13. Implementing Inheritance

Exam Objectives

1. Implement inheritance, including abstract and sealed classes.
2. Override methods, including that of Object class.
3. Implement polymorphism and differentiate object type versus reference type.

13.1 Create and use subclasses and superclasses

13.1.1 Understanding Inheritance

In general terms, when a class extends another class, we say that the class inherits from the other class. But this is a very vague view of inheritance. It is important to understand what exactly this class inherits from the other class and what exactly inheriting something means.

There are two things that a class contains - the instance fields defined in the class, i.e., **state**, the instance methods that provide behavior to the class, i.e., **implementation**. Furthermore, a class, on its own, also defines a **type**. Any of these three things can be inherited by the extending class. Thus, inheritance could be of state, it could be of implementation, and it could be of type.

When a class inherits something, it implies that it automatically gets that thing without you needing to explicitly define it in the class. This is very much like real life objects. When you say that a Poodle is a Dog, or a Beagle is a Dog, you implicitly know that a Poodle has a tail and that it barks. So, does a Beagle. You don't have to convey this information explicitly with a Poodle or a Beagle. In other words, both a Poodle and a Beagle inherit the tail and the barking behavior from Dog. Note that Poodle and Beagle do not contain a Dog. Poodle is a Dog. Beagle is a Dog. Being something is very different from containing something. If Poodles and Beagles contained a Dog, then both of them would always bark in exactly the same way, i.e., like a Dog. But you know that they don't bark the same way. Both of them do bark but they bark differently.

Inheritance of state

Only a class can contain state and therefore, only a class can extend another class. Furthermore, Java restricts a class from extending more than one class and that is why it is often said that Java does not support multiple inheritance. However, technically, it would be more precise to say that Java does not support multiple inheritance of state.

Inheritance of implementation

Before Java 8, only classes were allowed to have implementation. Starting with version 8, Java allows even interfaces to contain implementations in the form of "default" methods. Thus, a class can inherit implementation by extending a class and/or by implementing interfaces. This allows a class to inherit implementations from more than one types. This is one form of multiple implementation inheritance. However, due to the way default methods are inherited in a class, it is still not possible for a class to inherit more than one implementation of a method in Java. I will talk more about default methods later.

Inheritance of type

Java allows you to define a type using an interface as well as a class (enums and records are just special kinds of classes). Thus, a class can inherit behavior by extending a class and/or by implementing an interface. Since Java allows a class to implement multiple interfaces, it can be said that Java supports multiple inheritance of type.

13.1.2 Inheriting features from a class

To inherit features from another class, you have to extend that class using the **extends** keyword. For example, consider the following two classes:

```
public class Person{
  public String name;
  public String getName() {
     return name;
  }

  public static int personCount;
  public static int getPersonCount(){
     return personCount;
  }
}

public class Employee extends Person{
   public String employeeId;

   public static void main(String args[]){
      Employee ee = new Employee();
      ee.employeeId = "111";
      ee.name = "Amy";
      System.out.println(ee.getName());
   }

}
```

In the above code, Person is the **parent class** (also called the **super class** or the **base class**) and Employee is the **child class** (also called **sub class** or **derived class**). Since `Employee` extends `Person`, it automatically "gets" the `name` field as well the `getName` method. Thus, it is possible to access the `name` field and the `getName` method in an `Employee` object as if they were defined in the `Employee` class itself, just like the `employeeId` field.

Similarly, the `Employee` class also "gets" the static variable `personCount` and the static method `getPersonCount` from Person class. Thus, it is possible to access `personCount` and `getPersonCount` in an `Employee` class as if they were defined in the `Employee` class itself.

Members inherited by a class from its super class are as good as the members defined in the class itself and are therefore, passed on as inheritance to any subclass. For example, if you have a Manager class that extends an Employee class, the Manager class will inherit all members of the Employee class, which includes the members defined in Employee class as well as members inherited by Employee class from Person class.

```
public class Manager extends Employee {
   public String projectId;
   public static void main(String args[]){
      Manager m = new Manager();

      m.projectId = "OCPJP";

      m.employeeId = "111";
      m.name = "Amy";
      System.out.println(m.getName());
   }

}
```

There is no limit to how deep a chain of inheritance can go.

Inheriting constructors and initializers

Constructors are not considered members of a class and are therefore, never inherited. The following code proves it:

```
class Person{
  String name;
  Person(String name){
      this.name = name;
  }
}

public class Employee extends Person{

   public static void main(String args[]){
      Employee ee = new Employee("Bob");
   }

}
```

Employee does not inherit the String constructor of Person and that is why the compiler is not happy when you try to instantiate a new Employee using a String constructor.

Similarly, static and instance initializers of a class are also not considered to be members of a class and they are not inherited by a subclass either.

The java.lang.Object class

`java.lang.Object` is the root class of all classes. This means that if a class does not explicitly extend another class, it implicitly extends the `Object` class. Thus, every class inherits the members defined in the `Object` class either directly or through its parent. There are several methods in this class including the `equals` and the `toString` methods. It is because of the presence of these methods in the `Object` class that you can call them on any kind of object. I will talk more about them later.

Since Object is the root class of all classes, it is the only class in Java that does not have any parent.

Extending multiple classes

As I mentioned earlier, Java does not support multiple inheritance of state. Since classes contain state, it is not possible for a class to extend more than one class. Thus, something like the following will not compile:

```
public class Programmer{
}

//can't extend more than one class
public class Consultant extends Person, Programmer{

}
```

13.1.3 Inheritance and access modifiers

You might have noticed that I violated an important principle of OOP in Person and Employee classes. These classes are not well encapsulated. Recall that in the previous chapter, I explained why it is bad to have public fields. Here however, both the Person class, and since the Employee class extends Person, the Employee class have a field that is exposed to the whole world. As per the principle of encapsulation, I should make the name field private. However, if I make name private, Employee fails to compile with the following error message:

```
Error: name has private access in Person
```

How come, you ask? It turns out that access modifiers greatly impact inheritance. In fact, only those members that are visible to another class as per the rules of access modifiers are inherited in a subclass. Remember that a private member is not visible from anywhere except from the declaring class itself. Thus, a subclass cannot inherit a private member of the super class at all. That is why Employee will not inherit name from Person if you make name private.

With the above rule in mind, let us examine the impact of access modifiers on inheritance.

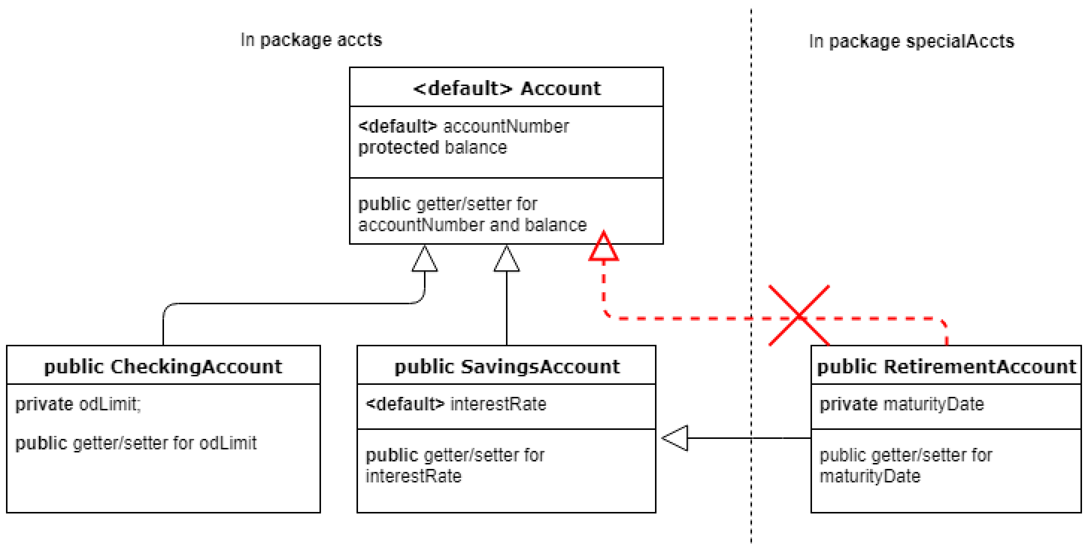
private - As explained above, private members are not inherited by a subclass.

default - Since a member with default access is visible only to a class in the same package, it can only be inherited in the subclass if the subclass belongs to the same package.

protected - This access modifier is actually built to overcome the restriction imposed by the default modifier. Recall that in addition to being visible to all classes of the same package, a protected member of a class is visible to another class that belongs to a different package if that class is a subclass of this class. Thus, even a subclass that belongs to another package inherits protected members of the super class.

public - public members are always inherited by a subclass.

