) So short can box to Short and then widen to Number in the example below.**
- You can combine var-args with either widening or boxing.

IMP:

Given:

```
class Eggs {
      int doX(Long x, Long y) { return 1; }
      int doX(long... x) { return 2; }
```

```
        int doX(Integer x, Integer y) { return 3; }
        int doX(Number n, Number m) { return 4; }
        public static void main(String[] args) {
                new Eggs().go();
        }
        void go() {
                short s = 7;
                System.out.print(doX(s,s) + " ");
                System.out.println(doX(7,7));
        }
}
```

What is the result?

A. `1 1`
B. `2 1`
C. `3 1`
D. `4 1`
E. `2 3`
F. `3 3`
G. `4 3`

Answer:

- **G** is correct. Two rules apply to the first invocation of `doX()`. You can't widen and then box in one step, and **var-args are always chosen last**. Therefore **you can't widen shorts to either ints or longs, and then box them to Integers or Longs.** But you can box shorts to Shorts and then widen them to Numbers, and this takes priority over using a var-args method. The second invocation uses a simple box from int to Integer.
- **A, B, C, D, E,** and **F** are incorrect based on the above. (Objective 3.1)

Garbage Collection (Objective 7.4)

- In Java, garbage collection (GC) provides automated memory management.
- The purpose of GC is to delete objects that can't be reached.
- Only the JVM decides when to run the GC, you can only suggest it.
- You can't know the GC algorithm for sure.
- Objects must be considered eligible before they can be garbage collected.
- **An object is eligible when no live thread can reach it.**
- To reach an object, you must have a live, reachable reference to that object.
- Java applications can run out of memory.
- **Islands of objects can be GCed, even though they refer to each other.**
- Request garbage collection with `System.gc();` (recommended).
- Class Object has a `finalize()` method.
- **The `finalize()` method is guaranteed to run once and only once before the garbage collector deletes an object.**
- The garbage collector makes no guarantees, `finalize()` may never run.
- You can uneligibilize an object for GC from within `finalize()`.

Given:

```
class CardBoard {
        Short story = 5;
        CardBoard go(CardBoard cb) {
                cb = null;
                return cb;
        }
        public static void main(String[] args) {
                CardBoard c1 = new CardBoard();
                CardBoard c2 = new CardBoard();
                CardBoard c3 = c1.go(c2);
                c1 = null;
                // do Stuff
        }
}
```

When `// doStuff` is reached, how many objects are eligible for GC?

A. 0
B. 1
C. 2
D. Compilation fails.
E. It is not possible to know.
F. An exception is thrown at runtime.
Answer:

- **C** is correct. Only one `CardBoard object (c1)` is eligible, but it has an associated **Short wrapper object that is also eligible.**
- **A, B, D, E,** and **F** are incorrect based on the above. (Objective 7.4)

Which is true? (Choose all that apply.)
A. The invocation of an object's `finalize()` method is always the last thing that happens before an object is garbage collected (GCed).
B. When a stack variable goes out of scope it is eligible for GC.
C. Some reference variables live on the stack, and some live on the `heap`.
D. Only objects that have no reference variables referring to them can be eligible for GC.
E. It's possible to request the GC via methods in either `java.lang.Runtime` or `java.lang.System` classes.
Answer:

- **C** and **E** are correct. **When an object has a reference variable, the reference variable lives inside the object, on the heap.**
- **A** is incorrect, because if, the first time an object's `finalize()` method runs, the object is saved from the GC, then the second time that object is about to be GCed, `finalize()` will not run. **B** is incorrect—stack variables are not dealt with by the GC. **D** is incorrect because objects can live in "islands of isolation" and be GC eligible. (Objective 7.4)

Operators

Relational Operators

- Relational operators always result in a `boolean` value (`true` or `false`).
- There are six relational operators: >, >=, <, <=, ==, and !=. The last two (== and !=) are sometimes referred to as *equality operators*.
- When comparing characters, Java uses the Unicode value of the character as the numerical value.
- Equality operators
  - There are two equality operators: == and !=.
- Four types of things can be tested: numbers, characters, booleans, and reference variables.
- **When comparing reference variables, == returns `true` only if both references refer to the same object.**

Given:

```
1. class Comp2 {
2.     public static void main(String[] args) {
3.           float f1 = 2.3f;
4.           float[][] f2 = {{42.0f}, {1.7f, 2.3f}, {2.6f, 2.7f}};
5.           float[] f3 = {2.7f};
6.           Long x = 42L;
7.           // insert code here
8.                 System.out.println("true");
9.     }
10. }
```

And the following five code fragments:

```
F1. if(f1 == f2)
```

```
F2. if(f1 == f2[2][1])
F3. if(x == f2[0][0])
F4. if(f1 == f2[1,1])
F5. if(f3 == f2[2])
```

What is true?

A. One of them will compile, only one will be `true`.
B. Two of them will compile, only one will be `true`.
C. Two of them will compile, two will be `true`.
D. Three of them will compile, only one will be `true`.
E. Three of them will compile, exactly two will be `true`.
F. Three of them will compile, exactly three will be `true`.

Answer:

- **D** is correct. Fragments `F2`, `F3`, and `F5` will compile, and only `F3` is `true`.
- **A, B, C, E,** and **F** are incorrect. `F1` is incorrect because you can't compare a primitive to an array. `F4` is incorrect syntax to access an element of a two-dimensional array. (Objective 7.6)

Given:

```
class Titanic {
       public static void main(String[] args) {
              Boolean b1 = true;
              boolean b2 = false;
              boolean b3 = true;
              if((b1 & b2) | (b2 & b3) & b3)
                     System.out.print("alpha ");
              if((b1 = false) | (b1 & b3) | (b1 | b2))
                     System.out.print("beta ");
       }
}
```

What is the result?

