***Method Declaration:***

* *A method declaration which specifies all the information needed to call the method. Two of the parts are method name and parameter list – is called method signature.*

|  |  |  |
| --- | --- | --- |
| *Element* | *Value in nap() example* | *Required?* |
| *Access Modifier* | *Public* | *No* |
| *Optional Specifier* | *Final* | *No* |
| *Return type* | *Void* | *Yes* |
| *Method name* | *Nap* | *Yes* |
| *Parameter List* | *Int minutes* | *Yes, but can be empty parentheses* |
| *Optional Exception List* | *Throws InterruptedException* | *No* |
| *Method body* | *-* | *Yes, but can be empty braces* |

*To call this method, just type its name followed by a single int value in parentheses. nap(10)*

***Access Modifiers:***

* *Java offers four choices of access modifiers*
* ***Private****: The private modifier means the method can be called only from within the same class.*
* ***Default(Package***-***Private access) –*** *With default access, the method can be called only from classes in same package. This one is tricky because there is no keyword for default access. You simply omit the access modifier.*
* ***Protected:*** *The protected modifier means the method can be called only from classes in the same package or subclasses.*
* ***Public:*** *The public modifier means the method can be called from any class.*

|  |
| --- |
| *public void walk1() {}*  *default void walk2() {} //Doesn’t compile*  *void public walk3() {} //Doesn’t compile*  *void walk4() {}* |

* *The walk1() method is a valid declaration with public access. The walk4() method is a valid declaration with default access. The walk2() method doesn’t compile because default is not a valid access modifier. The walk3() method doesn’t compile because the access modifier is specified after the return type.*

***Optional Specifiers:***

* *Optional specifiers come from the following list. Unlike with access modifiers, you can have multiple specifiers in the same method although not all combinations are legal.*
* *When this happens, you can specify them in any order. And since these specifiers are optional, you are not allowed to not have any of them at all. This means you can have zero or more specifiers in a method declaration.*
* *Static: The static modifier is used for class methods*
* *Abstract: The abstract modifier is used when a method body is not provided.*
* *Final: The final modifier is used when a method is not allowed to be overridden by a subclass*
* *Synchronized: The synchronized modifier is used with multithreaded code.*
* *Native: The native modifier is used when interacting with code written in another language such as C++.*
* *Strictfp: The strictfp modifier is used for making floating point calculations portable.*

|  |
| --- |
| *public void walk1() {}*  *public final void walk2() {}*  *public static final void walk3() {}*  *public final static void walk4() {}*  *public modifier void walk5() {} //Doesn’t compile*  *public void final walk6() {} // Doesn’t compile*  *final public void walk7() {}* |

* *The walk1() method is a valid declaration with no optional specifier. This is okay it is optional. The walk2() method is a valid declaration with final as the optional specifier. The walk3() and walk4() methods are valid declarations with both final and static as optional specifiers. The order of these two keywords doesn’t matter. The walk5() method doesn’t compile because modifier is not a valid optional specifier. The walk6() doesn’t compile because the optional specifier is after the return type. The walk7() method does compile. Java allows the optional specifiers to appear before the access modifier.*

***Return Type:***

* *The next item in a method declaration is the return type. The return type might be an actual Java type such as String or int.*
* *If there is no return type, the void keyword is used. This special return type void means without contents. In Java, there is no type there.*
* *A method must have a return type. If no value is returned, the return type is void. You cant omit the return type.*
* *When checking return types, you also have to look inside the method body. Methods with return type other than void are required to have a return statement inside the method body. This return statement must include primitive or object to be returned.*
* *Methods that have a return type of void are permitted to have a return statement with no value returned or omit the return statement entirely.*

|  |
| --- |
| *public void walk1() {}*  *public void walk2() { return; }*  *public String walk3() { return “ “;}*  *public String walk4 {} // Doesn’t compile*  *public walk5() {} // Doesn’t compile*  *public String int walk5() {} // Doesn’t compile*  *String walk7(int a) { if a==4) return “”;} //Doesn’t compile* |

* *Since the return type of the walk1() method is void, the return statement is optional. The walk2() method shows the optional return Statement that correctly doesn’t return anything.The walk3() method is a valid declaration with a String return type and a return statement that returns a String. The walk4() method doesn’t compile because the return statement is missing. The walk5() method doesn’t compile because the return statement is missing. The walk6() method doesn’t compile because it attempts to use two return types. You get only one return type. The walk7() method is little tricky, There is a return statement but it doesn’t always get to run. If a is 6, the return statement doesn’t get executed. Since the String always needs to be returned the compiler complains.*
* *When returning a value, it needs to be assignable to the return type. Imagine there is a local variable of that type to which it is assigned before being returned.*

|  |
| --- |
| *int integer() { return 9;}*  *int longMethod() { return 9L; }//Doesn’t compile* |

* *It is fairly mechanical exercise. You just add a line with local variable. The type of local variable matches the return type of the method. Then you return the local variable instead of the value directly.*

|  |
| --- |
| *int integerExpanded() {*  *int temp = 9;*  *return temp;*  *}*  *int longExpanded() {*  *int temp =9L; //Doesn’t compile*  *return temp;* |