For example, let's say you are designing an application that has three kinds of accounts: **CheckingAccount**, **SavingsAccount**, and **RetirementAccount**. All three classes need to have an account number and methods for checking balance. Besides these common features, each of these account types also have features that are specific only to that account type. The CheckingAccount has an **overdraftLimit**, the SavingsAccount has an **interestRate**, and the RetirementAccount has a **maturityDate**. The figure below shows the hierarchy of the classes.



Inheritance with access modifiers

Since all accounts are to have `accountNumber` and `balance`, I have put these members in a common super class called `Account`. The following is the code for these classes:

```
package accts;
class Account{
   int accountNumber;
   protected double balance;

   //public getter and setter methods for the above fields
}

package accts;
public class CheckingAccount extends Account{
   private double odLimit;
   //public getter and setter methods for odLimit
}
```

```
package accts;
public class SavingsAccount extends Account{
   double interestRate;
   //public getter and setter methods for interestRate
}


package specialAccts;
import accts.SavingsAccount;
public class RetirementAccount extends SavingsAccount{
   private String maturityDate;
   //getter and setter methods for maturityDate
   public static void main(String[] args){
       Account a = new Account(); //will not compile
       RetirementAccount ra = new RetirementAccount(); //valid

       ra.balance = 100.0; //valid
       ra.setAccountNumber(10); //valid
       ra.setInterestRate(7.0); //valid

       SavingsAccount sa = new SavingsAccount(); //valid
       sa.balance = 10.0; //will not compile
       ra.accountNumber = 10; //will not compile
       ra.interestRate = 7.0; //will not compile

   }
}
```

To make it interesting, I have put `RetirementAccount` into a different package than the rest of the classes. Let us now see the impact of the various access modifiers on these classes:

1. Since `Account` has default access, it is visible only in the `accts` package and thus `RetirementAccount` cannot access or extend the `Account` class.

2. Since `accountNumber` has default access, it is visible only in the `accts` package and thus, it will be inherited by all the subclasses of `Account` that belong to the same package. Thus, `accountNumber` will be inherited by `CheckingAccount` and `SavingsAccount`.

3. Since `balance` has protected access, it is visible in all the classes of the `accts` package and all the subclasses of the `Account` class irrespective of their package. Thus, it will be inherited by `CheckingAccount` and `SavingsAccount`. Since `Account` is not visible outside the `accts` package, making `balance` protected doesn't seem to make much sense **but it does**. Observe what happens with `RetirementAccount`.

4. Since `RetirementAccount` extends `SavingsAccount` but belongs to a different package, it only inherits the public and protected (but not default) members of `SavingsAccount`. This means, the protected field `balance` is passed on to the `RetirementAccount` as well. Since the `interestRate` field of `SavingsAccount` has default access, it is not visible in `RetirementAccount` and therefore, is not inherited by `RetirementAccount`.

5. Even though the `balance` field of `Account` is visible and inherited in `RetirementAccount`, `sa.balance` will still not compile because `RetirementAccount` class does not own `SavingsAccount`'s `balance` as explained in the previous chapter. In technical terms, code of `RetirementAccount` class is not responsible for the implementation of `SavingsAccount` class and so, `RetirementAccount` cannot access `SavingsAccount`'s `balance`. It can access its own `balance`, which it inherits from `SavingsAccount`, and which is why `ra.balance` compiles fine, but not `SavingsAccount`'s `balance`.

Note

Memory impact of access modifiers on sub classes

Even if a subclass does not inherit some of the instance variables of a superclass (because of access modifiers), a subclass object still consumes the space required to store all of the instance variables of the superclass. For example, when the JVM allocates memory for a `RetirementAccount` object, it includes space required to store all instance variables of `SavingsAccount` and `Account` irrespective of whether they are inherited in `RetirementAccount` or not. Thus, from a memory perspective, access modifiers have no impact.

So, on one hand, we are saying that a private member is not inherited by a subclass and on the other, we are saying that the subclass object does contain that member in its memory. This contradiction makes an intuitive understanding of the term "inheritance" difficult. Unfortunately, that is how it is. For example, in Section 8.2, The Java Language Specification says, "Members of a class that are declared private are not inherited by subclasses of that class." This shows that the JLS links the term inheritance with access modifiers and doesn't give any consideration to memory.

13.1.4 Inheritance of instance members vs static members

Inheritance of instance variables vs static variables

In an earlier example, you saw that the `Employee` class inherits instance as well as static variables of its superclass `Person`. However, there is a fundamental difference between the way instance and static variables are inherited. This difference is highlighted in the following code:

```java
public class Person{
  public String name;

  public static int personCount;
}

public class Employee extends Person{
   //inherits instance as well as static fields of Person
}

class TestClass{
   public static void main(String[] args){

       Person p = new Person();
       Employee e = new Employee();
       p.name = "Amy";
       e.name = "Betty";
       System.out.print(p.name+" ");
       System.out.println(e.name);

       Employee.personCount = 2;
       System.out.print(Person.personCount+" ");
       System.out.println(Employee.personCount);
   }
}
```

The above code generates the following output:

```
Amy  Betty
2 2
```

Observe that it prints different values for `name` - one for each object, but same value for `personCount`. What this means is that both the objects got their own personal copy of `name`, which they were able to manipulate without affecting each other's copy but the static variable `personCount` was shared by the two classes. When the code updated `personCount` of `Employee`, `personCount` of `Person` was updated as well. In fact, `Employee` did not get its own copy of `personCount` at all. It merely got access rights to `Person`'s `personCount`.

Inheritance of instance methods vs static methods

Since methods don't consume any space in an object (or a class), there is just one copy of a method anyway, irrespective of whether it is an instance method or a static method.

Conceptually, however, the difference that I highlighted above for variables also exists for methods. Conceptually, a subclass inherits its own copy of an instance method. This is proven by the fact that a subclass can **completely replace** the behavior of an inherited instance method for objects of the subclass by "**overriding**" it with a new implementation of its own without affecting the behavior of the superclass's implementation for objects of the superclass. On the other hand, a subclass merely gets access rights to a static method of its superclass and so, the subclass cannot change the behavior of the inherited static method. The subclass can "**hide**" the behavior of the superclass's static method by providing a new implementation but it cannot replace the super class's method.

Overriding and **hiding** are technical terms with precise meanings and are closely related to **polymorphism**. Their understanding is crucial for the exam as well as for being a good Java developer. I will dig deeper into these terms after we go over the nuts and bolts of extending classes and implementing interfaces.

13.1.5 Benefits of inheritance

First and foremost, Inheritance allows you to group classes by having them extend a common parent class. Functionality that is common to all such classes need to be defined only once in the parent class. For example, in the class hierarchy consisting of the **Account**, **CheckingAccount**, and **SavingsAccount** classes that I used at the beginning of this chapter, I put common functionality such as account number and method for checking balance in a parent class named **Account** and I put unique features of each of these account types in separate sub classes.

There are several advantages with this approach. Let's examine them one by one.

Code reuse

You can write logic for common fields only in one place and share that logic with all classes. In the above example, there is only one copy of the code to manage the account number. Since this logic is in the parent class, it is automatically inherited by all the child classes.

Having a common base class also makes it easy to add new functionality or modify the existing functionality that is common to all classes. This makes the code more extensible and maintainable overall.

