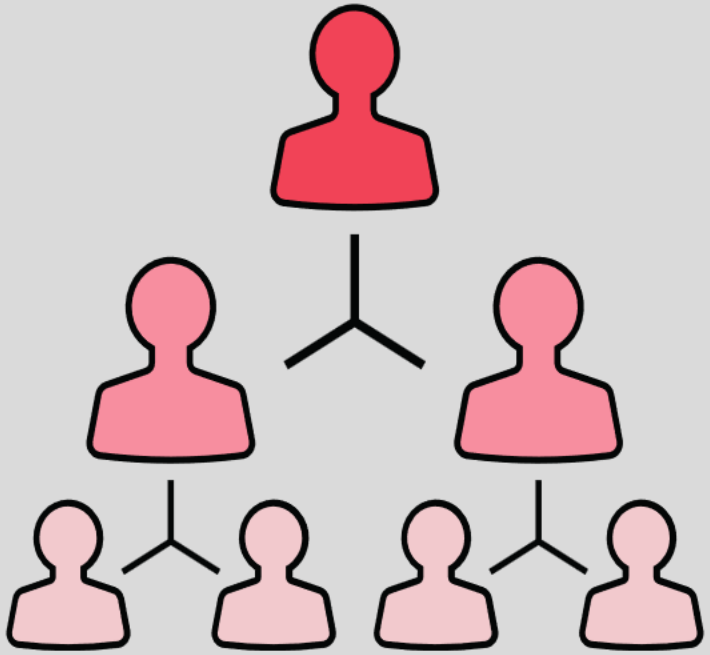
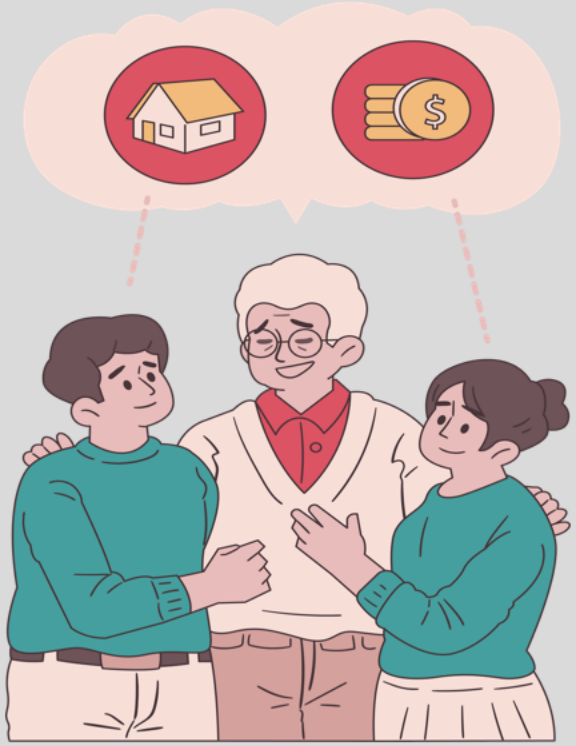
Inheritance

eazy bytes

For a second, let's think we are not technical people and try to answer the meaning of inheritance





In a non-technical sense, the concept of inheritance can be understood through an analogy with real-world scenarios. Imagine a family where certain traits, characteristics, or wealth are passed down from one generation to the next. This passing down of attributes/wealth from parents to children is similar to the idea of inheritance in programming.

Java Inheritance

eazy bytes

In your program, you might want to use the same action in different parts. There are various ways to do this in coding. One way is to copy the same code to all the places where you want that action. The issue with this approach is that if you ever need to change how the action works, you have to make changes in every place you copied the code.

For example, imagine you want the same action in three different parts of your program. You write the code for one part and then copy it to two other parts. The problem comes up when you need to update the code later. You have to make changes in all three places, which can be a challenging task.

Inheritance in object-oriented programming helps with this situation. Instead of copying the same code to different places, you create a class with the code, and other classes can inherit that code. If you ever need to adjust how the action works, you only have to make changes in one place—the original class. Inheritance not only makes code reuse easier but also allows you to customize the code without altering the existing code. And, there's more to inheritance than just reusing and customizing code.

In object-oriented programming, Inheritance stands as a fundamental principle. It enables the formation of a new class by incorporating code from an already existing class. The newly formed class takes on the title of a subclass, while the original class is referred to as the superclass. The superclass holds the code that the subclass reuses and modifies as needed. This relationship is often described as the subclass inheriting from the superclass. The superclass is alternatively called a base class or parent class, while the subclass may be referred to as a derived class or child class

Java Inheritance

eazy bytes

In Java, inheritance is a mechanism that allows one class to inherit the properties and behavior of another class. Just like how a child inherits certain physical traits and characteristics from their parent, a child class in Java inherits certain properties and behavior from its parent class. This allows you to reuse code and create more efficient and organized class hierarchies.

To understand about Inheritance, lets assume that you are trying to build Java classes representing various super heroes from marvel universe like shown below. If you see there is lot of duplicate code representing their name, age, how they eat, walk, sleep, use power using variables & methods. The only method that may have different implementation for each hero is how they use their power. To avoid duplicating the same code across multiple classes, you can create a parent class called Person that contains the shared properties and methods, and then inherit from that class to create subclasses for each superhero with their unique power implementation.

IronMan

```
public class  IronMan {

   String name;
   int age;

   public void eat(String food) {
   }
   public void walk() {
   }
   public void sleep() {
   }
   public void usePower() {
   }

}
```

SpiderMan

```
public class  SpiderMan {

   String name;
   int age;

   public void eat(String food) {
   }
   public void walk() {
   }
   public void sleep() {
   }
   public void usePower() {
   }

}
```

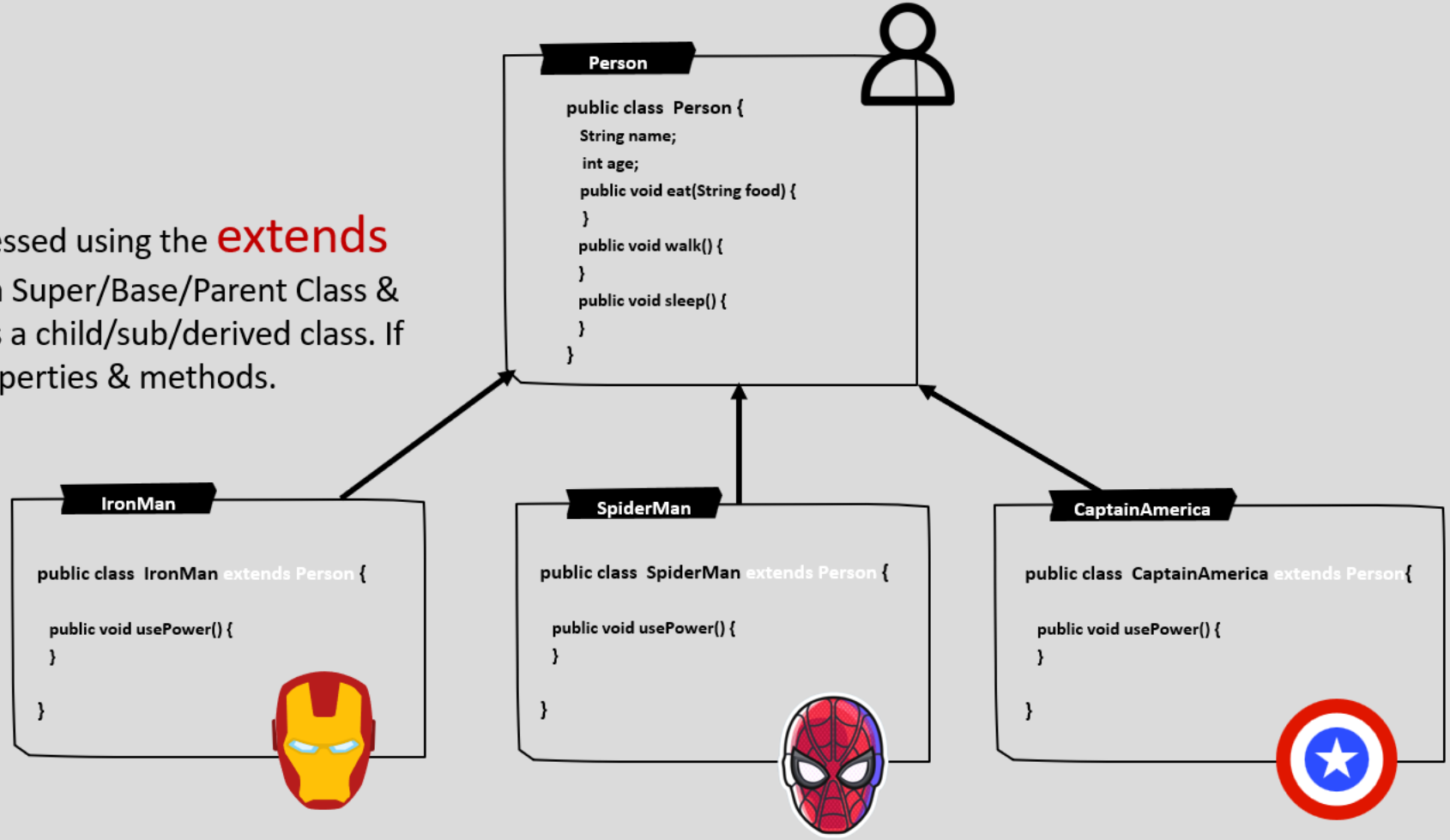
CaptainAmerica

```
public class  CaptainAmerica {

   String name;
   int age;

   public void eat(String food) {
   }
   public void walk() {
   }
   public void sleep() {
   }
   public void usePower() {
   }

}
```

Java Inheritance

eazy bytes

The parent-child relationship in Java is expressed using the extends keyword. Here the Person class is acting as a Super/Base/Parent Class & the remaining classes like SpiderMan acts as a child/sub/derived class. If needed, the child class can have its own properties & methods.

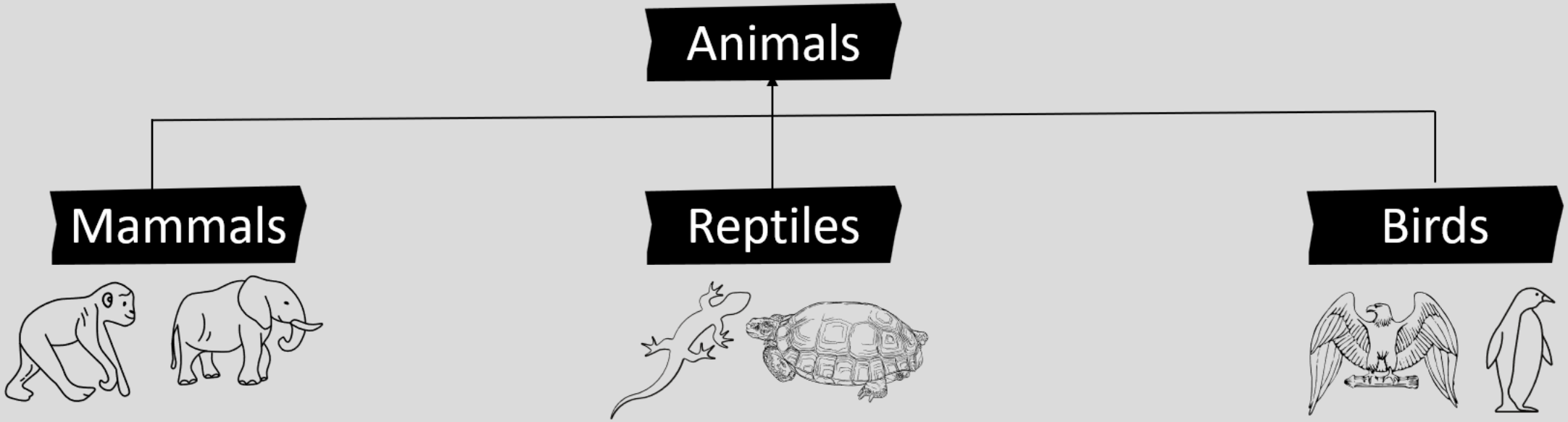


Constructors are not members, so they are not inherited by subclasses, but the constructor of the superclass can be invoked from the subclass.

Java Inheritance

eazy bytes

A real-life example of inheritance can be seen in the relationship between different types of animals. For example, consider the class hierarchy of animals:



In this hierarchy, each animal is a subclass of the Animal class. Each subclass inherits all the properties and behaviors of the Animal class, such as having a name, age, and the ability to move.

