**Identifiers**

1. Identifiers must start with a letter, a currency character ($), or a connecting character such as the underscore (\_). Identifiers cannot start with a number!
2. After the first character, identifiers can contain any combination of letters, currency characters, connecting characters, or numbers.
3. In practice, there is no limit to the number of characters an identifier can contain.
4. You can't use a Java keyword as an identifier. Table 1-1 lists all of the Java keywords including one new one for 5.0, enum.
5. Identifiers in Java are case-sensitive; foo and FOO are two different identifiers.

**Declaration Rules**

1. A source code file can have only one public class.
2. Package and import statements apply to all classes in the file.
3. A file can have more than one nonpublic class.
4. Files with no public classes have no naming restrictions.

**Class Modifiers fall into two categories:**

1. Access modifiers: public, protected, private., default
2. Non-access modifiers (including strictfp, final, and abstract).

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.

**strictfp**

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.

**Interfaces**

1. All interface methods are implicitly public and abstract. In other words, 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.
2. All variables defined in an interface are by default public, static, and final in other words, interfaces can declare only constants, not instance variables.
3. Interface methods must not be static.
4. Because interface methods are abstract, they cannot be marked final,
5. strictfp, or native. (More on these modifiers later.)
6. An interface can *extend* one or more other interfaces.
7. An interface cannot extend anything but another interface.
8. An interface cannot implement another interface or class.
9. An interface must be declared with the keyword interface.
10. The public modifier is required if you want the interface to have public rather than default access.

**Access Modifiers for Members**

**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).

For a subclass, if a member of its superclass is declared public, the subclass inherits that member regardless of whether both classes are in the same package.

**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. A subclass can't see, use, or even think about the private members of its superclass. You can, however, declare a matching method in the subclass. But regardless of how it looks, ***it is not an overriding method!*** It is simply a method that happens to have the same name as a private method (which you're not supposed to know about) in the superclass. The rules of overriding do not apply, so you can make this newly-declared-but-just-happens-to-match method declare new exceptions, or change the return type.

Using private members enables data protection (encapsulation), like setting an age field directly could lead to negative age value but using an accessor method will keep this under check.

**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.***

But what does it mean for a subclass-outside-the-package to have access to a superclass (parent) member? It means the subclass inherits the member. It does not, however, mean the subclass-outside-the-package can access the member using a reference to an instance of the superclass. In other words, protected = inheritance.

Protected does not mean that the subclass can treat the protected superclass member as though it were public. So if the subclass-outside-the-package gets a reference to the superclass (by, for example, creating an instance of the superclass somewhere in the subclass' code), the subclass cannot use the dot operator on the superclass reference to access the protected member. To a subclass-outside-the-package, a protected member might as well be default (or even private), when the subclass is using a reference to the superclass. **The subclass can see the** protected **member only through inheritance.**

But there's still one more issue we haven't looked at...what does a protected member look like to other classes trying to use the subclass-outside-the-package to get to the subclass' inherited protected superclass member? For example, using our previous Parent/Child classes, what happens if some other class—Neighbor, say—in the same package as the Child (subclass), has a reference to a Child instance and wants to access the member variable x? In other words, how does that protected member behave once the subclass has inherited it? Does it maintain its protected status, such that classes in the Child's package can see it?

No! Once the subclass-outside-the-package inherits the protected member, that member (as inherited by the subclass) becomes private to any code outside the subclass, with the exception of subclasses of the subclass. So if class Neighbor instantiates a Child object, then even if class Neighbor is in the same package as class Child, class Neighbor won't have access to the Child's inherited (but protected) variable x. The bottom line: when a subclass-outside-the-package inherits a protected member, the member is essentially private inside the subclass, such that only the subclass and its subclasses can access it. While in case of default access we can access the variable using a parent class instance in child class.

**Final Methods/Variables**

The final keyword prevents a method from being overridden in a subclass, and is often used to enforce the API functionality of a method.

public Record getRecord(int fileNumber, int recordNumber) {}

Method arguments are essentially the same as local variables. In the preceding example, the variables fileNumber and recordNumber will both follow all the rules applied to local variables. This means they can also have the modifier final.

**Abstract Methods**

A method can never, ever be marked as both abstract and final, or both abstract and private. Think about it—abstract methods must be implemented (which essentially means overridden by a subclass) whereas final and private methods cannot ever be overridden by a subclass.

**Finally, you need to know that the abstract modifier can never be combined with the static modifier.**

**Methods with Variable Argument Lists (var-args)**

1. **Var-arg type** When you declare a var-arg parameter, you must specify the type of the argument(s) this parameter of your method can receive. (This can be a primitive type or an object type.)
2. **Basic syntax** To declare a method using a var-arg parameter, you follow the type with an ellipsis (...), a space, and then the name of the array that will hold the parameters received.
3. **Other parameters** It's legal to have other parameters in a method that uses a var-arg.
4. **Var-args limits** The var-arg must be the last parameter in the method's signature, and you can have only one var-arg in a method.

**Local (Automatic/Stack/Method) Variables**

Local variables are variables declared within a method. That means the variable is not just initialized within the method, but also declared within the method. Just as the local variable starts its life inside the method, it's also destroyed when the method has completed. Local variables are always on the stack, not the heap. Just don't forget that while the local variable is on the stack, if the variable is an object reference, the object itself will still be created on the heap. There is no such thing as a stack object, only a stack variable.

It is possible to declare a local variable with the same name as an instance variable. It's known as *shadowing*

Comparison of modifiers on variables vs. methods

|  |  |  |
| --- | --- | --- |
| Local Variables | Instance variables | Methods |
| Final | Final  Public  Protected  Private  Static  Transient  Volatile | Final  Public  Protected  Private  Static  Abstract  Synchronized  Strictfp  native |

**Boolean variables bit depth is virtual machine dependent.**

**Final Variables**

A reference variable marked final can't ever be reassigned to refer to a different object. The data within the object can be modified, but the reference variable cannot be changed. In other words, a final reference still allows you to modify the state of the object it refers to, but you can't modify the reference variable to make it refer to a different object. 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. It can be applied only to instance variables.

**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. It can be applied only to instance variables.

**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. In other words, static members exist before you ever make a new instance of a class, and there will be only one copy of the static member regardless of the number of instances of that class. In other words, all instances of a given class share the same value for any given static variable.

Things you can mark as static:

1. Methods
2. Variables
3. A class nested within another class, but not within a method
4. Initialization blocks

Things you can't mark as static:

1. Constructors (makes no sense; a constructor is used only to create instances)
2. Classes (unless they are nested)
3. Interfaces
4. Method local inner classes
5. Inner class methods and instance variables
6. Local variables

**Declaring Enums:**

As of 5.0, Java lets you restrict a variable to having one of only a few pre-defined values—in other words, one value from an enumerated list.

Enums can be declared as their own separate class, or as a class member, however they must not be declared within a method.

The key point to remember is that the enum can be declared with only the public or default modifier, just like a non-inner class.

An enum declared outside a class must NOT be marked static, final, abstract, protected, or private.

Because an enum really is a special kind of class, you can do more than just list the enumerated constant values. You can add constructors, instance variables, methods and something really strange known as a *constant specific class body*.

**Inner Classes**

Inner classes let you define one class within another. They provide a type of scoping for your classes since you can make one class *a member of another class*. Just as classes have member *variables* and *methods*, a class can also have member *classes*. Sometimes, though, you find yourself designing a class where you discover you need behavior that belongs in a separate, specialized class, but also needs to be intimately tied to the class you're designing.

**Regular Inner Class**

An inner class is a full-fledged member of the enclosing (outer) class, so it can be marked with an access modifier as well as the abstract or final modifiers. (Never both abstract and final together— remember that abstract *must* be subclassed, whereas final *cannot* be subclassed).

An inner class instance shares a special relationship with an instance of the enclosing class. This relationship gives the inner class access to *all* of the outer class's members, including those marked private. To create an instance of an inner class, *you must have an instance of the outer class* to tie to the inner class.

MyOuter mo = new MyOuter(); // gotta get an instance!

**MyOuter.MyInner inner = mo.new MyInner();**

So the rules for an inner class referencing itself or the outer instance are as follows:

To reference the inner class instance itself, from *within* the inner class code, use this.

To reference the "*outer* this" (the outer class instance) from within the inner class code, use NameOfOuterClass.this (example, MyOuter.this).

**Member Modifiers Applied to Inner Classes** A regular inner class is a member of the outer class just as instance variables and methods are, so the following modifiers can be applied to an inner class:

final

abstract

public

private

protected

static—*but* static *turns it into a* static *nested class not an inner class.*

Strictfp

**Method-Local Inner Classes**

If you want to actually *use* the inner class (say, to invoke its methods), then you must make an instance of it somewhere *within the method but below the inner class definition*.

*A method-local inner class can be instantiated only within the method where the inner class is defined*.

Like regular inner class objects, the method-local inner class object shares a special relationship with the enclosing (outer) class object, and can access its private (or any other) members. However, *the inner class object cannot use the local variables of the method the inner class is in.* The local variables of the method live on the stack, and exist only for the lifetime of the method. You already know that the scope of a local variable is limited to the method the variable is declared in. When the method ends, the stack frame is blown away and the variable is history. But even after the method completes, the inner class object created within it might still be alive on the heap if, for example, a reference to it was passed into some other code and then stored in an instance variable. Because the local variables aren't guaranteed to be alive as long as the method-local inner class objects, the inner class object can't use them. *Unless the local variables are marked* final*! Only modifiers available are final and abstract.*

**Anonymous Inner Classes**

You can only call methods on an anonymous inner class reference that are defined in the reference variable type! The only difference between flavor one and flavor two is that flavor one creates an anonymous *subclass* of the specified *class* type, whereas flavor two creates an anonymous *implementer* of the specified *interface* type.

Cookable c = new Cookable() {

Declare a reference variable of type Cookable that, obviously, will refer to an object from a class that implements the Cookable interface. But, oh yes, we don't yet *have* a class that implements Cookable, so we're going to make one right here, right now. We don't need a name for the class, but it will be a class that implements Cookable, and this curly brace starts the definition of the new implementing class."One more thing to keep in mind about anonymous interface implementers—*they can implement only one interface*.

An anonymous inner class is always created as part of a statement; don't forget to close the statement after the class definition with a curly brace. This is a rare case in Java, a curly brace followed by a semicolon.

Because of polymorphism, the only methods you can call on an anonymous inner class reference are those defined in the reference variable class (or interface), even though the anonymous class is really a subclass or implementer of the reference variable type.

An anonymous inner class can extend one subclass *or* implement one interface. Unlike non-anonymous classes (inner or otherwise), an anonymous inner class cannot do both. In other words, it cannot both extend a class *and* implement an interface, nor can it implement more than one interface.