Information hiding

Isolating the common features to a common parent class helps you hide the features of individual sub classes to processes that don't need to know of those features. As discussed before, more exposure means more risk of inadvertent dependency, i.e., tighter coupling, which is not desirable. For example, if you have a process that loops through all accounts and prints their account numbers and balances, then this process doesn't need to know about overdraft limit, interest rate, or maturity date. If you just pass an array of Accounts to this process, that should be enough. Something like this:

```java
public class DumpAccountInfo{
  public static void printAccounts(Account[] accts){
    for(Account acct :  accts){
       System.out.println(acct.getAccountNumber()+" "+acct.getBalance());
    }
  }
}
```

Of course, the array will include all kinds of **Account** objects (i.e. objects of sub-classes of Account class) but the process will only see them as **Account** objects and not as **CheckingAccount**, **SavingsAccount**, or **RetirementAccount** objects. The above code doesn't care whether an Account object is really a SavingsAccount or a CheckingAccount. It just prints the account number and its balance irrespective of what kind of account it is. This is possible only due to inheritance. Without the common parent class, you would have to have three different methods - one for each of the three account types - to print this information.

Polymorphism

Finally, and most importantly, inheritance makes polymorphism possible. Polymorphism is a topic in its own right and I will discuss it soon in a section of its own.

13.2 Using super and this to access objects and constructors

13.2.1 Object initialization revisited

Recall that in the "Create and overload constructors" section of the "Working with Methods and Encapsulation" chapter, I talked about the four steps that a JVM takes while instantiating a class. Let's see how these steps are impacted when there is inheritance involved.

1. The first step was to load and initialize the class if it is not already loaded and initialized.

2. The second step was to allocate the memory required to hold the instance variables of the object in the heap space. Since the instance variables defined in a super class are also included in object of a subclass (whether they are accessible to the subclass or not is a different matter), the memory allocated by the JVM for a subclass object must include space for storing instance variables of the super class as well.

3. The third step was to initialize these variables to their default values. This means the inherited variables also need to be initialized to their default values.

4. The fourth step was to give that instance an opportunity to set the values of the instance variables as per the business logic of that class by executing code written in instance initializers and constructors. This step gets a little more complicated when the class extends another class. Remember that the whole purpose of inheriting features (i.e. variables as well as methods) of a superclass is for the subclass to be able to use those features! It can use these features even at the time of its initialization. But to be able to do that, those features have to be initialized first. This means that subclass cannot initialize its own features unless features of its superclass have been initialized.

You can see where this is going. The superclass cannot be initialized before the superclass of the superclass is initialized and so on until there is no super class left. This chain of initialization will stretch until the Object class because Object is the root class of all classes and it is the only class that has no super class.

You have seen that the initialization is triggered when you try to create an instance of that class using the new keyword. When the JVM encounters the new keyword applied to a class, it invokes the appropriate constructor of that class. But as discussed above, the execution of this constructor cannot proceed until the initialization of the superclass is complete.

13.2.2 Initializing super class using "super"

To ensure the initialization of the fields inherited from the superclass, a constructor must first invoke exactly one of its super class's constructors. This is done using the **super(<arguments>**) syntax. For example:

```
class Person{
  String name;
  Person(String name){
      this.name = name;
  }
}

public class Employee extends Person{
   public Employee(String s){
     super(s);
   }
   public static void main(String args[]){
      Employee ee = new Employee("Bob");
   }

}
```

The question that you might ask at this point is what about `Person` class. This class has no extends clause and that means it implicitly extends `java.lang.Object`. So where is the call to `Object`'s constructor in `Person`'s constructor?

Good question. The call to the super class's constructor is so important that if a class's constructor doesn't call any of the super class's constructors explicitly, the compiler automatically inserts a call to the super class's default constructor in the first line of that constructor. The compiler doesn't check the constructors that the super class has. It just assumes the presence of the default constructor and inserts a call to that constructor. Thus, basically, `Person`'s constructor is modified by the compiler as follows:

```
class Person{
  String name;
  Person(String name){
      super(); //<-- inserted automatically by the compiler
      this.name = name;
  }
}
```

This means that `Object`'s no-args constructor will be invoked first before proceeding with the execution of `Person`'s constructor. Armed with this knowledge, let us see what happens when I modify `Employee` as follows:

```
public class Employee extends Person{
   public Employee(String s){
      name = s;
   }
   public static void main(String args[]){
      Employee ee = new Employee("Bob");
   }

}
```

Any thoughts? That's right. It won't compile. Since Employee's constructor doesn't call super class's constructor explicitly the compiler inserts a call to super(); on its own. This means Employee's constructor really looks like this:

```
public Employee(String s){
      super(); //inserted automatically by the compiler
      name = s;
   }
```

But Person does not have any no-args constructor! Recall that a default no-args constructor is provided by the compiler to a class only if the class does not define any constructor at all. In this case, Person does define a constructor explicitly and so the compiler does not insert a no-args constructor in the Person class. Thus, Employee will fail to compile.

Let me make one more change. What if I remove all constructors from Employee class?

```
public class Employee extends Person{
   public static void main(String args[]){
      Employee ee = new Employee();
   }

}
```

Again, recall the rule about the default constructor. Since Employee does not define any constructor explicitly, the compiler inserts a no-args constructor automatically, which looks like this:

```
public class Employee extends Person{
   public Employee(){  //inserted by the compiler
       super(); //inserted by the compiler
   }
   public static void main(String args[]){
      Employee ee = new Employee();
   }

}
```

Observe that this default no-args constructor also contains a call to super(); (See how important invoking a super class's constructor is?). But as discussed above, Person doesn't have a no-args constructor and therefore, the above code will not compile!

Since an object can be initialized only once, call to super(<arguments>); can also be made only once. If you try to call it more than once, the code will fail to compile.

Invoking another constructor using `this(<arguments> )`

In the previous chapter, you saw how a constructor can invoke another constructor of the same class using the `this(<arguments> );` syntax. Recall that invocation of another constructor is only allowed if it is the first statement of a constructor. But this poses a small problem. How can both - `this(<arguments> );` and `super(<arguments> );`, be the first statement of a constructor at the same time? Well, they can't be. In fact, Java allows only one of the two statements in a constructor. In other words, if you call `this(<arguments> );` then you can't call `super(<arguments> );` and if you call `super(<arguments> );`, you can't call `this(<arguments> );`.

Note that this does not violate our original premise that the super class's features have to be initialized first before initializing subclass's features. If you call `this(<arguments> );`, you don't need to call `super(<arguments> );` anyway because the other constructor would have called `super(<arguments> );` and initialized the super class's features. If you call `super(<arguments> );`, then by prohibiting you from calling `this(<arguments> );`, Java prevents the invocation of the super class's constructor more than once.