* *This shows more clearly how you cant return a primitive in a method that returns an int.*

***Method Names:***

* *Method names follow the same rules as practiced via variable names.*

|  |
| --- |
| *public void walk1() {}*  *public void 2walk() {} // DOES NOT COMPILE*  *public walk3 void() {} // DOES NOT COMPILE*  *public void Walk\_$() {}*  *public \_() {} // DOES NOT COMPILE*  *public void() {} // DOES NOT COMPILE* |

* *The walk1() method is a valid declaration with a traditional name. the 2walk() method doesn’t compile because identifiers are not allowed to begin with numbers. The walk3() method doesn’t compile because the method name is before the return type. The Walk\_$() method is a valid declaration. While it is certainly isn’t good practice to start a method name with a capital letter and end with a punctuation it is legal. The \_ method is not allowed since it contains a single underscore. The final line of code doesn’t compile because the method name is missing.*

***Parameter List:***

* *Although the parameter list is required, it doesn’t have to contain any parameters. This means you can just have an empty pair of parentheses after the method name as follows*

*void nap() {}*

* *If you do have multiple parameters, you separate them with a comma. There are a couple more rules for the parameter list*

|  |
| --- |
| *public void walk1() {}*  *public void walk2 {} // DOES NOT COMPILE*  *public void walk3(int a) {}*  *public void walk4(int a; int b) {} // DOES NOT COMPILE*  *public void walk5(int a, int b) {}* |

* *The walk1() method is a valid declaration without any parameters. The walk2() method doesn’t compile because it is missing the parentheses around the parameter list. The walk3() method is a valid declaration with one parameter. The walk4() method doesn’t compile because the parameters are separated by a semicolon rather than a comma. Semicolons are for separating statements, not for parameter lists. The walk5() method is a valid declaration with two parameters.*

|  |
| --- |
| *public void zeroExceptions() {}*  *public void oneException() throws IllegalArgumentException*  *{}*  *public void twoExceptions() throws*  *IllegalArgumentException, InterruptedException {}* |

***Method Body:***

* *The final part of the method declaration is the method body except for abstract methods and interfaces. A method body is simply a code block. It has braces that contain zero or more java statements.*

|  |
| --- |
| *public void walk1() {}*  *public void walk2() // DOES NOT COMPILE*  *public void walk3(int a) { int name = 5; }* |

*The walk1() method is a valid declaration with an empty method body. The walk2() method doesn’t compile because it is missing braces around the empty method body. The walk3() is a valid declaration with one statement in the method body.*

***Working with varargs:***

* *A method may use a varargs parameter (variable argument) as if it is in array. It is a little different than an array, though. A varargs parameter must be the last element in a methods parameter list. This means you are allowed to only have one varargs parameter per method.*

|  |
| --- |
| *public void walk1(int... nums) {}*  *public void walk2(int start, int... nums) {}*  *public void walk3(int... nums, int start) {} // DOES NOT COMPILE*  *public void walk4(int... start, int... nums) {} // DOES NOT COMPILE* |

* *The walk1() method is a valid declaration with one varargs parameter. The walk2() method is a valid declaration with one int parameter and one varargs parameter. The walk3() and walk4() methods do not compile because they have a varargs parameter in a position that is not the last one.*
* *When calling a method with varargs parameter, you have a choice. You can pass in an array, or you can list the elements of the array and let Java create it for you. You can even omit the varargs values in the method call and Java will create an array of length zero for you.*

|  |
| --- |
| *public static void walk(int start, int… nums){*  *System.out.println(nums.length);*  *}*  *public static void main(String[] args){*  *walk(1); //0*  *walk(1,2) //1*  *walk(1,2,3) //2*  *walk(1, new int[]{4,5}) //2*  *}* |

* *Line 5 passes 1 as start but nothing else. This means Java creates an array of length 0 for nums. Line 6 passes 1 as start and one more value. Java converts this one value to an array of length 1. Line 7 passes 1 as start and two other values. Java converts these two values to an array of length 2. Line 8 passes 1 as start and an array of length 2 directly as nums. Java will create an empty array if no parameters are passed for a vararg. However it is still possible to pass null explicitly.*

*Walk(1, null) // throws a NullPointerException in walk()*

* *Since null isn’t an int, Java treats it as an array reference that happens to be null. It just passes on the null array object to walk. Then the walk() method throws an exception because it tries to determine the length of a null.*
* *Accessing varargs parameter is just like accessing an array. It uses array indexing.*

|  |
| --- |
| *public static void run(int… nums){*  *System.out.println(nums[1]);*  *}*  *public static void main(String[] args){*  *run(11,22); //22*  *}* |

* *Line 5 calls a varargs method with two parameters. When the method gets called, It sees an array of size 2. Since indexes are 0 based, 22 is printing.*

***Applying Access Modifiers:***

* *You already saw that there are four access modifiers: public, private, protected and default access.*
* ***Private****: only accessible with the same class*
* ***Default****(package-private) access: private plus other classes in the same package.*
* ***Protected:*** *Default access + child classes*
* *Public: Protected + classes in the other packages.*