However, each subclass also has its own unique properties and behaviors. For example, a Cat subclass might have a method for meowing, while a Lizard subclass might have a method for changing color.

This concept of inheritance allows us to write more efficient and flexible code. Instead of redefining common properties and behaviors for each subclass, we can define them in the Animal class and let each subclass inherit them. This saves time and reduces the likelihood of errors.

Java Inheritance

eazy bytes

In the whimsical world of coding, getting one class to inherit from another is like teaching a class the funky dance moves of another class. It's as easy as pie – just throw in the "extends" keyword along with the name of the superstar class you want your class to mimic. It's like saying, "Hey, subclass, check out these cool moves from the superclass!" The secret receipe looks something like this:

```
[modifiers] class <subclass-name> extends <superclass-name>
{
    // Code for the subclass goes here
}
```

For example, the following code declares a class Q, which inherits from class P, assuming that class P already exists:

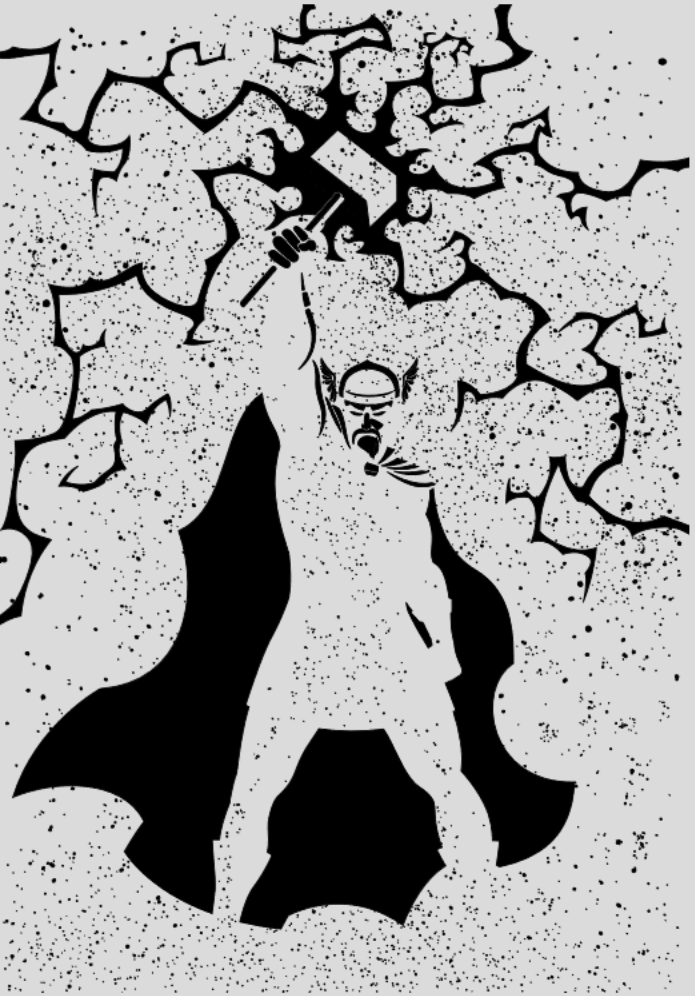
```
public class FunkySubclass extends SuperstarClass {
    // Code magic happens here!
}
```

If your subclass and superclass are living in different coding neighborhoods (packages), you might need to send out a friendly invitation. Use the "import" statement to bring the superclass name into the subclass's world, making it easier to call on that name in the "extends" clause. It's like saying, "Hey, superclass, you're invited to the subclass party – just gotta import your name for easy access!" 🎉📦

If not, you'll have to go for the full celebrity treatment and use the Superclass's fully qualified name. It's like saying, "Alright, Superclass, no shortcuts – we're calling you by your full, fancy name this time!" 🎩💼

Object class is the default Superclass

eazy bytes



The java.lang package contains the Object class that occupies the highest position in the class hierarchy tree. All classes in Java implicitly inherites Object class. As a result, any class you create or use will inherit the instance methods of Object. While it is not mandatory to utilize these methods, you may need to replace them with class-specific code if you choose to use them. Below are the few of the important methods of Object class,

protected Object clone() throws CloneNotSupportedException

Creates and returns a copy of this object.

public boolean equals(Object obj)

Indicates whether some other object is "equal to" this one.

protected void finalize() throws Throwable

Called by the garbage collector on an object when garbage collection determines that there are no more references to the object

public final Class getClass()

Returns the runtime class of an object.

public int hashCode()

Returns a hash code value for the object.

public String toString()

Returns a string representation of the object.

Object class is the default Superclass

eazy bytes

The following two class declarations for class Person are same,

```
// #1 – "extends Object" is implicitly added for class Person by compiler
public class Person {
     // Code for class Person goes here
}
```

```
// #2 – "extends Object" is explicitly added for class Person by developer
public class Person extends Object {

        // Code for class Person goes here
}
```

In the earlier sections, you didn't explicitly use the "extends" clause when declaring your classes. Instead, they quietly inherited from the Object class behind the scenes. That's why those classes could effortlessly tap into the methods of the Object class. Take a peek at this code snippet:

```
Person prsn = new Person();
int hc = prsn.hashCode();
String str = prsn.toString();
```

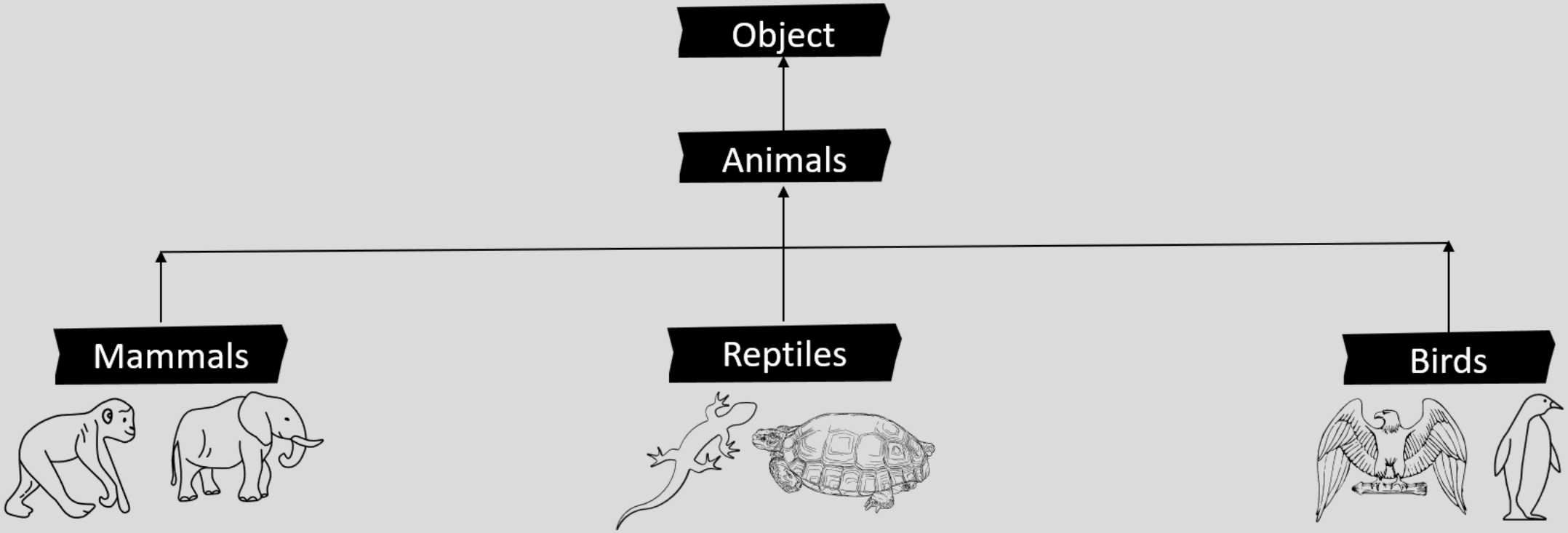
Notice that the Person class doesn't shout out its superclass using "extends." In the background, it inherits from the Object class, which happens to declare the hashCode() and toString() methods. Because Person is essentially a secret subclass of Object, it gets to casually use these methods as if they were always part of its own identity.

Surprise! You've been riding the inheritance train since your very first Java program, even if you didn't realize it. This section is just shedding some light on the magic of inheritance, showing off its cool trick of code reuse. Stick around; there are more perks of inheritance coming up in the next part of the story! 🚂✨

Hierarchical Relationship while using Inheritance

eazy bytes

All classes in an inheritance chain form a tree-like structure, which is known as an inheritance hierarchy or a class hierarchy. Now, the cool part—classes above a class in this hierarchy? They're the ancestors, like the wise elders of the coding tribe. And the classes below? Well, they're the descendants, the next-gen coders learning the ropes from their superiors. It's a family tree, but for classes! 🌳💻



In the figure, an example inheritance hierarchy is depicted for the Animals class and its descendants through a UML (Unified Modeling Language) diagram. In UML diagrams, a connection between a superclass and a subclass is represented by an arrow pointing from the subclass to the superclass.

Hierarchical Relationship while using Inheritance

eazy bytes

In the realm of Java, a class is granted the power of single inheritance—meaning it can have just one superclass (or parent). But hold on, this doesn't mean a class can't be a superhero to multiple other classes. In fact, it can play the role of superclass for several classes.

Here's the deal: every Java class, except the Object class, has a superclass. And guess who's sitting at the VIP spot right at the summit of all inheritance hierarchies? That's right—the Object class. It's like the grand master, overseeing the coding universe from the top! 🚀💻

At times, the term "immediate superclass" is employed to denote the ancestor class that is one level above in the inheritance hierarchy. In contrast, the term "superclass" is utilized to refer to an ancestor class at any level in the hierarchy. Similarly, the term "immediate subclass" is used to signify a descendant class that is one level below in the inheritance hierarchy, while "subclass" is employed for a descendant class at any level.

If a class is a descendant of another class, it is also a descendant of the ancestor of that class. For example, all descendants of the Animals class are also descendants of the Object class. All descendants of the Birds class are also descendants of the Animals class and the Object class.

"is-a" and "has-a" relatioships in Java

eazy bytes

In object-oriented programming, "is-a" and "has-a" are two fundamental concepts used to establish relationships between different classes. Lets see what is the difference between them

"Is-a" relationship is a way to express inheritance, which means that a subclass is a type of its superclass. In Java, you can use the keyword "extends" to establish an "is-a" relationship between two classes. In the below example, the class Dog "is-a" Animal, which means that it inherits all the properties and methods of the Animal class.

```
public class Animal {
   //...
}

public class Dog extends Animal {
   //...
}
```

In Java, inheritance is an is-a relationship. That is, we use inheritance only if there exists an is-a relationship between two classes. For example, SpiderMan is a Person, Car is a Vehicle, Orange is a Fruit. Here, SpiderMan can inherit from Person, Orange can inherit from Fruit, and so on.

On the other hand, "has-a" relationship is used to indicate that a class has a certain property or object of another class. In Java, you can use instance variables to establish a "has-a" relationship between two classes. For example:

```
public class Car {
   private Engine engine;

   public Car(Engine engine) {
      this.engine = engine;
   }

   //...
}

public class Engine {
   //...
}
```

In this example, the class Car "has-a" Engine, which means that it contains an object of the Engine class. The relationship is established through the instance variable "engine". has-a relationship is also known as Aggregation or Composition.

"is-a" and "has-a" relatioships in Java

eazy bytes

Below are the few more examples to understand "is-a" and "has-a" relationships between different classes

Is-A Relationship (Inheritance):

1.Vehicle -> Car, Truck, Motorcycle: Each of these is a kind of Vehicle.
2.Animal -> Dog, Cat, Bird: Each of these is a type of Animal.
3.Shape -> Circle, Square, Triangle: Each of these is a specific type of Shape.
4.Fruit -> Apple, Orange, Banana: Each of these is a specific type of Fruit.
5.Employee -> Manager, Developer, Tester: Each of these is a specialized type of Employee.

Has-A Relationship (Composition):

1.Car -> Engine, Tires, Steering Wheel: A Car has an Engine, Tires, and a Steering Wheel.
2.Computer -> Processor, Memory, Storage: A Computer has a Processor, Memory, and Storage.
3.Building -> Walls, Doors, Windows: A Building has Walls, Doors, and Windows.
4.Library -> Books, Shelves, Librarians: A Library has Books, Shelves, and Librarians.
5.Hospital -> Doctors, Nurses, Patients: A Hospital has Doctors, Nurses, and Patients.