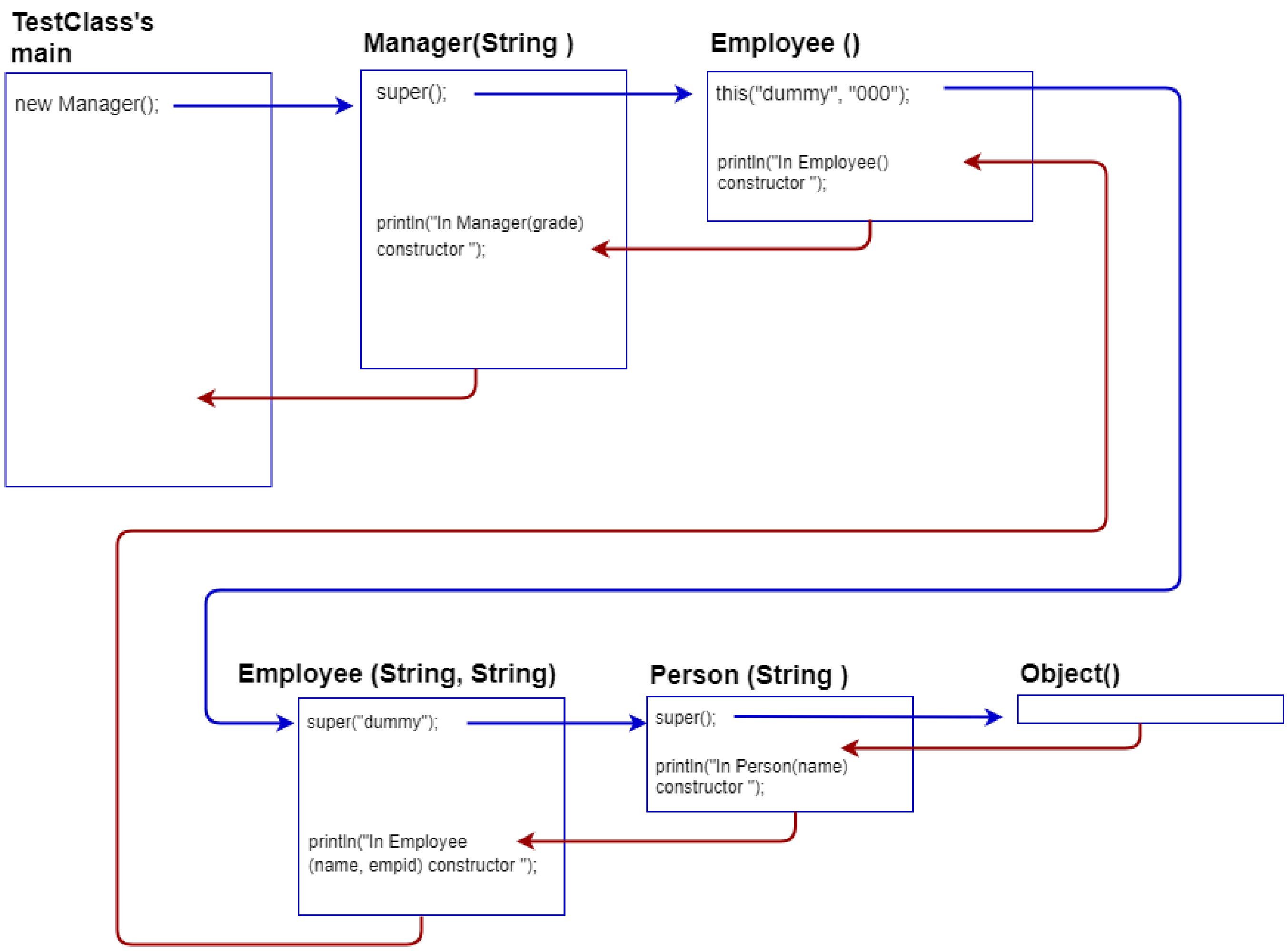
Based on the above discussion, can you determine the output generated when `TestClass` is run from the command line:

```java
class Person{
  String name;
  Person(String name){
     System.out.println("In Person's constructor ");
     this.name = name;
  }
}
class Employee extends Person{
  String empId;
  Employee(){
    this("dummy", "000");
    System.out.println("In Employee() constructor ");
  }

  Employee(String name, String empId){
    super(name);
    System.out.println("In Employee(name, empid) constructor ");
  }
}

class Manager extends Employee{
  String grade;
  Manager(String grade){
    System.out.println("In Manager(grade) constructor ");
    this.grade =  grade;
  }
}

class TestClass{
  public static void main(String[] args){
    Manager m = new Manager("A");
  }
}
```

In TestClass's main, an object of Manager class is being created using the only constructor it has. This constructor does not invoke superclass's constructor explicitly, therefore, the compiler inserts a call to `super();` as the first statement of this constructor. This takes us to the no-args constructor of Employee. Employee's no-args constructor invokes the two argument constructor of the same class explicitly using `this("dummy", "000")`. The two arguments constructor of Employee invokes its superclass's constructor using `super(name)`. This takes us to the Person's single argument constructor. Inside the `Person(String )` constructor, there is no explicit call to its superclass's constructor. So the compiler inserts `super();` as the first statement. This invokes `java.lang.Object`'s no-args constructor because Person implicitly extends `java.lang.Object`. Since Object class is the root class, this is the last constructor in the chain. Object's constructor doesn't produce any output.
Now, we have to start unwinding this chain of calls. After finishing the execution of Object's constructor, the control returns to next statement in Person's constructor, which is `System.out.println("In Person's constructor ")`. Thus, the first line on the output is `"In Person's constructor"`. Next, the name argument is assigned to the name instance variable and the call returns back to the Employee's constructor. The next statement in this constructor is `System.out.println("In Employee(name, empid) constructor ");`, which prints `"In Employee(name, empid) constructor "` and the control goes back to Employee's no-args constructor, which prints `"In Employee() constructor "`. The control returns to Manager's single argument constructor, which prints, `"In Manager(grade) constructor "`. This completes the chain of constructor invocation executed during creation of a Manager object. Thus, the output is:

```
In Person's constructor
In Employee(name, empid) constructor
In Employee() constructor
In Manager(grade) constructor
```

This chain of calls is illustrated in the following figure.


TestClass's main
new Manager();
Manager(String)
super();
println("In Manager(grade) constructor ");
Employee ()
this("dummy", "000");
println("In Employee() constructor ");
Employee (String, String)
super("dummy");
println("In Employee (name, empid) constructor ");
Person (String)
super();
println("In Person(name) constructor ");
Object()

Constructor chain

13.2.3 Using the implicit variable "super"

When a method is overridden by a subclass, it is impossible for any unrelated class to execute the super class's version of that method on a subclass object. However, the subclass itself can access its super class's version using an implicit variable named **"super"**. Every **instance method** of a class gets this variable. It is very much like the other implicit variable that you saw earlier, i.e., **"this"**. While `this` refers to the current object, `super` refers to the members that this class inherits from its parent class. The following code shows how it is used:

```
class InterestCalculator{
   public double computeInterest(double principle, int yrs, double rate){
       return principle*yrs*rate;
   }
}

class CompoundInterestCalculator extends InterestCalculator{
   public double computeInterest(double principle, int yrs, double rate){
        return principle*Math.pow(1 + rate, yrs) - principle;
   }

   //invoke this method from TestClass's main
   public void test(){
       double interest = super.computeInterest(100, 2, 0.1);
       System.out.println(interest); //prints 20.0

       interest = computeInterest(100, 2, 0.1);
       System.out.println(interest); //prints 21.000000000000014
   }
}
public class TestClass{
    public static void main(String[] args){
          CompoundInterestCalculator cic = new CompoundInterestCalculator();
          cic.test();
    }
}
```

The call to `super.computeInterest(100, 2, 0.1);` in the above code causes `InterestCalculator`'s version to be invoked while the call to `computeInterest(100, 2, 0.1)` causes `CompoundInterestCalculator`'s version to be invoked

The variable `super` is available only in instance methods of a class and can be used to access **any inherited member (static as well as instance)** of the super class. Are you now thinking about forming a chain of supers? Something like `super.super.someMethod()`? No, you can't do that because there is no member named super in the super class. In fact, you can't define a variable or a method named super in any class. This begs the question that how can a class access the grandparent's version of a method that is defined in its grand parent as well as in its parent class? For example, what if you have the following grandchild class and you want to access `InterestCalculator`'s version of `computeInterest` in this class:

```
class SubCompoundInterestCalculator
           extends CompoundInterestCalculator{
   public void test(){
       //invokes CompoundInterest's computeInterest
       super.computeInterest(100, 2, 0.1);

       //can't do super.super
       //super.super.computeInterest(100, 2, 0.1);
   }
}
```

The answer is you can't. There is no way to go more than one level higher. In other words, a subclass can only access its immediate super class's version of methods using super.

Note that this limitation applies only to methods that are overridden by the parent class. If the parent class does not override a method of its super class, then the parent class inherits that method from the grandparent and the child class can access the grandparent's version of the method as if it were the parent class's version.

