***Understanding Inheritance:***

* *When creating a new class in Java, you can define the class as inheriting from an existing class. Inheritance is the process by which a subclass automatically includes any public or protected members of the class including primitives, objects, or methods defined in the parent class*
* *We refer to any class that inherits from another class as a subclass or child class as it is considered as a descendant of that class.*
* *Alternatively, we refer to the class that the child inherits from as the superclass or parent class as it is considered as an ancestor class.*
* *When one class inherits from the parent class, all public and protected members are automatically available as part of private class. Package-private members are available if the child class is in the same package as the parent class. Private members are restricted to the class they are defined in and are never available via inheritance. This doesn’t mean the parent class doesn’t have private members that can hold data or modify an object; it just means that the class has no direct reference to them.*

|  |
| --- |
| *public class BigCat{*  *public double size;*  *}*  *public class Jaguar extends BigCat{*  *public Jaguar(){*  *size=10.2;*  *}*  *public void printDetails(){*  *System.out.println(size);*  *}}* |

*In the Jaguar class, size is accessible because it is marked public. Via Inheritance, the Jaguar subclass can read or write size as if it were its own member.*

***Single vs Multiple Inheritance:***

* *Java supports Single Inheritance by which a class may inherit from only one direct parent class. Java also supports multiple levels of inheritance, by which one class may extend another class, which in turn extends another class. You can have number of levels of inheritance, allowing each descendant to gain access to its ancestors members.*
* *To truly understand single inheritance, it may be helpful to contrast it with multiple inheritance, by which a class may have multiple direct parents. By design, java doesn’t support multiple inheritance in the language because multiple inheritance may lead to complex often difficult to maintain data models.*
* *It is possible in Java to prevent a class from being extended by marking the class as final. If you try to define from a class that inherits the final class then the class will fail to compile.*

***Inheriting Object:***

* *In Java, all classes inherit from a single class, java.lang.Object or Object for short. Furthermore, Object is the only class that doesn’t have a parent class.*
* *Compiler has been automatically inserting code into any class you write that doesn’t extend a specific class.*

*public class Zoo {}*

*public class Zoo extends java.lang.Object {}*

*The key is that when Java sees you define a class that doesn’t extend another class, it automatically adds the syntax extends java.lang.Object to the class definition. The result is that every class gains to any accessible methods in the Object class.*

* *For example, the toString() and equals() methods are available in the Object therefore they are accessible in all classes.*
* *Without being overridden in a subclass, though they may not be particularly useful. On the other hand, when you define a new class that extends an existing class, Java oesnt automatically extend the Object class. Since all the classes inherit from Object, extending an existing class means the child already inherits from Object by definition.*
* *Primitive types such as int and Boolean donot inherit from Object since they are not classes. Through autoboxing they can be assigned or passed as an instance of an associated wrapper class which doesn’t inherit Object.*

***Creating classes:***

* *Lets create two classes Animal.java and Lion.java in which the Lion class extends the Animal class. Assuming they are in the same package an import statement is not required in Lion.java to access the Animal class*

|  |
| --- |
| *public class Animal {*  *private int age;*  *protected String name;*  *public int getAge() {*  *return age;*  *}*  *public void setAge(int newAge) {*  *age = newAge;*  *}*  *}* |

*Lion.java*

|  |
| --- |
| *public class Lion extends Animal{*  *public void setProperties(int age, String n){*  *setAge(age);*  *name=n;*  *}*  *public void roar(){*  *System.out.print(name +”, age “ +getAge()+”says:Roar!”);*  *}*  *public static void main(String args[]){*  *var lion = new Lion();*  *lion.setProperties(3,”kion”);*  *lion.roar();*  *}}* |

* *The extends keyword is used to express the Lion class inherits the Animal class. When executed, the Lion program prints the following*

*Kion, age, 3, says:Roar!*

* *The instance variable age is marked as private and is not directly accessible from the subclass Lion. Therefore the following will not compile*

*public class Lion extends Animal{*

*…*

*Public void roar(){*

*System.out.print(“Lions age: “+age); //Doesn’t compile*

*}*

*….*

*}*

* *The age variable can be accessed indirectly through the getAge() and setAge() methods, which are marked as public in the Animal class. The name variable can be accessed by directly in the Lion class since it is marked as protected in Animal class.*

***Applying Class Access modifiers:***

* *You can also apply access modifiers to class definitions, since we have been adding the public access modifier to most classes upto now.*
* *In Java, a top level class is a class that is not defined inside another class. They can only have a public or package-private access. Applying public access to a class indicates that it can be referenced and used in any class. Applying default access, lack of access modifier indicates that the class can be accessed only be a class within the same package.*
* *An inner class is a class defined inside of another class and is the opposite of a top level class. In addition to public and package-private access, inner classes can also have a protected and private access. A Java file can have many top level classes but at most one public top level class. In fact, if there is no public class at all, there is no requirement that the single public class be the first class in the file.*
* *One benefit of using package-private access is that you ca define as many classes within the same Java file.*

***Accessing the THIS reference:***

* *What happens when a method parameter has the same name as an existing instance variable?*

|  |
| --- |
| *public class Flamingo{*  *private String color;*  *public void setColor(String color){*  *color=color;*  *}*  *Public static void main(String… unused){*  *Flamingo f = new Flamingo();*  *f.setColor(‘PINK”);*  *System.out.println(f.color);*  *}}* |

*If you said null, then you would be correct. Java uses the most granular scope so when it sees color= color, it thinks you are assigning the method parameter value to itself.*

* *The assignment completes successfully within the method, but the value of instance variable color is never modified and is null when printed in the main() method.*
* *The fix when you have a local variable with the same name as an instance variable is to use this reference or keyword. The this reference to the current instance of the class and can be used to access any member of the class, including inherited members.*
* *It can be used in any instance method, constructor and instance initializer block. It cannot be used when there is no implicit instance of the class, such as in a static method or static initializer block.*

*public void setColor(String color){*

*this.color = color;*

*}*

* *The corrected code will now print PINK as expected. In many cases, the this reference is optional. If Java encounters a variable or method it cannot find it will check the class hierarchy to see if it is available.*

|  |
| --- |
| *public class Duck{*  *private String color;*  *private int height;*  *private int length;*  *public void setData(int length, int theHeight){*  *length=this.length // Backwards-no good!*  *height= theHeight;*  *this.color=”white”;*  *}*  *public static void main(String args[]){*  *Duck b = new Duck();*  *b.setData(1,2);*  *System.out.print(b.length+ “ “ +b.height + “ “ + b.color); //0 2 white*  *}};* |

*Calling the SUPER Reference:*

* *In Java, a variable or method can be defined in both a parent class and a child class. When this happens, how do we reference the version in the parent class instead of the current class? To achieve this, you can use the super reference or keyword. The super reference is similar to this reference, except that it excludes any members found in the current class. In other words, the member must be accessible via inheritance.*

|  |
| --- |
| *class Mammal{*  *String type=”mammal”;*  *}*  *public class Bat extends Mammal{*  *String type=”bat”;*  *public String getType(){*  *return super.type + “:” + this.type;*  *}*  *public static void main(String… args){*  *System.out.print(new Bat().getType());*  *}}* |

|  |
| --- |
| *class Insect{*  *protected int numberOfLegs = 4;*  *String label =”buggy”;*  *}*  *public class Beetle extends Insect{*  *protected int numberOfLegs= 6;*  *short age=3;*  *public void printData(){*  *System.out.print(this.label);*  *System.out.print(super.label);*  *System.out.print(this.age);*  *System.out.print(super.age);*  *System.out.print(numberOfLegs);*  *}*  *public static void main(String []n){*  *new Beetle().printData();*  *}}* |