An argument-local inner class is declared, defined, and automatically instantiated as part of a method invocation. The key to remember is that the class is being defined within a method argument, so the syntax will end the class definition with a curly brace, followed by a closing parenthesis to end the method call, followed by a semicolon to end the statement: });

**Static Nested Classes**

A static nested class is simply *a class that's a static member of the enclosing class*:

class BigOuter {

static class Nested { }

}

The class itself isn't really "static"; there's no such thing as a static class. The static modifier in this case says that the nested class is *a static member of the outer class*. That means it can be accessed, as with other static members, *without having an instance of the outer class.* ***Just as a static method does not have access to the instance variables and non-static methods of the class, a static nested class does not have access to the instance variables and non-static methods of the outer class. Look for static nested classes with code that behaves like a non static (regular inner) class.***

**Immutable Objects**

1. No setter methods.
2. All fields are final, private.
3. Class is final.
4. If members are mutable objects then
5. Don’t provide methods to modify them.
6. Assign passed references by copying, similarly create a copy while returning the mutable objects

**Encapsulation**

Imagine you wrote the code for a class, and another dozen programmers from your company all wrote programs that used your class. Now imagine that later on, you didn't like the way the class behaved, because some of its instance variables were being set (by the other programmers from within their code) to values you hadn't anticipated. *Their* code brought out errors in *your* code. Well, it is a Java program, so you should be able just to ship out a newer version of the class, which they could replace in their programs without changing any of their own code. **The ability to make changes in your implementation code without breaking the code of others who use your code is a key benefit of encapsulation.** You want to hide implementation details behind a public programming interface.

For encapsulation:

Keep instance variables protected (with an access modifier, often private).

Make public accessor methods, and force calling code to use those methods rather than directly accessing the instance variable.

For the methods, use the JavaBeans naming convention of set<someProperty> and get<someProperty>.

**Inheritance**

It's also important to understand that the two most common reasons to use inheritance are

1. To promote code reuse
2. To use polymorphism

**IS-A**

In OO, the concept of IS-A is based on class inheritance or interface implementation. IS-A is a way of saying, "this thing is a type of that thing." For example, a Mustang is a type of horse, so in OO terms we can say, "Mustang IS-A Horse." Subaru IS-A Car.

**HAS-A**

HAS-A relationships are based on usage, rather than inheritance. In other words, class A HAS-A B if code in class A has a reference to an instance of class B.

**Polymorphism**

1. A reference variable can be of only one type, and once declared, that type can never be changed (although the object it references can change).
2. A reference is a variable, so it can be reassigned to other objects, (unless the reference is declared final).
3. A reference variable's type determines the methods that can be invoked on the object the variable is referencing.
4. A reference variable can refer to any object of the same type as the declared reference, or—this is the big one—**it can refer to any** *subtype* **of the declared type!**
5. A reference variable can be declared as a class type or an interface type. If the variable is declared as an interface type, it can reference any object of any class that *implements* the interface.
6. Polymorphic method invocations apply only to overridden *instance* methods.

Multiple inheritance is supported through interfaces only since allowing extension of two different classes might lead to ambiguity, two different implementations of the same method.

***Some languages (like C++) allow a class to extend more than one other class. This capability is known as "multiple inheritance." The reason that Java's creators chose not to allow multiple inheritance is that it can become quite messy. In a nutshell, the problem is that if a class extended two other classes, and both superclasses had, say, a doStuff() method, which version of doStuff() would the subclass inherit? This issue can lead to a scenario known as the "Deadly Diamond of Death," because of the shape of the class diagram that can be created in a multiple inheritance design. The diamond is formed when classes B and C both extend A, and both B and C inherit a method from A. If class D extends both B and C, and both B and C have overridden the method in A, class D has, in theory, inherited two different implementations of the same method. Drawn as a class diagram, the shape of the four classes looks like a diamond.***

**Polymorphic method invocations apply only to *instance methods.* You can always refer to an object with a more general reference variable type (a superclass or interface), but at runtime, the ONLY things that are dynamically selected based on the actual *object* (rather than the *reference* type) are instance methods. Not *static* methods. Not *variables*. Only overridden instance methods are dynamically invoked based on the real object's type.**

**Overriding**

The rules for overriding a method are as follows:

1. The argument list must exactly match that of the overridden method. If they don't match, you can end up with an overloaded method you didn't intend.
2. The return type must be the same as, or a subtype of, the return type declared in the original overridden method in the superclass. The access level can't be more restrictive than the overridden method's.
3. Instance methods can be overridden only if they are inherited by the subclass. A subclass within the same package as the instance's superclass can override any superclass method that is not marked private or final. A subclass in a different package can override only those non-final methods marked public or protected (since protected methods are inherited by the subclass).
4. The overriding method CAN throw any unchecked (runtime) exception, regardless of whether the overridden method declares the exception.
5. The overriding method must NOT throw checked exceptions that are new or broader than those declared by the overridden method. For example, a method that declares a FileNotFoundException cannot be overridden by a method that declares a SQLException, Exception, or any other non-runtime exception unless it's a subclass of FileNotFoundException.
6. The overriding method can throw narrower or fewer exceptions. Just because an overridden method "takes risks" doesn't mean that the overriding subclass' exception takes the same risks. Bottom line: an overriding method doesn't have to declare any exceptions that it will never throw, regardless of what the overridden method declares.
7. You cannot override a method marked final.
8. You cannot override a method marked static.
9. If a method can't be inherited (because it’s private), you cannot override it. Remember that overriding implies that you're reimplementing a method you inherited!
10. Using super to invoke an overridden method only applies to instance methods.
11. A subclass uses super.overriddenMethodName() to call the superclass version of an overridden method.

**Overloading**

Over*loaded* methods let you reuse the same method name in a class, but with different arguments.

1. Overloaded methods MUST change the argument list.
2. Overloaded methods CAN change the return type.
3. Overloaded methods CAN change the access modifier.
4. Overloaded methods CAN declare new or broader checked exceptions.
5. A method can be overloaded in the *same* class or in a *subclass*. In other words, if class A defines a doStuff(int i) method, the subclass B could define a doStuff(String s) method without overriding the superclass version that takes an int. So two methods with the same name but in different classes can still be considered overloaded, if the subclass inherits one version of the method and then declares another overloaded version in its class definition.
6. The choice of which overloaded method to call (in other words, the signature of the method) is NOT dynamically decided at runtime. Just remember, the *reference* type (not the object type) determines which overloaded method is invoked!
7. So it's true that polymorphism doesn't determine which overloaded version is called; polymorphism does come into play when the decision is about which overridden version of a method is called.
8. **In every case, when an exact match isn't found, the JVM uses the method with the smallest argument that is wider than the parameter.**
9. **For overloaded methods the calling is done in the following order of preference**
10. **Widening**
11. **Boxing**
12. **Var args**
13. Primitive widening uses the "smallest" method argument possible.
14. Used individually, boxing and var-args are compatible with overloading.
15. You CANNOT widen from one wrapper type to another. (IS-A fails.)
16. You CANNOT widen and then box. (An int can't become a Long.)
17. You can box and then widen. (An int can become an Object, via Integer.)
18. You can combine var-args with either widening or boxing.

|  |  |  |
| --- | --- | --- |
|  | **Overloaded Methods** | **Overridden Methods** |
| Argument(s). | Must change | Must not change. |
| Return type | Can change. | Can’t change except for covariant returns. |
| Exceptions | Can change. | Can reduce or eliminate.  Must not throw new or broader checked exceptions. |
| Access | Can change. | Must not make more restrictive (can be less restrictive). |
| Invocation | *Reference* type determines which overloaded version (based on declared argument types) is selected. Happens at *compile* time. The actual *method* that’s invoked is still a virtual method invocation that happens at runtime, but the compiler will already know the *signature* of the method to be invoked. So at runtime, the argument match will already have been nailed down, just not the *class* in which the method lives. | *Object* type (in other words, *the type of the actual instance on the heap*) determines which method is selected.  Happens at *runtime*. |

**Reference Variable Casting**

1. All the compiler can do is verifying that the two types are in the same inheritance tree, so that depending on whatever code might have come before the downcast. However, if the compiler knows with certainty that the cast could not possibly work, compilation will fail.
2. Implicit upcasting is allowed while implicit downcasting is not.

**Implementing an Interface**

1. Provide concrete (nonabstract) implementations for all methods from the declared interface.
2. Follow all the rules for legal overrides.
3. Declare no checked exceptions on implementation methods other than those declared by the interface method, or subclasses of those declared by the interface method.
4. Maintain the signature of the interface method, and maintain the same return type (or a subtype). (But it does not have to declare the exceptions declared in the interface method declaration.)
5. A class can implement more than one interface.
6. An interface can itself extend another interface, but never implement anything.
7. Overriding the interface methods is possible.

**Return Type Declarations**

**Return Types on Overloaded Methods**

So if you inherit a method but overload it in a subclass, you're not subject to the restrictions of overriding, which mean you can declare any return type you like. What you can't do is change only the return type. To overload a method, remember, you must change the argument list.

**Overriding and Return Types and Covariant Returns**

Or, as of Java 5, you're allowed to change the return type in the overriding method as long as the new return type is a *subtype* of the declared return type of the overridden (superclass) method.

**Constructors**

So if your super constructor (that is, the constructor of your immediate superclass/parent) has arguments, you must type in the call to super(), supplying the appropriate arguments. Crucial point: if your superclass does not have a no-argconstructor, you must type a constructor in your class (the subclass) because you need a place to put in the call to super with the appropriate arguments. Another way to put this is that if your superclass does *not* have a no-arg constructor, then in your subclass you will not be able to use the default constructor supplied by the compiler.

The first line in a constructor must be a call to super() or a call to this().In other words, if constructor A() has a call to this(), the compiler knows that constructor A() will not be the one to invoke super().The preceding rule means a constructor can never have both a call to super() and a call to this().

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.

Constructors are never inherited, thus they cannot be overridden.

Issues with calls to this():

1. May appear only as the first statement in a constructor.
2. The argument list determines which overloaded constructor is called.

**Static**

We don't actually need to initialize a static variable; static variables get the same default values instance variables get.

Finally, remember that *static methods can't be overridden*! This doesn't mean they can't be redefined in a subclass, but redefining and overriding isn’t the same thing.

Use static methods to implement behaviors that are not affected by the state of any instances, i.e. not specific to an instance or depending on the state of instance.

**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. 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.

**Literals, Assignments, and Variables**

**Integer Literals**

There are three ways to represent integer numbers in the Java language: decimal (base 10), octal (base 8), and hexadecimal (base 16).

**Decimal Literals**

int length = 343;

**Octal Literals**

int nine = 011;

You can have up to 21 digits in an octal number, not including the leading zero.

**Hexadecimal Literals**

int z = 0xDeadCafe;

You are allowed up to 16 digits in a hexadecimal number, not including the prefix 0x or the optional suffix extension L, which will be explained later.

**Floating-Point Literals**

Floating-point literals are defined as double (64 bits) by default, so if you want to assign a floating-point literal to a variable of type float (32 bits), you must attach the suffix F or f to the number

float g = 49837849.029847F; // OK; has the suffix "F"

You may also optionally attach a D or d to double literals, but it is not necessary because this is the default behavior.

**Boolean Literals**

Boolean literals are the source code representation for boolean values. A Boolean value can only be defined as true or false.