13.2.4 Order of initialization summarized

You have now seen all the steps, from loading the class to executing a constructor of the class, that a JVM takes to create an object of a class. Here is the order of these steps for quick reference:

1. If there is a super class, initialize static fields and execute static initializers of the super class in the order of their appearance (performed only once per class)
2. Initialize static fields and execute static initializers of the class in the order of their appearance (performed only once per class)
3. If there is a super class, initialize the instance fields and execute instance initializers of the super class in the order of their appearance
4. Execute super class's constructor
5. Initialize the instance fields and execute instance initializers of the class in the order of their appearance
6. Execute class's constructor

The following example illustrates the above steps:

```java
class A {
   static{ System.out.println("In A's static initializer"); }

   A(){ System.out.println("In A's constructor"); }

   { System.out.println("In A's instance initializer"); }
}

public class B extends A {
   static{ System.out.println("In B's static initializer"); }

   { System.out.println("In B's instance initializer"); }

   B(){ System.out.println("In B's constructor"); }

   public static void main(String[] args) {
      System.out.println("In B.main()");
      B b = new B();
      B b2 = new B(); //creating B's object again
   }
}
```

The following output is generated when class B is run:

```
In A's static initializer
In B's static initializer
In B.main()
In A's instance initializer
In A's constructor
In B's instance initializer
In B's constructor
In A's instance initializer
In A's constructor
In B's instance initializer
In B's constructor
```

You should observe the following points in the above output:

1. static initializers of A and B are invoked before B's main() is executed because the JVM needs to load the class before it can invoke a method on it.
2. Static initializers of A and B are invoked only once even though two objects of B are created.
3. Instance initializer of A is executed before A's constructor even though the instance initializer appears after the constructor in A's code.
4. While creating an object of B, A's instance initializer and constructor are invoked before B's instance initializer and constructor.
5. Instance initializer and constructor of A are invoked each time an object of B is created.

Exam Tip

Order of execution of various members of a class and its superclass can sometimes get very difficult to trace. However, the exam does not try to trick you too much on this topic. If you can follow the above example, you will not have any problem with the code in the real exam.

13.3 Using abstract, final, and sealed classes

13.3.1 Using abstract classes and abstract methods

I talked about **abstract class** as a **type of type** briefly in the "Kickstarter for Beginners" chapter. An abstract class is used when you want to capture the common features of multiple related object types while knowing that no object that exhibits only the features captured in the abstract class can actually exist. For example, if you are creating an application that deals with cats and dogs, you may want to capture the common features of cats and dogs into a class called Animal. But you know that you can't have just an Animal. You can have a cat or a dog but not something that is just an animal without being a cat, a dog or some other animal. In that sense, Animal is an **abstract concept**. Furniture is another such concept. If you want to have a Furniture, you have to get a chair or a table because Furniture doesn't exist on its own. It is an abstract concept.

Abstract classes are used to capture such abstract concepts. An abstract class can be defined by using the keyword `abstract` in its declaration. For example, the following is how you can define `Furniture`:

```
abstract class Furniture{
  private String material;
  //public getter and setter
}
```

Observe that `Furniture` is pretty much like a regular class with public and private fields and methods. However, since it is declared abstract, it cannot be instantiated. Thus, if you try to do `new Furniture()`, the compiler will generate an error saying, "`Furniture is abstract; cannot be instantiated`".

But Chair and Table are concrete concepts. They exist on their own. Since both of them are types of Furniture, you can model them as two classes that extend Furniture.

```
class Chair extends Furniture{
   //fields and methods specific to Chair
}
class Table extends Furniture{
   //fields and methods specific to Table
}
```

Since `Chair` and `Table` extend `Furniture`, they will inherit all of the features of `Furniture` just like any subclass inherits all of the features of its super class.

Classes that are not abstract are also called "concrete" classes because they represent real objects.

Adding an abstract method to an abstract class

Let's say every piece of furniture in a store has to provide a method for its assembly. Since the steps to assemble a chair will be different from steps to assemble a table, we can't include the steps for their assembly in Furniture class. In other words, what we are saying is that the declaration of the method to assemble is common to all furniture but their implementation is unique to each type of furniture. Java allows you to capture declaration without implementation in the form of an **abstract method**. An abstract method is declared by applying the **abstract** keyword to the method declaration. It must not have a method body either. For example,

```
abstract class Furniture{
  private String material;
  //public getter and setter

  public abstract void assemble(); //ends with a semicolon, no opening and closing
    curly braces
}
```

The benefit of adding abstract methods to a class is that other components can work with all objects uniformly. For example, a class that assembles furniture doesn't need to worry about whether it is assembling a Chair or a Table. It can simply call `assemble()` on any kind of furniture that it gets. Something like this:

```
class FurnitureAssembler{
   public static void assembleAllFurniture(Furniture[] allFurniture){
      for(Furniture f : allFurniture){
         f.assemble();
      }
   }
}
```

But adding an abstract method to the parent class has an impact on child classes in that the child classes are now obligated to provide an implementation for the abstract method. For example, we will now need to update the code for our Chair and Table classes as follows:

```
class Chair extends Furniture{
   //fields and methods specific to Chair
   public void assemble(){
     System.out.println("Assembling chair!");
   }
}
class Table extends Furniture{
   //fields and methods specific to Table
   public void assemble(){
     System.out.println("Assembling table!");
   }
}
```

This makes sense because if a piece of furniture actually exists then we must be able to assemble it as per our definition of `Furniture`. The only reason why a subclass of `Furniture` may be unable to have a method to assemble is if that subclass itself is abstract! For example, what if you have a subclass of `Furniture` called `FoldingFurniture`? `FoldingFurniture` is an abstract concept and can be modeled as an abstract class that extends `Furniture`. Since it is abstract, it can get away without implementing the `assemble` method. But any concrete subclass of `FoldingFurniture` will have to provide an implementation for the `assemble` method. If you are confused about how the code for `FurnitureAssembler` can invoke the `assemble` methods of Chairs and Tables when it does not even know about their existence, don't worry. I will talk about it in detail soon in the section about **polymorphism**.

13.3.2 Using final classes and final methods

You have seen the usage of **final** keyword while declaring variables. It implies that the variable is a **constant**, i.e., the value of the variable does not change throughout the execution of the program. The final keyword can also be applied to classes and methods (static as well as instance) and implies something similar. It means that the behavior of the class or the method cannot be changed by any subclass.

Observe that **final** is diametrically opposite of **abstract**. You make a class or a method abstract because you want a subclass of that class to provide different implementation for that class or that method as per the business logic of the subclass. On the other hand, you make a class or a method final because you don't want the behavior of the class or of the method to be changed at all by a subclass. In technical terms, we say that a final class cannot be **subclassed** and a final method cannot be **overridden**. I will discuss overriding in detail soon but you can now see why **abstract** and **final** cannot go together.

A final class doesn't necessarily have to have a final method. But practically, since a final class cannot be extended, none of its methods can be overridden anyway. The important thing is, a final class cannot have an abstract method. Why? Because there is absolutely no possibility of that abstract method ever getting implemented. It follows that if a final class inherits an abstract method from any of its ancestors, it must provide an implementation of that method.

Here is an example of a final class:

```
public final class Chair extends Furniture{

   //since assemble is an abstract method in Furniture, it must be implemented in this
    class
   public void assemble(){
       System.out.println("Assembling Chair");
   }
}
```

Observe that `Chair` hasn't declared any method as final explicitly (it **can** do so, but that would be redundant) and that it must implement the abstract method `assemble` that it inherits from its superclass.

A **final** method is defined similarly:

```
public abstract class Bed extends Furniture{

   //not necessary to implement assemble() because Bed is abstract

   public final int getNumberOfLegs(){
      return 4;
   }

   public static final void make(Bed b){
      System.out.println("Making Bed: "+b);
   }
}

public final class DoubleBed extends Bed{

   //must implement assemble because DoubleBed is not abstract
   public void assemble(){
       System.out.println("Assembling DoubleBed");
   }

   //Can't override getNumberOfLegs here because it is final in Bed

   final int getHeight(){
     return 18;
   }
}
```

Observe that even though `Bed` has a final method, `Bed` itself is not only not final, it is abstract. I made it abstract just to illustrate that any class irrespective of whether it is final, abstract, or neither, can contain a final method.
Java standard class library includes several final classes. Most important of them is `java.lang.String` class, which you have already seen in the chapter on String API.

13.3.3 Valid combinations of access modifiers, abstract, final, and static

Impact of access modifiers on abstract and final

The usage of abstract makes sense only in the presence of inheritance. If something is never inherited then there is no point in talking about whether it is abstract or final. You also know that a private member is never inherited. Based on this knowledge can you tell what will happen with the following code:

```
abstract class Sofa {
   private abstract void recline();
}
```

That's right. The compiler will reject the above code with the message, `"illegal combination of modifiers: abstract and private"`. Because a private method is never inherited, there is no way any subclass of `Sofa` can provide an implementation for this abstract method.
What if the `recline` method were protected or default? It would have been ok in that case because it is possible for a subclass to inherit methods with protected and default access.