*This program wouldn’t compile. Since label is defined in the parent class, it is accessible via both this and super references. On the other hand the variable age is defined only in the current class, making it accessible via this but not super. Remember, while this includes current and inherited members, super only includes inherited members. Even though both numberOfLegs variables are accessible in Beetle, Java checks outward starting with the narrowest scope. In this example, this.numberOfLegs and super.numberOfLegs refer to different variables with distinct values.*

* *Since this includes inherited members, you often only use super when you have a naming conflict via inheritance. For example, you have a method or variable defined in the current class that matches a method or variable in a parent class.*

***Declaring Constructors:***

* *A constructor is a special method that matches the name of the class and has no return type. It is called when a new instance of the class is created.*

|  |
| --- |
| *public class Bunny{*  *public Bunny() {*  *System.out.println(“Constructor”);*  *}* |

*The name of the constructor Bunny matches the name of the class Bunny, and there is no return type not even void. That makes this a constructor.*

*public class Bunny{*

*public bunny() {} //Doesn’t compile*

*public void Bunny() {}*

*}*

* *The first one doesn’t match the class name because Java is case sensitive. Since it doesn’t match, Java knows it cant be a constructor and is supposed to be a regular method. However, it is missing return type and doesn’t compile. The second method is a perfectly good method but it is not a constructor because it has a return type.*
* *Like method parameters, constructor parameters can be a valid class, array or primitive type, including generics but not include var.*

|  |
| --- |
| *class Bonobo{*  *public Bonobo(var food){ //Doesn’t compile*  *}}* |

* *A class can have multiple constructors, so long as each constructor has a unique signature. In this case, that means the constructor parameters must be distinct. Like methods with the same name but different signatures, declaring multiple constructors with different signatures is referred to as Constructor Overloading. The following Turtle class has four distinct overloaded constructors*

|  |
| --- |
| *public class Turtle{*  *private String name;*  *public Turtle(){*  *name=”John Doe”;*  *}*  *public Turtle(int age) {}*  *public Turtle(long age) {}*  *public Turtle(String newName, String… favouriteFoods) {*  *name=newName;*  *}}* |

* *Constructors are used when creating a new object. This process is called instantiation because it creates a new instance of the class. A constructor is called hen we write new followed by the name of the class we want to instantiate it.*

|  |
| --- |
| *new Turtle()* |

*When Java sees the new keyword, it allocates memory for the new object. It then looks for a constructor with a matching signature and calls it.*

***Default Constructor:***

* *Every class in Java has a constructor whether you code one or not. If you don’t include any constructors in the class, Java will create one for you without any parameters. This Java-created constructor is called the default constructor and is added anytime a class is declared without any constructors. We often refer to it as default no-argument constructor for clarity*

|  |
| --- |
| *public class Rabbit{*  *public static void main(String[] args){*  *Rabbit rabbit =new Rabbit(); //calls default constructor*  *}}* |

*In the Rabbit class, Java sees no constructor was coded and creates one. This default constructor is equivalent*

*public Rabbit() {}*

*The default constructor has an empty parameter list and an empty body. Java is happy to generate it for you and save you some typing. This is generated. This happens during the compile step. It is only in the compiled file with the .class extension that it makes an appearance.*

* *Default constructor is only supplied if there are no constructors present.*

|  |
| --- |
| *public class Rabbit1 {}*  *public class Rabbit2 {*  *public Rabbit2() {}*  *}*  *public class Rabbit3{*  *public Rabbit3(Boolean b) {}*  *}*  *public class Rabbit4 {*  *private Rabbit4() {}*  *}* |

* *Only Rabbit1 gets a default no-argument constructor. It doesn’t have a constructor coded, so Java generates a default no-argument constructor. Rabbit2 and Rabbit3 both have public constructors already. Rabbit4 has a private constructor. Since these three classes have a constructor defined, the default no-arg constructor is not inserted for you.*

|  |
| --- |
| *public class RabbitsMultiply{*  *public static void main(String args[]){*  *Rabbit r1 = new Rabbit();*  *Rabbit r2 = new Rabbit2();*  *Rabbit r3 = new Rabbit3(true);*  *Rabbit r4 = new Rabbit4(); //Doesn’t compile* |

* *Line 3 calls the generated default no argument constructor. Line 4 and 5 calls the user provided constructors. Line 6 doesn’t compile. Rabbit4 made the constructor private so that other classes couldn’t call it.*
* *Having only private constructors in a class tells the compiler not to provide a default no argument constructor. It also prevents other classes from instantiating the class. This is useful when a class has only static methods or the developer wants to have full control of all calls to create new instances of the class. Static methods in class including main() method may access private members including private constructors.*

***Calling Overloaded constructors with THIS()***

* *A single class can have multiple constructors. This is referred to as Constructor overloading because all the constructors have the same inherent name but a different signature.*

|  |
| --- |
| *public class Hamster{*  *private String color;*  *private int weight;*  *public Hamster(int weight){*  *this.weight= weight;*  *color=”brown”;*  *}*  *public Hamster(int weight, String color){*  *this.weight = weight;*  *this.color= color;*  *}}* |

* *One of the constructors take a single int parameter. The other takes an int and a String. These parameter lists are different so the constructors are successfully overloaded. There is a bit of duplication, as this.weight is assigned twice in the same way in both the constructors. In programming, even a bit of duplication tends to turn into a lot of duplication as we keep adding just more thing.*
* *Constructors can be called only by writing new before the name of the constructor.*

*public Hamster(int weight){*

*new Hamster(weight,”brown”); //Compiles but incorrect*

*}*

* *This attempt does compile. It doesn’t do what we want though. When this constructor is called it creates a new object with the default weight and color. It then constructs a different object with the desired weight and color and ignores the new object. In this manner we end up with two objects with one being discarded after it is created. That is not what we want. We want weight and color set on the object we are trying to instantiate in the first place.*
* *Java provides a solution – this() – yes, the same keyword we used to refer to instance members. When this() is used with parentheses, Java calls another constructor on the same instance of the class*

*public Hamster(int weight){*

*this(weight, “brown”);*

*}*

*Success! Now java calls the constructor that takes two parameters with weight and color set as expected.*

* *Calling this() has one special rule. If you choose to call it, the this() call must be the first statement in the constructor. The side effect of this is that there can be only one call to this() in any constructor.*

|  |
| --- |
| *public Hamster(int weight){*  *System.out.println(“in constructor”);*  *This(weight,”brown”); //Doesn’t compile* |

* *Even though a print statement on line4 doesn’t change any variables it is still a Java statement and is not allowed to be inserted before the call to this().*
* *There is one last rule for overloaded constructors you should be aware of.*

|  |
| --- |
| *public class Gopher{*  *public Gopher(int dugHoles){*  *this(5); // Doesn’t compile*  *}}* |

* *The compiler is capable of detecting that this constructor is calling itself infinitely. Since this code can never terminate, the compiler stops and reports this an error. Likewise, this doesn’t compile*

|  |
| --- |
| *public class Gopher{*  *public Gopher(){*  *this(5); //Doesn’t compile*  *}*  *public Gopher(int dugHoles){*  *this(); //Doesn’t compile*  *}}* |

*In this example, the constructors call each other, and the process continues infinitely. Since the compiler can detect this, it reports this as an error.*

***THIS vs THIS()***

* *Despite using the same keyword, this and this() are very different. The first, this, refers to an instance of the class, while the second, this() refers to a constructor call within the class.*