***Private Access:***

* *Private Access is easy. Only code in the same class can call private methods or access private fields. This is perfectly legal code because everything in one class:*

|  |
| --- |
| *package pond.duck;*  *public class FatherDuck{*  *private String noise=”quack”;*  *private void quack() {*  *System.out.println(noise); //Private access is ok*  *}*  *private void makeNoise(){*  *quack(); //Private access is ok*  *}}* |

*So far so good. FatherDuck makes a call to private method quack() on Line 8 and uses private instance variable noise on Line 5*

*Now we add another class:*

|  |
| --- |
| *package pond.duck;*  *public class BadDuckling() {*  *public void makeNoise() {*  *FatherDuck duck = new FatherDuck();*  *duck.quack(); //Doesn’t Compile*  *System.out.println(duck.noise); // Doesn’t compile* |

*BadDuckling is trying to access an instance variable and a method it has no business touching. On line 5, it tries to access a private method in another class. On line 6, it tries to access private instance variable in another class. Both generate compile errors.*

***Default(Package-Private) Access:***

* *When there is no access modifier, Java uses the default, which is package-private access. This means that the member s private to classes in the same package. In other words, only classes in the package may access it.*

|  |
| --- |
| *package pond.duck;*  *public class MotherDuck{*  *String noise =”quack”;*  *void quack(){*  *System.out.println(noise); //Default access is ok*  *}*  *private void makeNoise(){*  *quack(); // default access is ok*  *}}* |

*MotherDuck can refer to noise and call quack(). After all, members in the same class are certainly in the same package. The big difference s MontherDuck lets other classes in the same package access members whereas FatherDuck doesn’t due to being private. GoodDuckling has a much better experience than BadDuckling!*

|  |
| --- |
| *package pond.duck;*  *public class GoodDuckling{*  *public void makeNoise(){*  *MotherDuck duck = new MotherDuck();*  *duck.quack(); //default access*  *System.out.println(duck.noise); //default access* |

|  |
| --- |
| *package pond.swan;*  *import pond.duck.MotherDuck; // import another package*  *public class BadCygnet {*  *public void makeNoise() {*  *MotherDuck duck = new MotherDuck();*  *duck.quack(); // DOES NOT COMPILE*  *System.out.println(duck.noise); // DOES NOT COMPILE*  *}*  *}* |

*Protected Access:*

* *Protected access allows everything that default access allows and more. The protected access modifier adds the ability to access members of a parent class.*
* *First, create a Bird class and give protected access to its members*

|  |
| --- |
| *package pond.shore;*  *public class Bird{*  *protected String text=”floating”; //protected access*  *protected void floatInWater() //protected access*  *System.out.println(text);* |

*Next we create a subclass*

|  |
| --- |
| *package pond.goose;*  *import pond.shore.Bird; // in a different package*  *public class Gosling extends Bird{ // extends means create subclass*  *public void swim(){*  *floatInWater(); //calling protected member*  *System.out.println(text); //accessing protected member* |

* *This is a simple subclass. It extends the Bird class. Extending means creating a subclass that has access to any protected or public members of the parent class. Running this code prints floating twice. Once from calling floatInWater() and once from the print statement in swim(). Since Gosling is a subclass of Bird, it can access these members even though it is in a different package.*
* *Remember that protected also gives us access to everything that default access does. This means that a class in the same package as Bird can access its protected members.*

|  |
| --- |
| *package pond.shore; // same package*  *as Bird*  *public class BirdWatcher {*  *public void watchBird() {*  *Bird bird = new Bird();*  *bird.floatInWater(); // calling protected member*  *System.out.println(bird.text); // accessing protected member*  *}*  *}* |

*Since Bird and BirdWatcher are in the same package , BirdWatcher can access members of the Bird variable. The definition of protected allows access to subclasses and classes in the same package.*

*Now lets try something from a different package:*

|  |
| --- |
| *package pond.inland;*  *import pond.shore.Bird; // different package than Bird*  *public class BirdWatcherFromAfar {*  *public void watchBird() {*  *Bird bird = new Bird();*  *bird.floatInWater(); // DOES NOT COMPILE*  *System.out.println(bird.text); // DOES NOT COMPILE*  *}}* |

*BirdWatcherFromAfar is not in the same package as Bird and it doesn’t inherit from the Bird. This means that it doesn’t allow to access protected members from Bird.*

* *Subclasses and classes in the same package are the only ones allowed to access protected members.*

|  |
| --- |
| *package pond.swan;*  *2: import pond.shore.Bird; // in different package than Bird*  *3: public class Swan extends Bird { // but subclass of Bird*  *4: public void swim() {*  *5: floatInWater(); // subclass access to superclass*  *6: System.out.println(text); // subclass accessto superclass*  *7: }*  *8: public void helpOtherSwanSwim() {*  *9: Swan other = new Swan();*  *10: other.floatInWater(); // subclass access to superclass*  *11: System.out.println(other.text); // subclass access to superclass*  *13: }*  *14: public void helpOtherBirdSwim() {*  *15: Bird other = new Bird();*  *16: other.floatInWater(); // DOES NOT COMPILE*  *17: System.out.println(other.text); // DOES NOT COMPILE*  *18: }*  *19: }* |