What a subclass inherits from its superclass

eazy bytes

In Java, it's important to note that a subclass doesn't automatically acquire all aspects of its superclass. Instead, it selectively inherits non-private members from the superclass. This means that public, default and protected members get passed down to the subclass, while private members remain confined within the superclass and aren't accessible to the subclass.

It's crucial to recognize that constructors and initializers, both static and instance, don't fall under the category of members within a class. Hence, they do not undergo the inheritance process in Java.

Static members are inherited by subclasses

When a class member is marked as private, its accessibility is restricted solely to the class in which it is declared. This means that private class members are not passed down to subclasses and remain exclusive

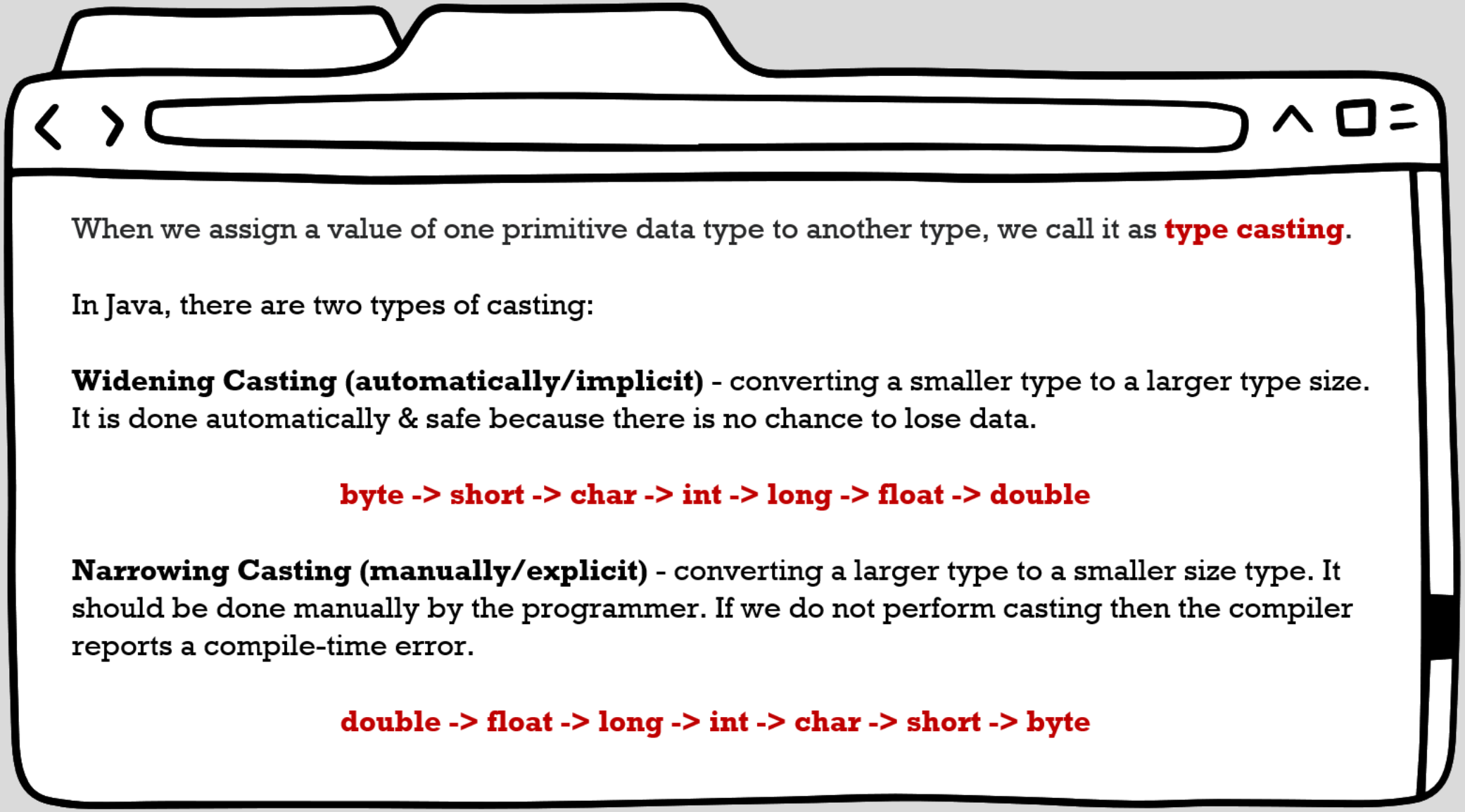
If a class member is declared public, it is accessible from any part of the program, given that the class itself is accessible. In the context of inheritance, a subclass inherits all public members of its superclass.

A class member at the package level/default is inherited exclusively when both the superclass and subclass reside within the same package. In cases where the superclass and subclass belong to different packages, the subclass does not inherit package-level members from its superclass.

When a class member is marked as protected, it is consistently accessible within the body of a subclass, regardless of whether the subclass is situated in the same package as the class or in a different package.

Type Casting in Java

eazy bytes



When we assign a value of one primitive data type to another type, we call it as **type casting**.

In Java, there are two types of casting:

Widening Casting (automatically/implicit) - converting a smaller type to a larger type size. It is done automatically & safe because there is no chance to lose data.

byte -> short -> char -> int -> long -> float -> double

Narrowing Casting (manually/explicit) - converting a larger type to a smaller size type. It should be done manually by the programmer. If we do not perform casting then the compiler reports a compile-time error.

double -> float -> long -> int -> char -> short -> byte

Upcasting and Downcasting in Java

eazy bytes

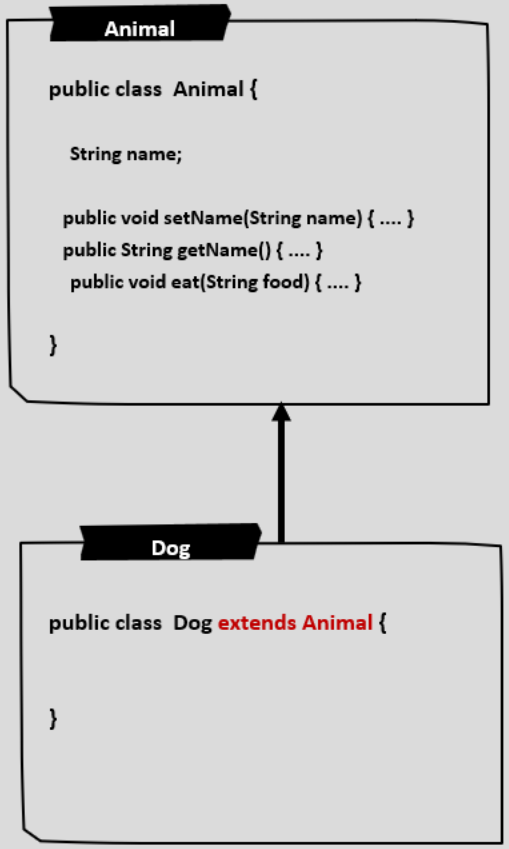
In Java, a class represents a type. By utilizing inheritance to establish an "is-a" relationship, you construct a subclass, which is a more specific type compared to its superclass. For instance, a Dog is a specific subtype of an Animal, and an Animal is, in turn, a particular type of Object. As you ascend within the inheritance hierarchy, you transition from a detailed type to a broader, more general type.

Inheritance in Java impacts client code. The client code written for the superclass can be seamlessly used with subclasses as well. In other words, if your code works with a class, it will also work with its subclasses, because subclasses guarantee the same behaviors as their superclasses, and potentially even more.

Consider the example of Animal and Dog where Dog extends Animal,

```
Animal anm;
anm = new Animal();
anm.setName("Scooby");
String name = anm.getName();
```

```
Animal anm;
anm = new Dog();
anm.setName("Scooby");
String name = anm.getName();
```



Both of the above codes compiles successfully. When you use the expression new Dog(), the type known to the compiler at compile time is Dog. Now, if you have a variable declared as Animal anm, its compile-time type is Animal. Since Dog inherits from Animal, you can safely say that an object of Dog "is-a" type of Animal. This kind of assignment, where you assign a subclass instance to a superclass variable, is termed upcasting in Java. Upcasting is essentially a widening conversion because you're assigning a more specific type (Dog) to a reference variable of a more general type (Animal). And, in Java, it's always allowed. In simpler terms, a Dog is always considered an Animal.

Upcasting and Downcasting in Java

eazy bytes

To determine if an assignment involves upcasting, examine the compile-time type (declared type) of the expression on the right side of the assignment operator. If the compile-time type of the right-hand operand is a subclass of the compile-time type of the left-hand operand, then upcasting is occurring. In essence, this reflects the concept that an object of a subclass "is-a" object of the superclass, making the assignment safe and permissible.

Upcasting is a highly beneficial aspect of inheritance, offering the ability to create polymorphic code that seamlessly interacts with both existing and future classes. This feature enables you to develop your application logic in reference to a superclass, ensuring compatibility with all current and potential subclasses. In essence, upcasting allows you to write generic code without the need to concern yourself with the specific type or class that the code will encounter during runtime. This flexibility enhances the adaptability and extensibility of your code, making it easier to accommodate new classes as your program evolves.

Below is a sample Utility code that is going to print the name of the given animal or it's subclasses. The logic inside it is not going to change, regardless you send parent class data type or child data type,

```
public class AnimalUtility {

    public static void printName(Animal animal) {
        System.out.println(animal.getName());
        animal.eat();
    }

}
```

```
public class UpcastingDemo {

    public static void main(String[] args) {

        Animal anm = new Animal();
        anm.setName("Scooby");
        AnimalUtility.printName(anm);

        anm = new Dog();
        anm.setName("Charlie");
        AnimalUtility.printName(anm);

        Cat cat = new Cat();
        cat.setName("Snoopy");
        AnimalUtility.printName(cat);

    }
}
```

Upcasting and Downcasting in Java

eazy bytes

Downcasting, also known as narrowing conversion, refers to the act of assigning a superclass reference to a subclass variable. Unlike upcasting, where the assignment moves up the class hierarchy, downcasting involves moving down the class hierarchy. It is the counterpart to upcasting. Unlike upcasting, the legality of downcasting cannot be guaranteed by the Java compiler during compile time. To illustrate, consider the following code snippet:

```
Animal anm;
Dog dog = new Dog();
anm = dog;  // upcasting works
dog = anm;  // compilation fails
```

The assignment anm = dog is permissible due to upcasting. However, attempting the assignment dog = anm is disallowed because it involves downcasting, where a variable of the superclass (Animal) is being assigned to a variable of the subclass (Dog). The compiler correctly assumes that every dog is an animal, making upcasting valid. However, not every animal is a dog, illustrating the reason why downcasting is restricted in this context.

In the earlier code snippet, if you are certain that the variable anm contains a reference to a Dog, and you want the downcasting to be successful at compile time, Java introduces an additional rule. To ensure that the downcast is approved during compilation, you must provide an extra assurance to the compiler. This assurance is given by including a typecast, commonly known as a cast, in the assignment. This is demonstrated as follows:

```
dog = (Dog) anm;  // Downcast will be successful this time as we are using a typecast
```

If we try to downcast to a class that is not a subclass of the original class, a ClassCastException is thrown at runtime.

instanceof Operator

eazy bytes

Want to ensure your downcasting is a slam dunk at runtime? Fear not, Java's got your back with the "instanceof" operator – your trusty sidekick in the runtime certainty game!

Picture this: you've got a reference variable, and you're itching to know if it's holding onto an object from a specific class or one of its cool subclasses. That's where the "instanceof" operator struts in with style. It's a dynamic duo – takes two operands and dishes out a snazzy boolean value, either a high-five-worthy true or a not-quite-right false.

Below is the syntax of instanceof operator,

```
<reference-variable> instanceof <type-name>
```

Prior to downcasting, it's advisable to employ the "instanceof" operator to confirm if the reference variable aligns with the anticipated type. For instance, if you aim to verify whether a variable of Animal type points to a Dog object at runtime, the following code snippet would suffice:

```java
Dog dog = new Dog();
Animal anm = dog; // upcasting
if (anm instanceof Dog) {
    // Perform downcasting confidently
    Dog dogInstance = (Dog) anm;
    // Proceed with Dog-specific operations
} else {
    // Handle situations where downcasting is not appropriate
}
```

This approach helps prevent unexpected issues during downcasting by ensuring that the reference variable is compatible with the desired type.

instanceof Operator

eazy bytes

Java 16 introduced a new feature called pattern matching. The instanceof operator can be enhanced to use pattern matching, making the code more concise. Instead of manually casting and checking, you can combine the type check and cast into a single expression.