***Calling Parent constructors with SUPER():***

* *In Java, the first statement of every constructor is either a call to another constructor within the class, using this() or a call to constructor in the direct parent class, using super(). If a parent constructor takes arguments, then the super() call also takes arguments.*

|  |
| --- |
| *public class Animal{*  *private int age;*  *public Animal(int age){*  *super(); //refers to constructor in java.lang.Object*  *this.age=age;*  *}}*  *public class Zebra extends Animal{*  *public Zebra(int age){*  *super(age); //Refers to the constructor in Animal*  *}*  *public Zebra() {*  *this(4); //Refers to constructor in Zebra with int argument*  *}}* |

* *In this first class, Animal, the first statement of the constructor is a call to the parent constructor defined in java.lang.Object which takes no arguments. In the second class, Zebra, the first statement of the first constructor is a call to Animal’s constructor which takes a single argument. This class Zebra also includes a second no-argument constructor that doesn’t call super() but instead calls the other constructor within the Zebra class using this(4).*
* *Like calling this(), calling super() can only be used as the first statement of the constructor. for example, the following two class definitions will not compile.*

|  |
| --- |
| *public class Zoo{*  *public Zoo(){*  *System.out.println(“Zoo Created”);*  *Super();*  *}}*  *public class Zoo{*  *public Zoo(){*  *super();*  *System.out.println(“Zoo created”);*  *Super();*  *}}* |

* *The first class will not compile because the call to the parent constructor must be the first statement of the constructor. In the second code snippet, super() is the first statement of the constructor, but it is also used as the third statement. Since super() can only be called once as the first statement of the constructor, the code will not compile.*
* *If the parent class has more than one constructor, the child class may use any valid parent constructor in its definition*

|  |
| --- |
| *public class Animal{*  *private int age;*  *private String name;*  *public Animal(int age, String name){*  *super();*  *this.age = age;*  *this.name = null;*  *}}*  *public class Gorilla extends Animal{*  *public Gorilla(int age){*  *super(age, “Gorilla”);*  *}*  *public Gorilla(){*  *super(5);*  *}}* |

*In this example, the first child constructor takes one argument, age and calls the parent constructor, which takes two arguments, age and name.*

*The second child constructor takes no arguments, and it calls the parent constructor, which takes one argument age.*

*In this example, notice that the child constructors are not required to call matching parent constructors. Any valid parent constructor is acceptable as long as the appropriate input parameters to the parent constructor are provided.*

***SUPER vs SUPER()***

* *Like this and this(), super and super() are unrelated in Java. The first, super, is used to reference members of the parent class, while the second, super() calls a parent constructor.*

***Understanding Compiler Enhancements:***

* *First line of every constructor is a call to either this() or super() but we have been creating classes and constructors . The answer is that the Java compiler automatically inserts a call to the no-argument constructor super() if you do not explicitly call this() or super() as the first line of a constructor.*
* *For example, the following three class and constructor definitions are equivalent, because the compiler will automatically convert them all*

|  |
| --- |
| *public class Donkey {}*  *public class Donkey {*  *public Donkey() {}*  *}*  *public class Donkey {*  *public Donkey(){*  *super();*  *}}* |

*Make sure you understand the differences between these three Donkey class definitions and why? Java will automatically convert them all to the last definition.*

***Are classes with only Private constructors considered final?***

* *Remember, a final class cannot be extended. What happens if you have a class that isn’t marked final but only contains private constructors can you extend the class?*

*The answer is “yes” but only an inner class defined in the class itself can extend it. An inner class is the only one that would have access to private constructor and be able to call super(). Other top level classes cannot extend such a class.*

***Missing a Default No-Argument Constructor:***

* *The default no-argument constructor is not required and is inserted by compiler only if there is no constructor defined in the class.*

*public class Mammal{*

*public Mammal(int age) {}*

*}*

*public class Elephant extends Mammal{ } //Doesn’t compile*

* *Since Elephant doesn’t define any constructors, the Java compiler will attempt to insert a default no-argument constructor. As a second compiler-time enhancement, it will also auto-insert a call to super() as the first line of the default no-argument constructor. One previous Elephant declaration is then converted by the compiler to the following declaration:*

*public class Elephant extends Mammal{*

*public Elephant() {*

*super(); //Doesn’t compile*

*}}*

* *Since the Mammal class has atleast one constructor declared, the compiler doesn’t insert a default no-argument constructor. Therefore, the super() call in the Elephant class declaration doesn’t compile. In this case, the Java compiler will not help, and you must create atleast one constructor in your child class that explicitly calls a parent constructor via the super() command. We can fix this by adding a call to a parent constructor that takes a fixed argument.*

*public class Elephant extends Mammal{*

*public Elephant() {*

*super(10);*

*}}*

* *This code will compile because we added a constructor with an explicit call to a parent constructor. Notice that the class Elephant now has a no argument constructor even though its parent class Mammal doesn’t.*
* *Subclasses may define explicit no-argument constructors even if their parent class do not, provided the constructor of the child maps to a parent constructor via an explicit call of the super() command. This means that subclasses of the Elephant can rely on compiler enhancements.*

*public class AfricanElephant extends Elephant {}*

***SUPER() always refer to the most direct parent:***

* *A class may have multiple ancestors via inheritance. AfricanElephant is a subclass of Elephant, which in turn is a subclass of Mammal. For constructors, through super() always refers to the most direct parent.*

***Constructors and Final Fields:***

* *Final static variables must be assigned a value only once. Instance variables marked final follow the same rules. They can be assigned values in the line in which they are declared or in an instance initializer.*

*public class MouseHouse{*

*private final int volume;*

*private static String name = “The Mouse House”;*

*{*

*Volume=10;*

*}*

*}*

* *Like final variables, once the value is assigned, it cant be changed. There is one more place they can assign a value- constructor. The constructor is part of the initialization process. So it is allowed to assign final instance variables in it. By the time the constructor completes, all final variables must be assigned a value.*

*public class MouseHouse{*

*private final int volume;*

*private final String type;*

*public MouseHouse() {*

*this.volume=10;*

*type=”happy”;*

*}}*

* *In our MouseHouse implementation, the values for volume and type are assigned in the constructor.*
* *Remember that this keyword is optional since the instance variables are part of the class declaration and there are no constructor parameters with the same name.*
* *Unlike local final variables, which are not required to have a value unless they are actually used, final instance variables must be assigned a value. Default values are not used for these variables. If they are not assigned a value in the line where they are declared or in an instance initializer, they must be assigned a value in constructor declaration. Failure to do so will result in a compiler error on that line declares the constructor.*

|  |
| --- |
| *public class MouseHouse{*  *private final int volume;*  *private final String type;*  *{*  *this.volume=10;*  *}*  *public MouseHouse(String type){*  *this.type=type;*  *}*  *public MouseHouse(){ //Doesn’t compile*  *this.volume = 2; //Doesn’t compile*  *}}* |

*In this example, the first constructor that takes a String argument compiles. Although a final instance variable can be assigned a value only once, each constructor can be assigned a value only once, each constructor is considered independently in terms of assignment. The second constructor doesn’t compile for two reasons. First, the constructor fails to set a value for the type variable. The compiler detects that a value is never set for type and reports an error on the line where volume is set.*

* *Be wary of any instance variables marked final. Make sur they are assigned a value on the same line declared, in an instance initializer or in a constructor. They should be assigned a value only once and failure to assign a value is considered a compiler error in the constructor.*
* *What about final instance variables when a constructor calls another constructor in the same class? In that case you have to follow the constructor logic pathway carefully, making sure every final instance variable is assigned a value exactly once.*
* *We can replace our previous bad constructor with the following one that does compile*

*public MouseHouse() {*

*this(null);*

*}*

*This constructor doesn’t perform any assignments to any final instance variables but it calls the MouseHouse(String) constructor which we observed compiles without any issue. We use null here to demonstrate that the variable doesn’t need an object value. We can assign a null value to final instance variables, as long as they are explicitly set.*

