**Java program**

|  |
| --- |
| A Java program is a collection of *objects* talking to other objects by invoking each other's *methods*. |

**Source File Declaration Rules**

|  |
| --- |
| * One public class per source file, the name of file = name of the public class. **Ex:** public class Dog{ } must be in Dog.java. * A file can have more than one nonpublic class. Files with no public classes can have any filename. * If the class is part of a package, the package statement must be the first line in the source code followed by import and class declaration. |

**Reference** this

|  |
| --- |
| this refers to the currently executing object. It is implicit, no need to preface member access code with it. |
| **Shadowing** [name collision] is declaration of a local variable with the same name as an instance variable. this resolves the name collision. |

**Naming Standards - Legal Identifiers**

|  |
| --- |
| * Identifiers must start with a letter, a currency character ($), an underscore ( \_ ). Cannot start with a number! * Cannot use a Java keyword as an identifier. * Identifiers in Java are case-sensitive. [foo and FOO are two different identifiers] |

**Access Modifiers**

|  |
| --- |
| *Every* class, method, and instance variable you declare has an access *control* [visibility], whether you type one or not.  **Public Members:** all classes, regardless of the package they belong to, can access the member.  **Private Members** are invisible to any code outside the member's own class.  **Defaultmember** [no modifiers are used] may be accessed only if the class accessing the member belongs to the same package.  **Protected member** can be accessed (through inheritance) by a subclass ***even if the subclass is in a different package.*** |

**Variable Declarations**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Primitives**: char, Boolean, byte, short, int, long [suffix L], double, float [suffix F]. If declared, primitive type cannot change, value can change.  **Primitive Ranges** [# types]: byte, short, int, long, float, double. All number types in Java are made up of a certain number of 8-bit bytes, and are *signed* (-/+). The leftmost bit is used to represent the sign [1 (-), 0 (+)]. The rest of the bits represent the value.   |  |  |  |  |  | | --- | --- | --- | --- | --- | | **Type** | **Bit** | **Byte** | **Minimum Range [**-2(bits-1)**]** | **Maximum Range [**2(bits-1)–1**]** | | **byte [Default: 0]** | 8 | 1 | -27 | 27-1 | | **short [Default: 0]** | 16 | 2 | -215 | 215-1 | | **int [Default: 0]** | 32 | 4 | -231 | 231-1 | | **long [Default: 0]** | 64 | 8 | -263 | 263-1 | | **float [Default: 0.0]** | 32 | 4 | n/a | n/a | | **Double [Default:0.0]** | 64 | 8 | n/a | n/a | | A boolean [Default: false] can be only true or false, the range is virtual-machine dependent. | | | | | | The char [16-bit Unicode character], an unsigned integer type. Default: '\u0000'  char has a larger range since short uses 1 bit to represent the sign, meaning fewer positive numbers are acceptable in a short. | | | | |   **.** |
| **Reference variables** refer to an object. A type can never be changed. Refers to an object of the declared type, or of a *subtype*.   * Can be of only one type, and once declared, that type can never be changed (although the object it references can change). * It is a variable, so it can be reassigned to other objects, (unless the reference is declared final). * Type determines the methods that can be invoked on the object the variable is referencing. * Can refer to any object of the same type as the declared reference, or **it can refer to any** *subtype* **of the declared type!** * A reference variable can be declared as a class or an interface type. It can reference any object of any class that *implements* the interface. |
| **Instance variable:** [field, property, attribute] are defined inside the class, outside of method, initialized when the class is instantiated. The fields belong to each unique object. Can only be final, transient, four access levels. |
| **Local (Automatic/Stack/Method) Variables** are variables declared within a method. It starts life inside the method, destroyed when the method has completed, lives only within the scope of the method. *The local variable is on the stack, if the variable is an object reference, the object itself will still be created on the heap.* Before a local variable can be *used*, it must be *initialized* with a value. Unlike instance variables—local variables don't get default values. But the value of local variable can be passed out of the method to take on a new life. |
| **Final Variables:** once final **primitive** **variable** is assigned a value, it can't be altered. A final **reference variable** allows you to modify the state of the object it refers to, but can't ever be reassigned to refer to a different object. There are no final objects, only final references. |
| **Transient Variables**: the JVM skips (ignore) transient variable when you attempt to serialize the object containing it. Serialization lets you save ("flatten") an object by writing its state to a special type of I/O stream. With serialization you can save an object to a file, or even ship it over a wire for reinflating (deserializing) at the other end, in another JVM. Serialization has been added to the exam as of Java 5. |
| **Volatile Variables** tell the JVM that a thread accessing the variable must always reconcile its own private copy of the variable with the master copy in memory. Volatile, transient can be applied only to instance variables. |
| **Static Variables and Methods** exist independently of any instances created for the class.   * All static members exist before you ever make a new instance of a class. * There will be only one copy of a static member regardless of the number of instances of that class. * No need to initialize a static variable since they get the same default values instance variables get. * A static method can't access a non-static (instance) variable, because there is no instance! * To access a static method (or variable) use the dot operator on the class name. An object reference variable can access static member. * Static methods can't be overridden but can be redefined! |
| **Variable Scope**:   * **Static variables** are created when the class is loaded, and they survive as long as the class stays loaded in JVM. * **Instance variables** are created when a new instance is created, and they live until the instance is removed. * **Local variables** live as long as their method remains on the stack. * **Block variables** live only as long as the code block is executing. |