Below is the syntax of the pattern-matching instanceof operator:

```
<reference-variable> instanceof <type-name> <new-variable-name>
```

Sample code that uses instanceof operator along with the pattern matching. By the time expression inside the if statement executes, the downcasting will be completed automcatically if the instanceof operator return true. The dogInstance variable can be used inside the if block to execute any Dog specific operations,

```
Dog dog = new Dog();
Animal anm = dog; // upcasting
if (anm instanceof Dog dogInstance) {
    // dogInstance is a variable of type Dog.
    // Proceed with Dog-specific operations
} else {
    // Handle situations where downcasting is not appropriate
}
```

instanceof Operator

eazy bytes

The following use of the instanceof operator will generate a compile-time error because the String class is not related to Animal class and is not in the inheritance chain of the Animal class:

```
String str = ”Hello";
if (str instanceof Animal) {  // compilation fails

}
```

Static Binding and Dynamic Binding in Java

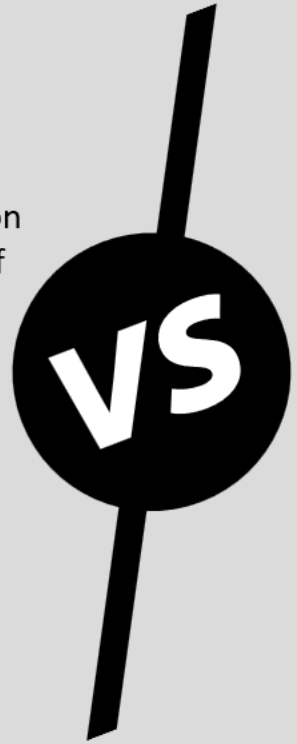
eazy bytes

In object-oriented programming, classes consist of methods and fields, and code is written to interact with these components. Binding refers to the determination of which method's code or field will be utilized when the code is executed. Essentially, binding is the decision-making process that establishes the connection between the code and the specific method or field it accesses. This binding process occurs at two stages: during compile time and at runtime.

Static Binding

Early binding, also referred to as static binding or compile-time binding, occurs when the binding process takes place during compile time. In this context, early binding denotes the association between the code and the specific method or field at the time of compilation.

When type of the object is determined at compiled time(by the compiler), it is known as static binding. private, final and static members (methods and variables) use static binding



Dynamic Binding

Late binding, alternatively termed dynamic binding or runtime binding, occurs when the binding process takes place during program execution rather than during the compilation phase.

When type of the object is determined at run-time, it is known as dynamic binding.

Static Binding and Dynamic Binding in Java

eazy bytes

In the example, obj is of type A, but at runtime it refers to an instance of B. During compilation, the method call obj.show() is bound to the method A.show() because the reference type is A. This is an example of static binding.

In the example, obj is of type A, but at runtime it refers to an instance of B. During runtime, the method call obj.show() is bound to the method B.show() because the actual object type is B. This is an example of dynamic binding.

```
class A {
    void show () {
        System.out.println("Show method in class A");
    }
}

class B extends A {
    void show () {
        System.out.println("Show method in class B");
    }
}

public class Main {

    public static void main(String[] args) {
        A a = new A();
        B b = new B();
        a.show(); // Output: Show method in class A
        b.show(); // Output: Show method in class B
        A obj = new B();
        obj.show(); // Output: Show method in class B
    }

}
```

Polymorphism in Java

eazy bytes

In Java, polymorphism refers to the ability to perform a single action in multiple ways. The term "polymorphism" is derived from the Greek words "poly" and "morphs", where "poly" means many and "morphs" means forms, hence it means "many forms".

In real life also we see many examples of polymorphism. For instance a woman can play multiple roles in a day like a mother, a wife, an employee, a sister. So the same person possesses different behavior in different situations. Polymorphism is considered one of the important features of Object-Oriented Programming.

Based on type of binding, there are two types of polymorphism in Java:

1) **compile-time or static polymorphism**
2) **runtime or dynamic polymorphism**

We can perform polymorphism in java by method overriding (dynamic) and method overloading (static).

Method Overriding in Java Inheritance

eazy bytes

Method overriding is a feature of Java inheritance that allows a subclass to provide its own implementation of a method that is already defined in its parent class. To override a method, the subclass must define a method with the same name, return type, and parameters as the method in the parent class. The access level of the overriding method cannot be more restrictive than the method being overridden.

When a method is called on an object of the subclass, the JVM looks for an implementation of the method in the subclass. If an implementation is found, it is used instead of the implementation in the parent class. Method overriding is useful when you want to provide a specialized implementation of a method in a subclass that is different from the implementation in the parent class. This allows for greater flexibility and customization in the behavior of objects in the subclass.

Method overriding happens only with non-static methods. A private method cannot be overridden.

For example, think like IronMan has special walking style, may be due to his Iron suite ☺ . In this scenario, the child can override the walk() method by using the same method signature as present in parent along with the @override annotation. When intending to override a method within your class, it is advisable to annotate the method with @Override. When a method is marked with @Override, the compiler ensures that the method genuinely overrides a corresponding method in the superclass. Failure to meet this criterion results in a compile-time error.

IronMan

```
public class  IronMan extends Person {

        public void usePower() {

        }

        @Override
        public void walk() {

         }

}
```

What is a annotation ?

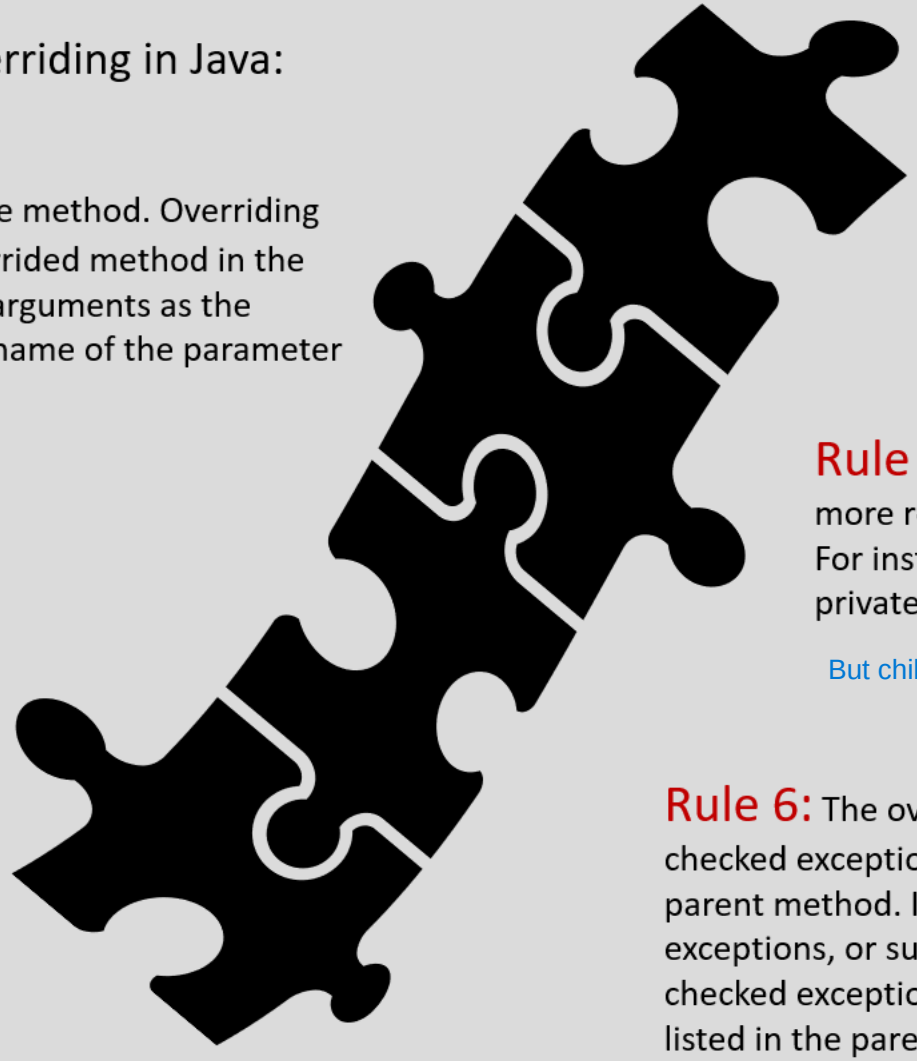
Annotations in Java are a form of metadata that provide information about a program's code, particularly its classes, methods, fields, and parameters. Annotations are defined using the @ symbol followed by an annotation type, and they can be added to code elements such as classes, methods, or fields.

Annotations are used to convey information to the compiler, runtime, or other tools that process the program.

Method Overriding rules

eazy bytes

Here are the rules for method overriding in Java:



Rule 1: The method must be an instance method. Overriding does not apply to static methods. The overrided method in the child class must have the same name and arguments as the method in the parent class. Note that the name of the parameter does not matter.

Rule 2: The return type of the child method can be same or a subclass of the return type in the parent method.

Rule 3: Only inherited methods can be overridden, which means that methods can be overridden only in child classes.

Rule 4: Constructors, private & final methods cannot be overridden as they are not inherited.

Rule 5: The access modifier of the child method can't be more restrictive than the access modifier of the parent method. For instance, if the parent method is protected, then using private in the child's overridden method is not allowed.

But child overridden method can be public

Rule 6: The overridden method is not allowed to introduce new checked exceptions to the set of checked exceptions specified in the parent method. It has the flexibility to eliminate one or all checked exceptions, or substitute an checked exception with another checked exception that is a descendant of the checked exception listed in the parent method.

runtime or dynamic polymorphism using method overriding

eazy bytes

Runtime polymorphism, also known as dynamic method dispatch, refers to the ability of an object-oriented programming language to determine at runtime which implementation of a method to invoke, based on the actual type of the object the method is being called on. This is in contrast to static method dispatch, where the method to be called is determined at compile-time based on the type of the reference variable.

Upcasting is often used in conjunction with runtime polymorphism, as it allows a superclass reference variable to refer to objects of its subclasses, thereby enabling dynamic method dispatch.

```
public class DynamicPoly {

    public static void main(String[] args) {
        Person person = new IronMan(); // upcasting
        person.walk(); // IronMan walk method is going to be invoked
    }

}
```

We are calling the walk method by the reference variable of Person class. Since it refers to the subclass object and subclass method overrides the Parent class method, the subclass method is invoked at runtime.
Since method invocation is determined by the JVM not compiler, it is known as runtime polymorphism.

How to access overridden method in Super class from Sub class ?

eazy bytes

Occasionally, there might be a requirement to invoke the overridden method from a subclass. In such cases, a subclass can employ the keyword "super" as a qualifier to invoke the overridden method of its superclass. It's essential to recognize that the Object class has no superclass, and attempting to use the keyword "super" in the Object class is not permissible. However, it's worth noting that writing code for the Object class is unnecessary for programmers since it is an integral part of the Java class library.

```java
public class Person {


    public void eat(String food) {
        System.out.println("Person is eating the food : " + food);
    }

}


public class IronMan extends Person {


    @Override
    public void eat(String food) {
        super.eat("Pasta");
        System.out.println("IronMan is eating the food : " + food);
    }

    public void callOverriddenEatMethod() {
        super.eat("Pasta");
    }

}
```

The IronMan class overrides the eat() method of the Person class. Note the super.eat() method call inside the eat() method and the callOverriddenEat() methods of the IronMan class. It will call the eat() method of the Person class.

Directly invoking an instance method of a superclass's superclass is not possible in Java. The keyword super allows you to call an overridden method of the immediate superclass (ancestor) only. In a scenario where there are three classes – A, B, and C – with B inheriting from A and C inheriting from B, there is no direct mechanism for class C to call methods of class A. To achieve this, you must create a method in class B that calls a method in class A. Then, class C can call the method in class B, which, in turn, invokes the method in class A. This indirect approach is necessary for invoking methods of the distant superclass in the hierarchy.

Method Overloading in Java

eazy bytes

Method overloading is a feature in Java that allows you to define multiple methods with the same name in the same class. However, the methods must have different parameter lists (i.e., different number of parameters, different data types of parameters, or both). The return type, access level, and throws clause of a method play no role in making it an overloaded method. Using different names for parameters does not make the method overloaded.

When you call an overloaded method, Java will choose the most appropriate method based on the arguments passed. Java looks at the number and data types of the arguments to determine which method to call.