**Character Literals**

A char literal is represented by a single character in single quotes. You can also type in the Unicode value of the character, using the Unicode notation of prefixing the value with \u as follows:

char letterN = '\u004E'; // The letter 'N'

Remember, characters are just 16-bit unsigned integers under the hood. That means you can assign a number literal, assuming it will fit into the unsigned 16-bit range (65535 or less).

**Assignment**

A variable referring to an object is just that—a *reference* variable. A reference variable bit holder contains bits representing a *way to get to the object*.

We know that a literal integer is always an int, but more importantly, the result of an expression involving anything int-sized or smaller is always an int. In other words, add two bytes together and you'll get an int—even if those two bytes are tiny. Multiply an int and a short and you'll get an int. Divide a short by a byte and you'll get…an int.

**Primitive Casting**

Typically, an implicit cast happens when you're doing a widening conversion. In other words, putting a smaller thing (say, a byte) into a bigger container (like an int).

The large-value-into-small-container conversion is referred to as *narrowing* and requires an explicit cast, where you tell the compiler that you're aware of the danger and accept full responsibility. You don't get a runtime error, even when the value being narrowed is too large for the type. When you narrow a primitive, Java simply truncates the higher-order bits that won't fit. In other words, it loses all the bits to the left of the bits you're narrowing to.

**Variable Scope**

Static variables have the longest scope; they are created when the class is loaded, and they survive as long as the class stays loaded in the JVM.

Instance variables are the next most long-lived; they are created when a new instance is created, and they live until the instance is removed.

Local variables are next; they live as long as their method remains on the stack. As we'll soon see, however, local variables can be alive, and still be "out of scope".

Block variables live only as long as the code block is executing.

Most VM implementations have reference pointing to memory location which contains the pointer to the object it is referring to. Microsoft VM though uses the references directly pointing to the object location in memory.

Narrowing a primitive truncates the *high order* bits.

Integer expressions always result in an int-sized result, never smaller.

Floating-point numbers are implicitly doubles (64 bits).

**Arrays**

*Array elements are always, always, always given default values, regardless of where the array itself is declared or instantiated. (i.e. static, instance, local).*

It makes no difference if you're passing primitive or reference variables, you are always passing a copy of the bits in the variable. So for a primitive variable, you're passing a copy of the bits representing the value.

**Constructing Arrays**

int[][] myArray = new int[3][];

Notice that only the first brackets are given a size. That's acceptable in Java, since the JVM needs to know only the size of the object assigned to the variable myArray.

int[] dots = {6,x,8};

Dog puppy = new Dog("Frodo");

Dog[] myDogs = {puppy, new Dog("Clover"), new Dog("Aiko")};

int[][] scores = {{5,2,4,7}, {9,2}, {3,4}};

The preceding code creates a total of four objects on the heap.

**Arrays of Primitives**

Primitive arrays can accept any value that can be promoted implicitly to the declared type of the array. For example, an int array can hold any value that can fit into a 32-bit int variable.

**Arrays of Object References**

If the declared array type is a class, you can put objects of any subclass of the declared type into the array.

**Array Reference Assignments for One-Dimensional Arrays**

int[] splats;

int[] dats = new int[4];

char[] letters = new char[5];

splats = dats; // OK, dats refers to an int array

splats = letters; // NOT OK, letters refers to a char array

Arrays that hold object references, as opposed to primitives, aren't as restrictive. Just as you can put a Honda object in a Car array (because Honda extends Car), you can assign an array of type Honda to a Car array reference variable.

**Array Reference Assignments for Multidimensional Arrays**

When 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.

int[][] books = new int[3][];

int[] numbers = new int[6];

int aNumber = 7;

books[0] = aNumber; // NO, expecting an int array not an int

books[0] = numbers; // OK, numbers is an int array

**Initialization Blocks**

A *static* initialization block runs *once*, when the class is first loaded. An *instance* initialization block runs once *every time a new instance is created.* Instance init block code runs rightafter the call to super() in a constructor, in other words, after all super-constructors have run. When it's time for initialization blocks to run, if a class has more than one, they will run in the order in which they appear in the class file…in other words, from the top down.

**Using Wrapper Conversion Utilities**

**xxxValue()**

When you need to convert the value of a wrapped numeric to a primitive, use one of the many xxxValue() methods.

**parseXxx() and valueOf()**

parseXxx() returns the named primitive.

valueOf() returns a newly created wrapped object of the type that invoked the method.

**toXxxString() (Binary, Hexadecimal, Octal)**

String s3 = Integer.toHexString(254); // convert 254 to hex

System.out.println("254 is " + s3); // result: "254 is fe"

**In order to save memory, two instances of the following wrapper objects will always be == when their** primitive values are the same:

Boolean

Byte

Character from \u0000 to \u007f (7f is 127 in decimal)

Short and Integer from -128 to 127

*If either operand is a* String*, the + operator becomes a* String *concatenation operator. If both operands are numbers, the + operator is the addition operator.*

A switch's expression must evaluate to a char, byte, short, int, or, as of Java 5, an enum, as of Java 7 a String ***The*** default ***case doesn’t have to come at the end of the*** switch

The first part of the for statement lets you declare and initialize zero, one, or multiple variables of the same type inside the parentheses after the for keyword. The rule to remember is this: *You can have only one test expression*. You can have any number of iterator expressions. **Keep in mind that barring a forced exit, evaluating the iteration expression and then evaluating the conditional expression are always the last two things that happen in a** for **loop!.**

***Labeled*** continue ***and*** break ***statements must be inside the loop that has the same label name; otherwise, the code will not compile.***

**Writing Code Using control flow Statements**

The only legal expression in an if statement is a boolean expression, in other words an expression that resolves to a boolean or a boolean variable.

Watch out for boolean assignments (=) that can be mistaken for Boolean equality (==) tests:

boolean x = false; if (x = true) { } // an assignment, so x will always be true!

switch statements can evaluate only to enums, string or the byte, short, int, and char data types. You can't say,

long s = 30; switch(s) { }

The case constant must be a literal or final variable, or a constant expression, including an enum or string. You cannot have a case that includes a nonfinal variable, or a range of values.

If the condition in a switch statement matches a case constant, execution will run through all code in the switch following the matching case statement until a break statement or the end of the switch statement is encountered. In other words, the matching case is just the entry point into the case block, but unless there's a break statement, the matching case is not the only case code that runs.

The default keyword should be used in a switch statement if you want to run some code when none of the case values match the conditional value.

The default block can be located anywhere in the switch block, so if no case matches, the default block will be entered, and if the default does not contain a break, then code will continue to execute to the end of the switch or until the break statement is encountered.

You can initialize more than one variable of the same type in the first part of the basic for loop declaration; each initialization must be separated by a comma.

You cannot use a number (old C-style language construct) or anything that does not evaluate to a boolean value as a condition for an if statement or looping construct. You can't, for example, say if(x), unless x is a boolean variable.

An unlabeled break statement will cause the current iteration of the innermost looping construct to stop and the line of code following the loop to run.

An unlabeled continue statement will cause: the current iteration of the innermost loop to stop, the condition of that loop to be checked, and if the condition is met, the loop to run again.

If the break statement or the continue statement is labeled, it will cause similar action to occur on the labeled loop, not the innermost loop.

**Formatting**

%[arg\_index$][flags][width][.precision]conversion char

The values within [ ] are optional. In other words, the only required elements of a format string are the % and a conversion character.

arg\_index: An integer followed directly by a $, this indicates which argument should be printed in this position.

flags:

"-" Left justify this argument

"+" Include a sign (+ or -) with this argument

"0" Pad this argument with zeroes

"," Use locale-specific grouping separators (i.e., the comma in 123,456)

"(" Enclose negative numbers in parentheses

width: This value indicates the minimum number of characters to print. (If you want nice even columns, you'll use this value extensively.)

precision: For the exam you'll only need this when formatting a floating-point number, and in the case of floating point numbers, precision indicates the number of digits to print after the decimal point.

conversion: The type of argument you'll be formatting. You'll need to know:

b boolean

c char

d integer

f floating point

s string

int i1 = -123;

int i2 = 12345;

System.out.printf(">%1$(7d< \n", i1);

System.out.printf(">%0,7d< \n", i2);

System.out.format(">%+-7d< \n", i2);

System.out.printf(">%2$b + %1$5d< \n", i1, false);

This produces:

> (123)<

>012,345<

>+12345 <

>false + -123<

**Exception Handling**

|  |
| --- |
| Object  Error  Checked Exception  Runtime Exception/ Unchecked Exception  Error Subclasses  Exception  Throwable |

A finally block encloses code that is always executed at some point after the try block, whether an exception was thrown or not. Even if there is a return statement in the try block, the finally block executes right after the return statement is encountered, and before the return executes!

***It is illegal to use a*** try ***clause without either a*** catch ***clause or a*** finally ***clause. A*** try ***clause by itself will result in a compiler error. Any*** catch ***clauses must immediately follow the*** try ***block. Any*** finally ***clause must immediately follow the last*** catch ***clause (or it must immediately follow the*** try ***block if there is no*** catch***). It is legal to omit either the*** catch ***clause or the*** finally ***clause, but not both.***

*Each method must either handle all checked exceptions by supplying a* catch *clause or list each unhandled checked exception as a thrown exception.*

RuntimeException, Error, and all of their subtypes are unchecked exceptions and unchecked exceptions do not have to be specified or handled.

Both Exception and Error share a common superclass, Throwable, thus both can be thrown using the throw keyword. When an Error or a subclass of Error is thrown, it's unchecked.

The only exception to the finally-will-always-be-called rule is that a finally will not be invoked if the JVM shuts down. That could happen if code from the try or catch blocks calls System.exit().

Just because finally is invoked does not mean it will complete. Code in the finally block could itself raise an exception or issue a System.exit().

Uncaught exceptions propagate back through the call stack, starting from the method where the exception is thrown and ending with either the first method that has a corresponding catch for that exception type or a JVM shutdown (which happens if the exception gets to main(), and main() is "ducking" the exception by declaring it).

You can create your own exceptions, normally by extending Exception or one of its subtypes. Your exception will then be considered a checked exception, and the compiler will enforce the handle or declare rule for that exception. If you extend the RuntimeException class or one of its subtypes then an unchecked exception gets created.

All catch blocks must be ordered from most specific to most general. If you have a catch clause for both IOException and Exception, you must put the catch for IOException first in your code. Otherwise, the IOException would be caught by catch(Exception e), because a catch argument can catch the specified exception or any of its subtypes! The compiler will stop you from defining catch clauses that can never be reached.

**The try-with-resources statement** is a try statement that declares one or more resources. A resource is an object that must be closed after the program is finished with it. The try-with-resources statement ensures that each resource is closed at the end of the statement. Any object that implements java.lang.AutoCloseable, which includes all objects which implement java.io.Closeable, can be used as a resource.

You may declare one or more resources in a try-with-resources statement.

A try-with-resources statement can have catch and finally blocks just like an ordinary try statement. In a try-with-resources statement, any catch or finally block is run after the resources declared have been closed.