***Class Initialization***

* *First, you need to initialize the class, which involves invoking all the static members in the class hierarchy, starting with the highest superclass and working downward.*
* *This is often referred to as loading of the class. The JVM controls when the class is initialized, although you can assume your class is loaded before it is used. The class may be instantiated when the program first starts, when a static member of the class is referenced, or shortly before an instance of the class is created.*
* *The most important rule with class initialization is that it happens at most once for each class. The class may also never be loaded if it is not used in the program.*

***Initialize class X:***

* *If there is a superclass Y of X, then initialize class Y first.*
* *Process all static variable declarations in the order they appear in the class.*
* *Process all static initializers in the order they appear in the class.*

|  |
| --- |
| *public class Animal{*  *static{*  *System.out.print(“A”);*  *}*  *public class Hippo extends Animal{*  *static{*  *System.out.print(“B”);*  *}*  *public static void main(String[] args){*  *System.out.print(“C”);*  *new Hippo(); new Hippo(); new Hippo();* |

*It prints ABC exactly once. Sinc ethe main() method is inside the Hippo class, the class will be initialized first, starting with the superclass and printing AB. Afterward, the main() method is executing, printing c. Even though the main() method creates three instances, the class is loaded only once.*

*Why did the Hippo program prints “C” after AB*

* *In the previous example, the Hippo class was initialized before the main() method was executed. This happened because our main() method was inside the class being executed, so it had to be loaded on startup.*

*What if you called Hippo inside another program?*

*public class HippoFriend{*

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

*System.out.print(“C”);*

*new Hippo();*

*}}*

* *Assuming that the class isn’t referenced anywhere else, this program will likely print CAB, with the Hippo class not being loaded until it is needed inside the main() method. We say likely, because the rules for when classes are loaded are determined by the Java at runtime.*

***Instance Initialization:***

* *An instance is initialized anytime the new keyword is used. Instance Initialization is a bit more complicated than class initialization, because a class or superclass may have many constructors declared but only a handful used as part of instance initialization.*
* *First, start at the lowest level constructor where the new keyword is used. The first line of every constructor is a call to this() or super() and if omitted, the compiler will automatically insert a call to the parent no-argument constructor super().*
* *Then progress upward and note the order of constructors. Finally, initialize each class starting with the superclass, processing each instance initializer and constructor in reverse order in which it is called.*

***Instance Initializer of X:***

* *If there is a superclass Y of X, then initialize the instance of Y first.*
* *Process all the instance variable declarations in the order they appear in the class.*
* *Process all the instance initializers in the order they appear in the class.*
* *Initialize the constructor including any overloaded constructors referenced with this().*

|  |
| --- |
| *public class ZooTickets{*  *private String name=”BestZoo”;*  *{ System.out.println(name+”-“); }*  *private static int COUNT =0;*  *static{ System.out.print(COUNT+”-“);}*  *static{COUNT +=10; System.out.print(COUNT+”-“);}*  *public ZooTickets() {*  *System.out.print(“z-“);*  *}*  *public static void main(String… patrons){*  *new ZooTickets();*  *}}* |

*The output is as follows- 0-10-BestZoo-z-*

*First, we have to initialize the class. Since there is no superclass declared, which means the superclass is Object, we can start with the static components of ZooTickets. In this case, lines 4,5,6 are executed, printing 0- and 10-. Next, we initialize the instance. Again, since there is no superclass declared, we start with the instance components. Lines 2 and 3 are executed, which prints BestZoo-. Finally, we run the constructor on lines 8-10, which outputs z-.*

|  |
| --- |
| *class Primate{*  *public Primate() {*  *System.out.print(“Primate-“);*  *}}*  *Class Ape extends Primate{*  *public Ape(int fur){*  *System.out.print(“Ape1-“);*  *}*  *public Ape() {*  *System.out.print(“Ape2-“);*  *}}*  *public class Chimpanzee extends Ape{*  *public Chimpanzee(){*  *super(2);*  *System.out.print(“Chimpanzee-“);*  *}*  *public static void main(String args[]){*  *new Chimpanzee();*  *}}* |

*The compiler inserts the super() command as the first statement of both Primate and Ape constructors. The code will execute with parent constructors called first and yields the following output:*

*Primate-Ape1-Chimpanzee*

*Notice that only one of the two Ape() constructors is called. You need to start with the call to new Chimpanzee() to determine which constructors will be executed. Constructors are executed from the bottom-up, but since the first line of every constructor is a call to another constructor, the flow actually ends up with the parent constructor executed before the child constructor.*

|  |
| --- |
| *public class Cuttlefish{*  *private String name=”Swimmy”;*  *{System.out.println(name);}*  *private static int COUNT = 0;*  *static {System.out.println(COUNT); }*  *{ COUNT++; System.out.println(COUNT); }*  *public Cuttlefish() {*  *System.out.println(“Constructor”);*  *}*  *public static void main(String[] args){*  *System.out.println(“Ready”);*  *new Cuttlefish();*  *}}* |

*0*

*Ready*

*Swimmy*

*1*

*Constructor*

*There is no superclass declared, so we can skip any steps that relate to inheritance. We first process the static variables and static initializers lines 4 and 5 with line 5 printing 0. Now that the static initializers are out of the way main() method can run, which prints Ready . Lines 2, 3 and 6 are processed, with line 3 printing swimmy and line 6 printing 1.*

*Finally the constructor is run on lines 8-10 which prints constructor.*

|  |
| --- |
| *Class GiraffeFamily{*  *static { System.out.print(“A”); }*  *{System.out.print(“B”);}*  *public GiraffeFamily(String name){*  *this(1);*  *System.out.print(“C”);*  *}*  *public GiraffeFamily(){*  *System.out.print(“D”);*  *}*  *public GiraffeFamily(int stripes){*  *System.out.print(“E”);*  *}}*  *public class Okapi extends GiraffeFamily{*  *static{ System.out.print(“F”); }*  *public Okapi(int stripes){*  *super(“sugar”);*  *System.out.print(“G”);*  *}*  *{ System.out.print(“H”) ; }*  *public static void main(String[] args){*  *new Okapi(1);*  *System.out.println();*  *new Okapi(2);*  *}}* |

*The program prints the following:*

*AFBECHG*

*BECHG*

* *Start with initializing the Okapi class. Since it has a superclass GiraffeFamily, initialize it first printing A on line 2. Next, initialize the Okapi class, printing F on line 19. After the classes are initialized, execute the main() method on line 27. The first line of the main() method creates a new Okapi Object, triggering the instance initialization process.*
* *Per the first rule, the super class instance of GiraffeFamily is initialized first. Per our third rule, the instance initializer in the superclass GiraffeeFamily is called, and B is printed on line3. Per our fourth rule, we initialize the constructors. In this case, this involves calling the constructor on line5 which in turn calls the overloaded constructor on line 14. The result is that EC is printed, as the constructor bodies are unwound in the reverse order that they were called.*
* *The process then continues with the initialization of the Okapi instance itself. Per the third and fourth rules, H is printed on line 25 and G is printed on line 23 respectively. The process is a lot simpler when you don’t have to call any overloaded constructors.*
* *Line 29 then inserts a line break in the output. Finally, line 30 initializes a new Okapi object. The order and initialization are the same as line 28, the class initialization, so BECHG is printed again. Notice that D is never printed as only two of the three constructors in the superclass GiraffeFamily is added.*