*Swan is not in the same package as Bird but does extend it which implies it has access to the protected members of Bird since its subclass. And it does line 5 and 6, refer to protected members via inheriting them. Line 10 and 11 also successfully use protected members of Bird. This is allowed because these lines refer to a Swan object. Swan inherits from Bird, so this is okay. It is sort of a two-phase check. The Swan class is allowed to use protected members of Bird, and we are referring to a Swan object. Granted, it is a Swan object created on line 9 rather than an inherited one, but it is still a Swan object. Lines 16 and 17 do not compile. Wait a minute. They are almost exactly the same as lines 10 and 11! There’s one key difference. This time a Bird reference is used rather than inheritance. It is created on line 15. Bird is in a different package, and this code isn’t inheriting from Bird, so it doesn’t get to use protected members. Say what now? We just got through saying repeatedly that Swan inherits from Bird. And it does. However,* *the variable reference isn’t a Swan. The code just happens to be in the Swan class.It’s okay to be confused. Looking at it a different way, the protected rules apply under two scenarios: A member is used without referring to a variable. This is the case on lines 5 and 6. In this case, we are taking advantage of inheritance and protected access is allowed. A member is used through a variable. This is the case on lines 10, 11, 16, and 17. In this case, the rules for the reference type of the variable are what matter. If it is a subclass, protected access is allowed. This works for references to the same class or a subclass.*

|  |
| --- |
| *package pond.goose;*  *import pond.shore.Bird;*  *public class Goose extends Bird {*  *public void helpGooseSwim() {*  *Goose other = new Goose();*  *other.floatInWater();*  *System.out.println(other.text);*  *}*  *public void helpOtherGooseSwim() {*  *Bird other = new Goose();*  *other.floatInWater(); // DOES NOT COMPILE*  *System.out.println(other.text); // DOES NOT COMPILE*  *}*  *}* |

*The first method is fine. In fact, it is equivalent to the Swan example. Goose extends Bird.Since we are in the Goose subclass and referring to a Goose reference, it can access protected members. The second method is a problem. Although the object happens to be a Goose, it is stored in a Bird reference. We are not allowed to refer to members of the Bird class since we*

*are not in the same package and the reference type of other is not a subclass of Goose.*

|  |
| --- |
| *package pond.duck;*  *import pond.goose.Goose;*  *public class GooseWatcher {*  *public void watch() {*  *Goose goose = new Goose();*  *goose.floatInWater(); // DOES NOT COMPILE*  *}*  *}* |

*This code doesn’t compile because we are not in the goose object. The floatInWater() method is declared in Bird. GooseWatcher is not in the same package as Bird, nor does it extend Bird.Goose extends Bird. That only lets Goose refer to floatInWater() and not callers of Goose.*

***PUBLIC ACCESS:***

* *Public means anyone can access the member from anywhere. The Java Module System redefines “anywhere” and it becomes possible to restrict access to public code. When a code sample, you can assume it isn’t in a module unless explicitly stated.*

|  |
| --- |
| *public class DuckTeacher {*  *public String name = "helpful"; // public access*  *public void swim() { // public access*  *System.out.println("swim");*  *}*  *}* |

*DuckTeacher allows access to any class that wants it. Now we can try it*

|  |
| --- |
| *package pond.goose;*  *import pond.duck.DuckTeacher;*  *public class LostDuckling {*  *public void swim() {*  *DuckTeacher teacher = new DuckTeacher();*  *teacher.swim(); //*  *allowed*  *System.out.println("Thanks" + teacher.name); //*  *allowed*  *}*  *}* |

***Access Modifiers:***

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| ***Method in \_\_ can access a \_\_member*** | ***private*** | ***Default(package-private)*** | ***protected*** | ***public*** |
| *The same class* | *Yes* | *Yes* | *Yes* | *Yes* |
| *Another class in the same package* | *No* | *Yes* | *Yes* | *Yes* |
| *In a subclass in a different package* | *No* | *No* | *Yes* | *Yes* |
| *An unrelated class in different package* | *No* | *No* | *No* | *Yes* |

***Applying the static keyword:***

* *When the static keyword is applied to a variable, method or class, it applies to the class rather than the specific instance of the class. Static methods don’t require an instance of the class. They are shared among all users of the class. Static variable as being a member of the single class object that exists independently of any instances of that class.*

|  |
| --- |
| *public class Koala{*  *public static int count=0;*  *public static void main(String args[]){*  *System.out.println(count);* |

* *Here the JVM basically calls Koala.main() to get the program started. We can have a KoalaTester that does nothing but call the main() method.*

|  |
| --- |
| *public class KoalaTester{*  *public static void main(String[] args){*  *Koala.main(new String[0]);*  *}}* |