An exception can be thrown from the block of code associated with the try-with-resources statement. In the example writeToFileZipFileContents, an exception can be thrown from the try block, and up to two exceptions can be thrown from the try-with-resources statement when it tries to close the ZipFile and BufferedWriter objects. If an exception is thrown from the try block and one or more exceptions are thrown from the try-with-resources statement, then those exceptions thrown from the try-with-resources statement are suppressed, and the exception thrown by the block is the one that is thrown by the writeToFileZipFileContents method. You can retrieve these suppressed exceptions by calling the Throwable.getSuppressed method from the exception thrown by the try block.

**JDBC**

Short for Java Database Connectivity, a Java API that enables Java programs to execute SQL statements. This allows Java programs to interact with any SQL-compliant database. Since nearly all relational database management systems (DBMSs) support SQL, and because Java itself runs on most platforms, JDBC makes it possible to write a single database application that can run on different platforms and interact with different DBMSs.

JDBC is similar to ODBC, but is designed specifically for Java programs, whereas ODBC is language-independent

Components of JDBC - JDBC Components, Connection Pools, Data Sources (A DataSource class brings another level of abstraction than directly using a connection object. Data source can be referenced by JNDI. Data Source may point to RDBMS, file System, any DBMS etc.), and MultiPools (pool of connection pools).

Loading Drivers - Class.forName(”Driver”);

Getting connection - Connection con = DriverManager.getConnection(url,“myLogin”, “myPassword”);

**JDBC Statements**

java.sql.Statement - Top most interface which provides basic methods useful for executing SELECT, INSERT, UPDATE and DELETE SQL statements.

java.sql.PreparedStatement - An enhanced version of java.sql.Statement which allows precompiled queries with parameters. It is more efficient to use java.sql.PreparedStatement if you have to specify parameters to your SQL queries. If you want to execute a Statement object many times, it will normally reduce execution time to use a PreparedStatement object instead.

The main feature of a PreparedStatement object is that, unlike a Statement object, it is given an SQL statement when it is created. The advantage to this is that in most cases, this SQL statement will be sent to the DBMS right away, where it will be compiled. As a result, the PreparedStatement object contains not just an SQL statement, but an SQL statement that has been precompiled. This means that when the PreparedStatement is executed, the DBMS can just run the PreparedStatement’s SQL statement without having to compile it first.

Although PreparedStatement objects can be used for SQL statements with no parameters, you will probably use them most often for SQL statements that take parameters. The advantage of using SQL statements that take parameters is that you can use the same statement and supply it with different values each time you execute it.

java.sql.CallableStatement - Allows you to execute stored procedures within a RDBMS which supports stored procedures (MySQL doesn’t support stored procedures at the moment).

Batch Updates - If you want to execute a set of statements, i.e. SQL statements at a time then we use batch update statement. resultset=pst.batchUpdate();

Let’s say there are 100 records need to be insert. If we execute normal statemets the no of transactions will be 100 (in terms of connection making to DB). using batch updates we can add 100 rec to batch and the no of transactions will be only one in this case. This will reduce the burdon on db, which is very costly in terms of resources.

**Stored procedure**

A stored procedure is a group of SQL statements that form a logical unit and perform a particular task. Stored procedures are used to encapsulate a set of operations or queries to execute on a database server. For example, operations on an employee database (hire, fire, promote, lookup) could be coded as stored procedures executed by application code. Stored procedures can be compiled and executed with different parameters and results, and they may have any combination of input, output, and input/output parameters. You can call a stored procedure using Callable statements.

AutoCommit - The DML operations by default are committed. If we wish to avoid the commit by default, setAutoCommit(false) has to be called on the Connection object. Once the statements are executed, commit() has to be called on the Connection object explicitly.

**RowSet**

The interface that adds support to the JDBC API for the JavaBeansTM component model. A rowset, which can be used as a JavaBeans component in a visual Bean development environment, can be created and configured at design time and executed at run time.

The RowSet interface provides a set of JavaBeans properties that allow a RowSet instance to be configured to connect to a JDBC data source and read some data from the data source. A group of setter methods (setInt, setBytes, setString, and so on) provide a way to pass input parameters to a rowset’s command property. This command is the SQL query the rowset uses when it gets its data from a relational database, which is generally the case.

The RowSet interface supports JavaBeans events, allowing other components in an application to be notified when an event occurs on a rowset, such as a change in its value.

The RowSet interface is unique in that it is intended to be implemented using the rest of the JDBC API. In other words, a RowSet implementation is a layer of software that executes “on top” of a JDBC driver. Implementations of the RowSet interface can be provided by anyone, including JDBC driver vendors who want to provide a RowSet implementation as part of their JDBC products.

A RowSet object may make a connection with a data source and maintain that connection throughout its life cycle, in which case it is called a connected rowset. A rowset may also make a connection with a data source, get data from it, and then close the connection. Such a rowset is called a disconnected rowset. A disconnected rowset may make changes to its data while it is disconnected and then send the changes back to the original source of the data, but it must reestablish a connection to do so.

**ResultSet**

A table of data representing a database result set, which is usually generated by executing a statement that queries the database.

A ResultSet object maintains a cursor pointing to its current row of data. Initially the cursor is positioned before the first row. The next method moves the cursor to the next row, and because it returns false when there are no more rows in the ResultSet object, it can be used in a while loop to iterate through the result set.

A default ResultSet object is not updatable and has a cursor that moves forward only. Thus, you can iterate through it only once and only from the first row to the last row. It is possible to produce ResultSet objects that are scrollable and/or updatable. The following code fragment, in which con is a valid Connection object, illustrates how to make a result set that is scrollable and insensitive to updates by others, and that is updatable. See ResultSet fields for other options.

**SQLWarning**

SQLWarning objects are a subclass of SQLException that deal with database access warnings. Warnings do not stop the execution of an application, as exceptions do.

They simply alert the user that something did not happen as planned. A warning can be reported on a Connection object, a Statement object (including PreparedStatement and CallableStatement objects), or a ResultSet object. Each of these classes has a getWarnings method, which you must invoke in order to see the first warning reported on the calling object.

E.g.

SQLWarning warning = stmt.getWarnings();

if (warning != null) {

while (warning != null) {

System.out.println(”Message: ” + warning.getMessage());

System.out.println(”SQLState: ” + warning.getSQLState());

System.out.print(”Vendor error code: “);

System.out.println(warning.getErrorCode());

warning = warning.getNextWarning();

}

}

How many types of JDBC Drivers are present and what are they?

There are 4 types of JDBC Drivers

• JDBC-ODBC Bridge Driver

• Native API Partly Java Driver

• Network protocol Driver

• JDBC Net pure Java Driver (Fastest)

**File I/O**

|  |
| --- |
| ObjectInput  DataInput  InputStream  Closeable  StringBufferInputStream  ObjectInputStream  ByteArrayInputStream  (Reading froma buffer containing the bytes)  FileInputStream  (Reading from a file)  FilterInputStream  PipedInputStream  SequenceInputStream  BufferedInputStream  DataInputStream  DataInputStream  dwdwefwfhth  LineNumberInputStream  PushbackInputStream |

**InputStream**

This abstract class is the superclass of all classes representing an input stream of bytes. Applications that need to define a subclass of InputStream must always provide a method that returns the next byte of input.

**ByteArrayInputStream**

A ByteArrayInputStream contains an internal buffer that contains bytes that may be read from the stream. An internal counter keeps track of the next byte to be supplied by the read method. Closing a ByteArrayInputStream has no effect. The methods in this class can be called after the stream has been closed without generating an IOException.

**FileInputStream**

A FileInputStream obtains input bytes from a file in a file system. What files are available depends on the host environment.FileInputStream is meant for reading streams of raw bytes such as image data. For reading streams of characters, consider using FileReader.

**FilterInputStream**

A FilterInputStream contains some other input stream, which it uses as its basic source of data, possibly transforming the data along the way or providing additional functionality. The class FilterInputStream itself simply overrides all methods of InputStream with versions that pass all requests to the contained input stream. Subclasses of FilterInputStream may further override some of these methods and may also provide additional methods and fields.

**BufferedInputStream**

A BufferedInputStream adds functionality to another input stream-namely, the ability to buffer the input and to support the mark and reset methods. When the BufferedInputStream is created, an internal buffer array is created. As bytes from the stream are read or skipped, the internal buffer is refilled as necessary from the contained input stream, many bytes at a time. The mark operation remembers a point in the input stream and the reset operation causes all the bytes read since the most recent mark operation to be reread before new bytes are taken from the contained input stream.

**DataInputStream**

A data input stream lets an application read primitive Java data types from an underlying input stream in a machine-independent way. An application uses a data output stream to write data that can later be read by a data input stream. DataInputStream is not necessarily safe for multithreaded access. Thread safety is optional and is the responsibility of users of methods in this class. One final point about most of DataInputStream’s methods: when the end of the stream is reached, they throw an EOFException.

**PushbackInputStream**

A PushbackInputStream adds functionality to another input stream, namely the ability to "push back" or "unread" one byte. This is useful in situations where it is convenient for a fragment of code to read an indefinite number of data bytes that are delimited by a particular byte value; after reading the terminating byte, the code fragment can "unread" it, so that the next read operation on the input stream will reread the byte that was pushed back. For example, bytes representing the characters constituting an identifier might be terminated by a byte representing an operator character; a method whose job is to read just an identifier can read until it sees the operator and then push the operator back to be re-read.

**PipedInputStream**

A piped input stream should be connected to a piped output stream; the piped input stream then provides whatever data bytes are written to the piped output stream. Typically, data is read from a PipedInputStream object by one thread and data is written to the corresponding PipedOutputStream by some other thread. Attempting to use both objects from a single thread is not recommended, as it may deadlock the thread. The piped input stream contains a buffer, decoupling read operations from write operations, within limits. A pipe is said to be broken if a thread that was providing data bytes to the connected piped output stream is no longer alive.

**SequenceInputStream**

A SequenceInputStream represents the logical concatenation of other input streams. It starts out with an ordered collection of input streams and reads from the first one until end of file is reached, whereupon it reads from the second one, and so on, until end of file is reached on the last of the contained input streams.

|  |
| --- |
| ObjectOutput  DataOutput  Flushable  Closeable  OutputStream  ByteArrayOutputStream  (Reading froma buffer containing the bytes)  ObjectOutputStream  PipedOutputStream  FilterOutputStream  FileOutputStream  (Reading from a file)  BufferedOutputStream  PrintStream  DataInputStream  dwdwefwfhth  DataOutputStream  DataInputStream  dwdwefwfhth |

**OutputStream**

This abstract class is the superclass of all classes representing an output stream of bytes. An output stream accepts output bytes and sends them to some sink. Applications that need to define a subclass of OutputStream must always provide at least a method that writes one byte of output.

**ByteArrayOutputStream**

This class implements an output stream in which the data is written into a byte array. The buffer automatically grows as data is written to it. The data can be retrieved using toByteArray() and toString(). Closing a ByteArrayOutputStream has no effect. The methods in this class can be called after the stream has been closed without generating an IOException.