***Reviewing Constructor Rules:***

* *The first statement of every constructor is a call to an overloaded constructor via this() or a direct parent constructor via super().*
* *If the first statement of a constructor is not to a call to this() or super(), then the compiler will insert a no default argument constructor super() as the first statement of the constructor.*
* *Calling this() and super() after the first statement of a constructor results in a compile time error.*
* *If the parent doesn’t have a no argument constructor, then every constructor in the child class must start with an explicit this() or super() constructor class.*
* *If the parent class doesn’t have a no argument constructor and the child doesn’t define any constructors then the child class will not compile.*
* *If a class only defines private constructors then it cannot be extended by a top-level class.*
* *All final instance variables must be assigned a value exactly once by the end of the constructor. Any final instance variables not assigned a value will be reported as a compiler error on the line the constructor is declared.*

***Inheriting Members:***

* *One of the biggest strengths is leveraging its inheritance model to simplify code.*
* *Java classes may use any public or protected member of the parent class including methods, primitives or object references. If the parent class and child class are part of the same package then the child class may use any package-private members defined in the parent class. Finally a child class may never access a private member of the parent class, at least not through any direct reference. To reference a member in a parent class you can just call it directly*

*class Fish{*

*protected int size;*

*private int age;*

*public Fish(int age){*

*this.age = age;*

*}*

*public int getAge(){*

*return age;*

*}}*

*public class Shark extends Fish{*

*private int numberOfFins=8;*

*public Shark(int age){*

*super(age);*

*this.size=4;*

*}*

*public void displaySharkDetails(){*

*System.out.print(“Shark with age:” +getAge());*

*System.out.print(" and "+size+" meters long");*

*System.out.print(" with "+numberOfFins+" fins");*

*}}*

* *In the child class, we use the public method getAge() and protected member size to access values in the parent class. You can use this to access visible members of the current or a parent class., and you can use super to access visible members of a parent class.*

*public void displaySharkDetails() {*

*System.out.print("Shark with age: "+super.getAge());*

*System.out.print(" and "+super.size+" meters long");*

*System.out.print(" with "+this.numberOfFins+"*

*fins");*

*}*

* *In this example, getAge() and size can be accessed with this or super since they are defined in the parent class while numberOfFins can only be accessed with this and not super since it is not inherited property.*

***Inheriting Methods:***

* *Inheriting a class not only grants access to inherited methods in the parent class but also sets the stage for collisions between methods defined in both the parent class and the subclass.*

*Overriding a Method:*

* *What if there is a method defined in both parent and child classes with the same signature? For example, you may want to define a new version of the method and have it behave differently for that subclass. The solution is to override the method in the child class. In Java, overriding a method occurs when a subclass declares a new implementation for an inherited method with the same signature and compatible return type. Remember that a method signature includes the name of the method and method parameters.*
* *When you override the method you may reference the parent version of the method using the super keyword. In this manner, the keywords this and super allow you to select between the current and previous versions of a method, respectively.*

*public class Canine{*

*public double getAverageWeight(){*

*return 50;*

*}*

*public class Wolf extends Canine{*

*public double getAverageWeight(){*

*return super.getAverageWeight() + 20;*

*}*

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

*System.out.println(new Canine().getAverageWeight()); //50.0*

*System.out.println(new Wolf().getAverageWeight()); //70.0*

*}}*

***Method Overriding and Recursive calls:***

* *If you remove super keyword in the above code, the compiler wouldn’t call the current wolf method since it would think you were executing a recursive method call. A recursive method is one that calls itself as part of execution. It is common in programming but must have a termination condition that triggers the end of the recursion at some point in depth. In the above program there is no end so program would attempt to call the code infinitely and produce a StackoverflowError at runtime.*
* *To override a method, you must follow the number of rules. The compiler performs the following checks when you override a method:*
* *The method in the child class must have the same signature as the method in the parent class*
* *The method in the child class must be at least as accessible as the method in the parent class.*
* *The method in the child class may not declare a checked exception that is new or broader than the class of any exception declared in the parent class method.*
* *If the method returns a value, it must be the same or a subtype of the method in the parent class, known as covariant return types*

*Defining Subtype and Supertype:*

* *When discussing inheritance and polymorphism we often use the word subtype rather than subclass since Java includes interfaces. A subtype is the relationship between two types where one type inherits the other. If we define X to be a subtype of Y, then one of the following is true:*
* *X and Y are classes, and X is a subclass of Y*
* *X and Y are interfaces and X is a subinterface of Y*
* *X is a class and Y is an interface and X implements Y(either directly or through an inherited class)*
* *Likewise, a supertype s the reciprocal relationship between two types where one type is the ancestor of the other. A subclass is a subtype but not all subtypes are subclasses/*
* *The first rule of overriding a method is somewhat self explanatory. If two methods have the same name but different signatures the methods are overloaded not overridden. Overloaded methods are considered independent and donot share the same polymorphic properties as overridden methods.*

*Overloading vs Overriding:*

* *Overloading and overriding a method are similar in that they both involve redefining a method using the same name. They differ in that an overloaded method will use a different list of method parameters. This distinction allows overloaded methods a great deal more freedom in syntax than an overridden method would have.*
* *For example, compare the overloaded fly() with the overridden eat() in the Eagle class.*

|  |
| --- |
| *public class Bird{*  *public void fly(){*  *System.out.println(“Bird is flying”);*  *}*  *public void eat(int food){*  *System.out.println(“Bird is eating “+food+” units of food”);*  *}}*  *public class Eagle extends Bird{*  *public int fly(int height){*  *System.out.println(“Bird is flying at”+height+” metres”);*  *return height;*  *}*  *public int eat(int food){*  *System.out.println(“Bird is eating “+food+” units of food”);*  *return food;*  *}}* |

*The fly() method is overloaded in the subclass Eagle, since the signature changes from no-argument method with one int argument. Because the method is being overloaded and not overridden, the return type can be changed from void to int.*

* *The eat() method is overridden in the subclass Eagle, since the signature is the same as it is in parent class Bird they both take a single argument int. Because the method is being overridden, the return type of the method in Eagle class must be compatible with the return type for the method in Bird class.*
* *In this example, the return type int is not a subtype of void; therefore the compiler will throw an exception on this method definition.*
* *Anytime you see a method with same name as a method in the parent class determine the method is being overloaded or overridden first.*

|  |
| --- |
| *public class Camel{*  *public int getNumberOfHumps(){*  *return 1;*  *}*  *public class BatrianCamel extends Camel{*  *private int getNumberOfHumps(){ //Doesn’t compile*  *return 2;*  *}*  *public class Rider{*  *public static void main(String args[]){*  *Camel c = new BatrianCamel();*  *System.out.println(c.getNumberOfHumps());*  *}}* |

*In this example, BatrianCamel attempts to override the getNumberOfHumps() method defined in the parent class but fails because the access modifier private is more restrictive than the one defined in the parent version of the method. The reference type for the object is Camel, where the method is declared public, but the object is actually an instance of type BatricanCamel which is declared as private.*

* *Java avoids these types of ambiguity problems by limiting overriding a method to access modifiers that are as accessible than the version in the inherited method.*
* *The third rule says that overriding a method cannot declare new checked exceptions or checked exceptions broader than the inherited method. This is done by similar polymorphic reasons as limiting access modifiers.*
* *In other words, you could end up with an object that is more restrictive than the reference type it is assigned to, resulting in checked exception that isn’t handled or declared. If a broader checked exception is declared in the overriding method, the code will not compile.*
* *public class Reptile {*

*protected void sleepInShell() throws IOException{}*

*protected void hideInShell() throws NumberFormatException {}*

*protected void exitShell() throws FileNotFoundException {}*

*}*

*public class GalapagoseTortise extends Reptile{*

*public void sleepInShell() throws FileNotFoundException{}*

*public void hideInShell() throws IllegalArgumentException{}*

*public void exitShell() throws IOException{} //Doesn’t compile*

*In this example, we have three overridden methods. These overridden methods use the more accessible public modifier, which is allowed per our second rule over overridden methods. The overridden sleepInShell()method declares FileNotFoundException which is a subclass of exception declared in the inherited method, IOException. Per our third rule of overridden methods, this is a successful override since the exception is narrower in the overridden methods.*