* *In addition to main() methods static methods may have two main purposes*
* *For utility or helper methods that don’t require any object state. Since there is no need to access instance variables, having static methods eliminates the need for the caller to instantiate an object just to call the method.*
* *For state that is shared by all instances of a class, like a counter. All instances must share the same state. Methods that merely use that state should be static as well.*
* *Accessing a static member is easy. You just put the class name before the method or variable and you are done. You can use an instance of the object to call a static method. The compiler checks for the type of the reference and uses that instead of the object.*

|  |
| --- |
| *5: Koala k = new Koala();*  *6: System.out.println(k.count); // k is a Koala*  *7: k = null;*  *8: System.out.println(k.count); // k is still a Koala* |

*The code outputs 0 twice. Line 6 sees that k is a koala and count is a static variable, so it reads that static variable. Line 8 does the same thing. Java doesn’t care that k happens to be null*

***Static vs Instance:***

* *A static member cannot call an instance member without referencing an instance of the class.*

|  |
| --- |
| *public class Static {*  *private String name = "Static class";*  *public static void first() { }*  *public static void second() { }*  *public void third() { System.out.println(name); }*  *public static void main(String args[]) {*  *first();*  *second();*  *third(); // DOES NOT COMPILE*  *}*  *}* |

* *The compiler will give you an error about making a static reference to a non static method. If we fix this by static to third() we create a new problem. Now third() is referring to nonstatic name. Adding static to name as well would solve the problem. Another solution would have been to call a third as an instance method – for eg. New Static().third()*
* *A static method or instance method can call a static method because static methods don’t require an object to use. Only an instance method can call another instance method on the same class without using a reference variable, because instance methods do require an object.*

|  |
| --- |
| *public class Gorilla {*  *2: public static int count;*  *3: public static void addGorilla() { count++; }*  *4: public void babyGorilla() { count++; }*  *5: public void announceBabies() {*  *6: addGorilla();*  *7: babyGorilla();*  *8: }*  *9: public static void announceBabiesToEveryone() {*  *10: addGorilla();*  *11: babyGorilla(); // DOES NOT COMPILE*  *12: }*  *13: public int total;*  *14: public static double average*  *15: = total / count; // DOES NOT COMPILE*  *16: }* |

*Line 3 and 4 are fine because the static and instance methods can refer to the static variable. Lines 5-8 are fine because an instance method can call a static method. Line 11 doesn’t compile because a static method cannot call an instance method. Similarly Line 15 doesn’t compile because a static variable is trying to use an instance variable.*

*A common use for static variables is counting number of instances.*

|  |
| --- |
| *public class Counter {*  *private static int count;*  *public Counter() { count++; }*  *public static void main(String[] args) {*  *Counter c1 = new Counter();*  *Counter c2 = new Counter();*  *Counter c3 = new Counter();*  *System.out.println(count); // 3*  *}*  *}* |

*Each time the constructor gets called, it increments count by 1. This example relies on the fact that static and instance variables are automatically initialized to default value for that type which is 0 for int. Also notice that we dint write Counter.count. It isn’t necessary because we are already in that class so that compiler can infer it.*

*Each object has a copy of the instance variables. There is only one copy of the code for the instance methods. Each instance of the class can call it as many times as it would like. However, each call of an instance method gets space out of the stock for method parameters and local variables. The same thing happens for static methods. There is one copy of the code Parameters and local variables go on to the stack.*

*Only data gets its own copy. There is no duplicate copies of the code itself.*

***Static Variables:***

* *Some static variables are meant to change as the program runs. Counters are a common example of this. We want the count to increase over time. Just as with instane variables, you can initialize a static variable on the line it is declared.*

|  |
| --- |
| *public class Initializers {*  *private static int counter = 0; //initialization*  *}* |

* *Other static variables are meant to never change during the program. This type of variable is known as a constant. It uses the final modifier to ensure the variable never changes. Constants use the modifier static final and a different naming convention than other variables.*

|  |
| --- |
| *public class Initializers {*  *private static final int NUM\_BUCKETS = 45;*  *public static void main(String[] args) {*  *NUM\_BUCKETS = 5; // DOES NOT COMPILE*  *}*  *}* |

* *The compiler will make sure that you don’t accidentally update the final variable.*

|  |
| --- |
| *private static final ArrayList<String> values = new ArrayList<>();*  *public static void main(String[] args) {*  *values.add("changed");*  *}* |

* *It actually does compile since values is a reference variable. We are allowed to call methods on reference variables. All the compiler can do is check that we don’t try to reassign the final values to point to a different object.*

***Static Initialization:***

* *Static Initializers add the static keyword to specify they should be run when the class is first loaded.*

|  |
| --- |
| *private static final int NUM\_SECONDS\_PER\_MINUTE;*  *private static final int NUM\_MINUTES\_PER\_HOUR;*  *private static final int NUM\_SECONDS\_PER\_HOUR;*  *static{*  *NUM\_SECONDS\_PER\_MINUTE=60;*  *NUM\_MINUTES\_PER\_HOUR=60;*  *}*  *static{*  *NUM\_SECONDS\_PER\_HOUR = NUM\_SECONDS\_PER\_MINUTE\* NUM\_MINUTES\_PER\_HOUR;* |