**FilterOutputStream**

This class is the superclass of all classes that filter output streams. These streams sit on top of an already existing output stream (the underlying output stream) which it uses as its basic sink of data, but possibly transforming the data along the way or providing additional functionality. The class FilterOutputStream itself simply overrides all methods of OutputStream with versions that pass all requests to the underlying output stream. Subclasses of FilterOutputStream may further override some of these methods as well as provide additional methods and fields.

**FileOutputStream**

A file output stream is an output stream for writing data to a File or to a FileDescriptor. Whether or not a file is available or may be created depends upon the underlying platform. Some platforms, in particular, allow a file to be opened for writing by only one FileOutputStream (or other file-writing object) at a time. In such situations the constructors in this class will fail if the file involved is already open. FileOutputStream is meant for writing streams of raw bytes such as image data. For writing streams of characters, consider using FileWriter.

**BufferedOutputStream**

The class implements a buffered output stream. By setting up such an output stream, an application can write bytes to the underlying output stream without necessarily causing a call to the underlying system for each byte written.

**DataOutputStream**

A data output stream lets an application write primitive Java data types to an output stream in a portable way. An application can then use a data input stream to read the data back in.

**PrintStream**

A PrintStream adds functionality to another output stream, namely the ability to print representations of various data values conveniently. Two other features are provided as well. Unlike other output streams, a PrintStream never throws an IOException; instead, exceptional situations merely set an internal flag that can be tested via the checkError method. Optionally, a PrintStream can be created so as to flush automatically; this means that the flush method is automatically invoked after a byte array is written, one of the println methods is invoked, or a newline character or byte ('\n') is written.

All characters printed by a PrintStream are converted into bytes using the platform's default character encoding. The [PrintWriter](http://java.sun.com/javase/6/docs/api/java/io/PrintWriter.html) class should be used in situations that require writing characters rather than bytes.

**PipedOutputStream**

A piped output stream can be connected to a piped input stream to create a communications pipe. The piped output stream is the sending end of the pipe. Typically, data is written to a PipedOutputStream object by one thread and data is read from the connected PipedInputStream by some other thread. Attempting to use both objects from a single thread is not recommended as it may deadlock the thread. The pipe is said to be broken if a thread that was reading data bytes from the connected piped input stream is no longer alive.

PipedInputStream sIn = PipedInputStream();

PipedOutputStream sOut = PipedOutputStream(sIn);

**Serialization**

When you serialize an object, Java serialization takes care of saving that object's entire "object graph." That means a deep copy of everything the saved object needs to be restored.

If you mark an instance variable with transient, then serialization will simply skip it during serialization.

private void writeObject(ObjectOutputStream os) {

// your code for saving the Collar variables

}

private void readObject(ObjectInputStream os) {

// your code to read the Collar state, create a new Collar,

// and assign it to the Dog

}

1. Like most I/O-related methods writeObject() can throw exceptions. You can declare them or handle them but we recommend handling them.

2. When you invoke defaultWriteObject() from within writeObject() you're telling the JVM to do the normal serialization process for this object. When implementing writeObject(), you will typically request the normal serialization process, *and* do some custom writing and reading too.

3. In this case we decided to write an extra int (the collar size) to the stream that's creating the serialized Dog. You can write extra stuff before and/or after you invoke defaultWriteObject(). BUT…when you read it back in, you have to read the extra stuff in the same order you wrote it. Again, we chose to handle rather than declare the exceptions.

5. When it's time to deserialize, defaultReadObject() handles the normal deserialization you'd get if you didn't implement a readObject() method.

6. Finally we build a new Collar object for the Dog using the collar size that we manually serialized. (We had to invoke readInt() *after* we invoked defaultReadObject() or the streamed data would be out of sync!)

7. If you have variables marked transient, they will not be restored to their original state (unless you implement defaultReadObject()), but will instead be given the default value for that data type.

8. If you are a serializable class, but your superclass is NOT serializable, then any instance variables you INHERIT from that superclass will be reset to the values they were given during the original construction of the object. This is because the nonserializable class constructor WILL run!

**ObjectOutputStream**

An ObjectOutputStream writes primitive data types and graphs of Java objects to an OutputStream. The objects can be read (reconstituted) using an ObjectInputStream. The default serialization mechanism for an object writes the class of the object, the class signature, and the values of all non-transient and non-static fields. References to other objects (except in transient or static fields) cause those objects to be written also. Enum constants are serialized differently than ordinary serializable or externalizable objects. The serialized form of an enum constant consists solely of its name; field values of the constant are not transmitted. To serialize an enum constant, ObjectOutputStream writes the string returned by the constant's name method. The process by which enum constants are serialized cannot be customized; any class-specific writeObject and writeReplace methods defined by enum types are ignored during serialization. Similarly, any serialPersistentFields or serialVersionUID field declarations are also ignored--all enum types have a fixed serialVersionUID of 0L.

**ObjectInputStream**

ObjectInputStream is used to recover those objects previously serialized. Other uses include passing objects between hosts using a socket stream or for marshaling and unmarshaling arguments and parameters in a remote communication system. Only objects that support the java.io.Serializable or java.io.Externalizable interface can be read from streams. No-arg constructors are invoked for the non-serializable classes and then the fields of the serializable classes are restored from the stream starting with the serializable class closest to java.lang.object and finishing with the object's most specific class.

The readObjectNoData method is responsible for initializing the state of the object for its particular class in the event that the serialization stream does not list the given class as a superclass of the object being deserialized. This may occur in cases where the receiving party uses a different version of the deserialized instance's class than the sending party, and the receiver's version extends classes that are not extended by the sender's version. To deserialize an enum constant, ObjectInputStream reads the constant name from the stream; the deserialized constant is then obtained by calling the static method Enum.valueOf(Class, String) with the enum constant's base type and the received constant name as arguments.

**Collections**

A *collections framework* is a unified architecture for representing and manipulating collections. All collections frameworks contain the following:

**Interfaces:** These are abstract data types that represent collections. Interfaces allow collections to be manipulated independently of the details of their representation. In object-oriented languages, interfaces generally form a hierarchy.

**Implementations:** These are the concrete implementations of the collection interfaces. In essence, they are reusable data structures.

**Algorithms:** These are the methods that perform useful computations, such as searching and sorting, on objects that implement collection interfaces. The algorithms are said to be *polymorphic*: that is, the same method can be used on many different implementations of the appropriate collection interface. In essence, algorithms are reusable functionality.

Collections come in four basic flavors:

1. **Lists** *Lists* of things (classes that implement List).
2. **Sets** *Unique* things (classes that implement Set).
3. **Maps** Things with a *unique* ID (classes that implement Map).
4. **Queues** Things arranged by the order in which they are to be processed.

|  |  |
| --- | --- |
| **Java.lang.Comparable** | **Java.util.Comparator** |
| int objOne.compareTo(objTwo) | int compare(objOne, objTwo) |
| Returns negative if objOne < objTwo  zero if objOne == objTwo positive if objOne > objTwo | Same as Comparable |
| You must modify the class whose  instances you want to sort. | You build a class separate from the class whose instances you want to sort. |
| Only **one** sort sequence can be created | **Many** sort sequences can be created |
| Implemented frequently in the API by:  String, Wrapper classes, Date, Calendar... | Meant to be implemented to sort instances of third-party  classes. |

* The sort() methods for both the Collections class and the Arrays class are static methods, and that they alter the objects they are sorting, instead of returning a different sorted object.
* There's one Big Truth you need to know to understand why generics run without problems—the JVM has no idea that your ArrayList was supposed to hold only Integers. The typing information does not exist at runtime! All your generic code is strictly for the compiler. Through a process called "type erasure," the compiler does all of its verifications on your generic code and then strips the type information out of the class byte code. At runtime, ALL collection code—both legacy and new Java 5 code you write using generics—looks exactly like the pre-generic version of collections. None of your typing information exists at runtime.
* **public static** <T **extends** Object & Comparable<? **super** T>>

T max(Collection<T> coll)

This is an example of giving *multiple bounds* for a type parameter, using the syntax T1 & T2 ... & Tn. A type variable with multiple bounds is known to be a subtype of all of the types listed in the bound. When a multiple bound is used, the first type mentioned in the bound is used as the erasure of the type variable.

**Searching Arrays and Collections**

The Collections class and the Arrays class both provide methods that allow you to search for a specific element. When searching through collections or arrays, the following rules apply:

1. Searches are performed using the binarySearch() method.
2. Successful searches return the int index of the element being searched.
3. Unsuccessful searches return an int index that represents the *insertion point*. The insertion point is the place in the collection/array where the element would be inserted to keep the collection/array properly sorted. Because positive return values and 0 indicate successful searches, the binarySearch() method uses negative numbers to indicate insertion points. Since 0 is a valid result for a successful search, the first available insertion point is -1. Therefore, the actual insertion point is represented as (-(insertion point) -1). For instance, if the insertion point of a search is at element 2, the actual insertion point returned will be -3.
4. The collection/array being searched must be sorted before you can search it.
5. If you attempt to search an array or collection that has not already been sorted, the results of the search will not be predictable.
6. If the collection/array you want to search was sorted in natural order, it *must* be searched in natural order. (This is accomplished by NOT sending a Comparator as an argument to the binarySearch() method.)
7. If the collection/array you want to search was sorted using a Comparator, it *must* be searched using the same Comparator, which is passed as the second argument to the binarySearch() method. Remember that Comparators cannot be used when searching arrays of primitives.

**Generics**

1. Generics let you enforce compile-time type safety on Collections (or other classes and methods declared using generic type parameters).
2. An ArrayList<Animal> can accept references of type Dog, Cat, or any other subtype of Animal (subclass, or if Animal is an interface, implementation).
3. When using generic collections, a cast is not needed to get (declared type) elements out of the collection. With non-generic collections, a cast is required:

List<String> gList = new ArrayList<String>();

List list = new ArrayList();

// more code

String s = gList.get(0); // no cast needed

String s = (String)list.get(0); // cast required

1. You can pass a generic collection into a method that takes a non-generic collection, but the results may be disastrous. The compiler can't stop the method from inserting the wrong type into the previously type safe collection.
2. If the compiler can recognize that non-type-safe code is potentially endangering something you originally declared as type-safe, you will get a compiler warning. For instance, if you pass a List<String> into a method declared as void foo(List aList) { aList.add(anInteger); } the compiler will issue a warning because the add() method is potentially an "unsafe operation. Remember that "compiles without error" is not the same as "compiles without warnings."
3. Generic type information does not exist at runtime—it is for compile-time safety only. Mixing generics with legacy code can create compiled code that may throw an exception at runtime.
4. Polymorphic assignment applies only to the base type, not the generic type parameter. You can say

List<Animal> aList = new ArrayList<Animal>(); // yes

You can't say

List<Animal> aList = new ArrayList<Dog>(); // no

1. The polymorphic assignment rule applies everywhere an assignment can be made. The following are NOT allowed:

void foo(List<Animal> aList) { } // cannot take a List<Dog>

List<Animal> bar() { } // cannot return a List<Dog>

1. Wildcard syntax allows a generic method, accept subtypes (or supertypes) of the declared type of the method argument:

void addD(List<Dog> d) {} // can take only <Dog>

void addD(List<? extends Dog>) {} // take a <Dog> or <Beagle>

1. The wildcard keyword extends is used to mean either "extends" or "implements." So in <? extends Dog>, Dog can be a class or an interface.
2. **When using a wildcard, List<? extends Dog>, the collection can be accessed but not modified.**
3. **When using a wildcard, List<?>, any generic type can be assigned to the reference, but for access only, no modifications.**
4. As a workaround to modify the list in case of wildcard capture the helper methods can be used.