* *The overridden hideInShell() method declares an IllegalArgumentException, which is a superclass of the exception declared in the inherited method, NumberFormatException. While this seems to be an invalid override since the overridden method uses a broader exception both of these exceptions are unchecked, so the third rule doesn’t apply.*
* *The third overridden exitShell() method declares IOException, which is the superclass f the exception declared in the inherited method, FileNotFoundException. Since these re checked exceptions and IOException is broader, the overridden exitShell() method doesn’t compile in the GalapagosTortoise class.*
* *The fourth and final rule around overriding a method is probably the most complicated, as it requires knowing the relationships between the return types. The overriding method must use a return type that a covariant with the return type of the inherited method.*
* *public class Rhino {*

*protected CharSequence getName(){*

*return “Rhino”;*

*}*

*protected String getColor(){*

*return “grey, black or white”;*

*}*

*class JavanRhino extends Rhino{*

*public String getName(){*

*return “javan rhino”;*

*}*

*public CharSequence getColor(){*

*return “grey”;*

*}}*

* *The subclass JavanRhino attempts to override two methods from Rhino:getName() and getColor(). Both overridden methods have the same name and signature as the inherited methods. The overridden methods also have a broader access modifier, public, than the inherited methods. Per the second rule, a broader access modifier s acceptable.*
* *String implements the CharSequence interface, making String a subtype of CharSequence. Therefore the return type of the getName() in JavanRhino is covariant with the return type of getName()in Rhino.*
* *On the other hand, the overridden getColor() method doesn’t compile because CharSequence is not a subtype of String. To put in other way, all String values are CharSequence values, but not all charSequence values are String values.*
* *For example, a StringBuilder is a CharSequence but not a String. you need to know if the return type of the overriding method is the same or subtype of the return type of the inherited method.*

*Simple test for Covariance:*

* *Given an inherited return type A and an overriding return type B, can you assign an instance of B to a reference variable for A without a cast ? if so then the are covariant. This rule applies to all the primitive types and object types alike. If one of the return types is void, then both must be void, as nothing is covariant with void except itself.*
* *The last three rules of overriding a method may seem arbitrary or confusing but they are needed for consistency.*

*Overriding a Generic method:*

* *Overriding methods is complicated enough, but add generics to it and things only get challenging. You cannot overload methods by changing the generic type due to type erasure. To review, only one of the two methods is allowed in a class because type erasure will reduce both set of arguments to List Input.*

|  |
| --- |
| *public class LongTailAnimal{*  *protected void chew(List<Object> input) {}*  *protected void chew(List<Double> input) {} //Doesn’t compile* |

*For the same reason you also cant overload a generic method in parent class*

*public class LongTailAnimal{*

*protected void chew(List<Object> input){} //Doesn’t compile*

*}*

*public class Anteater extends LongTailAnimal{*

*protected void chew(List<Double> input) {} // Doesn’t compile*

*}*

*Both of these examples fail to compile because of type erasure. In the compiled form, the generic type is dropped and it appears as an invalid overloaded method.*

*Generic Method Parameters:*

* *You can override a method with generic parameters, but you must watch the signature including the generic type exactly. For example, this version of the AntEater class does compile because it uses the same generic type in the overridden method as the one defined in the parent class:*

*public class LongTailAnimal{*

*protected void chew(List<String> input) {}*

*}*

*public class AntEater extends LongTailAnimal{*

*protected void chew(ArrayList<Double> input){}*

*}*

*Yes these classes do compile. However they are considered overloaded methods not overridden methods. Because the signature isn’t same. Type Erasure doesn’t change the fact that the one of the method arguments is a List and the other is ArrayList.*

*Generics and Wildcards:*

* *Java includes support for generic wildcards using the question mark(?) character. It even supports bounded wildcards.*

*void sing1(List<?> v){} //unbounded wildcard*

*void sing2(List<? super String> v){} //lower bounded wildcard*

*void sing3(List<? extends String> v){} //upper bounded wildcard*

* *Using generics with wildcards, overloaded methods and overridden methods can get quite complicated.*

***Generic Return Types:***

* *When you are working with overridden methods that return generics, the return values must be covariant. In terms of generics, this means that the return type of the class or interface declared in the overriding method must be a subtype of the class defined in the parent class. The generic parameter type must match its parents type exactly.*
* *Given the following declaration of the Mammal class , which of the two subclasses Monkey and Goat compile?*

*public class Mammal{*

*public List<CharSequence> play() {….}*

*public CharSequence sleep() {…}*

*}*

*public class Monkey extends Mammal{*

*public ArrayList<CharSequence> play() {…}*

*}*

*public class Goat extends Mammal{*

*public List<String> play() {…} //Doesn’t compile*

*public String sleep() {….}*

*}*

* *The Monkey class compiles because ArrayList is a subtype of List. The play() method n the Goat class doesn’t compile, though. For the return types to be covariant, the generic type parameter must match.*
* *Even though the String is a subtype of CharSequence it doesn’t exactly match the generic type defined in the Mammal class. Therefore, it is considered to be an invalid override.*
* *Notice that the sleep() method in the Goat class does compile since String is a subtype of CharSequence. This example shows that covariance applies to the return type, just not the generic parameter type.*
* *It might be helpful for you to apply type erasure to questions involving generics to ensure that they compile properly. Once you have determined which methods are overridden and which are being overloaded work backward making sure that the generic types match for the overridden methods. Generic methods cannot be overloaded by changing the generic parameter type only.*

*Redeclaring Private Methods:*

* *In Java, you can’t override private methods since they are not inherited. Just because a child class doesn’t have access to the parent method doesn’t mean that child class can’t define its own version of the method. It just means new method is not an overridden version of the parent class’s method.*
* *Java permits you to redeclare a new method in the child class with the same or modified signature as the method in the parent class. This method in the child class is a separate and independent method, unrelated to the parent version’s method so none of the rules for overriding methods is invoked.*

*public class Camel{*

*private String getNumberOfHumps(){*

*return “Undefined”;*

*}}*

*public class DromedaryCamel extends Camel{*

*private int getNumberOfHumps(){*

*return 1;*

*}}*

*This code compiles without issue. Notice that the return type differs in the child method from String to int. In this example, the method getNumberofHumps() in the parent class is redeclared, so the method in the child class is a new method and not an override of the method in the parent class. If the method in the parent class were public or protected, the method in the child class wouldn’t compile because it would violate two rules of overriding methods.*

*Hiding static Methods:*

* *A hidden method occurs when a child class defines a static method with the same name and signature as an inherited static method defined in a parent class.*
* *Method hiding is similar but not exactly the same as method overriding. The previous four rules for overriding a method must be followed when a method is hidden.*
* *The method defined in the child class must be marked as static if it is marked as static in a parent class.*
* *If it is a method hiding if the two methods are marked static, and method overriding f they are not marked static. If one is marked static and the other is not, the class will not compile.*

*public class Bear{*

*public static void eat() {*

*System.out.println(“Bear is eating”);*

*}}*

*public class Panda extends Bear{*

*public static void eat(){*

*System.out.println(“Panda is chewing….”);*

*}*

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

*eat();*

*}}*

* *In this example, the code compiles and runs. The eat() method in the Panda class hides the eat() method in the Bear class printing “Panda is chewing” at runtime. Because they are both marked as static, this is not considered as an overridden method.*