**Assignment**

|  |
| --- |
| Instance variables and objects live on the **heap**. Local variables live on the **stack**. |
| There are three ways to represent integer numbers in the Java language: decimal (base 10), octal (base 8), and hexadecimal (base 16). |
| **Octal** literals use only the digits 0 to 7. Represent an integer in octal form by placing a zero in front of the number. [up to 21 digits] |
| **Hexadecimal** literalsuse 16 distinct symbols, not including the prefix 0x [0-9, A-F]. [up to 16 digits – case insensitive] |
| **Variables** are bit holders, with a designated type. byte val = 6 [bit pattern (the byte holder) is 00000110, representing the 8 bits]. |
| **Assigning Primitive Variable to Another Primitive Variable:** int a = 6; int b = a; We used the variable a *only* to copy its contents to b but if we change the contents of a or b, the other variable won't be affected. a and b do not share a single value; they have identical copies. |
| **Reference variable** bit holder contains bits representing a *way to get to the object*. (it points to something, we don't know what that really is). The variable's value is *not* the object, but a value representing a specific object on the heap [Or null if the reference variable is not assigned]. |
| Assign a newly created object: Button b = new Button(); Makes a reference variable named b, of type Button. Creates a new Button object on the heap. Assigns the newly created Button object to the reference variable b. The space for the Button reference variable is created but not an actual Button object. |
| **Assigning reference Variable to Another Reference Variable:** An assignment of reference variable to another means the contents (bit pattern) of one variable are *copied* into another. If we assign an existing instance of an object to a new reference variable, then two reference variables will hold the same bit pattern—a bit pattern referring to a specific object on the heap.  Dimension a = new Dimension(5,10); Dimension b = a; but if we change the contents of a or b, the other variable will be affected.  **Exception:** String objects are immutable; you can't change the value of a String object. x changes but y stays the same. |
| **Pass-by-value**: the called method can't change the caller's variable because the copy of the variable is passed. The reference variable can change the state of the object it refers to, but it can't refer to a different object! Java is pass-by-value for all variables running within a single VM. |
| **Passing Object Reference Variables:** When an object variable is passed into a method, the copy of object *reference is passed* [not the actual object itself] since a reference variable holds bits that represent (to the underlying VM) a way to get to a specific object in memory (on the heap).  A copy of a variable means you get a copy of the bits in that variable, so when you pass a reference variable, you're passing a copy of the bits representing how to get to a specific object. In other words, both the caller and the called method will now have identical copies of the reference, and thus both will refer to the same exact (*not* a copy) object on the heap. **Method modifies the original object**, any changes to the object that occur inside the method are being made to the object whose reference was passed. **Except for a string object.** |
| **Result:** Java passes objects by passing the reference variable instead but it is not pass-by-reference for objects. **Pass-by-value** means pass-by-variable-value [pass-by-copy-ofthe-variable!]. |
| **For a primitive variable**, a copy of the bits representing the value is passed to a method. **Ex:** if you pass an int variable with the value of 3, you're passing a copy of the bits representing 3. The called method then gets its own copy of the value. A method does not change the variable after it was passed to the method. It is passed by value, which means pass-by-copy-of-the-bits-in-the-variable. |
| **For an object reference variable**, a copy of the bits representing the reference to an object is passed. The called method then gets its own copy of the reference variable. But because two identical reference variables refer to the exact same object, if the called method modifies the object (by invoking setter methods), the caller will see that the object the caller's original variable refers to has also been changed. |