The WildcardError example produces a capture error when compiled:

import java.util.List;

public class WildcardError {

void foo(List<?> i) {

i.set(0, i.get(0));

}

}

In this example, the compiler processes the i input parameter as being of type Object. When the foo method invokes List.set(int, E), the compiler is not able to confirm the type of object that is being inserted into the list, and an error is produced. When this type of error occurs it typically means that the compiler believes that you are assigning the wrong type to a variable. Generics were added to the Java language for this reason — to enforce type safety at compile time.

The WildcardError example generates the following error when compiled by Oracle's JDK 7 javac implementation:

WildcardError.java:6: error: method set in interface List<E> cannot be applied to given types;

i.set(0, i.get(0));

^

required: int,CAP#1

found: int,Object

reason: actual argument Object cannot be converted to CAP#1 by method invocation conversion

where E is a type-variable:

E extends Object declared in interface List

where CAP#1 is a fresh type-variable:

CAP#1 extends Object from capture of ?

1 error

In this example, the code is attempting to perform a safe operation, so how can you work around the compiler error? You can fix it by writing a private helper method which captures the wildcard. In this case, you can work around the problem by creating the private helper method, fooHelper, as shown in WildcardFixed:

public class WildcardFixed {

void foo(List<?> i) {

fooHelper(i);

}

// Helper method created so that the wildcard can be captured

// through type inference.

private <T> void fooHelper(List<T> l) {

l.set(0, l.get(0));

}

}

1. Declaration conventions for generics use T for type and E for element:

public interface List<E> // API declaration for List

boolean add(E o) // List.add() declaration

1. The generics type identifier can be used in class, method, and variable declarations:

class Foo<T> { } // a class

T anInstance; // an instance variable

Foo(T aRef) {} // a constructor argument

void bar(T aRef) {} // a method argument

T baz() {} // a return type

The compiler will substitute the actual type.

1. You can use more than one parameterized type in a declaration:

public class UseTwo<T, X> { }

1. You can declare a generic method using a type not defined in the class:

public <T> void makeList(T t) { }

It is NOT using T as the return type. This method has a void return type, but to use T within the method's argument you must declare the <T>, which happens before the return type.

1. Multiple Bounds
2. The preceding example illustrates the use of a type parameter with a single bound, but a type parameter can have multiple bounds:

<T extends B1 & B2 & B3>

A type variable with multiple bounds is a subtype of all the types listed in the bound. If one of the bounds is a class, it must be specified first. For example:

Class A { /\* ... \*/ }

interface B { /\* ... \*/ }

interface C { /\* ... \*/ }

class D <T extends A & B & C> { /\* ... \*/ }

If bound A is not specified first, you get a compile-time error:

class D <T extends B & A & C> { /\* ... \*/ } // compile-time error

1. Effects of Type Erasure and Bridge Methods

Sometimes type erasure causes a situation that you may not have anticipated. The following example shows how this can occur. The example (described in Bridge Methods) shows how a compiler sometimes creates a synthetic method, called a bridge method, as part of the type erasure process.

Given the following two classes:

public class Node<T> {

public T data;

public Node(T data) { this.data = data; }

public void setData(T data) {

System.out.println("Node.setData");

this.data = data;

}

}

public class MyNode extends Node<Integer> {

public MyNode(Integer data) { super(data); }

public void setData(Integer data) {

System.out.println("MyNode.setData");

super.setData(data);

}

}

Consider the following code:

MyNode mn = new MyNode(5);

Node n = mn; // A raw type - compiler throws an unchecked warning

n.setData("Hello");

Integer x = mn.data; // Causes a ClassCastException to be thrown.

After type erasure, this code becomes:

MyNode mn = new MyNode(5);

Node n = (MyNode)mn; // A raw type - compiler throws an unchecked warning

n.setData("Hello");

Integer x = (String)mn.data; // Causes a ClassCastException to be thrown.

Here is what happens as the code is executed:

n.setData("Hello"); causes the method setData(Object) to be executed on the object of class MyNode. (The MyNode class inherited setData(Object) from Node.)

In the body of setData(Object), the data field of the object referenced by n is assigned to a String.

The data field of that same object, referenced via mn, can be accessed and is expected to be an integer (since mn is a MyNode which is a Node<Integer>.

Trying to assign a String to an Integer causes a ClassCastException from a cast inserted at the assignment by a Java compiler.

Bridge Methods

When compiling a class or interface that extends a parameterized class or implements a parameterized interface, the compiler may need to create a synthetic method, called a bridge method, as part of the type erasure process. You normally don't need to worry about bridge methods, but you might be puzzled if one appears in a stack trace.

After type erasure, the Node and MyNode classes become:

public class Node {

public Object data;

public Node(Object data) { this.data = data; }

public void setData(Object data) {

System.out.println("Node.setData");

this.data = data;

}

}

public class MyNode extends Node {

public MyNode(Integer data) { super(data); }

public void setData(Integer data) {

System.out.println("MyNode.setData");

super.setData(data);

}

}

After type erasure, the method signatures do not match. The Node method becomes setData(Object) and the MyNode method becomes setData(Integer). Therefore, the MyNode setData method does not override the Node setData method.

To solve this problem and preserve the polymorphism of generic types after type erasure, a Java compiler generates a bridge method to ensure that subtyping works as expected. For the MyNode class, the compiler generates the following bridge method for setData:

class MyNode extends Node {

// Bridge method generated by the compiler

//

public void setData(Object data) {

setData((Integer) data);

}

public void setData(Integer data) {

System.out.println("MyNode.setData");

super.setData(data);

}

// ...

}

As you can see, the bridge method, which has the same method signature as the Node class's setData method after type erasure, delegates to the original setData method.

1. Generics constructor

A constructor can be declared as generic, independently of whether the class that the constructor is declared in is itself generic. A constructor is generic if it declares one or more type variables. These type variables are known as the formal type parameters of the constructor. The form of the formal type parameter list is identical to a type parameter list of a generic class or interface. The interface constructor is generic.

class Test {

//Generics constructor

public <T> Test(T item){

System.out.println("Value of the item: " + item);

System.out.println("Type of the item: "

+ item.getClass().getName());

}

}

**Annotations**

1. A form of meta-data, provide data about a program that is not part of the program itself.
2. Helps the compiler to detect errors or suppress warnings.
3. Software tools can make use of annotations to generate any code or XML files and so forth.
4. Helps for runtime processing sometimes.
5. If @Retention annotation is not declared, the retention policy defaults to CLASS. Possible Values (SOURCE (discarded by the compiler), CLASS (retained by the compiler, discarded by JVM), RUNTIME (retained by JVM)
6. If Target is not declared, the declared type may be used on any program element. Possible Values (ANNOTATION\_TYPE, CONSTRUCTOR, FIELD, LOCAL\_VARIABLE, METHOD, TYPE, PACKAGE, PARAMETER)
7. Marker annotation does not contains any member declarations. By looking at presence of the annotation, we can do some specific actions.
8. Rules for defining Java annotations:

Annotation declaration should start with an ‘at’ sign like @, following with an interface keyword, following with the annotation name.

Method declarations should not have any parameters.

Method declarations should not have any throws clauses.

Return types of the method should be one of the following:-

primitives

String

Class

enum

array of the above types

The below annotations are used for annotations applied to other annotations:-

@Retention: – Specifies how the marked annotation is stored — whether in code only, compiled into the class, or available at runtime through reflection.

@Documented: – Marks another annotation for inclusion in the documentation.

@Target: – Marks another annotation to restrict what kind of java elements the annotation may be applied to

@Inherited: – Marks another annotation to be inherited to subclasses of annotated class (by default annotations are not inherited to subclasses).

1. Built in annotations - @Override, @Deprecated, @SuppressWarnings.

**Garbage Collection**

In Java, garbage collection (GC) provides automated memory management. The purpose of GC is to delete objects that can't be reached. Objects must be considered eligible before they can be garbage collected. An object is eligible when no live thread can reach it. Islands of objects can be GCed, even though they refer to each other.

Only the JVM decides when to run the GC, you can only suggest it. Request garbage collection with System.gc();.

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 make an object ineligible for GC from within finalize().

Java 7 introduced the G1 collector intended to be replacement of CMS collectors it supports both compaction and allows for more predictable pause times being met since it calculates the number of regions which can be swept to meet the user specified pause time goal.

Java 8 has removed PermGen space and introduced metaspace for class definitions. Internalized Strings will be moved to heap space.

Behavior-Based Tuning

For the parallel collector, Java SE provides two garbage collection tuning parameters that are based on achieving a specified behavior of the application: maximum pause time goal and application throughput goal; see the section The Parallel Collector. (These two options are not available in the other collectors.) Note that these behaviors cannot always be met. The application requires a heap large enough to at least hold all of the live data. In addition, a minimum heap size may preclude reaching these desired goals.

Maximum Pause Time Goal

The pause time is the duration during which the garbage collector stops the application and recovers space that is no longer in use. The intent of the maximum pause time goal is to limit the longest of these pauses. An average time for pauses and a variance on that average is maintained by the garbage collector. The average is taken from the start of the execution but is weighted so that more recent pauses count more heavily. If the average plus the variance of the pause times is greater than the maximum pause time goal, then the garbage collector considers that the goal is not being met.

The maximum pause time goal is specified with the command-line option -XX:MaxGCPauseMillis=<nnn>. This is interpreted as a hint to the garbage collector that pause times of <nnn> milliseconds or less are desired. The garbage collector will adjust the Java heap size and other parameters related to garbage collection in an attempt to keep garbage collection pauses shorter than <nnn> milliseconds. By default there is no maximum pause time goal. These adjustments may cause garbage collector to occur more frequently, reducing the overall throughput of the application. The garbage collector tries to meet any pause time goal before the throughput goal. In some cases, though, the desired pause time goal cannot be met.

Throughput Goal

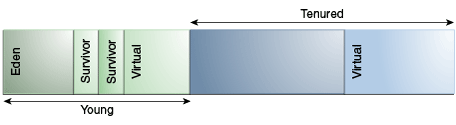
The throughput goal is measured in terms of the time spent collecting garbage and the time spent outside of garbage collection (referred to as application time). The goal is specified by the command-line option -XX:GCTimeRatio=<nnn>. The ratio of garbage collection time to application time is 1 / (1 + <nnn>). For example, -XX:GCTimeRatio=19 sets a goal of 1/20th or 5% of the total time for garbage collection.