*public class Bear{*

*public static void sneeze(){*

*System.out.println(“Bear is sneezing…”);*

*}*

*public void hibernate() {*

*System.out.println(“Bear is hibernating…”);*

*}*

*public static void laugh() {*

*System.out.println(“Bear is laughing…”);*

*}}*

*public class Panda extends Bear{*

*public void sneeze() {*

*System.out.println(“Panda sneezes quietly..”);*

*}*

*public static void hibernate() {*

*System.out.println(“Panda is going to sleep..”);*

*}*

*protected static void laugh() {*

*System.out.println(“Panda is laughing…”);*

*}}*

*In this example, sneeze() is marked static in the parent class but not in the child class. The compiler detects that you are trying to override using an instance method. However sneeze() is a static method that should be hidden, causing the compiler to generate an error. In the second method, hibernate() is an instance member in the parent class but a static method in the child class. In this scenario, the compiler thinks that you are trying to hide a static method. Because hibernate() is an instance method that should be overridden, the compiler generates an error. Finally, the laugh method doesn’t compile. Even though both the versions of the method are marked static, the version in Panda has a more restrictive access modifier than the one it inherits and it breaks the second rule for overriding methods. The four rules for overriding methods must be followed when hiding static methods.*

***Creating final methods:***

* *Final methods cannot be replaced. By marking a method final, you forbid a child class from replacing this method. This rule is in place both when you override a method and when you hide a method. In other words, you cannot hide a static method in a child class if It is marked final in the parent class.*

|  |
| --- |
| *public class Bird{*  *public final boolean hasFeathers(){*  *return true;*  *}*  *public final static void flyaway(){}*  *}*  *public class Penguin extends Bird{*  *public final boolean hasFeathers(){ //Doesn’t compile*  *return false;*  *}*  *public final static void flyaway(){}//Doesn’t compile*  *}* |

*In this example, the instance method hasFeathers() is marked as final in the parent class Bird, so the child class Penguin cannot override the parent method, resulting in a compiler error. The static method flyaway() is also marked final, so it cannot be hidden in the subclass. In this example, whether or not the child method used the final keyword is irrelevant the code will not compile either way.*

* *This rule applies only to inherited methods. For example, if the two methods are marked private in the parent Bird class then the Penguin class as defined would compile. In that case, the private methods would be redeclared not overridden or hidden.*

*Why mark a method as final?*

* *Although marking methods as final prevents them from being overridden, it does have advantages in practice. For example you would mark a method as final where you are defining a parent class and want to guarantee certain behavior of a method in the parent class regardless of which child is invoking the method.*
* *In this previous example, with Bird, the author of the parent class may want to ensure the method hasFeathers() always returns true, regardless of the child class instance on which it is invoked.*
* *The reason methods are not commonly marked as final in practice, though, is that it may be difficult for the author of a parent class method to consider all of the possible ways her child class may be used. Although all adult birds have feathers a baby chick doesn’t therefore, if you have an instance of a Bird that a chick it would not have feathers. For this reason, the final modifier is often used when the author of the parent class wants to guarantee certain behavior at the cost of limiting polymorphism.*

*Hiding Variables:*

* *Method overriding has many rules when two methods have the same signature and are defined in both the parent and child classes. The rules for variables with the same name in the parent and child classes are a lot simpler.*
* *In fact, Java doesn’t allow variables to be overridden. Variables can be hidden though. A hidden variable occurs when a child class defines a variable with the same name as an inherited variable defined in the parent class.*
* *This creates two distinct copies of the variable within an instance of the child class one instance defined in the parent class and one defined in the child class.*
* *As when hiding a static method, you cant override a variable, you can only hide it. Lets take a look at a hidden variable*

|  |
| --- |
| *class Carnivore{*  *protected boolean hasFur= false;*  *}*  *public class Meerkat extends Carnivore{*  *protected boolean hasFur = true;*  *public static void main(String[] args){*  *Meerkat m = new Meerkat();*  *Carnivore c=m;*  *System.out.println(m.hasFur);*  *System.out.println(c.hasFur);*  *}}* |

*It prints true followed by false. Both of these classes define a hasFur variable, but with different values. Even though there is only one object created by the main() method, both variables exist independently of each other. The output changes depending on the reference variables used.*

* *Overriding a method replaces the parent method on all reference variables other than super whereas hiding a method or variable replaces the member only if a child reference type is used.*

***Understanding Polymorphism:***

* *Java supports polymorphism, the property of an object to take on different forms. To put this precisely, a Java object may be accessed using a reference with the same type as the object, a reference that is a superclass of the object, or a reference that defines an interface the object implements either directly or through a superclass. Furthermore, a cast is not required if the object is being reassigned to a supertype or interface of the object.*
* *An interface can define abstract methods*
* *A class can implement any number of interfaces.*
* *A class implements an interface by overriding the inherited abstract methods.*
* *An object that implements an interface can be assigned to a reference for that an interface.*

*public class Primate{*

*public boolean hasHair() {*

*return true;*

*}}*

*public interface HasTail{*

*public abstract boolean isTailStripped();*

*}*

*public class Lemur extends Primate implements HasTail{*

*public boolean isTailStriped() {*

*return false;*

*}*

*public int age = 10;*

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

*Lemur lemur = new Lemur();*

*System.out.println(lemur.age);*

*HasTail hasTail = lemur;*

*System.out.println(hasTail.isTailStriped());*

*Primate primate = lemur;*

*System.out.println(primate.hasHair());*

*}}*

*This code compiles and prints the following output:*

*10*

*False*

*True*

*The most important thing to note about this example is that only one object, Lemur is created and referenced.*

*Polymorphism enables an instance of Lemur to be reassigned or passed to a method using one of its supertypes such as Primate or HasTail.*

*Once the object has been assigned to a new reference type, only the methods and variables available at that reference type are called on the object without an explicit cast. For example, the following snippets wouldn’t compile:*

|  |
| --- |
| *HasTail hasTail = lemur;*  *System.out.println(hasTail.age); //Doesn’t compile*  *Primate primate = lemur;*  *System.out.println(primate.isTailStriped()); //doesn’t compile* |

*In this example, the reference hasTail has direct access only to methods defined with the HasTail interface, therefore it doesn’t know the variable age is part of the object. Likewise, the reference primate has access only to methods defined in the Primate class and it doesn’t have direct access to the isTailStripped() method.*

*Object vs Reference:*

* *In Java, all objects are accessed by a reference so a developer never have direct access to the object itself. Conceptually though, you should consider the object as the entity that exists in memory allocated by the Java runtime environment.*
* *Regardless of the type of the reference you have for the object in memory, the object itself doesn’t change. For example, since all the objects inherit java.lang.Object they can all be reassigned to java.lang.Object*

*Lemur lemur = new Lemur();*

*Object lemurAsObject = lemur;*

* *Even though the lemur object has been assigned to a reference with a different type, the object itself has not changed and still exists as a lemur object in memory.*
* *What has changed, then is our ability to access methods within Lemur class with the lemurAsObject reference.*
* *To summarize this principle with the following two rules:*

*The type of the object determines which properties exist within the object in memory.*

*The type of the reference to the object determines which methods and variables are accessible to the Java program.*

*It therefore follows that successfully changing a reference of an object to a new reference type may give you access to new properties of the Object, those properties existed before the reference change occurred.*

*Casting Object:*

* *In the previous example, we created a single instance of a Lemur object and accessed it via superclass and interface references. Once we changed the reference type, though, we lost access to more specific members defined in the subclass that still exist within the object. We can reclaim those references by casting the object back to the specific subclass it came from:*