**Behavior**

|  |
| --- |
| **Behavior** (**methods**) are where the class' logic is stored and data gets manipulated. **Naming standards:** the first letter should be lowercase, and then normal camelCase rules should be used. In addition, the names should typically be verb-noun pairs. **Ex**: getBalance. |
| **Final Methods** prevents a method from being overridden in a subclass for safety and security. public final void showSample(){}   * **Arguments:** values between the parentheses when you're *invoking* a method: doStuff("a", 2); * **Parameters:** values in *method's signature* that method receives: void doStuff(String s, int a) { } * **Final Arguments** appear in a method declaration. It must keep the same value when it was passed into the method. |
| **Returning a Value**:   * null with an object reference return type: return null * An array: return new String[] {"Fred", "Barney", "Wilma"} or int[] nums = {1,2,3}; return nums; * A primitive return type: char c = 'c'; return c; * A primitive return type that can be explicitly cast to the declared return type. float f = 32.5f; return (int) f; * Cannotreturn anything from a method with a void return type. * An object reference return type, return any object type that can be implicitly cast to the declared return type. return new Horse(); * Any object that passes the IS-A test (test true the instanceof operator) can be returned from that method. |
| **Abstract Methods** is a method that's been *declared* as abstract but not *implemented*. No curly braces, no method body but a semicolon. It is illegal to have even a single abstract method in a class that is not explicitly declared abstract. public abstract void showSample();  *The first concrete [non-abstract] subclass of an abstract class must implement all abstract methods of the superclass.*  A method can never be marked as both abstract and final, abstract and private, or static. |
| **Synchronized Methods** can be accessed by only one thread at a time. public synchronized Record retrieveUserInfo(int id){} |
| **Native Method** modifier indicates that a method is implemented in platform-dependent code, often in C. |
| **Strictfp Method** forces floating points to adhere to the IEEE 754 standard. You can predict how your floating points will behave regardless of the underlying platform the JVM is running on. It can only modify a class or method declaration, a variable can never be strictfp. |
| **Methods with Variable Argument Lists (var-args):** methods that can take a variable number of arguments.   * **Var-arg type:** specifies the type of the argument(s) this parameter of your method can receive. * **Basic syntax:** void doStuff3(Animal... animal), void doStuff2(char c, int... x) { } * **Other parameters** It's legal to have other parameters in a method that uses a var-arg, must the last parameter in the method. |
| **Constructor:** objects are constructed. When an object is made, at least one constructor is invoked. Every class has a constructor, although if you don't create one explicitly, the compiler will build one for you. Constructor is a method without a return type.   * Constructors are invoked at runtime. * It's desirable for a class to have a no-arg constructor. Overloaded constructors offer flexible ways to instantiate objects from your class. * Constructors must not have a return type. Constructors are used to initialize instance variable state. * Constructor declarations have all of the normal access modifiers, and they can take arguments (including var-args). * Constructors must have the same name as the class in which they are declared. * If you don't type a constructor into your class code, a default constructor [no-arg constructor] is be automatically generated by the compiler.   If you you've typed any other constructor(s) into your class code, the compiler won't provide the no-arg constructor   * Constructors can't be marked static (they are after all associated with object instantiation), final or abstract. * this() always means a call to another constructor in the same class. **The first line in a constructor: a call to super() or a call to this().** * Every constructor invokes the constructor of its superclass with an (implicit) call to super(), unless the constructor.   If you do not type a constructor or call to super() or this(), the compiler will insert a no-arg call to super(), the first statement in constructor.   * Abstract classes have constructors, and those constructors are always called when a concrete subclass is instantiated. * Interfaces do not have constructors. Interfaces are not part of an object's inheritance tree. * Constructors are never inherited, they are not methods, and cannot be overridden. |
| **Initialization blocks** [usually appear at the top of the class], methods and constructors are the places where operations can be performed.   * Init blocks execute in the order they appear. * Static init blocks run once, when the class is first loaded. * Instance init blocks run every time a class instance is created. * Instance init blocks run after the constructor's call to super(). |

**Package**

|  |
| --- |
| **Naming Standards:** use reverse domain names, appended with division and/or project names. **Ex**: if your domain name is geeksanonymous.com, and you're working on the client code [class Utilities].  Name your package com.geeksanonymous.steps.client.  That changes the name of your class to com.geeksanonymous.steps.client.Utilities.  There might be a name collision within your company but no collisions with classes developed outside your company. |

**Class**

|  |
| --- |
| The **benefits of Object Orientation** (OO): flexibility, maintainability, extensibility. |
| **Class** is atemplate that describes the state, behavior of the object. Put classes into packages. That object will have its own state and behaviors defined by its class. The names should be nouns. **Ex**: Dog. Every class is a subclass of class Object [gets equals, toString methods] |
| **Coupling** [how classes interact with each other] is the degree to which one class knows about another class.  **Cohesion** [how a single class is designed]: every class should have a focused set of responsibilities. |
| **Object** is an instance of the class created at runtime when JVM encounters the *new* keyword. **State** (**instance variables**) - each object have its own unique set of instance variables as defined in the class. The values assigned to an object's instance variables make up the object's state. |
| **Default Access** [*package*-level access]: A class can be seen only by classes within the same package. **Ex:** if class A has default access and class B are in different packages, class B won't be able to create an instance of class A, or even declare a variable or return type of class A. |
| **Public Access**:*all* classes can access to a public class. If a public class in a different package, you need to import the class. |
| **Non-access Class Modifiers:** a class can be modified using the keyword final, abstract, or strictfp. A class can be public final. |
| strictfpconforms a class to the IEEE 754 standard rules for floating points. Without that modifier, floating points used in the methods might behave in a platform-dependent way. If you don't declare a class as strictfp, you can declare a method as strictfp. |
| **Final Classes:** no class can extend a final class. It is absolute guarantee that none of the methods in that class will ever be overridden. **Ex:** the String final class cannot be subclassed. You'll almost never make a final class, since it obliterates a key benefit of OO - extensibility. |
| **Abstract Classes** can never be instantiated. Its sole purpose is to be extended. You can't make objects out of it because the class is too *abstract*. A class cannot be both abstract and final. **Ex:** a class Car that has generic methods common to all vehicles. But you don't want anyone creating a generic, abstract Car object. You need to instantiate actual car types such as BMWBoxster and SubaruOutback.   * The abstract methods end in a semicolon [no curly braces]. * If one method is abstract, the whole class must be declared abstract then both the method and the class must be marked abstract. * You can put non-abstract methods in an abstract class. All concrete subclasses (not abstract) inherit method implementations. |

**Enum**

|  |
| --- |
| As of Java 5, you can restrict a variable to having one of only a few pre-defined values, one value [**enum**] from an enumerated list. **Enums** are items in the enumerated list. Enums shouldn’t be caps. Enums can be declared as their own class, or class member, they must not be declared within a method! The semicolon is optional after declaration. Enum values are objects that can each have their own instance variables. |
| Every enum has a static method, *values(),* that returns an array of the enum's values in the order they're declared.   * You cannot invoke an enum constructor directly. * You can define more than one argument to the constructor, you can overload the enum constructors, just as any normal class constructor. |