Here is an example of method overloading in Java:

```
public class Example {
   public void print(int n) {
      System.out.println("Printing an integer: " + n);
   }
   public void print(double d) {
      System.out.println("Printing a double: " + d);
   }
   public void print(String s) {
      System.out.println("Printing a string: " + s);
   }
}
```

In this example, we have three methods with the same name, print, but each method has a different parameter list. The first method takes an integer, the second method takes a double, and the third method takes a string.

When we call the print method, Java will automatically choose the most appropriate method based on the type of argument passed. For example:

```
Example ex = new Example();
ex.print(10);       // Calls the first method
ex.print(3.14);     // Calls the second method
ex.print("Hello");  // Calls the third method
```

Method overloading results in compile-time or static polymorphism

Method Overloading in Java Inheritance

eazy bytes

For method overloading, the location of method declaration, whether in a parent or child class/interface, is not relevant. The method overloading in the following example is the same as the previous one.

```java
public class Person {


    public void eat(String food) {
        System.out.println("Person is eating the food : " + food);
    }

}


public class IronMan extends Person {


    @Override
    public void eat(String food) {
      System.out.println("IronMan is eating the food : " + food);
    }

    // overloaded method 1
    public void eat() {

    }

    // overloaded method 2
    public void eat(String food, int quantity) {

    }

}
```

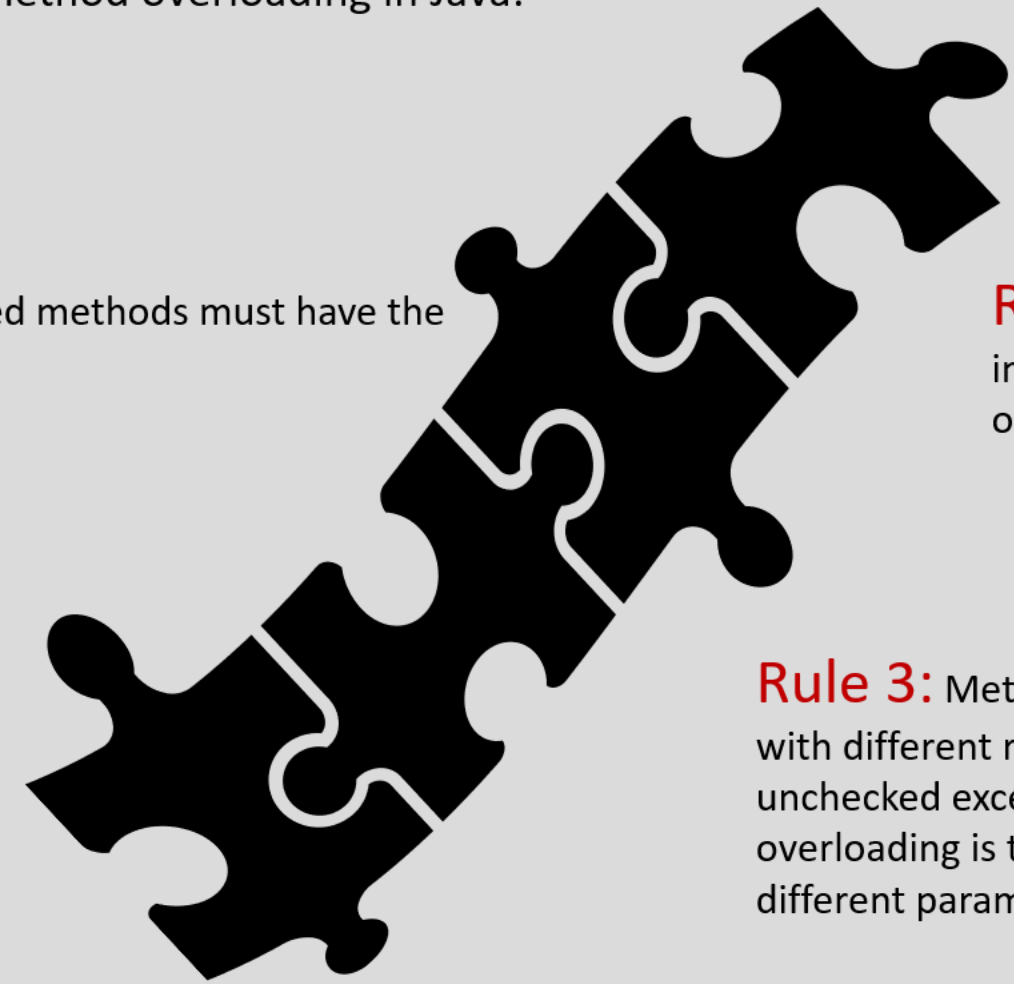
To determine if two methods in a class can be considered overloaded, follow a straightforward rule: list the names of the methods, and then list the types of their parameters from left to right, separating the method names and parameter types with any chosen separator. If the resulting lists for two methods with the same name are different, then those methods are considered overloaded. Conversely, if the lists are the same, the methods are not overloaded.

A private non-static method can be overloaded only by a non-static method of the same class.

Method Overloading rules

eazy bytes

Here are the rules for method overloading in Java:



Rule 1: The overloaded methods must have the same name

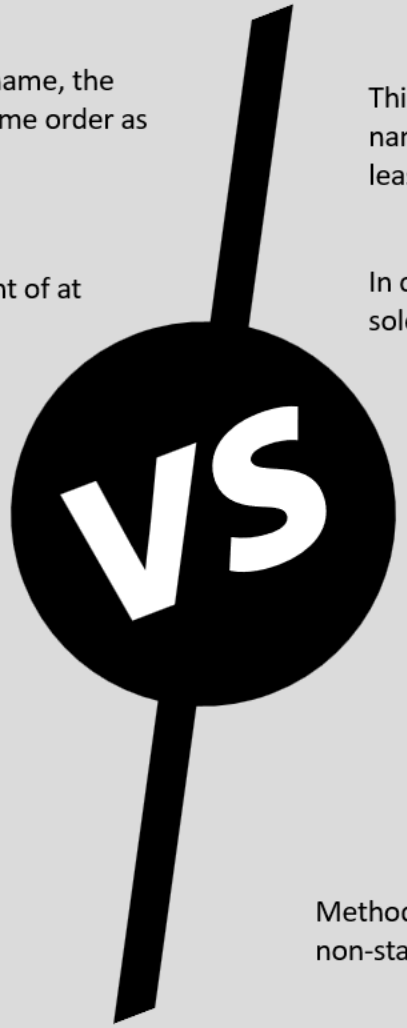
Rule 2: The overloaded methods must differ in the number of parameters, order or type of their parameters.

Rule 3: Method overloading in Java may involve methods with different return types, access modifiers, and checked or unchecked exceptions. However, the defining feature of method overloading is that the methods have the same name but different parameters.

”Method overriding" vs ”Method overloading"

eazy bytes

	”Method overriding"	”Method overloading"
1	This occurs when a class declares a method that shares the identical name, the same number of parameters, and the same parameter types in the same order as specified by its superclass.	This occurs takes place when a class articulates multiple methods under the same name. For methods sharing the same name, it is imperative that they differ in at least one aspect—whether it be the number of parameters, their types, or order.
2	Overriding is a concept tied to inheritance and requires the involvement of at least two classes.	In contrast to overriding, overloading is unrelated to inheritance and can pertains solely to a single class.
3	The return type of the overridden method must be compatible for assignment substitution with the return type of the overriding method.	The overloading of methods is not influenced by the return types of the methods.
4	The overridden method is not allowed to include an additional throws clause compared to the overriding method. It may have an equivalent or a less restrictive set of exceptions in its list compared to the overridden method.	The throws clauses of overloaded methods do not impact the process of overloading.
5	Overriding is exclusive to instance (non-static) methods.	Method overloading is applicable to any method, whether it is static or non-static.



Method overloading is resolved at compile time, contrasting with method overriding, which is resolved at runtime. During compilation, the compiler determines the specific version of the overloaded method that will be called by matching the actual parameters with the formal parameters of the overloaded methods.

Method hiding in Java Inheritance

eazy bytes

A class inherits all non-private static methods from its superclass. The act of redefining an inherited static method in a class is referred to as method hiding. In this context, the redefined static method in a subclass is said to hide the static method of its superclass. It is worth noting that when a non-static method is redefined in a class, this process is known as method overriding.

Consider the following example code:

```
public class Vehicle {

    public static void start() {
        System.out.println("Vehicle starting...");
    }

}
```

```
public class Car extends Vehicle {

    public static void start() {
        System.out.println("Car starting...");
    }

}
```

The start() method in Car is an example of method hiding.

It's important to note that when invoking a static method, you have the option to use either the class name or a reference variable. Using the class name eliminates any ambiguity in method binding. In this case, the compiler binds the static method as defined (or redefined) in the class without any confusion.

But lets try to understand how method binding works when we use reference variable.

Method hiding in Java Inheritance

eazy bytes

```java
public class TestMethodHiding {

    public static void main(String[] args) {

        Vehicle vehicle = new Car();
        Car car = new Car();

        Vehicle.start(); // 1
        vehicle.start(); // 2

        Car.start(); // 3
        car.start(); // 4
        ((Vehicle) car).start(); // 5

        vehicle = car; // 6
        vehicle.start(); // 7
        ((Car) vehicle).start(); // 8

    }
}
```

In the first call, Vehicle.start(), the invocation is performed using the class name. The compiler binds this call to execute the start() method of the Vehicle class.

In the second call, vehicle.start(), the invocation is made using the reference variable vehicle. Since the compile-time type (or declared type) of the vehicle variable is Vehicle, the compiler binds this call to execute the start() method of the Vehicle class.

In the third call, Car.start(), the invocation is performed using the class name. The compiler binds this call to execute the start() method of the Car class.

In the fourth call, car.start(), the invocation is made using the reference variable car. Since the compile-time type (or declared type) of the car variable is Car, the compiler binds this call to execute the start() method of the Car class.

For the fifth call, ((Vehicle) car).start(), some clarification is necessary. The compile-time type of the car variable is Car. However, when a typecast (Vehicle) is applied to the car variable, the compile-time type of the expression (Vehicle) car is effectively changed to Vehicle. Upon calling the start() method on this expression, the compiler binds it according to its compile-time type, which is now Vehicle. As a result, the print() method of the Vehicle class will be invoked.

In the initial statement at 6, a Car object's reference is assigned to the vehicle reference variable. Following this assignment, vehicle now points to an object of the Car class. When the call to the start() method occurs at 7, the compiler evaluates the compile-time type (or declared type) of the vehicle variable, determining it to be Vehicle. Consequently, the compiler associates the call vehicle.start() with the start() method of the Vehicle class.

In contrast, the last call to the start() method is linked to the start() method of the Car class. This association arises from the typecast (Car), which sets the type of the entire expression to Car.

Field hiding in Java Inheritance

eazy bytes

In a class, the declaration of a field, whether static or non-static, results in the hiding of an inherited field with the same name in its superclass. The considerations for field hiding do not involve the type of the field or its access level; instead, it is solely determined by the field name. Field access, in the context of field hiding, employs early binding, meaning that the compiler uses the compile-time type of the class to bind the field access.

Consider the following example code:

```java
public class Vehicle {

    public int horsePower = 120;
    public String color = "White";
    public double turningRadius = 5.23;

}

public class Car extends Vehicle {

    public int horsePower = 150;
    public String color = "Black";
    public String turningRadius = "6.23";
    public boolean isAutomatic = true;

}
```

The field declarations horsePower, color, and turningRadius within class Car effectively conceal the inherited fields from class Vehicle. It is crucial to highlight that the act of having the same field name in a class is sufficient to hide the corresponding field in its superclass. The data types of the hiding and hidden fields are irrelevant. For instance, even if the data type of turningRadius in class Vehicle is double, while the data type of turningRadius in class Car is String, the field turningRadius in class Car still conceals the field turningRadius in class Vehicle.