* *All static initializers run when the class is first used in the order they are defined. The statements in them run and assign any static variables as needed.*

|  |
| --- |
| *14: private static int one;*  *15: private static final int two;*  *16: private static final int three = 3;*  *17: private static final int four; // DOES NOT COMPILE*  *18: static {*  *19: one = 1;*  *20: two = 2;*  *21: three = 3; // DOES NOT COMPILE*  *22: two = 4; // DOES NOT COMPILE*  *23: }* |

* *Line 14 declares a static variable that is not final. It can be assigned as many times as we like. Line 15 declares a final variable without initializing it. This means we can initialize it exactly once in a static block. Line 22 doesn’t compile because this is the second attempt. Line 16 declares a final variable and initializes it at the same time. We are not allowed to assign it again, so line 21 doesn’t compile. Line 17 declares a final variable that never gets initialized. The compiler gives a compiler error because it knows that the static blocks are the only place the variable could possibly get initialized. Since the programmer forgot, this is clearly an error.*

***Try to avoid Static and Instance Initializers:***

* *Using static and instance initializers can make your code much harder to read. Everything that could be done in an instance initializer could be done in a constructor instead. There is a common case to use a static initializer when you need to initialize a static field and the code to do so requires more than one line. This often occurs when you want to initialize a collection like an ArrayList. When you do need to use a static initializer put all the static initialization in the same block.*

***Static Imports:***

* *Imports are convenient because you don’t need to specify where each class comes from each time you use it. There is another type of import called as static import. Regular imports are for importing classes. Static imports are for importing static members of classes.*
* *Just like regular imports, you can use a wildcard or import a specific member. The idea is that you shouldn’t have to specify where each static method or variables comes from each time you use it.*
* *In a large program, static imports can be overused. When importing from too many places, it can be hard to remember where each static member comes from.*

|  |
| --- |
| *import java.util.List;*  *import java.util.Arrays;*  *public class Imports {*  *public static void main(String[] args) {*  *List<String> list = Arrays.asList("one", "two");*  *}*  *}* |

*Rewriting this with static imports:*

|  |
| --- |
| *import java.util.List;*  *import static java.util.Arrays.asList; // static*  *import*  *public class StaticImports {*  *public static void main(String[] args) {*  *List<String> list = asList("one", "two"); // no*  *Arrays.*  *}*  *}* |

*In the above example, we are specifically importing the asList method. This means that anytime we refer to asList in the class it is Arrays.asList()*

|  |
| --- |
| *import static java.util.Arrays; // DOES NOT COMPILE*  *2: import static java.util.Arrays.asList;*  *3: static import java.util.Arrays.\*; // DOES NOT COMPILE*  *4: public class BadStaticImports {*  *5: public static void main(String[] args) {*  *6: Arrays.asList("one"); // DOES NOT COMPILE*  *7: } }* |

*Line 1 tries to use a static import to import a class. Static imports are only for importing static members. Regular imports are for importing a class.*

***Passing data among Methods:***

* *Java is a “pass-by-value” language. This means that a copy of the variable is made and the method receives the copy. Assignments made in the method do not affect the caller.*

|  |
| --- |
| *public static void main(String[] args) {*  *3: int num = 4;*  *4: newNumber(num);*  *5: System.out.println(num); // 4*  *6: }*  *7: public static void newNumber(int num) {*  *8: num = 8;*  *9: }* |

* *On line 3, num is assigned to the value of the 4. On line4, we call a method. On line 8, the num parameter in the method gets set to 8. Although this parameter has the same name as the variable on line 3, this is a coincidence.*

|  |
| --- |
| *public static void main(String[] args) {*  *String name = "Webby";*  *speak(name);*  *System.out.println(name);*  *}*  *public static void speak(String name) {*  *name = "Sparky";*  *}* |

*The correct answer is webby. The variable assignment is only the method parameter and doesn’t affect the caller. This is because we can call methods on parameters. As an example, here is the code that calls a method on the StringBuilder passed into the method.*

|  |
| --- |
| *public static void main(String[] args) {*  *StringBuilder name = new StringBuilder();*  *speak(name);*  *System.out.println(name); // Webby*  *}*  *public static void speak(StringBuilder s) {*  *s.append("Webby");*  *}* |

*In this case, the output is Webby because the method merely calls a method on the parameter. It doesn’t reassign name to a different object. The variable s is a copy of the variable name. Both point to the same StringBuilder , which means that changes made to the StringBuilder are available to both the references.*

***Pass-By-Value vs Pass-By-Reference:***

* *Different languages handle parameters in different ways. Pass-by-value is used by many languages, including Java. In this example, the swap method doesn’t change the original values. It only changes a and b within the method.*

|  |
| --- |
| *public static void main(String args[]){*  *int original1 = 1;*  *int original2 = 2;*  *swap(original1, original2);*  *System.out.println(original1); //1*  *System.out.println(original2); //2*  *}*  *public static void swap(int a, int b){*  *int temp=a;*  *a=b;*  *b=temp;*  *}* |