**Encapsulation**

|  |
| --- |
| **Encapsulation:** hiding implementation details behind a public programming interface. Without encapsulation protection, the variable can be set to a wrong number but an accessor method, setProperty(int wt), could check for an inappropriate number. |
| * Keep instance variables protected (with an access modifier, often private). * Make public accessor methods [set/get<someProperty>]. Use methods rather than directly accessing the instance variable. |
| * **Setter method** [accessor, changes a value]**:** public, void return type and an argument that represents the property type. * **Getter method** [mutator, retrieves a value]**:** public, no arguments, a return type matches the type of the setter method for that property. |

**Inheritance**

|  |
| --- |
| **Inheritance**: a superclass is extended with more specific subclasses. The superclass knows nothing of the classes that inherit from it. A subclass is given accessible instance variables and methods defined by the superclass. Subclass methods can override superclass methods.  **Reuse**: create a generic version of a class with the intention of creating more specialized subclasses that inherit from it. All specialized subclasses are guaranteed to have the capabilities of the more generic superclass. **No multiple inheritance**: *A class can have one superclass.* |
| * If a subclass *inherits* a member, the subclass *has* the member. * If a member of its superclass is public, the subclass inherits that member regardless of whether both classes are in the same package. * If a member of its superclass is declared private, a subclass can't inherit it. * A private method cannot be overridden by a subclass. |

**Polymorphism**

|  |
| --- |
| **Polymorphism** ("many forms"): treat any *subclass* of superclass as a superclass. The methods you can call on a reference are dependent on the *declared* type of the variable. *All* Java objects are polymorphic since they pass the IS-A test for their own type and for class Object. |
| **Polymorphic method invocations apply only to *instance methods.*** You can refer to an object with a more general reference variable type (a superclass or interface), but at runtime, things are dynamically selected based on the actual *object* (not *reference* type) are instance methods. |

**Is-A vs. Has-A Relationship**

|  |
| --- |
| **IS-A** is based on class inheritance or interface implementation. [ "A is a type of B."] The instanceof method returns true if the reference variable being tested is of the type being compared to. If the expression (Foo instanceof Bar) is true, then class Foo IS-A Bar. |
| **HAS-A is** based on usage. Alass A HAS-A B if code in class A has a reference to an instance of class B. A Horse HAS-A Halter. |

**Interface**

|  |
| --- |
| **Interface** is a companion to inheritance. Interface is a contract for *what* a class can do, without saying *how* the class will do it, 100% abstract superclass. The names are adjectives like Runnable, Serializable. By defining an interface for Bounceable, any class that wants to be treated as a Bounceable thing can simply implement the Bounceable interface and provide code for the interface's two methods. |
| * Like an abstract class, an interface defines abstract methods that end with semicolon rather than curly braces. * An abstract class has both abstract and non-abstract methods, an interface can have only abstract methods. * All interface methods are implicitly public and abstract, you do not need to actually type the public or abstract. * Only constants-public, static, and final—, not instance variables. public static final int x = 1; * No static interface methods. Interface methods are abstract, they cannot be marked final, strictfp, or native. * An interface can *extend* one or more other interfaces. **Ex:** interface Bounceable extends Moveable, Spherical * It cannot implement another interface or class. public class Ball implements Bounceable, Serializable, Runnable{} |
| Unless the implementing class is abstract, the implementing class must provide implementations for all methods defined in the interface. Implementation classes must adhere to the same rules for method implementation as a class extending an abstract class:   * Provide concrete (nonabstract) implementations for all methods from the declared interface. * Declare no checked exceptions on implementation methods other than those declared by the interface method. * Follow all the rules for legal overrides, maintain the signature of the interface method, and maintain the same return type (or a subtype). * An implementation class can be abstract, implementation methods does not have to be provided (implement in the first concrete class) |

**Overloading vs. Overriding**

|  |
| --- |
| **Over*loaded* methods** let you reuse the same method name in a class, but with different arguments (optionally, a different return type).   * Overloaded methods MUST change the argument list. * Overloaded methods CAN change the return type. * Overloaded methods CAN change the access modifier. * Overloaded methods CAN declare new or broader checked exceptions. * A method can be overloaded in the *same* class or in a *subclass*. * When an exact argument match isn't found, the JVM uses the method with the smallest argument that is wider than the parameter. * Widening beats boxing, widening beats var-args, boxing beats var-args. * *Reference* type determines which overloaded version (based on arguments) is selected at *compile* time.   The actual *method* invoked is a virtual method invocation that happens at runtime. |
| **Overriding:** class that inherits a method from a superclass can override the method (except final method). It defines behavior specific to a particular subclass type. You *must* implement the abstract method in the subclass ***unless the subclass is also abstract.*** |
| * The argument must not change. * 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 overriding method cannot have a more restrictive access modifier than the method being overridden. * Instance methods can be overridden only if they are inherited by the subclass. (unless it’s private, final or static) * The overriding method must NOT throw checked exceptions that are new or broader than those declared by the overridden method. * *Object* type (*the type of the actual instance on the heap*) determines which method is selected at *runtime*. |
| **Invoking a Superclass Version of an Overridden Method:** super.method(args){} It only applies to instance methods. |

**Arrays**

|  |
| --- |
| **Arrays** are objects that store multiple variables of the same type, or variables that are all subclasses of the same type.   * Array is the instance of Object. * An array instance variable that's declared but not explicitly initialized will have a value of null. * The array is always an object on the heap, even if the array is declared to hold primitive elements. * *Array elements are always given default values, regardless of where the array itself is declared or instantiated*. * Any object that passes the "IS-A" test for the declared array type can be assigned to an element of that array. * The reference variable declared can be reassigned to any int array, but cannot be reassigned to anything that is not an int array |