When using the simple names horsePower, color and turningRadius in class Car, they reference the hiding fields, not the inherited fields. Therefore, if one uses the simple name color in class Car, it pertains to the field color declared in class Car, not in class Vehicle. To reference the field color in class Parent from within class Car, the keyword super needs to be employed, for example, super.color

The story of constructors & Inheritance together

eazy bytes

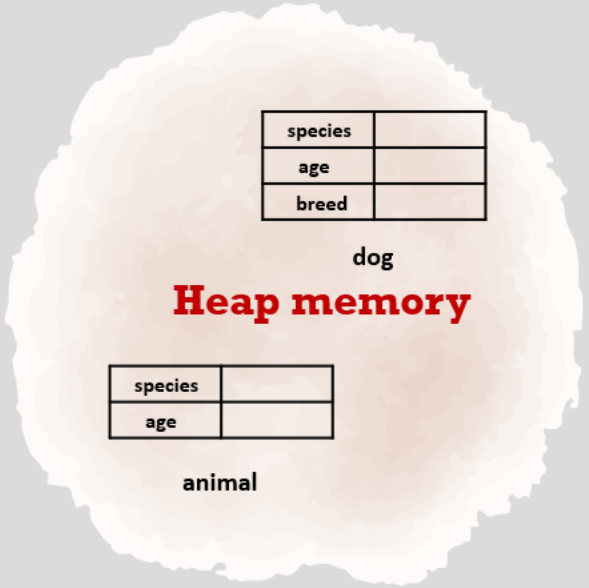
An object contains two essential aspects: state and behavior. The state is represented by instance variables within a class, and the behavior is captured by instance methods. Each object of a class preserves its unique state. Upon creating an object, memory allocation occurs for all instance variables declared in the class, as well as those declared in its ancestor classes at all levels.

Consider the following example code:

```java
public class Animal {

    private String species;
    protected int age;

    // other code

}

public class Dog extends Animal {

    protected String breed;

    // other code

}
```

When an object of class Animal is created, memory is allocated only for the instance variables that are declared in class Animal. When an object of class Dog is created, memory is allocated for all instance variables in class Animal and class Dog.

Below images show on how the memory allocation is going to happen for the objects of Animal and Dog,



When you create an object of a class, the object contains instance variables from the class and all of its ancestors. But how is this being achivied ?

The story of constructors & Inheritance together

eazy bytes

Consider the following example code:

```
public class Parent {

    public Parent() {
        System.out.println("Inside Parent() constructor.");
    }

}

public class Child extends Parent {

    public Child() {
        System.out.println("Inside Child() constructor.");
    }

}

public class SomeClass {
    public static void main(String[] args) {
        Child child = new Child();
    }
}
```

The output from the SomeClass class indicates that the constructor of the Parent class is invoked first, followed by the constructor of the Child class. Notably, the constructor of the Object class is implicitly called before the constructor of the Parent class, though this detail is not directly printable since the Object class is not modifiable.

The question arises: "How is the constructor of the Parent class invoked?" The answer lies in the rule that when an object of a class is instantiated, memory is allocated for all instance variables, including those in all ancestor classes. To ensure proper initialization, constructors for all classes must be invoked. The compiler facilitates this rule by automatically inserting a call to the immediate ancestor's no-args constructor as the first statement in every constructor you define for your class by using super() . This process is integral to enforcing proper initialization.

Moreover, the keyword super is versatile in its usage and also signifies the immediate ancestor of a class. When followed by parentheses, it specifically refers to the constructor of the superclass.

If a superclass constructor accepts parameters, you should pass a list of parameters within parentheses similar to a method call by using super().
The following are examples of calling the constructor of a superclass:

```
super(); // Call no-args constructor of superclass
super("John Doe"); // Call superclass constructor that accepts a String argument
super("John Doe", 25); // Call superclass constructor that accepts String and int argument
```

The story of constructors & Inheritance together

eazy bytes

Consider the following example code. The Child2 class will not compile. Let's try to understand why ?

```
public class Parent2 {

    String name = "Unknown";

    public Parent2(String name) {
        this.name = name;
    }

}

public class Child2 extends Parent2 {
    // Empty class
}
```

Since we haven't included any constructor for the Child2 class. Consequently, the compiler will automatically add a default no-args constructor. In doing so, it attempts to insert a super() call as the initial statement within the no-args constructor. This super() call aims to invoke the no-args constructor of the Parent2 class. However, a complication arises because the Parent2 class lacks a no-args constructor. This discrepancy is the cause of the aforementioned error.

To fix this, we can follow any of the below approaches,

1) Adding a no-args constructor to the Parent2 class

2) Add a no-args constructor to the Child2 class and explicitly call the constructor of the Parent2 class with a String argument

3) Include a constructor in the Child2 class that accepts a String argument and forwards the value to the constructor of the Parent2 class. This enables the creation of a Child2 instance by providing the child's name as a parameter to its constructor.

Each class is required to invoke the constructor of its superclass either directly or indirectly within its own constructors. In the event that the superclass lacks a default (no-args) constructor, explicit calls to any other constructors of the superclass become necessary.

this and super keywords in Java

eazy bytes

In Java, this and super are two keywords used to refer to the current object and its superclass respectively. Here's a brief explanation & comparision of them

this is a reference variable in Java that refers to the current object. It is often used to access instance variables and methods of the current object. For example, if you have a class with an instance variable called name, you can refer to that variable using this.name

```java
public class Person {
    private String name;
    private int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
}
```

this() can be used to invoke other constructors from a constructor inside the same class and when it is being used, it should be the very first statement inside a constructor.

this keyword is a reserved keyword in java i.e, we can't use it as an identifier.
Same with super keyword as well.
We can use this as well as super anywhere except in static area.

super is a keyword in Java that refers to the superclass of the current object. It can be used to access the superclass's instance variables and methods. Here's an example of how super can be used in a subclass:

```java
public class Animal {
    public void makeSound() {
        System.out.println("The animal makes a sound");
    }
}

public class Dog extends Animal {
    public void makeSound() {
        super.makeSound();
        System.out.println("The dog barks");
    }
}
```

In this example, the Dog class extends the Animal class and overrides its makeSound method. The super.makeSound() line in the Dog class calls the makeSound method of the Animal class before printing "The dog barks".

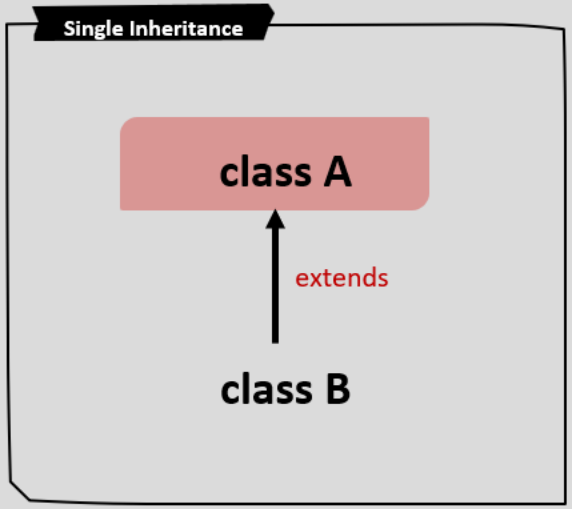
super() can be used to invoke super class constructors from a subclass constructor and when it is being used, it should be the very first statement inside a constructor.

The most common use of super keyword is that it eliminates the confusion between the superclasses and subclasses that have methods with same name.

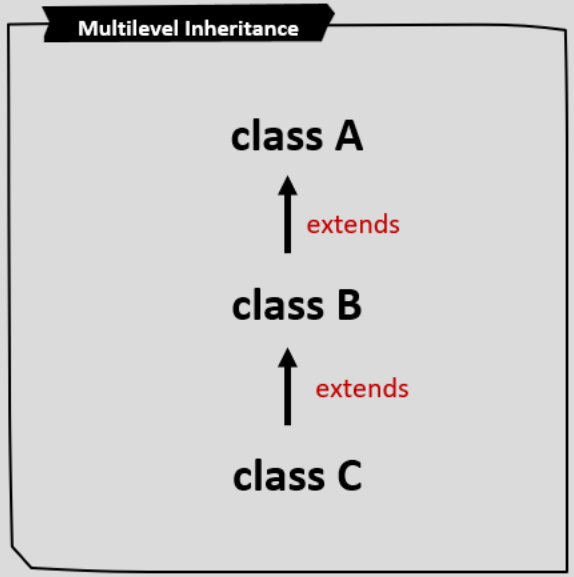
Types of Inheritance

eazy bytes

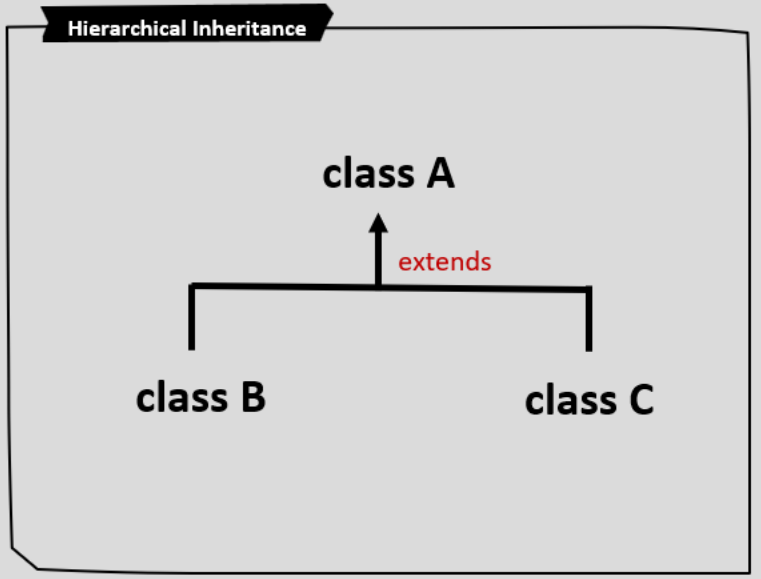
There are five different forms of inheritance that exist.



Single inheritance involves a solitary subclass that inherits from a lone superclass.



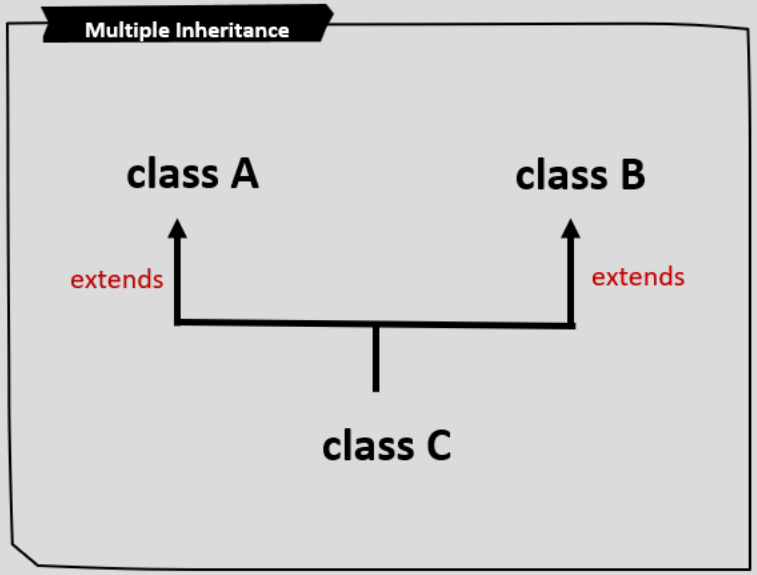
Multilevel inheritance involves a subclass that extends from a superclass and subsequently serves as a superclass for another class.



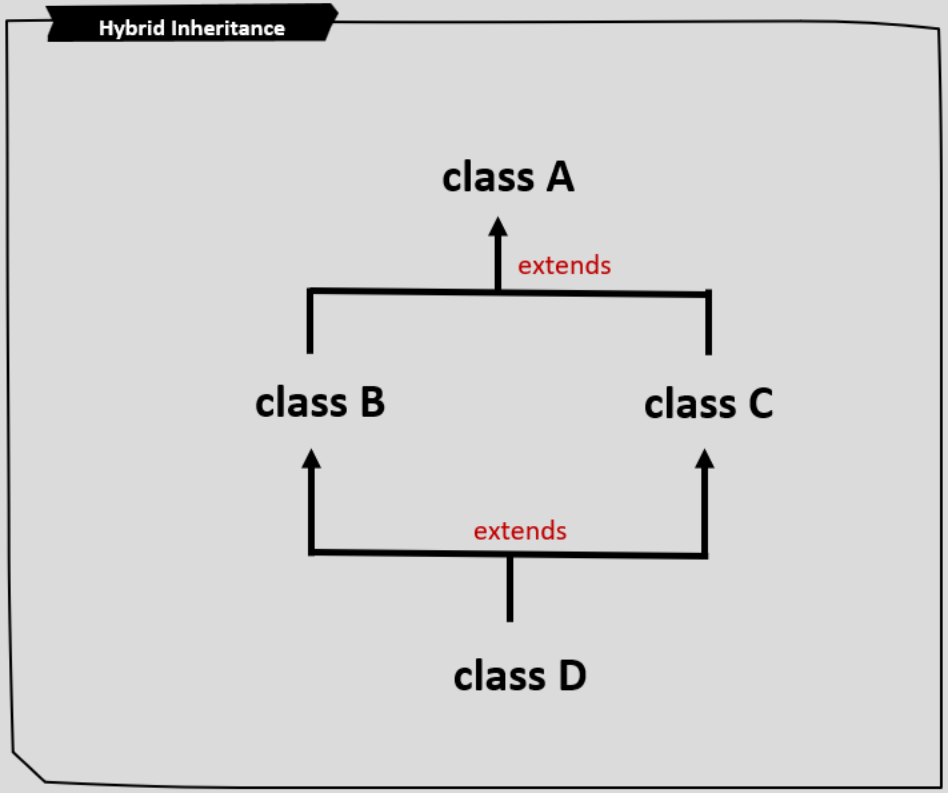
Hierarchical inheritance is defined as multiple subclasses that inherit from a common superclass.