Let us now see what happens when you make a private method final:

```
class Sofa {
   private final void recline(){
   }
}
```

What do you think will happen? It won't compile, you say? Wrong! If a method cannot be inherited at all by any subclass, is it even possible to override it? No. Thus, a private method is, practically, already final! The compiler accepts the above code because there is no contradiction between private and final. Marking a private method as final is, therefore, not wrong but rather redundant.

abstract and static

The abstract keyword is strictly about overriding and static methods can never be overridden. Therefore, abstract and static cannot go together. Thus, the following code will not compile:

```
abstract class Bed extends Furniture{
   static abstract void getWidth();
}
```

The compiler will generate an error message saying, `"illegal combination of modifiers: abstract and static"`.

final and static

Although a static method cannot be overridden, it is nevertheless inherited by a subclass and can be hidden by the subclass. I will discuss the difference between overriding and hiding in detail later but for now, just remember that final prevents a subclass from hiding the super class's static method. Thus, the following code will compile fine:

```
class Bed{
   static final int getWidth(){
      return 36;
   }
}
```

But the following will not because `getWidth` is final in `Bed`:

```
class DoubleBed extends Bed{
   static int getWidth(){ //will not compile
     return 60;
   }
}
```

Summary of the application of access modifiers, final, abstract, and static keywords

Based on the previous discussion, let me summarize the rules of abstract, concrete classes, and final classes and methods:

1. An **abstract class** doesn't necessarily have to have an abstract method but if a class has an abstract method, it must be declared abstract. In other words, a concrete class cannot have an abstract method.

2. An abstract class cannot be instantiated irrespective of whether it has an abstract method or not.

3. A **final class** or a **final method** cannot be abstract.
4. A **final class** cannot contain an **abstract method** but an **abstract class** may contain a **final method**.
5. A **private** method is always **final**.
6. A **private** method can never be **abstract**.
7. A **static method** can be **final** but can never be **abstract**.

Exam Tip

The exam tests you on various combinations of access modifiers and final, abstract, and static keywords as applied to classes and methods. You should be very clear about where you can and cannot apply these modifiers.

The exam, however, does not try to trick you on the order of access modifiers and final and abstract keywords. For example, you will not be asked to pick the right declaration out of, say, the following three:

```
public abstract void m();
final public void m(){ }
abstract protected void m();
```

As far as the compiler is concerned, the return type must be specified immediately before the method name. It doesn't care about the order of the other modifiers. Thus, all of the above three methods are valid.

Nevertheless, you should know that, conventionally, the sequence of modifiers in a method declaration is as follows:

`<access modifier> <static> <final or abstract> <return type> methodName(<parameter list> )`

If you ever get confused about their order, just recall the signature of the main method as:

`public static final void main(String[] args)`.

Of course, the main method does not have be final. I have mentioned it above to show the position of final in a method declaration.

13.3.4 Using sealed classes and interfaces

You saw earlier that a non-final class can be extended by any other class, provided it is accessible. For example, once you create a public class `Furniture`, you don't care who extends it. Anyone can create a subclass of Furniture and reuse the functionality that you provided in the Furniture class. In general, that is a good thing, but not always.

In OO development, there is often a need to model an entity such that it would be accessible to all but extensible only by a selected few. In other words, you want a restricted hierarchy of classes. For example, let's say you are developing a library for valuation of financial instruments such as stocks and bonds. You want to be able refer to all financial instruments using a common base class so that you could invoke methods that are common to all financial instruments without worrying about the exact instrument. However, as the author of the library, you also know that some functionality of your library, or to put it in concrete terms, some methods in your library, cannot deal will all kinds of financial instruments. In fact, you know that many methods of your library can deal with only two kinds of financial instruments - Stock and Bond. You don't want a new kind of financial instrument created without your knowledge because, may be, a method in the library has a switch like this:

```
FinancialInstrument fi = //get it somehow
switch(fi){
  case fi instanceof BOND b -> valueBond(b);
  case fi instanceof Stock s -> valueStock(s);
  default -> throw new RuntimeException("Can't deal with this financial instrument!");
}
```

The default block is not really adding any functionality but, as you learned earlier, it is necessary in the above switch to make it exhaustive. If anyone creates a new type of FinancialInstrument and passes it to this code, they will get an exception at run time. This is a waste of time for them. Ideally, they should be made aware at development time itself that they should not be creating a subclass of FinancialInstrument because your library will not be able to handle it. Well, this is where sealed classes come into picture.

Note

Extracting business logic from individual classes and putting it into one common class using a switch is a variation of the classic visitor pattern. It is not required for the exam but you should still read more about this pattern because it is often discussed in technical interviews.

Creating sealed classes

You could define your FinancialInstrument class like this:

```
public abstract sealed class FinancialInstrument permits Stock, Bond{
}
```

The two new keywords used above are sealed and permits. Using these two, you are telling the compiler that no class, except Stock and Bond, can extend FinancialInstrument. As soon as you do that, the compiler will want to make sure that Stock and Bond classes actually exist and that they extend FinancialInstrument. Not only that, it will also want to ensure that their subtypes (if they can exist) are also known at compile time. So, you will need to create those classes as well. Once you do that, you can remove the default block from the switch code written above because now the compiler knows that there are only two possible subclasses of FinancialInstrument and both of them are being handled in the switch and the switch is exhaustive without the default block.

Let me now go over the **rules for creating sealed classes** quickly:

1. Only **classes** and **interfaces** can be sealed. Records are always final, so sealing them doesn't make sense. Enums are also always final, except when an enum constant has a class body (not important for the exam), in which case they are implicitly sealed.

2. All **direct subclasses** of a sealed class must themselves also be declared **`final`**, **`sealed`**, or **`non-sealed`**. The usage of `final` and `sealed` restrict the inheritance hierarchy at development time itself and allow the compiler to make assumptions about it, so it is obvious why a subtype has to be either of them. But `non-sealed` is a bit tricky. Marking a class `non-sealed` frees it from the clutches of a bounded class hierarchy, i.e., it brings things back to normal. Once you make a class `non-sealed`, any class is now free to extend it without being `sealed` or `final`.

3. All **direct subclasses** of the sealed class must be listed in the **permits** clause **except** when they are defined in the **same source file**. If all the subtypes are coded in the same source file, the compiler infers the permits clause on it own and saves the developer from typing a few keystrokes without affecting readability of the code much.

4. All **direct subclasses** of the sealed class must belong to the same package as that of the sealed class **except** when they are part of a same **named module** (in which case, they may belong to different packages). Since this rule is regarding modules, which I haven't discussed yet, you may come back to it later but it is meant to ensure that all of the direct subtypes of a sealed type come from the same provider/developer. Since a named module is sure to have classes from the same provider, Java allows a sealed type to have its direct subtypes in any package of that module. However, packages in an unnamed module may come from multiple providers and since it does not make sense for a sealed class to allow other providers to create its subclasses, Java prohibits immediate subclasses from belonging to a different package.

Using sealed interfaces

Sealing an interface limits the types that can extend or implement that interface. Since records and enums are also allowed to implement interfaces, the permits clause of a sealed interface can, therefore, include all types, i.e., classes, interfaces, enums, and records. The rules for a sealed interface are the same as those for a sealed class but since an interface cannot be final, a sub-interface of a sealed interface must be either sealed or non-sealed.

The above rules are simple but they allow creation of many legal and illegal scenarios. The exam requires you to know and apply the above rules to identify valid type definitions. So, let us work out a few examples.

Example 1

```java
//in file FinancialInstrument.java
package instruments;
public sealed class FinancialInstrument permits Stock, Bond{ }
```

```
non-sealed class Stock { }
sealed class Bond extends FinancialInstrument permits GovtBond{ }
final class Option extends FinancialInstrument { }

//In file GovtBond.java
package safeinstruments;
final class GovtBond extends Bond{ }
```

The above code has the following issues:

1. `FinancialInstrument` is invalid because it lists a class (`Stock`) in its permits clause that does not extend `FinancialInstrument`.
2. `Stock` is invalid because it uses `non-sealed` modfier but does not extend any sealed class.
3. `Bond` is invalid because it permits `GovtBond`, which is not a member of the same package.
4. `GovtBond` is invalid because it extends a sealed class but is not in the same package as the sealed class. For the purpose of the exam, do not assume that the classes belong to a module if the problem statement does not explicitly specify it.
5. `Option` is interesting. This class is defined in the same file as its sealed superclass `FinancialInstrument`. If `FinancialInstrument` did not have a `permits` clause, `Option` would have been a valid subclass. However, `FinancialInstrument` does have a `permits` clause and `Option` is not in it and so, `Option` is invalid. This shows that if you specify the `permits` clause explicitly, you must list all the subclasses in it.