**Casting**

|  |
| --- |
| **Primitive/Reference Variable Casting** lets you convert values from one type to another. There are two types: implicit and explicit casting:  An **implicit cast**: no need write code for the cast; the conversion happens automatically, a widening conversion [byte to int].  An **explicit cast**: must write code for the cast; a narrowing conversion [int to byte], possible loss of precision. |

**Wrapper classes**

|  |
| --- |
| * Wrapper class exists for every primitive in Java. The name of the class is the same as type of primitive variable except for char [Character] * Primitive values could be treated as objects. It is used for adding to Collections, returning from a method with an object return value.   Note: Java 5 Started handling autoboxing (and unboxing) automatically.   * It provides functions for primitives. **Ex:** functions for converting primitives to and from String objects. * *Wrapper objects are immutable*. Once they have been given a value, that value cannot be changed. * Wrapper reference variables can be null. * *None of the wrapper classes will widen from one to another!* |
| The static **valueOf() method** takes 2 arguments [a String representation of the appropriate type of primitive and an additional argument, int radix, which indicates in what base (for example binary, octal, or hexadecimal) the first argument is represented.  Integer i2 = Integer.valueOf("101011", 2); // converts 101011 to 43, assigns the value 43 to the Integer object i2  Float f2 = Float.valueOf("3.14f"); // assigns 3.14 to the Float object f2 |
| **xxxValue()** returns a newly created wrapped object of the type that invoked the method. Any numeric wrapper can be converted to any primitive numeric type. Integer i2 = new Integer(42); byte b = i2.byteValue(); |
| parseXxx() returns the named primitive. double d4=Double.parseDouble("3.14");long L2=Long.parseLong("101010", 2); |
| Class Object has a toString() method. Since all other Java classes inherit from class Object, all other Java classes have a toString() method. The toString() method returns a String with the value of the primitive wrapped in the object.  Double d = new Double("3.14"); System.out.println("d = "+ d.toString() ); // result is d = 3.14 |
| Wrapper class is immutable, second object is created and its value is set to 568. The equals() method determines whether two instances of a given class are "meaningfully equivalent" but it's up to the creator of the class to determine what "equivalent" means for objects of the class in question. For all the wrapper classes, two objects are equal if they are of the same type and have the same value. |

**Garbage Collection**

|  |
| --- |
| **Garbage Collection** is automatic memory management in Java.   * The heap is where Java objects live, and it's the only part of memory that is in any way involved in the garbage collection process. * Garbage collection makes sure that the heap has a free space by deleting any objects that are not reachable by the Java program running. * A Java program is a cycle of creating the objects it needs on the heap, and then discarding them using the garbage collector. * The JVM decides when to run the garbage collector. You can ask the JVM to run the garbage collector, but the JVM might not comply. * *An object is eligible for garbage collection when no live thread can access it*, GC might/might not delete the object. * Garbage collection cannot ensure that there is enough memory, only that the memory that is available will be managed very efficiently. * If memory is low, the garbage collector will run before it throws an OutOfMemoryException. |
| **Make Objects Eligible for Collection:** An object becomes eligible for GC when there are no more reachable references to it. |
| **Set the reference variable that refers to the object to null**.  StringBuffer sb = new StringBuffer("hello");  sb = null; // the StringBuffer object is eligible for collection |
| **Reassigning a Reference Variable**  StringBuffer s1 = new StringBuffer("hello");  StringBuffer s2 = new StringBuffer("goodbye");  s1 = s2; // s1 refers to the "goodbye" object, the StringBuffer "hello" is eligible for collection |
| **Isolating a Reference:**  StringBuffer s1 = new StringBuffer("hello");  StringBuffer temp = s1;  s1 = null; // the StringBuffer object is eligible for collection |
| **Forcing Garbage Collection**: garbage collection cannot be forced. It is possible to suggest to the JVM that it perform garbage collection. However, there are no guarantees the JVM will actually remove all of the unused objects from memory. Ask for GC: **System.gc();** |
| The **Runtime class** is a special class that has a single object (a Singleton) for each main program. The Runtime object provides a mechanism for communicating directly with the virtual machine. To get the Runtime instance, use the method Runtime.getRuntime(), which returns the Singleton. Once you have the Singleton, invoke the GC using the gc() method. [Runtime rt = Runtime.getRuntime(); rt.gc()] |
| **The finalize() Method** is inherited by all classes from class Object. It is a mechanism to run some code just before your object is deleted by the GC. The finalize() method for any given object might run, but you can't count on it. It is recommended not to override finalize() at all.   * For any given object, finalize() will be called only once (at most) by the garbage collector. * Calling finalize() can actually result in saving an object from deletion. * Any code can go into finalize(), you could even write code that passes a reference to the object in question back to another object, effectively *uneligiblizing* the object for GC. |

**Thread**

|  |
| --- |
| * Every Java program has from one to many threads. * Each thread has its own little execution stack; each thread has its own lifecycle [threads can be alive or dead]. * The programmer cause at least one thread to run in a Java program, the one with the main() method at the bottom of the stack. |