Types of Inheritance

eazy bytes



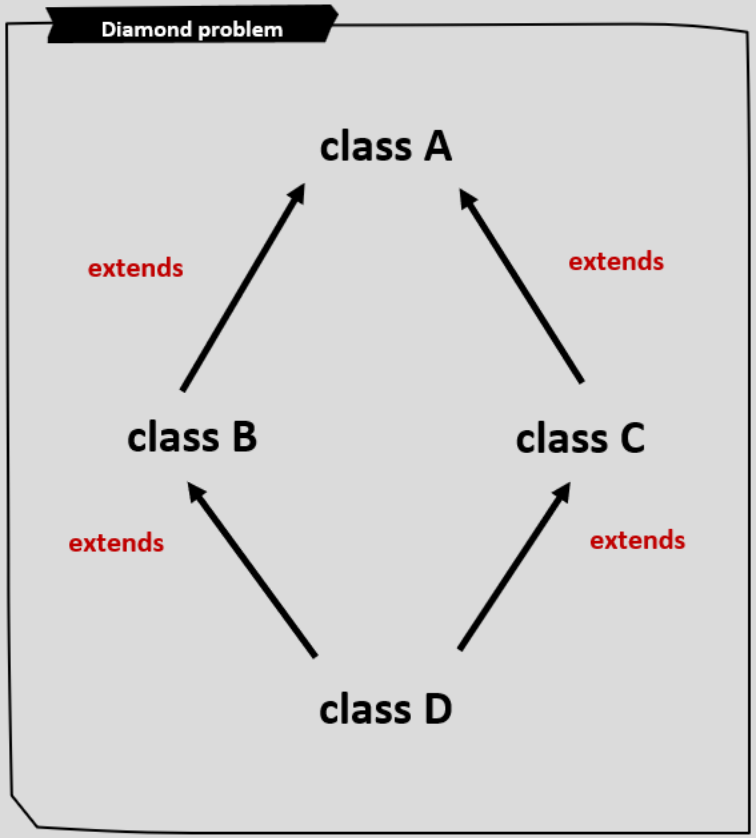
Multiple inheritance refers to a subclass inheriting from more than one superclass, which is not supported in Java. Nevertheless, multiple inheritance can be accomplished using interfaces.



Hybrid inheritance involves the amalgamation of two or more types of inheritance.

Why multiple inheritance with classes is not allowed

eazy bytes



Multiple inheritance is not allowed in Java with classes because it can lead to a number of potential problems and complexities, such as the "diamond problem" where there are conflicting implementations of a method from multiple parent classes.

```java
class A
{
 public void sample() {
    System.out.println("class A sample() method");
 }
}
```

"diamond problem"

```java
class B extends A
{
    @Override
    public void sample()
    {
    System.out.println("class B sample() method");
    }
}
```

```java
class C extends A
{
    @Override
    public void sample()
    {
    System.out.println("class C sample() method");
    }
}
```

```java
//not supported in Java
public class D extends B,C
{
    public static void main(String args[])
    {
        D d = new D();
        //creates ambiguity which sample() method to invoke
        d.sample();
    }
}
```

Abstraction in Java

eazy bytes

In Java, abstraction is a fundamental concept that involves simplifying complex systems by modeling classes based on essential characteristics and ignoring unnecessary details. It is one of the four pillars of object-oriented programming, alongside encapsulation, inheritance, and polymorphism.

Abstraction allows developers to focus on the essential features of an object while hiding the non-essential details. This is typically achieved through the use of abstract classes and interfaces. Abstract classes can have abstract methods (methods without a body) that must be implemented by the concrete (non-abstract) subclasses. Interfaces, on the other hand, define a contract that implementing classes must adhere to.

By utilizing abstraction, developers can create a blueprint or template for classes, defining the methods and properties that are necessary for a specific type of object without specifying how they are implemented. This promotes code reusability, maintainability, and a clear separation of concerns in the software design. Abstraction is crucial in building scalable and modular systems in Java and other object-oriented programming languages.

There are two ways to achieve abstraction in java

- **Abstract class (0 to 100%)**
- **Interface (100%)**

abstract methods and classes

eazy bytes

Sometimes, you might design a Java class to represent a concept rather than representing tangible objects. Consider the scenario of developing classes for various educational subjects. A subject is an abstract concept; it doesn't have a physical existence. If someone asks you to provide details about a subject, your initial inquiry might be, “Which subject are you referring to?” It makes sense to discuss specific subjects like mathematics or history.

In Java, you can create a class for which objects cannot be instantiated; its sole purpose is to represent an abstract idea shared among objects of other classes. Such a class is termed an abstract class. Conversely, a "concrete class" is one that is not abstract, and instances of it can be created. Up until now, all the classes you've created have been concrete classes.

To designate a class as abstract in Java, you utilize the abstract keyword in the class declaration.

```
public abstract class Subject {
    // other code
}
```

As the Subject class is marked as abstract, creating an object of this class is not permitted, despite having a public constructor (that gets automatically added by the compiler). However, you can declare a variable of an abstract class, similar to how you declare variables for concrete classes. The code snippet below illustrates both valid and invalid use cases of the Subject class:

```
Subject sub;     // Compiles successfully
new Subject();   // Compilation fails
```

abstract methods and classes

eazy bytes

Examining the definition of the Subject class, it appears similar to any other concrete class, except for the inclusion of the abstract keyword in the declaration. Like any class, it encompasses instance variables and methods that define the state and behavior of its objects. The declaration of a class as abstract signals that it contains incomplete method definitions (behaviors) for its objects. Therefore, it is considered incomplete for the purpose of creating objects.

An incomplete method in a class refers to a method that is declared but lacks a body. It's important to clarify that the term "missing body" does not imply an empty body; rather, it signifies the absence of any body. In Java, the body of a method is denoted by braces. For incomplete methods, these braces are replaced with a semicolon.

When a method is incomplete, the abstract keyword is utilized in its declaration to explicitly indicate its abstract or incomplete nature. Let's consider an example with a Subject class – a class that represents a generic subject. The teach method, for instance, may not have a concrete implementation because how a subject is taught depends on the specific subject in question. Despite not knowing the exact teaching methodology, it is certain that a subject should be teachable. In this scenario, where the behavior name (teach) is known, but the implementation is uncertain, declaring teach as an abstract method (or incomplete method) in the Subject class is a fitting choice.

```
public abstract class Subject {

    public abstract void teach(); // abstract method

}
```

abstract methods and classes

eazy bytes

Declaring a class as abstract does not imply the necessity of having at least one abstract method. An abstract class can encompass entirely concrete methods, consist solely of abstract methods, or include a combination of both. The key characteristic is that an object of an abstract class cannot be instantiated.

However, if a class contains an abstract method, whether declared directly or inherited, it is imperative to declare the class as abstract. In essence, the presence of an abstract method mandates the declaration of the class as abstract, signifying that the class is incomplete and intended for extension by subclasses.

The utility of an abstract class lies in its inherent promotion of inheritance, at least in theory. Without inheritance, an abstract class becomes essentially ineffective. Consider the Subject class as an example; until concrete implementations are provided for its abstract methods, the other components such as instance variables, concrete methods, and constructors remain functionally useless. The practical value of an abstract class is realized when subclasses are created, tasked with overriding the abstract methods and supplying meaningful implementations.

Declaring an abstract class as final is not allowed. Remember that a final class cannot be extended or subclassed, and this contradicts the fundamental purpose of an abstract class. Abstract classes, by nature, require subclasses to provide meaningful implementations, making the declaration of an abstract class as final incompatible with its intended usage.

Declaring all constructors of an abstract class as private is not advisable, as it would render the abstract class unable to be subclassed. It's important to note that when an object of a class is created, the constructors of all ancestor classes, including abstract classes, are invariably invoked.

abstract methods and classes

eazy bytes

Declaring an abstract method as private is not allowed. It's important to remember that private methods are not inherited, and therefore, they cannot be overridden. The essence of an abstract method lies in the expectation that a subclass should be capable of overriding it and providing a concrete implementation.

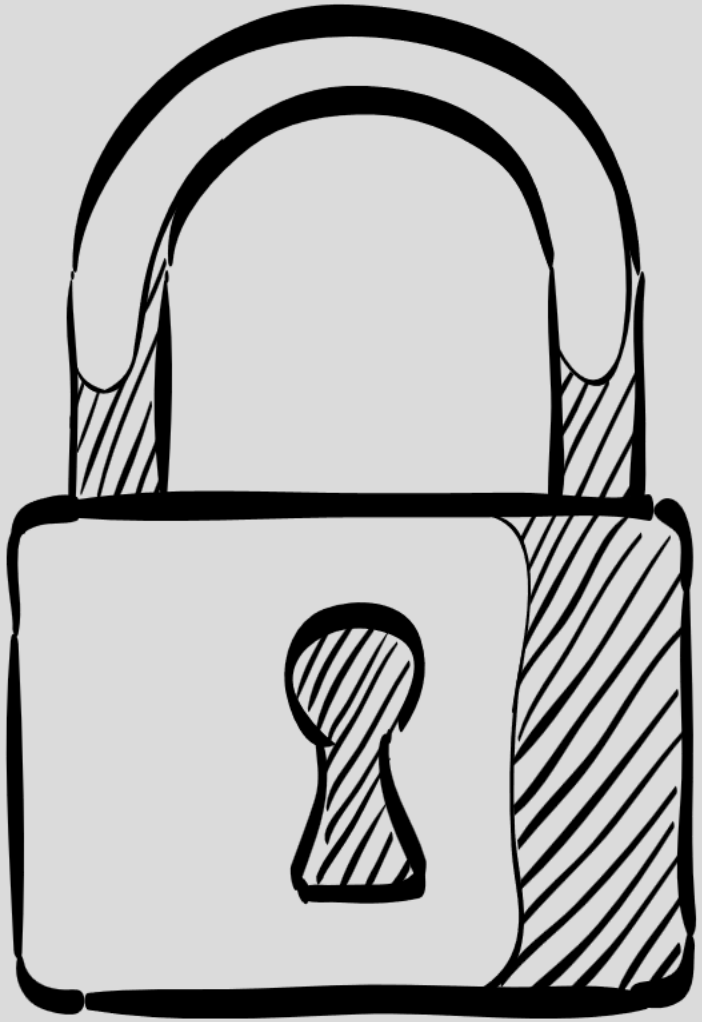
Declaring an abstract method as static is not permissible. It's essential to recognize that abstract methods must be overridden and implemented by a subclass. Since static methods cannot be overridden, the concept of an abstract method is incompatible with the static modifier. Nonetheless, a static method can be hidden, though it cannot be overridden.

An abstract class can be extended by another abstract class and the sub class can maintain the methods abstract as well

```
public abstract class A {
    public abstract void method1();
}

public abstract class B extends A {
    public abstract void method1();
}

public class C extends B {

    public void method1() {
        // Code goes here
    }
}
```

final keyword in Java

eazy bytes



In Java, the final keyword is used to indicate that a variable, method, or class cannot be modified or extended after it has been defined.