*Primate primate = new Lemur(); ///Implicit cast*

*Lemur lemur2 = primate; //doesn’t compile*

*System.out.println(lemur2.age);*

*Lemur lemur3 = (Lemur) primate; //Explicit cast*

*System.out.println(lemur3.age);*

* *In this example, we first create a Lemur object and implicitly cast it to a Primate reference. Since Lemur is a subclass of Primate, this can be done without a cast operator. Next, we try to convert the primate reference back to a lemur reference, lemur2, without an explicit cast. The result is that the code will not compile. In the second example, though, we explicitly cast the object to a subclass of the object Primate, and we gain access to all the methods and fields available to the Lemur class.*
* *When casting objects, you do not need a cast operator if the current reference is a subtype of the target type. This is referred to as an implicit cast or type conversion. Alternatively, if the current reference is not a subtype of the target type, then you need to perform an explicit cast with a compatible type. If the underlying object is not compatible with the type, then a ClassCastException will be thrown at runtime.*
* *Casting a reference from a subtype to a supertype doesn’t require an explicit cast.*
* *Casting a reference from a supertype to a subtype requires an explicit cast.*
* *The compiler disallows casts to an unrelated class.*
* *At runtime, an invalid cast of a reference to an unrelated type results in a ClassCastException.*
* *While the compiler can enforce rules about casting to unrelated types of classes, it cannot do the same for interfaces since a subclass may implement the interface.*
* *Casting is not without limitations. Even though two classes can share a related hierarchy, that doesn’t mean an instance of one can automatically be cast to another public class Rodent {}*

*public class Capybara extends Rodent {*

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

*Rodent rodent = new Rodent();*

*Capybara capybara = (Capybara) rodent; //ClassCastException*

*}*

*}*

*This code creates an instance of Rodent and then tries to cast it to a subclass of Rodent, capybara. Although this code will compile, it will throw a ClassCastException at runtime since the object being referenced is not an instance of the Capybara class. The thing to keep in mind in this example is the Rodent object created doesn’t inherit the Capybara class in anyway.*

***The InstanceOf Operator:***

* *The instanceOf operator, which can be used to check whether the object belongs to a particular class or interface and to prevent ClassCastExceptions at runtime. The below code snippet doesn’t throw an exception at runtime and performs the cast only if the instanceOf operator returns true*

*If(rodent instanceof Capybara) {*

*Capybara capybara = (Capybara) rodent;*

*}*

*Just as the compiler doesn’t allow casting an object to unrelated types, it also doesn’t allow instanceof to be used with unrelated types.*

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

*Fish fish = new Fish();*

*If(fish instanceof Bird) { //Doesn’t compile*

*Bird bird = (Bird) fish; //Doesn’t compile*

*}}*

*In this snippet, neither the instanceof operator nor the explicit cast operation compile.*

***Polymorphism and Method Overriding:***

* *In Java, polymorphism states that when you override a method, you replace all calls to it, even those defined in the parent class.*

*class Penguin{*

*public int getHeight() {return 3;}*

*public void printInfo(){*

*System.out.print(this.getHeight());*

*}}*

*public class EmperorPenguin extends Penguin{*

*public int getHeight() {return 8;}*

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

*new EmperorPenguin().printInfo();*

*}}*

*In this example, the object being operated on in memory is an EmperorPenguin. The getHeight() method is overridden in the subclass, meaning all calls to it are replaced at runtime. Despite printInfo() being defined in the Penguin class calling getHeight() on the object calls the method associated with the precise object in memory not the current reference type when it is called. Even using this reference, which is optional in this example, doesn’t call the parent version because the method has been replaced.*

*The facet of polymorphism that replaces methods via overriding is one of the most important properties in all Java. It allows you to create complex inheritance models with subclasses that have their own custom implementation of overridden methods. It also means the parent class doesn’t need to be updated to use the custom or overridden method. If the method is properly overridden the overridden version will be used in all places that it is called. You can choose to limit the polymorphic behavior by marking methods, final which prevents them from being overridden by a subclass.*

*Calling the Parent version of an Overridden method:*

* *There is no exception to overriding a method where the parent method can still be called, and that is when the super reference is used.*
* *How can you modify our EmperorPenguin example to print 3 as defined in the Penguin getHeight() method? You could try calling super.getHeight() in the printInfo method of the Penguin class*

*class Penguin{*

*…*

*public void printInfo() {*

*System.out.print(super.getHeight()); //Doesn’t compile*

*}*

*Unfortunately, this doesn’t compile, as super refers to the superclass of Penguin, in this case is an Object. The solution is to override printInfo() in the EmperorPenguin subclass and add super there.*

*public class EmperorPenguin extends Penguin{*

*…*

*public void printInfo() {*

*System.out.print(super.getHeight());*

*}…*

*}*

*The new version of EmperorPenguin uses the getHeight() method declared in the parent class and prints 3.*

*Overriding vs Hiding Members:*

* *While method overriding replaces the method everywhere it is called static method and variable hiding doesn’t. hiding members is not a form of polymorphism since the methods and variables maintain their individual properties.*
* *Unlike method overriding, hiding members is very sensitive to the reference type and location where the member is being used.*

*class Penguin {*

*public static int getHeight() { return 3; }*

*public void printInfo() {*

*System.out.println(this.getHeight());*

*}*

*}*

*public class CrestedPenguin extends Penguin {*

*public static int getHeight() { return 8; }*

*public static void main(String... fish) {*

*new CrestedPenguin().printInfo();*

*}*

*}*

* *The CrestedPenguin example is nearly identical to our previous EmporerPenguin example, although as you probably already guessed, it prints 3 instead of 8. The getHeight() method is static and is therefore hidden, not overridden. The result is that calling getHeight() in CrestedPenguin returns a different value than calling it in the Penguin, even if the underlying object is the same. Contrast this with overriding a method, where it returns the same value for an object regardless of which class it is called in.*
* *What about the fact that we used this to access a static method in this.getHeight()? As discussed in Chapter 7, while you are permitted to use an instance reference to access a static variable or method, it is often discouraged. In fact, the compiler will warn you when you access static members in a non-static way. In this case, the this reference had no impact on the program output.*
* *Besides the location, the reference type can also determine the value you get when you are working with hidden members.*

*class Marsupial {*

*protected int age = 2;*

*public static boolean isBiped() {*

*return false;*

*}*

*}*

*public class Kangaroo extends Marsupial {*

*protected int age = 6;*

*public static boolean isBiped() {*

*return true;*

*}*

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

*Kangaroo joey = new Kangaroo();*

*Marsupial moey = joey;*

*System.out.println(joey.isBiped());*

*System.out.println(moey.isBiped());*

*System.out.println(joey.age);*

*System.out.println(moey.age);*

*}*

*}*

*The program prints the following:*

*true*

*false*

*62*

* *Remember, in this example, only one object, of type Kangaroo, is created and stored in memory. Since static methods can only be hidden, not overridden, Java uses the reference type to determine which version of isBiped() should be called, resulting in joey.isBiped() printing true and moey.isBiped() printing false.*
* *Likewise, the age variable is hidden, not overridden, so the reference type is used to determine which value to output. This results in joey.age returning 6 and moey.age returning 2.*

*DON’T HIDE MEMBERS IN PRACTICE*

* *Although Java allows you to hide variables and static methods, it is considered an extremely poor coding practice. As you saw in the previous example, the value of the variable or method can change depending on what reference is used, making your code very confusing, difficult to follow, and challenging for others to maintain.*
* *This is further compounded when you start modifying the value of the variable in both the parent and child methods, since it may not be clear which variable you’re updating.*
* *When you’re defining a new variable or static method in a child class, it is considered good coding practice to select a name that is not already used by an inherited member.*
* *Redeclaring private methods and variables is considered less problematic, though, because the child class does not have access to the variable in the parent class to begin*

*with.*