A. `beta`
B. `alpha`
C. `alpha beta`
D. Compilation fails.
E. No output is produced.
F. An exception is thrown at runtime.

Answer:

- **E** is correct. In the second `if` test, the leftmost expression is an assignment, not a comparison. Once `b1` has been set to `false`, the remaining tests are all `false`.
- **A, B, C, D,** and **F** are incorrect based on the above. (Objective 7.6)

instanceof Operator

- **`instanceof` is for reference variables only, and checks for whether the object is of a particular type.**
- The `instanceof` operator can be used only to test objects (or `null`) against class types that are in the same class hierarchy.
- **For interfaces, an object passes the `instanceof` test if any of its superclasses implement the interface on the right side of the `instanceof` operator.**

  Given:

```
1. class Example {
2.     public static void main(String[] args) {
3.            Short s = 15;
4.            Boolean b;
5.            // insert code here
6.     }
7. }
```

Which, inserted independently at line 5, will compile? (Choose all that apply.)
A. `b = (Number instanceof s);`
B. `b = (s instanceof Short);`
C. `b = s.instanceof(Short);`
D. `b = (s instanceof Number);`
E. `b = s.instanceof(Object);`
F. `b = (s instanceof String);`
Answer:
- **B** and **D** correctly use boxing and `instanceof` together.
- **A** is incorrect because the operands are reversed. **C** and **E** use incorrect `instanceof` syntax. **F** is wrong because `Short` isn't in the same inheritance tree as `String`. (Objective 7.6)

Arithmetic Operators

- There are four primary math operators: add, subtract, multiply, and divide.
- The remainder operator (%), returns the remainder of a division.

```
public class Sixties {
        public static void main(String[] args) {
                int x = 5;
                int y = 7;
                System.out.print(((y * 2) % x));
                System.out.print(" " + (y % x));
        }
}
```

What is the result?
A. `1 1`
B. `1 2`
C. `2 1`
D. `2 2`
E. `4 1`
F. `4 2`
G. Compilation fails.
H. An exception is thrown at runtime.
Answer:
- **F** is correct. The % (remainder a.k.a. modulus) operator returns the remainder of a division operation.
- **A, B, C, D, E, G,** and **H** are incorrect based on the above. (Objective 7.6)

- Expressions are evaluated from left to right, unless you add parentheses, or unless some operators in the expression have higher precedence than others.
- **The *, /, and % operators have higher precedence than + and -.**

String Concatenation Operator

- If either operand is a `String`, the + operator concatenates the operands.
- If both operands are numeric, the + operator adds the operands.

Given:

```
class Feline {
       public static void main(String[] args) {
              Long x = 42L;
              Long y = 44L;
              System.out.print(" " + 7 + 2 + " "); // concatenates
              System.out.print(foo() + x + 5 + " "); // concatenates
              System.out.println(x + y + foo()); // adds x and y
       }
       static String foo() { return "foo"; }
```

```
}
```
What is the result?
A. `9 foo47 86foo`
B. `9 foo47 4244foo`
C. `9 foo425 86foo`
D. `9 foo425 4244foo`
E. `72 foo47 86foo`
F. `72 foo47 4244foo`
G. `72 foo425 86foo`
H. `72 foo425 4244foo`
I. Compilation fails.
Answer:
- **G** is correct. **Concatenation runs from left to right, and if either operand is a `String`, the operands are concatenated. If both operands are numbers they are added together. Unboxing works in conjunction with concatenation.**
- **A, B, C, D, E, F, H,** and **I** are incorrect based on the above. (Objective 7.6)

Increment/Decrement Operators

- **Prefix operators (++ and --) run before the value is used in the expression.**
- Postfix operators (++ and --) run after the value is used in the expression.
- **In any expression, both operands are fully evaluated *before* the operator is applied.**
- **Variables marked `final` cannot be incremented or decremented.**

Ternary (Conditional Operator)

- Returns one of two values based on whether a `boolean` expression is `true` or `false`.
- Returns the value after the `?` if the expression is `true`.
- Returns the value after the `:` if the expression is `false`.

Logical Operators

- The exam covers six "logical" operators: &, |, ^, !, &&, and ||.
- Logical operators work with two expressions (except for !) that must resolve to `boolean` values.
- The && and & operators return `true` only if both operands are `true`.
- The || and | operators return `true` if either or both operands are `true`.
- The && and || operators are known as short-circuit operators.
- The && operator does not evaluate the right operand if the left operand is `false`.
- The || does not evaluate the right operand if the left operand is `true`.
- **The & and | operators always evaluate both operands and don't short circuit.**
- **The ^ operator (called the "logical XOR"), returns `true` if exactly one operand is `true`. To remember, XORing means inequality test.. so if one is true and other false then it returns true whereas if both are true or false then it returns false.**

Given:
```
class Fork {
      public static void main(String[] args) {
            if(args.length == 1 | args[1].equals("test")) {
                  System.out.println("test case");
            } else {
                  System.out.println("production " + args[0]);
            }
      }
}
```
And the command-line invocation:
```
java Fork live2
```
What is the result?
A. `test case`
B. `production`