When used with a variable, final means that the value of the variable cannot be changed once it has been assigned. For example:

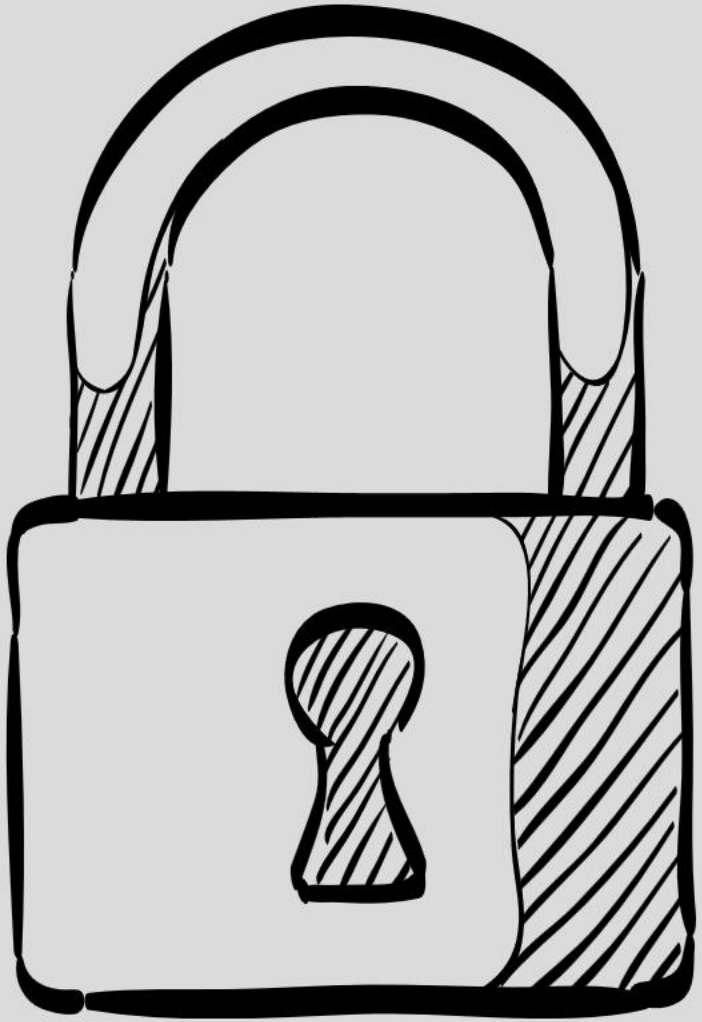
```
final int x = 5;
x = 10; // compilation error
```

When used with a class, final means that the class cannot be subclassed. Using final keyword inside the class definition, we can disable inheritance. For example:

```
final public class Animal {
    // Class definition
}

// compilation error
public class Dog extends Animal {
    // Class definition
}
```

final keyword in Java

eazy bytes



When used with a method, final means that the method cannot be overridden by any subclasses. For example:

```java
public class Animal {
    public final void eat() {
        System.out.println("The animal is eating");
    }
}

public class Dog extends Animal {
    public void eat() { // compilation error
        System.out.println("The dog is eating");
    }
}
```

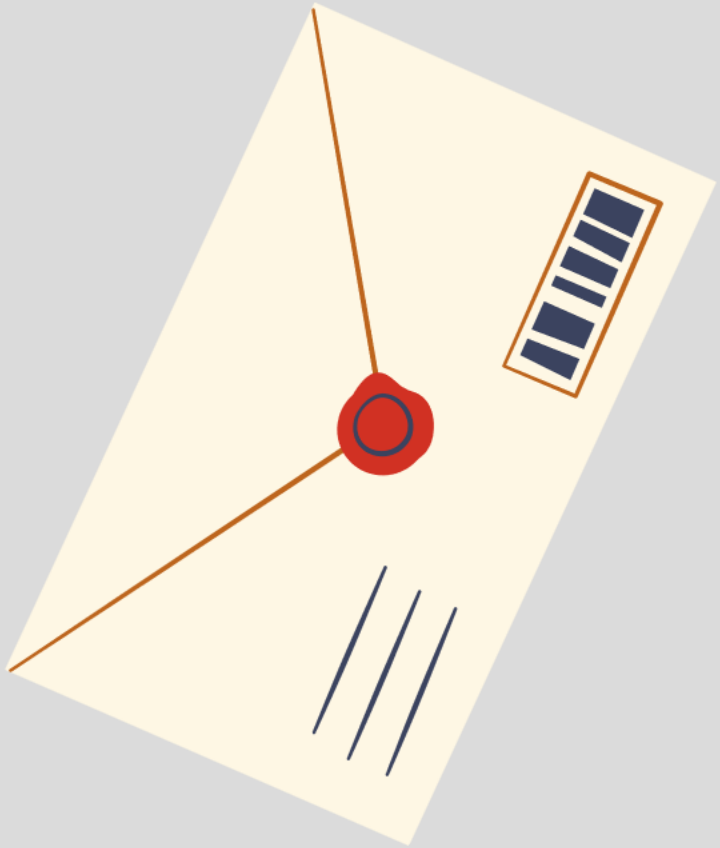
The final keyword can also be used with method parameters, which means that the value of the parameter cannot be changed within the method. Additionally, it can be used with local variables to indicate that their value should not be changed once initialized.

Overall, the final keyword is useful for ensuring that certain values, methods, or classes cannot be modified or extended, which can improve the stability and security of a program.

From Java 17

Sealed classes & interfaces

eazy bytes



Sealed classes and interfaces are a new feature in Java 17 that provide enhanced control over class and interface inheritance. They allow a developer to restrict the inheritance hierarchy of a class or interface to a predefined set of subtypes, thereby improving code safety and maintainability.

A sealed class or interface is declared with the "sealed" modifier and specifies a list of permitted subclasses or subinterfaces using the "permits" keyword.

For example, the following declaration of Person specifies two permitted subclasses Employee, Student :

```
public sealed class Person
    permits Employee, Student {
}
```

Constraints on Permitted Subclasses

- They must be in the same module as the sealed class (if the sealed class is in a named module) or in the same package (if the sealed class is in the unnamed module).
- They must directly extend the sealed class.
- They must have exactly one of the following modifiers to describe how it continues the sealing initiated by its superclass:

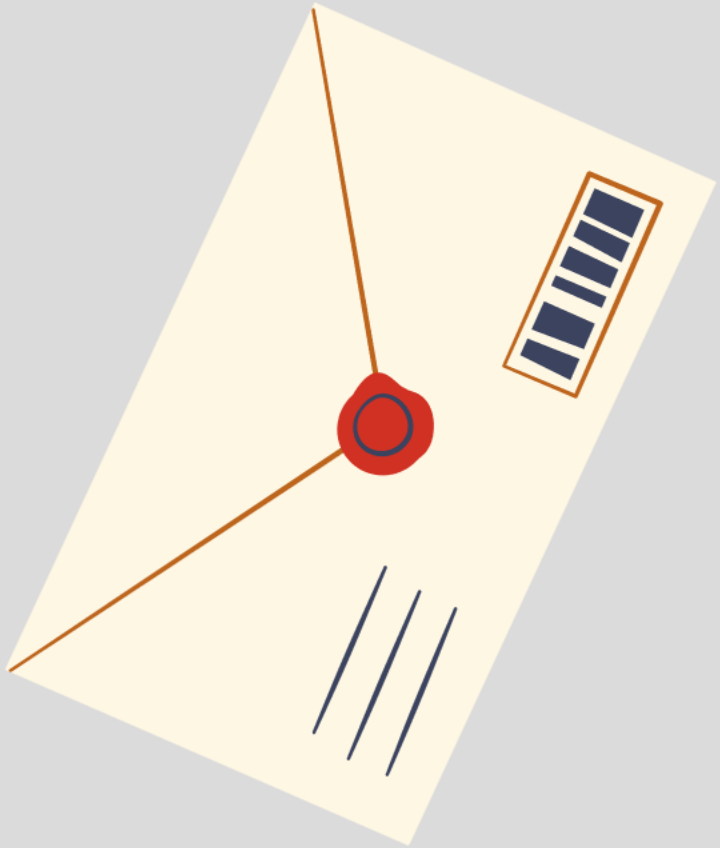
  - ✓ final: Cannot be extended further
  - ✓ sealed: Can only be extended by its permitted subclasses
  - ✓ non-sealed: Can be extended by unknown subclasses; a sealed class cannot prevent its permitted subclasses from doing this

From Java 17

Sealed classes & interfaces

eazy bytes

The following example declares a sealed interface named Animal. Only the classes Dog, Cat can implement it,



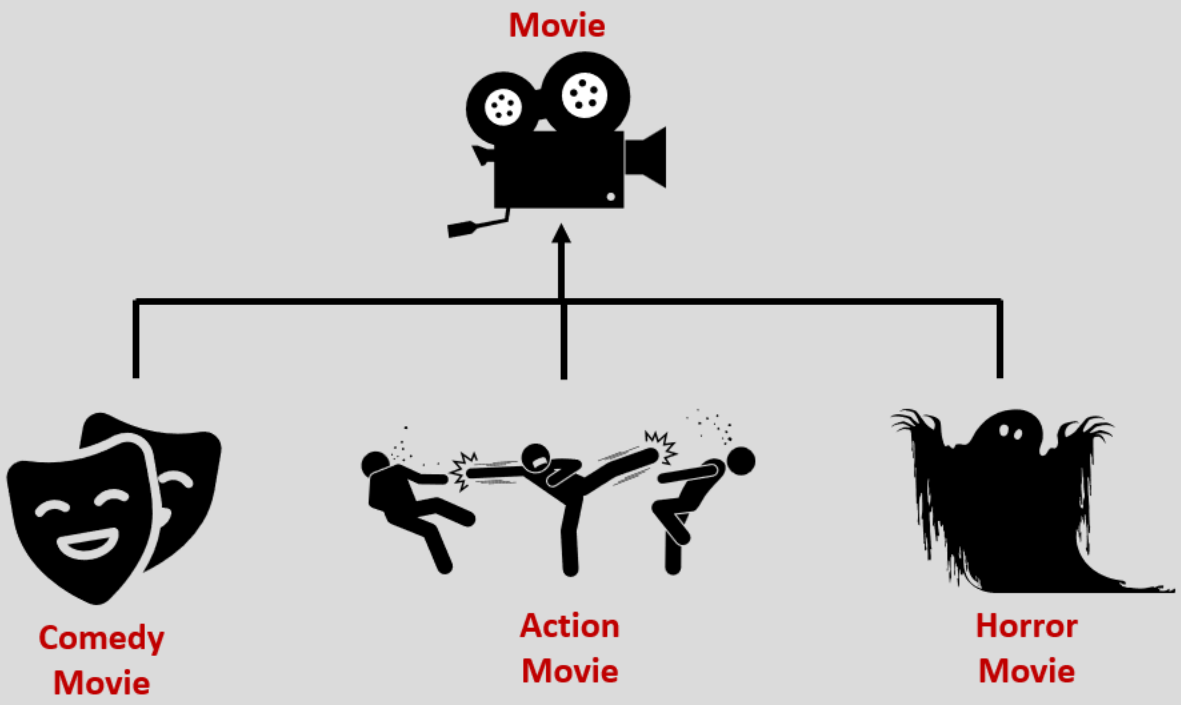
```
sealed interface Animal
    permits Dog, Cat {

    public void makeSound();

}
```

Overall, sealed classes and interfaces provide a useful tool for designing more secure and flexible class hierarchies in Java, particularly in larger codebases where inheritance can become complex and difficult to manage.

Polymorphism using Inheritance

eazy bytes

Let's assume we have a parent class called "Movie" that has three child classes called "ActionMovie", "ComedyMovie" and "HorrorMovie". Each of these child classes extends the parent "Movie" class and has its own unique methods and properties.




```
public class Movie {

    private String title;
    private int length;

    public Movie(String title, int length) {
        this.title = title;
        this.length = length;
    }

    public String getTitle() {
        return title;
    }

    public int getLength() {
        return length;
    }

    public void play() {
        System.out.println("Playing " + title);
    }
}
```

Polymorphism using Inheritance

eazy bytes

```
public class ActionMovie extends Movie {

    public ActionMovie(String title, int length) {
        super(title, length);
    }

}
```

```
public class ComedyMovie extends Movie {

    public ComedyMovie(String title, int length) {
        super(title, length);
    }

}
```

```
public class HorrorMovie extends Movie {

    public HorrorMovie(String title, int length) {
        super(title, length);
    }

}
```

Polymorphism using Inheritance

eazy bytes

```
public class Main {

    public static void main(String[] args) {

        Movie[] movies = new Movie[3];
        movies[0] = new ActionMovie("Die Hard", 120);
        movies[1] = new ComedyMovie("The Hangover", 90);
        movies[2] = new HorrorMovie("The Shining", 150);

        for (Movie movie : movies) {
            movie.play(); // Demo of Polymorphism
        }
    }
}
```

In this example, we created an array of three movies of different genres. We then used a for loop to call the play() method for each movie, which is a method inherited from the parent class "Movie."

Polymorphism allows you to write code that is based on a parent or base class, making it more generic and flexible. This means that your code doesn't need to change when new subclasses are added. In the main method, this generic code can handle any instances that are a "Movie" or a subclass of "Movie" that are returned from the factory method.