*The other approach is pass-by-reference. It is used by default n few languages.*

*To review, Java uses pass-by-value to get data into a method. Assigning a new primitive or reference to a parameter doesn’t change the caller. Calling methods on a reference to an object can affect the caller.*

* *A copy is made of the primitive or reference and returned from the method. Most of the time, this returned value is used. For example, it might be stored in a variable. If the returned value is not used, the result is ignored.*

|  |
| --- |
| 1*: public class ReturningValues {*  *2: public static void main(String[] args) {*  *3: int number = 1; //number=1*  *4: String letters = "abc"; //letters=abc*  *5: number(number); //number=1*  *6: letters = letters(letters); //letters=abcd*  7*: System.out.println(number + letters); //*  *1abcd*  *8: }*  *9: public static int number(int number) {*  *10: number++;*  *11: return number;*  *12: }*  *13: public static String letters(String letters) {*  *14: letters += "d";*  *15: return letters;*  *16: }*  *17: }* |

***Overloading Methods:***

* *Method Overloading occurs when methods have the same name but different method signatures which means they differ by method parameters. System.out.println and StringBuilder’s append methods provide many overloaded versions, so you can pass just about anything to them without having to think about them.*
* *In both of these examples, the only change was the type of the parameter. Overloading also allows us different number of parameters.*
* *Everything other than method name can vary for overloading methods. This means there can be different access modifiers, specifiers like static, return types and exception lists. These are all valid overloaded methods:*

|  |
| --- |
| *public void fly(int numMiles)*  *public void fly(short numFeet)*  *public boolean fly() {return false; }*  *void fly(int numMiles, short numFeet) {}*  *public void fly(short numFeet, int numMiles) throws Exception {}* |

*As you can see, we can overload by changing anything in the parameter list. We can have a different type, more types, or the same types in a different order. Also notice that the return type, access modifier and exception list are irrelevant to overloading.*

* *This is not a valid Method overloading*

*public void fly(int numMiles) {}*

*public int fly(int numMiles) {} // DOES NOT COMPILE*

*The method doesn’t compile because it differs from the original only by return type. The parameter list are the same, so they are duplicate methods as far as Java is concerned.*

* *public void fly(int numMiles) {}*

*public static void fly(int numMiles) {} // DOES NOT COMPILE*

*Again the parameter list is same. You cannot have methods where the only difference is that one is an instance method and one is static method.*

* *public void fly(int[] length) {}*

*public void fly(int… lengths) {} //Doesn’t compile*

*Java treats varargs as if they are an array. This means that the method signature is the same for both the methods. Since we are not allowed to overload methods with the same parameter list, the code doesn’t compile. Even though the code doesn’t look the same, it compiles to the same parameter list.*

*fly(new int[]{1,2,3});*

*However, you can call the varargs version with standalone parameters.*

*fly(1,2,3);*

*Obviously, this means they don’t compile exactly the same. The parameter list is the same, though.*

***Autoboxing:***

|  |
| --- |
| *public void fly(Integer numMiles)*  *public void fly(int numMiles)* |

*Java will match the int numMiles version. Java tries to use the most specific parameter list it can find. When the primitive int version isn’t present, it will autobox. However, when the primitive int version is provided, there is no reason for Java to do the extra work of autoboxing.*

***Reference Types:***

* *Given the rule that Java picks the most specific version of a method that it can, what is the output of the below code:*

|  |
| --- |
| *public class ReferenceTypes{*  *public void fly(String s){*  *System.out.print(“string”);*  *}*  *public void fly(Object o){*  *System.out.print(“object”);*  *}*  *public static void main(String args[]){*  *ReferenceTypes r = new ReferenceTypes();*  *r.fly(“test”);*  *System.out.print(“-“);*  *r.fly(56);* |

*The answer is string-object. The first call is a String and finds a direct match. There is no reason to use the object version when the nice String parameter list just waiting to be called. The second call looks for an int parameter list. When it doesn’t find one, it autoboxes to Integer. Since it doesn’t find a match, it gets the Object one.*

|  |
| --- |
| *public static void print(Iterable i){*  *System.out.print(“I”);*  *}*  *public static void print(CharSequence c){*  *System.out.print(“C”);*  *}*  *public static void print(Object o){*  *System.out.print(“O”);*  *}*  *public static void main(String args[]){*  *print(“abc”);*  *print(new ArrayList<>());*  *print(LocalDate.of(2019, Month.JULY, 4));* |

*The answer is CIO. The code is due to promotion. The first call to print passes a String. String and StringBuilder implement the CharSequence interface.*

*The second call to print() passes an ArrayList. In this case, Iterable is an interface for classes you can iterate over.*

*The final call to print passes a LocalDate. This is another class for which Object signature is being used.*

*Primitives:*

* *Primitives work in similar way to reference variables. Java tries to find the most specific matching overloaded method.*

|  |
| --- |
| *public class Plane{*  *public void fly(int i){*  *System.out.print(“int”);*  *}*  *public void fly(long l){*  *System.out.print(“long”);*  *}*  *public static void main(String args[]){*  *Plane p = new Plane();*  *p.fly(“123”);*  *System.out.print(“-“);*  *p.fly(123L);*  *}*  *}* |