**CERTIFICATION OBJECTIVE**

**Operators**

|  |
| --- |
| Java operators aren't typically overloaded. There are a few exceptional operators that come overloaded:   * The + operator can be used to add two numeric primitives together, or to perform a concatenation operation if either operand is a String. * The &, |, and ^ operators can all be used to compare values, or to perform the bit-twiddling capabilities. * Java uses the Unicode value of the character as the numerical value, for comparison. * You can compare a character primitive with any number (though it isn't great programming style). |
| **Conditional Operator** is a *ternary* operator. The operator: x = (boolean expression) ? value to assign if true : value to assign if false |
| *The short-circuit && operator is so named because it doesn't waste its time on pointless evaluations*. && evaluates the left side of the operation first (operand one), and if it resolves to false, the && operator doesn't look at the right side of the expression (operand two) since the && operator already *knows* that the complete expression can't possibly be true. |
| || evaluates to true if EITHER of the operands is true. If the first operand in an OR operation is true, the result will be true, || doesn't waste time looking at the right side of the equation. If the first operand is false, however, the short-circuit || has to evaluate the second operand to see if the result of the OR operation will be true or false. |
| ***Non-short-circuit*** logical operators [(&), (|)] evaluate both sides of the expression, always! They're inefficient. |
| The **(^) an exclusive-OR operator** is related to the non-short-circuit operators, it always evaluates *both* the left and right operands in an expression. For an exclusive-OR (^) expression to be true, EXACTLY one operand must be true. |
| **"Equality" Operators:** (==) compares the value in the variable—the bit pattern. The equals tests if objects are "meaningfully equivalent” |
| **Primitive variables:** (==) tests variables for ”meaningful equality”. |
| **Reference variables:** compares addresses, the test shows if references refer to the same object. |
| **Enum equality:** use either the == operator or the equals() method to determine if two variables are referring to the same enum constant. |
| The instanceof is used for object reference variables only, it checks whether an object is of a particular type [class or interface].  Test an object reference against its own class type, or its superclasses. A*ny* Object reference will evaluate to true: b instanceof Object  **Ex:** interface Foo { } class A implements Foo { } class B extends A { } A a = new A(); B b = new B();  ***The following are true:*** a instanceof Foo, b instanceof A, b instanceof Foo, a instanceof Object. |
| **BITWISE OPERATORS** |

**String**

|  |
| --- |
| **Strings Are Immutable Objects** once a String object is created, it can never be changed. The VM cannot change an old value, so it creates a new String object. **Ex:** String s=”abc"; s=s.concat(”abc"); There are three Strings created [including the String that is being concatenated], original String did not change but it is not referenced [“lost” in memory]. StringBuffer and StringBuilder objects can change. **Concatenation:** If either operand is a String, (+) is a String concatenation operator. If both operands are numbers (+) is the addition operator. |
| **Strings and Memory:** To make Java more memory efficient, the JVM sets aside a special area of memory called the "**String constant pool**." When the compiler encounters a String literal, it checks the pool to see if an identical String already exists. If a match is found, the reference to the new literal is directed to the existing String, and no new String literal object is created. (The existing String has an additional reference.)  **Why immutable String?**  If several reference variables refer to the same String without even knowing it, it is bad if any of them could change then String's value. |
| **Creating and Declaring New Strings:**  **a)**String s = "abc"; // creates one String object and one reference variable  **b)**String s = new String("abc"); // creates two objects,and one reference variable  **a)** “abc" goes in the pool and s will refer to it.  **b)**Java creates a new String object in normal (non-pool) memory, and s will refer to it. In addition, the literal "abc" will be placed in the pool. |
| **The StringBuffer and StringBuilder Classes:** The java.lang.StringBuffer and java.lang.StringBuilder classes [added in Java 5] should be used when you modify strings of characters. If you choose to do a lot of manipulations with String objects, you will end up with a lot of abandoned String objects in the String pool. StringBuffer and StringBuilder can be modified over and over again without leaving behind many String objects. |
| **StringBuffer vs. StringBuilder:** StringBuilder [added in Java 5] has exactly the same API as the StringBuffer class, except StringBuilder is not thread safe [not synchronized]. Use StringBuilder instead of StringBuffer whenever possible because StringBuilder will run faster. |
| String objects are immutable; you can't change the value of a String object. x changes but y stays the same.  String x = "Java"; String y = x; x = x + " Bean"; System.out.println(y); // Java   * A new string is created (or a matching String is found in the String pool), leaving the original String object untouched. * The reference used to modify the String is then assigned the brand new String object.   **Ex:** 1. String s = "Fred"; 2. String t = s; 3. t.toUpperCase();  The original String object s is created. Both t and s reference the same String object. a brand new String object is created, then abandoned. |

**Flow Control**

|  |
| --- |
| break - execution jumps immediately to the 1st statement after the for loop, stops the entire loop, must be inside loop or switch statement.  continue – stops the current execution, must be inside of the loop.  return - Execution jumps immediately back to the calling method.  System.exit() - All program execution stops; the VM shuts down. |
| The labeled varieties are needed only in situations where you have a nested loop, and need to indicate which of the nested loops you want to break from, or from which of the nested loops you want to continue with the next iteration. |