The time spent in garbage collection is the total time for both the young generation and old generation collections combined. If the throughput goal is not being met, then the sizes of the generations are increased in an effort to increase the time that the application can run between collections.

Footprint Goal

If the throughput and maximum pause time goals have been met, then the garbage collector reduces the size of the heap until one of the goals (invariably the throughput goal) cannot be met. The goal that is not being met is then addressed.

Efficient collection is made possible by focusing on the fact that a majority of objects "die young." To optimize for this scenario, memory is managed in generations (memory pools holding objects of different ages). Garbage collection occurs in each generation when the generation fills up. The vast majority of objects are allocated in a pool dedicated to young objects (the young generation), and most objects die there. When the young generation fills up, it causes a minor collection in which only the young generation is collected; garbage in other generations is not reclaimed. Minor collections can be optimized, assuming that the weak generational hypothesis holds and most objects in the young generation are garbage and can be reclaimed. The costs of such collections are, to the first order, proportional to the number of live objects being collected; a young generation full of dead objects is collected very quickly. Typically, some fraction of the surviving objects from the young generation are moved to the tenured generation during each minor collection. Eventually, the tenured generation will fill up and must be collected, resulting in a major collection, in which the entire heap is collected. Major collections usually last much longer than minor collections because a significantly larger number of objects are involved.

  
[Description of "Figure 3-2 Default Arrangement of Generations, Except for Parallel Collector and G1"](http://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/img_text/jsgct_dt_001_armgnt_gn.html)

At initialization, a maximum address space is virtually reserved but not allocated to physical memory unless it is needed. The complete address space reserved for object memory can be divided into the young and tenured generations.

The young generation consists of eden and two survivor spaces. Most objects are initially allocated in eden. One survivor space is empty at any time, and serves as the destination of any live objects in eden; the other survivor space is the destination during the next copying collection. Objects are copied between survivor spaces in this way until they are old enough to be tenured (copied to the tenured generation).

By default, the young generation size is controlled by the parameter NewRatio. For example, setting -XX:NewRatio=3 (default value = 2) means that the ratio between the young and tenured generation is 1:3. In other words, the combined size of the eden and survivor spaces will be one-fourth of the total heap size.

**The following are general guidelines for server applications:**

First decide the maximum heap size you can afford to give the virtual machine. Then plot your performance metric against young generation sizes to find the best setting.

Note that the maximum heap size should always be smaller than the amount of memory installed on the machine to avoid excessive page faults and thrashing.

If the total heap size is fixed, then increasing the young generation size requires reducing the tenured generation size. Keep the tenured generation large enough to hold all the live data used by the application at any given time, plus some amount of slack space (10 to 20% or more).

Subject to the previously stated constraint on the tenured generation:

Grant plenty of memory to the young generation.

Increase the young generation size as you increase the number of processors, because allocation can be parallelized.

**Selecting a Collector**

Unless your application has rather strict pause time requirements, first run your application and allow the VM to select a collector. If necessary, adjust the heap size to improve performance. If the performance still does not meet your goals, then use the following guidelines as a starting point for selecting a collector.

If the application has a small data set (up to approximately 100 MB), then

Select the serial collector with the option -XX:+UseSerialGC.

If the application will be run on a single processor and there are no pause time requirements, then let the VM select the collector, or select the serial collector with the option -XX:+UseSerialGC.

If (a) peak application performance is the first priority and (b) there are no pause time requirements or pauses of 1 second or longer are acceptable, then let the VM select the collector, or select the parallel collector with -XX:+UseParallelGC.

If response time is more important than overall throughput and garbage collection pauses must be kept shorter than approximately 1 second, then select the concurrent collector with -XX:+UseConcMarkSweepGC or -XX:+UseG1GC.

**The Parallel Collector**

* The parallel collector (also referred to here as the throughput collector) is a generational collector similar to the serial collector; the primary difference is that multiple threads are used to speed up garbage collection. The parallel collector is enabled with the command-line option -XX:+UseParallelGC. By default, with this option, both minor and major collections are executed in parallel to further reduce garbage collection overhead.
* Enabling the parallel collector should make the collection pauses shorter. Because multiple garbage collector threads are participating in a minor collection, some fragmentation is possible due to promotions from the young generation to the tenured generation during the collection. Each garbage collection thread involved in a minor collection reserves a part of the tenured generation for promotions and the division of the available space into these "promotion buffers" can cause a fragmentation effect. Reducing the number of garbage collector threads and increasing the size of the tenured generation will reduce this fragmentation effect.
* The parallel collector (also known as the throughput collector) performs minor collections in parallel, which can significantly reduce garbage collection overhead. It is intended for applications with medium-sized to large-sized data sets that are run on multiprocessor or multithreaded hardware. The parallel collector is selected by default on certain hardware and operating system configurations, or can be explicitly enabled with the option -XX:+UseParallelGC. This option also enables parallel compaction for old generation. If required It can be disabled using –XX:-useParallelOldGC.
* Generation Size Adjustments: The statistics such as average pause time kept by the collector are updated at the end of each collection. The tests to determine if the goals have been met are then made and any needed adjustments to the size of a generation is made. The exception is that explicit garbage collections (for example, calls to System.gc()) are ignored in terms of keeping statistics and making adjustments to the sizes of generations. Growing and shrinking the size of a generation is done by increments that are a fixed percentage of the size of the generation so that a generation steps up or down toward its desired size. Growing and shrinking are done at different rates. By default a generation grows in increments of 20% and shrinks in increments of 5%. The percentage for growing is controlled by the command-line option -XX:YoungGenerationSizeIncrement=<Y> for the young generation and -XX:TenuredGenerationSizeIncrement=<T> for the tenured generation. The percentage by which a generation shrinks is adjusted by the command-line flag -XX:AdaptiveSizeDecrementScaleFactor=<D>. If the growth increment is X percent, then the decrement for shrinking is X/D percent.
* If the collector decides to grow a generation at startup, then there is a supplemental percentage is added to the increment. This supplement decays with the number of collections and has no long-term effect. The intent of the supplement is to increase startup performance. There is no supplement to the percentage for shrinking. If the maximum pause time goal is not being met, then the size of only one generation is shrunk at a time. If the pause times of both generations are above the goal, then the size of the generation with the larger pause time is shrunk first. If the throughput goal is not being met, the sizes of both generations are increased. Each is increased in proportion to its respective contribution to the total garbage collection time. For example, if the garbage collection time of the young generation is 25% of the total collection time and if a full increment of the young generation would be by 20%, then the young generation would be increased by 5%.

**Concurrent Mark Sweep (CMS) Collector**

* Results in shorter pause time although throughput takes a hit since it shares processor resources with the running application threads.
* Performs both minor and major collection. Both are done in same manner.
* The first step is mark where all application threads are stopped to identify all reachable object roots.
* Then it traces all the references from the roots without blocking the application.
* The next step is remark where any objects which could have been left untraced due to updates by application threads after the marking and tracing was finished. All application threads are paused again while executing this step. This pause tends to be little longer than the earlier one during mark phase.
* The sweep is carried out in parallel without blocvking the application execution.
* The CMS collector is enabled with the command-line option -XX:+UseConcMarkSweepGC.
* if the CMS collector is unable to finish reclaiming the unreachable objects before the tenured generation fills up, or if an allocation cannot be satisfied with the available free space blocks in the tenured generation, then the application is paused and the collection is completed with all the application threads stopped. The inability to complete a collection concurrently is referred to as **concurrent mode failure** and indicates the need to adjust the CMS collector parameters.
* Since application threads and the garbage collector thread run concurrently during a major collection, objects that are traced by the garbage collector thread may subsequently become unreachable by the time collection process ends. Such unreachable objects that have not yet been reclaimed are referred to as floating garbage.
* Based on recent history, the CMS collector maintains estimates of the time remaining before the tenured generation will be exhausted and of the time needed for a concurrent collection cycle. Using these dynamic estimates, a concurrent collection cycle is started with the aim of completing the collection cycle before the tenured generation is exhausted. These estimates are padded for safety, because concurrent mode failure can be very costly.
* A concurrent collection also starts if the occupancy of the tenured generation exceeds an initiating occupancy (a percentage of the tenured generation). The default value for this initiating occupancy threshold is approximately 92%, but the value is subject to change from release to release. This value can be manually adjusted using the command-line option -XX:CMSInitiatingOccupancyFraction=<N>, where <N> is an integral percentage (0 to 100) of the tenured generation size.

**Garbage-First Collector**

* The Garbage-First (G1) garbage collector is fully supported in Oracle JDK 7 update 4 and later releases. The G1 collector is a server-style garbage collector, targeted for multi-processor machines with large memories. It meets garbage collection (GC) pause time goals with high probability, while achieving high throughput.
* The G1 GC is a regionalized and generational garbage collector, which means that the Java object heap (heap) is divided into a number of equally sized regions. Upon startup, the Java Virtual Machine (JVM) sets the region size. The region sizes can vary from 1 MB to 32 MB depending on the heap size. The goal is to have no more than 2048 regions. The eden, survivor, and old generations are logical sets of these regions and are not contiguous.
* **A Young GC in G1:** Live objects are evacuated (i.e., copied or moved) to one or more survivor regions. If the aging threshold is met, some of the objects are promoted to old generation regions. This is a stop the world (STW) pause.
* **Old Generation Collection with G1:**

| **Phase** | **Description** |
| --- | --- |
| (1) Initial Mark *(Stop the World Event)* | This is a stop the world event. With G1, it is piggybacked on a normal young GC. Mark survivor regions (root regions) which may have references to objects in old generation. |
| (2) Root Region Scanning | Scan survivor regions for references into the old generation. This happens while the application continues to run. The phase must be completed before a young GC can occur. |
| (3) Concurrent Marking | Find live objects over the entire heap. This happens while the application is running. This phase can be interrupted by young generation garbage collections. A concurrent marking phase is started when the occupancy of the entire Java heap reaches the value of the parameter InitiatingHeapOccupancyPercent. The default value of InitiatingHeapOccupancyPercent is 45. To ensure that the semantics of snapshot-at-the beginning are met, G1 GC requires that all the concurrent updates to the object graph made by the application threads leave the previous reference known for marking purposes.  This is achieved by the use of the Pre-Write barriers (not to be confused with Post-Write barriers discussed later and memory barriers that relate to multithreaded programming). Their function is to, whenever you write to a field while G1 Concurrent Marking is active, store the previous referee in the so-called log buffers, to be processed by the concurrent marking threads. |
| (4) Remark *(Stop the World Event)* | Completes the marking of live object in the heap. For G1, it briefly stops the application threads to stop the inflow of the concurrent update logs and processes the little amount of them that is left over, and marks whatever still-unmarked objects that were live when the concurrent marking cycle was initiated. This phase also performs some additional cleaning, e.g. reference processing (see the Evacuation Pause log) or class unloading. |
| (5) Cleanup *(Stop the World Event and Concurrent)* | * Performs accounting on live objects and completely free regions. (Stop the world) * Scrubs the Remembered Sets. (Stop the world) * Reset the empty regions and return them to the free list. (Concurrent). |
| (\*) Copying/ Evacuation Pause Mixed *(Stop the World Event)* | These are the stop the world pauses to evacuate or copy live objects to new unused regions. This can be done with young generation regions which are logged as [GC pause (young)]. Or both young and old generation regions which are logged as [GC Pause (mixed)].  After the mark phase completes, G1 knows which regions are mostly empty. It collects in these regions first, which usually yields a large amount of free space. This is why this method of garbage collection is called Garbage-First. As the name suggests, G1 concentrates its collection and compaction activity on the areas of the heap that are likely to be full of reclaimable objects, that is, garbage. G1 uses a pause prediction model to meet a user-defined pause time target and selects the number of regions to collect based on the specified pause time target.  This evacuation is performed in parallel on multi-processors, to decrease pause times and increase throughput. Thus, with each garbage collection, G1 continuously works to reduce fragmentation, working within the user defined pause times. **This is beyond the capability of both the previous methods. CMS (Concurrent Mark Sweep) garbage collection does not do compaction. ParallelOld garbage collection performs only whole-heap compaction, which results in considerable pause times.** |