*The answer is int-long. The first call passes an int and sees an exact match. If we comment out the overloaded method with int parameter list, the output becomes long-long. Java has no problem calling a larger primitive. However, it will not do so unless a better match is not found.*

*Note that Java can only accept wider types. An int can be passed to a method taking a long parameter. Java will not automatically convert to a narrower type. If you want to pass a long method taking an int parameter, you have to add a cast to explicitly say narrowing is okay.*

*GENERICS:*

* *These aren’t valid method overloads*

*public void walk(List<String> strings) {}*

*public void walk(List<Integer> integers) {} //Doesn’t compile*

*Java has a concept called* ***type erasure*** *where generics are used only at compile time. This means that the compiled code looks like this*

*public void walk(List strings) {}*

*public void walk(list Integers) {} //Doesn’t compile*

*We clearly cant have two methods with the same method signature, so this doesn’t compile. Method overloads must differ in at least one of the method parameters.*

*ARRAYS:*

* *Unlike previous example the code is just fine*

*public static void walk(int[] ints) {}*

*public static void walk(Integer[] integers) {}*

*Arrays have been around since the beginning of Java. They specify the actual types and don’t participate in type erasure.*

*Putting it Altogether:*

* *So far, all the rules for when an overloaded method is called should be logical. Java calls the most specific method it can. When some of the types interact, the Java rules focus on backward compatibility.*
* *A long time ago, autoboxing and varargs doesn’t exist. Since old code still needs to work, this means autoboxing and varargs come last when Java looks at overloaded methods.*

|  |  |
| --- | --- |
| *Rule* | *Example* |
| *Exact match by type* | *String glide(int I, int j)* |
| *Larger primitive type* | *String glide(long I, long j)* |
| *Autoboxed type* | *String glide(Integer I, Integer j)* |
| *Varargs* | *String glide(int…)* |

|  |
| --- |
| *public class Glider2 {*  *public static String glide(String s){*  *return “1”;*  *}*  *public static String glide(String… s){*  *return “2”;*  *}*  *public static String glide(Object o){*  *return “3”;*  *}*  *public static String glide(String s, String t){*  *return “4”;*  *}*  *public static void main(String[] args){*  *System.out.print(glide(“a”));*  *System.out.print(glide(“a”,”b”));*  *System.out.print(glide(“a”,”b”,”c”));*  *}}* |

*It prints out 142. The first call matches the signature taking a single String because that is the most specific match. The second call matches the signature, taking two String parameters since that is the exact match. It isn’t until the third call that the varargs version is used since there are no better matches.*

*As accommodating as Java is with trying to find a match, it will do only one conversion.*

*public class TooManyConversions{*

*public static void play(Long l){}*

*public static void play(Long… l){}*

*public static void main(String[] args){*

*play(4); //Doesn’t compile*

*play(4L);//calls the Long version*

***Encapsulating Data:***

* *public class Swan{*

*int numberEggs; //instance variable*

*}*

*This is a class with a field which isn’t private. Since there is a default(package-private) access that means any class in the package can set numberEggs. We no longer have control of what gets set in your own class. A caller could even write this way:*

*mother.numberEggs=-1;*

*We don’t want Swan to have negative number of eggs!.*

* *Encapsulation to the rescue. Encapsulation means only methods in a class with the variables can refer to the instance variables. Callers are required to use these methods.*

|  |
| --- |
| *public class Swan{*  *private int numberEggs; //private*  *public int getNumberEggs(){*  *return numberEggs;*  *}*  *public void setNumberEggs(int newNumber){*  *if(newNumber>=0){*  *numberEggs= newNumber;*  *}}* |

*Now, numberEggs is private. This means only the code within the class can read or write the value of numberEggs. Since we wrote the class, we know better than to set a negative number of Eggs. We added a getter method to read the value which is called as accessor or getter. We also added a method to update the value, which is called mutator method or a setter. The setter has an if statement in this example to prevent setting the instance variable to an invalid value. The guard condition protects the instance variable*

*For encapsulation, remember that data(an instance variable) is private and getters/setters are public. Java defines a naming convention for getters or setters*

|  |  |
| --- | --- |
| *Rule* | *Example* |
| *Getter methods most frequently begin with is if the property is a boolean* | *public boolean isHappy(){*  *return happy;*  *}* |
| *Getter methods begin with get if the property is not boolean* | *public int getNumberEggs(){*  *return numberEggs;*  *}* |
| *Setter methods begin with set* | *public void setHappy(boolean \_happy){*  *happy = \_happy;*  *}* |

*For data to be encapsulated, you don’t have to provide getters and setters. As long as instance variables are private, you are good.*

*public class Swan{*

*private int numEggs;*

*public void layEgg(){*

*numEggs++;*

*}*

*public void printEggCount(){*

*System.out.println(numEggs);*

*}*

*To review you can tell it is well encapsulated class because the numEggs instance variable is private. Only methods can retrieve and update the value.*