**Handling Exceptions**

|  |
| --- |
| **Catching an Exception: Exception** is an occurrence that alters the normal program flow. When an exceptional event occurs, an exception is "**thrown**." The code handling an exception is called an "**exception handler**," and it "catches" the thrown exception. **Exception handling** transfers the execution of a program to an appropriate exception handler when an exception occurs. |
| try( "risky code goes here") defines a block of code in which exceptions may occur. Try must be se used with catch and/or finally, never alone. |
| catch clauses [one or more] match a specific exception to a block of code that handles it. catch blocks immediately follow the try block. catch blocks must all follow each other, without any other statements in between. Once control jumps to the catch block, it never returns to complete the balance of the try block. |
| A finally block encloses code that is always executed at some point after the try block, whether an exception was thrown or not. finally block immediately follows the try block. This is the place to close your files, release your network sockets, and perform any other cleanup your code requires. If the try block executes with no exceptions, the finally block is executed immediately after the try block completes. If there was an exception thrown, the finally block executes immediately after the proper catch block completes. |
| The **call stack** is the chain of methods that your program executes to get to the current method. If your program starts in method main() and it calls method a(), which calls method b(), which in turn calls method c(). The call stack [bottom-up]: main, a, b, c. The last method called is at the top of the stack, while the first calling method is at the bottom. We move back from the current method to the previously called method. An exception that's never caught will cause your application to stop running.  It's possible to grow the stack so large that the OS runs out of space to store the call Stack, a StackOverflowError. |
| Class Throwable provides printStackTrace()that prints stack trace from where the exception occurred. It prints the most recently entered method first and continues down, printing the name of each method as it works its way down the call stack (**unwinding the stack**) from the top. try {}catch (IndexOutOfBoundsException e) {e.printStackTrace();} |
| ***Resist the temptation to write a single catchall exception handler:*** try {}catch (Exception e) {e.printStackTrace();} |
| **Exception Matching:** the handlers for the most specific exceptions must always be placed above those for more general exceptions:  try {} catch (FileNotFoundException ex) {}catch (IOException e) {} |
| **Exception Declaration:** The exceptions that a method can throw must be *declared* (except RuntimeException). The list of thrown exceptions is part of a method's public interface. The throws: void myFunction() throws MyException1, MyException2 {}  (Just because the method declares that it throws an exception doesn't mean it always will. It just tells the world that it might.)  Any method that might throw an exception must declare the exception. That includes methods that aren't actually throwing it directly, but are "ducking" and letting the exception pass down to the next method in the stack. |
| But all non-RuntimeExceptions are considered "checked" exceptions, because the compiler checks to be certain you've acknowledged that "bad things could happen here." Runtime exceptions are *unchecked* exceptions. They derive from java.lang.RuntimeException.  A checked exception must be caught somewhere in your code. If you invoke a method that throws a checked exception but you don't catch the checked exception somewhere, your code will not compile. That's why they're called checked exceptions; the compiler checks to make sure that they're handled or declared. |
| **NOTE:** *Each method must either handle all checked exceptions by supplying a* catch *clause or list each unhandled checked exception as a thrown exception* ( "catch or declare" requirement). |
| You can create your own exception: class MyException extends Exception { } |
| All exceptions are instances of class Exception, subclass of java.lang.Exception. Exception class derives from the class Throwable (which derives from the class Object). There are two subclasses that derive from Throwable: Exception and Error. Error classes represent unusual situations that are not caused by program errors. **Ex**: the JVM running out of memory. Application won't be able to recover from an Error, so you're not required to handle them. Errors are technically not exceptions because they do not derive from class Exception. |
| **Assertions** let you test your assumptions during development. [The Java 6 compiler will use the assert keyword by default.]   * Assertions are disabled by default, you can choose to enable or disable your assertions at runtime.   **Enabling Assertions at Runtime**: java –ea [or –enableassertions] com.geeksanonymous.TestClass  **Disabling Assertions at Runtime:** java -da [or-disableassertions] com.geeksanonymous.TestClass  **Selective:** java -ea -da:com.geeksanonymous.Foo[ disable assertions in a single class, but keep them enabled for all others]   * The assertion code evaporates when the program is deployed, leaving no overhead or debugging code to track down and remove. * Assertions let your code stay cleaner and tighter, assertions are inactive unless specifically "turned on" (enabled), the code runs though it. * You assertion is true, code keeps running. If assertion is false, then a stop-the-world AssertionError is thrown. * **Declaration 1**: assert (y > x); // more code assuming y is greater than x   **Declaration** 2: assert (y > x): "y is " + y + " x is " + x; // more code assuming y is greater than x  The second expression [must return any value] generate a message that displays in the stack trace to give you a little more debugging info.   * Don't Use Assertions to Validate Arguments to a Public Method but throw an exception if the value is invalid. * Do Use Assertions to Validate Arguments to a Private Method. * Don't Use Assertions to Validate Command-Line Arguments |

**File Navigation and I/O**

|  |
| --- |
| **File Objects** represent the actual files (but not the data in the files) or directories that exist on a computer's physical disk. |
| **FileReader** class is used to read only single character, the whole stream of characters or a fixed number of characters from files. |
| **BufferedReader** class is used to read large chunks of data from a file at once, and keep this data in a buffer. When you ask for the next character or line of data, it is retrieved from the buffer, which minimizes the number of file operations performed. |
| **FileWriter** class is used to write to character files. Its write() methods allow you to write character(s) or Strings to a file. |
| **BufferedWriter** writes relatively large chunks of data to a file at once, minimizing the number of times that slow, which minimizes the number of file operations performed. |
| **PrintWriter:** new methods like format(), printf(), and append() make PrintWriters very flexible and powerful. |
| **Console** [ Java 6] convenience class provides methods to read input from the console and write formatted output to the console. |
| * boolean exists(): method returns true if it can find the actual file. * Boolean createNewFile(): creates a new file if it doesn't already exist. * flush():the last of the data you thought you had already written actually gets out to the file [When reading a file, no flushing is required]. * close(): method should be used when you are done using the file. It frees up expensive and limited operating system resources. |