Example 2

```
//in file FinancialInstrument.java
public abstract sealed class FinancialInstrument permits Stock, Bond{ }
sealed class Stock extends FinancialInstrument{ }
non-sealed class Bond extends FinancialInstrument permits GovtBond{ }
final sealed class GovtBond extends Bond{ }
final class PStock extends Stock{ }
```

Let us analyze each definition:

1. It is ok for an `abstract` class to be `sealed`, so `FinancialInstrument` is valid.
2. A `sealed` class that does not have a valid subclass is invalid (it would be like a `final` class actually, but it is not allowed). However, the `permits` clause can be omitted if all of the subclasses of the sealed class are defined in the same source file. So, even though `Stock` does not have a `permits` clause, it is valid. If you comment out `PStock`, then `Stock` would become invalid.
3. A `non-sealed` class cannot have a `permits` clause, so, `Bond` is invalid .
4. Since a final class cannot have any subclass, it doesn't make sense to mark it `sealed`. Therefore, `GovtBond` is invalid.
5. There is no issue with `PStock` because it is defined in the same source file as its sealed superclass.

13.3.5 Quiz ✎

Q 1. Given the following definition:

```
sealed interface Reader permits SerialReader, Data{
  default String getHeader(){ return "Reader Header"; }
}
```

Which of the following set of definitions will compile without any error?
(Assume that all of the code is in a single source file.)

Select 2 correct options

A.

```
non-sealed interface SerialReader extends Reader{ }
final record Data(String s) implements Reader{ }
```

B.

```
sealed interface SerialReader extends Reader{ }
enum Data implements Reader{ STREAM, FILE}
```

C.

```
sealed interface SerialReader extends Reader{ }
final class Data implements SerialReader{ }
```

D.

```
interface SerialReader extends Reader{ }
interface Data extends Reader{ }
```

E.

```
non-sealed interface SerialReader extends Reader{ }
class Data implements Reader{ }
```

F.

```
non-sealed interface SerialReader extends Reader{ }
non-sealed interface Data extends Reader{ }
```

Answer
The correct answer is **A**, **F**.

As per the given definition, `Reader` is an interface. Therefore, `SerialReader` and `Data` can be interfaces, classes, records, or enums because all of them are allowed to implement interfaces. **Option A** is fine. Records are implicitly final and it is allowed to explicitly make them final. **Option B** is invalid because a sealed interface must have a valid subtype but no subtype is defined in this option. **Option C** is invalid because `Data` doesn't directly implement the `Reader` interface. It does so indirectly but, apparently, that doesn't count! **Option D** is invalid because sub-interfaces of a sealed interfaces must be declared sealed or non-sealed. **Option E** is invalid because `Data` is not sealed, non-sealed, or final. **Option F** is fine.

13.4 Enable polymorphism by overriding methods

13.4.1 What is polymorphism

In simple terms, **Polymorphism** refers to the ability of an object to exhibit behaviors associated with different types. In other words, if the same object behaves differently depending on which "side" of that object you are looking at, then that object is **polymorphic**. For example, if you model apples using an Apple class that extends a Fruit class, then an apple can behave as an Apple as well as a Fruit. Later on, if you have a RedApple class that extends the Apple class, then a red apple will behave as an Apple and a Fruit besides behaving as a RedApple. Similarly, an object of a StockPrice class that you saw earlier behaves as a Readable as well as a Movable besides behaving as a StockPrice! Thus, apples and stock prices are polymorphic objects.

It is very important to understand that the actual object doesn't change at all. The object itself always remains the same. A red apple will always be a red apple. It doesn't suddenly morph into an apple or a fruit. It has always been an apple and a fruit besides being a red apple. It follows that if an object doesn't already support a particular behavior, it won't suddenly start supporting that behavior. A red apple will never morph into a green apple no matter what you do.

Remember that when we talk about the behavior of an object in the context of polymorphism, we are essentially talking about its instance methods. We are not talking about its static methods because static methods define the behavior of a class and not the behavior of the object. (Yes, it is true that you can access a static method using an object reference instead of the class name but that is just a peculiarity of the Java language and has nothing to do with polymorphism). We are not talking about instance variables either because variables merely store data (either primitive data or references to other objects) and do not possess any behavior. Also, the variables would not be visible to any other class anyway if the class is well encapsulated.

Polymorphism in Java

Java allows objects to be polymorphic by letting a class extend another class and/or implement interfaces. Since every class in Java implicitly extends `java.lang.Object`, you can say that every object is polymorphic because every object exhibits the behavior of at least two classes (except an instance of `java.lang.Object`, of course).

Thus, an object can be as polymorphic as the number of classes it extends (directly as well as indirectly) and the number of interfaces it implements (directly as well as indirectly through ancestors).

Importance of Polymorphism

In the "Kickstarter for Beginners" chapter I talked about how a component can be easily replaced with another component if they promise to honor the same behavior. This is made possible only because of **polymorphism**. If an object is allowed to behave only as a single type, then you can never replace that object with another object of a different type.

For example, if all you want is a Fruit, then it shouldn't matter whether you are given an Apple or an Orange. Both of them promise to honor the behavior described by Fruit and are, therefore, equally acceptable fruits. The situation is illustrated by the following code:

```
abstract class Fruit{ //must be declared abstract because it has an abstract method
   abstract void consume();
}
class Apple extends Fruit{
   void consume(){
       System.out.println("Consuming Apple...");
   }
}
class Orange extends Fruit{
   void consume(){
       System.out.println("Consuming Orange...");
   }
}
class Person{
   void eatFruit(Fruit f){
      f.consume();
   }
}
class TestClass{
   public static void main(String[] args){
      Apple a = new Apple();
      Orange o = new Orange();
      Person p = new Person();
      p.eatFruit(a);
      p.eatFruit(o);
   }
}
```

Observe that a Person object has no knowledge about what it is eating. All it cares about is Fruit. As long as you pass it a Fruit, it is fine. You can easily pass an Orange to a Person instead of an Apple. This is possible only because Apple and Orange are polymorphic, they behave like a Fruit besides behaving like an Apple or an Orange respectively.

Without polymorphism, your code would look something like this:

```
class Apple{
   void consume(){
       System.out.println("Consuming Apple...");
   }
}
class Orange{
   void consume(){
       System.out.println("Consuming Orange...");
   }
}
class Person{
   void eatApple(Apple a){
      a.consume();
   }
   void eatOrange(Orange o){
      o.consume();
   }
}
class TestClass{
   public static void main(String[] args){
      Apple a = new Apple();
      Orange o = new Orange();
      Person p = new Person();
      p.eatApple(a);
      p.eatOrange(o);
   }
}
```

Observe that in the absence of a common `Fruit` class, `Person` must have two different methods - one for eating `Apple` and one for eating `Orange`. The user of the `Person` class, i.e., `TestClass`, also has to call two different methods. What if you wanted to feed a banana to a person? You would have to code a new method in `Person` to accept a `Banana` and also have to change the code in `TestClass`. And you know these are not the only kinds of fruit in the world, right? You get the idea. Imagine how many eat methods you would have to write!

But that is not the only problem. Let us say you code a method for each of the fruits that you know about in `Person` class and deliver this class to other people for use. What if you or someone else comes to know about an entirely new kind of fruit and wants to feed that fruit to a `Person` object? You can't change the code of the `Person` class at this stage because then you would have to make everyone using the old class get this new version of the `Person` class from you. Depending on the diversity of people in different projects, this could become a nightmare every time a new fruit is discovered.