* G1 copies objects from one or more regions of the heap to a single region on the heap, and in the process both compacts and frees up memory. Applications running today with either the CMS or the ParallelOld garbage collector would benefit switching to G1 if the application has one or more of the following traits.
  + More than 50% of the Java heap is occupied with live data.
  + The rate of object allocation rate or promotion varies significantly.
  + Undesired long garbage collection or compaction pauses (longer than 0.5 to 1 second)
* G1 is planned as the long term replacement for the Concurrent Mark-Sweep Collector (CMS). Comparing G1 with CMS, there are differences that make G1 a better solution. One difference is that G1 is a compacting collector. G1 compacts sufficiently to completely avoid the use of fine-grained free lists for allocation, and instead relies on regions. This considerably simplifies parts of the collector, and mostly eliminates potential fragmentation issues. Also, G1 offers more predictable garbage collection pauses than the CMS collector, and allows users to specify desired pause targets.
* In G1, the failure (exhaustion of the Java heap) occurs while G1 is copying live data out of one region (evacuating) into another region. The copying is done to compact the live data. If a free (empty) region cannot be found during the evacuation of a region being garbage collected, then an allocation failure occurs (because there is no space to allocate the live objects from the region being evacuated) and a stop-the-world (STW) full collection is done.

**Multithreading**

**Defining, Instantiating, and Starting Threads**

A *thread of execution* is an individual process (a "lightweight" process) that has its own call stack. In Java,

there is *one thread per call stack*—or, to think of it in reverse, *one call stack per thread*.

The JVM, which gets its turn at the CPU by whatever scheduling mechanism the underlying OS uses, operates like a mini-OS and schedules *its* own threads regardless of the underlying operating system.

Threads can be created by extending Thread and overriding the public void run() method.

Thread objects can also be created by calling the Thread constructor that takes a Runnable argument. The Runnable object is said to be the *target* of the thread.

If you create a thread using the no-arg constructor, the thread will call its own run() method when it's time to start working. That's exactly what you want when you extend Thread, but when you use Runnable, you need to tell the new thread to use *your* run()method rather than its own. The Runnable you pass to the Thread constructor is called the *target* or the *target Runnable.*

You can call start() on a Thread object only once. If start() is called more than once on a Thread object, it will throw a RuntimeException.

It is legal to create many Thread objects using the same Runnable object as the target.

When a Thread object is created, it does not become a *thread of execution* until its start() method is invoked. When a Thread object exists but hasn't been started, it is in the *new* state and is not considered *alive*.

A thread is done being a thread when its target run() method completes.

**Transitioning Between Thread States**

Once a new thread is started, it will always enter the runnable state. The thread scheduler can move a thread back and forth between the runnable state and the running state.

For a typical single-processor machine, only one thread can be running at a time, although many threads may be in the runnable state.

There is no guarantee that the order in which threads were started determines the order in which they'll run.

There's no guarantee that threads will take turns in any fair way. It's up to the thread scheduler, as determined by the particular virtual machine implementation. If you want a guarantee that your threads will take turns regardless of the underlying JVM, you can use the sleep() method. This prevents one thread from hogging the running process while another thread starves. (In most cases, though, yield() works well enough to encourage your threads to play together nicely.). sleep() and yield() methods are static methods defined in such a way that they affect the current thread instance only.

A running thread may enter a blocked/waiting state by a wait(), sleep(), or join() call.

A running thread may enter a blocked/waiting state because it can't acquire the lock for a synchronized block of code.

When the sleep or wait is over, or an object's lock becomes available, the thread can only reenter the runnable state. It will not *go* directly from waiting to running (well, for all practical purposes anyway).

A dead thread cannot be started again.

|  |
| --- |
|  |

**Sleep, Yield, Join and Interrupt**

Sleeping is used to delay execution for a period of time, and no locks are released when a thread goes to sleep.

A sleeping thread is guaranteed to sleep for at least the time specified in the argument to the sleep() method (unless it's interrupted), but there is no guarantee as to when the newly awakened thread will actually return to running.

The sleep() method is a static method that sleeps the currently executing thread's state. One thread *cannot* tell another thread to sleep.

The setPriority() method is used on Thread objects to give threads a priority of between 1 (low) and 10 (high), although priorities are not guaranteed, and not all JVMs recognize 10 distinct priority levels—some levels may be treated as effectively equal. If not explicitly set, a thread's priority will have the same priority as the priority of the thread that created it. The yield() method *may* cause a running thread to back out if there are runnable threads of the same priority. There is no guarantee that this will happen, and there is no guarantee that when the thread backs out there will be a *different* thread selected to run. A thread might yield and then immediately reenter the running state.

The closest thing to a guarantee is that at any given time, when a thread is running it will usually not have a lower priority than any thread in the runnable state. If a low-priority thread is running when a high-priority thread enters runnable, the JVM will usually preempt the running low-priority thread and put the high-priority thread in.

When one thread calls the join() method of another thread, the currently running thread will wait until the thread it joins with has completed. Think of the join() method as saying, "Hey thread, I want to join on to the end of you. Let me know when you're done, so I can enter the runnable state."

Besides sleep, yield, join, we also have the following scenarios in which a thread might leave the running state:

The thread's run() method completes.

A call to wait() on an object (we don't call wait() on a *thread*, as we'll see in a moment).

A thread can't acquire the *lock* on the object whose method code it's attempting to run.

The thread scheduler can decide to move the current thread from running to runnable in order to give another thread a chance to run. No reason is needed—the thread scheduler can trade threads in and out whenever it likes.

On using an interrupt call on a thread its gets interrupted. The post interruption behavior can be defined by handling the interrupted exception, or by checking on the interrupted flag for the thread instance.

**Concurrent Access Problems and Synchronized Threads**

synchronized methods prevent more than one thread from accessing an object's critical method code simultaneously.

You can use the synchronized keyword as a method modifier, or to start a synchronized block of code. Only methods (or blocks) can be synchronized, not variables or classes. To synchronize a block of code (in other words, a scope smaller than the whole method), you must specify an argument that is the object whose lock you want to synchronize on.

While only one thread can be accessing synchronized code of a particular instance, multiple threads can still access the same object's *un*synchronized code.

When a thread goes to sleep, its locks will be unavailable to other threads.

static methods can be synchronized, using the lock from the java.lang.Class instance representing that class.

A thread can acquire more than one lock. For example, a thread can enter a synchronized method, thus acquiring a lock, and then immediately invoke a synchronized method on a different object, thus acquiring that lock as well. As the stack unwinds, locks are released again. Also, if a thread acquires a lock and then attempts to call a synchronized method on that same object, it’s allowed to do so. The JVM knows that this thread already has the lock for this object, so the thread is free to call other synchronized methods on the same object, using the lock the thread already has. This is termed as reentrant synchronization.

Reads and writes are atomic for reference variables and for most primitive variables (all types except long and double).

Reads and writes are atomic for all variables declared volatile (including long and double variables).

**Communicating with Objects by Waiting and Notifying**

All three methods—wait(), notify(), and notifyAll()—must be called from within a synchronized context! A thread invokes wait() or notify() on a particular object, and the thread must currently hold the lock on that object.

The wait() method lets a thread say, "there's nothing for me to do now, so put me in your waiting pool and notify me when something happens that I care about." Basically, a wait() call means "wait me in your pool," or "add me to your waiting list." When the thread waits, it temporarily releases the lock for other threads to use, but it will need it again to continue execution.

The notify() method is used to send a signal to one and only one of the threads that are waiting in that same object's waiting pool.

The notify() method can NOT specify which waiting thread to notify.

The method notifyAll() works in the same way as notify(), only it sends the signal to *all* of the threads waiting on the object.

**Deadlocked Threads**

Deadlocking is when thread execution grinds to a halt because the code is waiting for locks to be removed from objects.

Deadlocking can occur when a locked object attempts to access another locked object that is trying to access the first locked object. In other words, both threads are waiting for each other's locks to be released; therefore, the locks will *never* be released!

Starvation occurs when thread is not able to gain access of resources like CPU etc. due to presence of high priority/ greedy threads.

Livelock occurs when threads are busy giving responses to each other/ or context switching.

**High level concurrency objects:**

[Lock objects](file:///D:\Documents\Resources%20(Books%20and%20Links)\J2EE%20&%20DB%20Resources\Core%20Java\tutorial\essential\concurrency\newlocks.html) support locking idioms that simplify many concurrent applications.

[Executors](file:///D:\Documents\Resources%20(Books%20and%20Links)\J2EE%20&%20DB%20Resources\Core%20Java\tutorial\essential\concurrency\executors.html) define a high-level API for launching and managing threads. Executor implementations provided by java.util.concurrent provide thread pool management suitable for large-scale applications. There are three different interfaces Executor, ExecutorService and ScheduledExecutorService.

[Concurrent collections](file:///D:\Documents\Resources%20(Books%20and%20Links)\J2EE%20&%20DB%20Resources\Core%20Java\tutorial\essential\concurrency\collections.html) make it easier to manage large collections of data, and can greatly reduce the need for synchronization.

[Atomic variables](file:///D:\Documents\Resources%20(Books%20and%20Links)\J2EE%20&%20DB%20Resources\Core%20Java\tutorial\essential\concurrency\atomicvars.html) have features that minimize synchronization and help avoid memory consistency errors.

ThreadPools consist of worker threads. These are used to execute multiple threads.

|  |
| --- |
| Collection  List  Queue  Set  SortedSet  PriorityQueue  DataInputStream  dwdwefwfhth  LinkedList  Vector  ArrayList  TreeSet  (Reading from a file)  LinkedHashSet  (Reading froma buffer containing the bytes)  HashSet  Object  Map  SortedMap  Collections  Array  HashMap  HashTable  LinkedHashMap  TreeMap |

**Also read the Collections and Generics material from the SUN tutorial.**