**Serialization**

|  |
| --- |
| **Serialization** allows saving this object and all of its instance variables unless a variable is transient.   * ObjectOutputStream.writeObject() // serialize objects and write them to a stream * ObjectInputStream.readObject() // read the stream and deserialize objects * ObjectOutput[Input]Stream class [*higher*-level class] in java.io package, can be wrapped around FileOutput[Input]Stream [*lower*-level class]. * When you serialize an object, it saves object's entire "object graph." A deep copy of everything the saved object needs to be restored. * If a superclass is Serializable then all subclasses of that class automatically implement Serializable implicitly. * Serialization is not for statics: Static variables are not as part of the object's state…because they do not belong to the object! |
| **An object constructed using** new**:**   1. All instance variables are assigned default values. 2. The constructor is invoked, which immediately invokes the superclass constructor. 3. All superclass constructors complete. 4. Instance variables that are initialized as part of their declaration are assigned their initial value. 5. The constructor completes. |
| **Deserializing object:** No normal initialization happens! We want the values saved as part of the serialized state of the object to be reassigned.   1. The constructor does not run. 2. Instance variables are NOT given their initially assigned values. 3. If variables marked transient, they will not be restored to their original state (unless you implement readObject()). 4. Object references marked transient will always be reset to null, regardless of whether they were initialized at the time of declaration. 5. The class of the serialized object directly extends Object, or has ONLY serializable classes in its inheritance tree. |

**Parsing, Tokenizing, and Formatting**

|  |
| --- |
| **Regex engines** search textual data using instructions that are coded into ***expressions***. When you invoke a regex engine, you pass it the chunk of textual data you want it to process (a String or a stream), and you pass it the expression you want it to use to search through the data.  A regex search runs from left to right, and once a source's character has been used in a match, it cannot be reused. **Exceptions:** |
| **Simple Searches:** Data sources use zero-based source: abaaaba, index: 0123456  import java.util.regex.\*;  Pattern p = Pattern.compile("ab"); // the expression  Matcher m = p.matcher("abaaaba"); // the source  System.out.println("Pattern is " + m.pattern());  while(m.find())  System.out.print(m.start() + " " + m.group()); |
| **Searches Using Metacharacters**  \d - numeric digits, use it as "\\d".  \s - whitespace character.  \w - word character (letters, digits, or "\_" (underscore)).  [abc] - search only for a, b or c characters.  [a-f] - search only for a, b, c, d, e, or f characters.  0[xX][0-9a-fA-F] - a hexadecimal literal. 1st char - "0", 2nd char -"x" or an "X", 3rd char 0 -9, a-f or A-F.  [a-fA-F] - searches for the first six letters of the alphabet, both cases, it's NOT looking for an fA combination:.  "^" - negates the characters specified, nested brackets to create a union of sets.  "&&" - specifies the intersection of sets.  "." (dot) – wildcard. [“\\.” – not a metacharacter]  [^abc]- search for anything anything but a's, b's, or c's in a file you could say  proj1([^,])\*- search for zero or more characters that aren't a comma. |
| **Quantifiers** specifies an amount of characters [+ - 1 or more occurrences, \* - zero or more occurrences, ? - zero or one occurrence]  \d+ - search for one or more digits in a row.  0[xX]([0-9a-fA-F])+ - Once we've found our 0x or 0X, you can find from one to many occurrences of hex digits.  \d\d\d([-\s])?\d\d\d\d – 7 digits in a row, or 3 digits followed by a dash or a space followed by 4 digits |
| **Tokenizing** is the process of taking big pieces of source data, breaking them into little pieces, and storing the little pieces in variables.  **Tokens** are the actual pieces of data. **Delimiters** are the expressions that *separate* the tokens from each other.  When we tokenize source data, the delimiters themselves are discarded, and all that we are left with are the tokens.  The String split() method takes a regex expression as its argument, returns a String array of tokens produced by the split. |
| String[] tokens = ("ab.cde.fg").split("\\.");  for(String s : tokens)  System.out.println(">" + s + "<"); |
| **Formatting with printf() and format():** printf("format string", argument(s));  The construction of format strings: %[arg\_index$][flags][width][.precision]conversion char   * The values within [ ] are optional. The only required elements of a format string are the % and a conversion character. * **arg\_index** An integer followed directly by a $, this indicates which argument should be printed in this position. * **Flags:**   + - * "-" Left justify this argument       * "+" Include a sign (+ or -) with this argument       * "0" Pad this argument with zeroes       * "," Use locale-specific grouping separators (i.e., the comma in 123,456)       * "(" Enclose negative numbers in parentheses * **width** indicates the minimum number of characters to print. * **precision** used for formatting a floating-point number, it indicates the number of digits to print after the decimal point. * **conversion** The type of argument you'll be formatting. You'll need to know:   + - * b boolean       * c char       * d integer       * f floating point       * s string   Let's see some of these formatting strings in action:  int i1 = -123;  int i2 = 12345;  System.out.printf(">%1$(7d< \n", i1); > (123) <  System.out.printf(">%0,7d< \n", i2); >012,345<  System.out.format(">%+-7d< \n", i2); >+12345 <  System.out.printf(">%2$b + %1$5d< \n", i1, false); >false + -123< |
| **Scanner Tokenizing** |