If you think logically, it shouldn't matter to a Person if a new kind of fruit is discovered. As long as it is a fruit, a `Person` should be able to accept it. This is where Polymorphism shines. In our first design `Person`'s eat method accepted just a `Fruit`. If you find a new fruit, say Custard Apple (amazing fruit, btw), all you have to do is to define a `CustardApple` class that extends `Fruit` and pass objects of this class to `Person`'s `eatFruit(Fruit )` method. No change is required in the `Person` class. This is possible only because `CustardApple` honors the contract defined by `Fruit` by saying it extends `Fruit`.

Another important thing about Polymorphism is that it allows components to be switched not only at compile time but also at **run time**. You can actually introduce completely new classes that extend existing classes (or implement existing interfaces) into an application that is already running. An existing class or interface serves as a contract for a specific behavior and if a new class promises to honor this contract by extending that class or by implement that interface, then any code can make use of instances of this new class in places where it requires instances of the existing class or interface.

Always remember that the whole objective of polymorphism is to enable classes to become standardized components that can be easily exchanged without any impact on other components. The various seemingly confusing rules that you will soon see, only exist to achieve this goal. If you ever get confused about a particular rule, always think about how it affects the interoperability of one component with another.

Note

The ability to transparently replace an object with another is called "substitutability". The substitutability principle states that objects of a type may be replaced with objects of its subtype without altering any of the desirable properties of the program.

Polymorphism and Inheritance

As you saw above, methods of a class or an interface basically form a **contract** that a subclass or the implementing class promises to fulfill. When a class says that it extends another class or says that it implements an interface, it agrees to have the methods the superclass or the interface has. However, the exact things that these methods must do is not covered by this contract. In other words, the contract is not so detailed as to affect the internal logic of the methods. The internal logic of a method is only governed by an informal contract that is implicit in the name of a method. For example, if the superclass declares a method named `computeInterest`, it is reasonable to expect that the subclass‘implementation would compute some kind of interest in this method. This is not an issue with abstract methods because a subclass is required to implement abstract methods on its own, but it may be a problem in case of non-abstract methods. What if the superclass computes interest in one way but the subclass wants to compute interest in another way? If you think about it, this is an important aspect of componentizing a class. You want to componentize classes so that you could easily replace one component with another. But why would you replace one component with another if they behaved exactly the same? You wouldn't.

What you really want is the ability to replace a component with another component with which you can interact in the same way, but which behaves differently. This is exactly what polymorphism in Java achieves. Java allows a subclass to provide its own implementation of the method if it does not like the behavior provided by the inherited method. The technical term for this is **"overriding"**. A subclass is free to override a method that it inherits with its own implementation.

13.4.2 Overriding methods

As discussed earlier in this chapter, only **non-private instance methods** can be **overridden**. There are several rules that you need to follow while overriding a method. These rules govern the **accessibility**, **return type**, **parameter types**, and the **throws clause** of the method. The best way to understand and remember these rules is to keep in mind that these rules are there to make sure that an object of a class can be replaced with an object of a subclass without breaking existing code. The objective is to be able to replace one component with another transparently while giving flexibility to a subclass in implementing its methods.

Whenever you override a method in a subclass, think about what will happen to the code that depends on the superclass object and if you pass it a subclass object instead of the superclass object. If this replacement does not require the code to be recompiled, then you have overridden the method correctly. From this perspective, the following rules are applicable for method hiding (which happens for non-private static methods) as well.

Let me now list all of the rules and then I will show you the reasoning behind these rules with an example. (Remember that an "overridden method" means the method in the superclass and an "overriding method" method means the method in the subclass).

1. **Accessibility** - An overriding method must not be less accessible than the overridden method. This means that if the overridden method is `protected`, you can't make the overriding method default because default is less accessible than `protected`. You can make the overriding method more accessible. Thus, you can make it `public`.

2. **Return type** - The return type of the overriding method must either be the same as the return type of the overridden method or it must be a sub type. For example, if the return type of the overridden method is `Fruit`, then the return type of the overriding method can either be `Fruit` or any subclass of `Fruit` such as `Apple` or `Mango`. This is called the rule of "**covariant returns**".

   In the case of primitive return types, the Java Language Specification mandates that the return type of the overriding method must match exactly to the return type of the overridden method even though it does define supertype-subtype relationships between numeric types.

3. **parameters** - The list of parameters that the overriding method takes must match exactly to that of the overridden method in terms of types and order (parameter names don't matter). Indeed, if there is a difference between the types and/or order of parameters, it would not be an override but an overload. I have already discussed overloading of methods.

4. **throws clause** - An overriding method cannot put a wider exception (i.e. a superclass exception) in its throws clause than the ones present in the throws clause of the overridden method. For example, if the overridden method throws `IOException`, the overriding method cannot throw `Exception` because `Exception` is a superclass of `IOException`. The overriding method may throw a subclass exception such as `FileNotFoundException`.

   The overriding method cannot throw a new exception that is not listed in the throws clause of the overridden method either.

   The overriding method may decide to not have a throws clause altogether though.

   Note that this rule applies only to checked exceptions because these are the only ones the compiler cares about. There is no rule regarding unchecked exceptions.

Let's see how these rules works with respect to the following classes:

```
class InterestCalculator{
   Number computeInterest(double principle, double yrs, double rate) throws Exception {
         if(yrs<0) throw new IllegalArgumentException("yrs should be > 0");
         return principle*yrs*rate;
   }
}
class Account{
   double balance; double rate;
   Account(double balance, double rate){
      this.balance = balance;
      this.rate = rate;
   }
   double getInterest( InterestCalculator ic, double yrs ){
       try{
          Number n = ic.computeInterest(balance, yrs, rate);
          return n.doubleValue();
       }catch(Exception e){
           e.printStackTrace();
       }
       return 0.0;
   }
}
class AccountManager{
   public static void main(String[] args){
      Account a = new Account(100, 0.2);
      InterestCalculator ic = new InterestCalculator();
      double interest = a.getInterest(ic, 2);
      System.out.println(interest);
   }

}
```

We have an `Account` class that uses an `InterestCalculator` object to compute interest. We also have an `AccountManager` class that manages accounts and also the way accounts compute interest. `AccountManager` knows the interest calculation logic that accounts must use to compute interest. As of now, `AccountManager` creates an `InterestCalculator` object and passes it on to an `Account` object to compute interest.

Our objective is to make an Account return compound interest. To achieve this, all we need to do is to create a subclass of `InterestCalculator` named `CompoundInterestCalculator` and pass an instance of this new class to `Account`'s `getInterest` method from `AccountManager`'s `main` :

```
public class CompoundInterestCalculator extends InterestCalculator {
   public Double computeInterest(double principle, double yrs, double rate){
         return principle*Math.pow(1 + rate, yrs) - principle;
   }
}
```

```
class AccountManager{
   public static void main(String[] args){
      Account a = new Account(100, 0.2);
      InterestCalculator ic = new CompoundInterestCalculator();
      double interest = a.getInterest(ic, 2);
      System.out.println(interest);
   }
}
```

Observe that the `Account` class has no idea about the change that we did. The `Account` class does not need recompilation because it does not even know about the existence of `CompountInterest-Calculator`. It interacts with the `CompountInterestCalculator` object as if it were just another `InterestCalculator` object.

Now, let's see how violating the rules of overriding affects Account class.

1. **Accessibility** - If you make `computeInterest` in `CompountInterestCalculator` private, the JVM will be facing two contradictory directions. The call to `ic.computeInterest(...)` in `Account` expects the JVM to invoke the method on the actual object referred to by `ic`. But the type of the actual object is `CompoundInterestCalculator` and `CompoundInter-estCalculator` expects the JVM to reject any attempt to invoke this method from outside the `CompoundInterestCalculator` class. There is no way to obey both the directions. Therefore, the compiler rejects the restriction on making this method private.

   Making the method public in `CompoundInterestCalculator` is ok because the `Account` class doesn't care if classes from other packages are able to access `CompoundInterestCalculator`'s `computeInterest`.

2. **Return type** - When `Account`'s `getInterest` calls `ic.computeInterest`, it expects to get back a `Number`. So, when `CompoundInterestCalculator`'s `computeInterest` returns a `Dou-ble`, `Account` has no problem because a `Double` is-a `Number`. But if you change the return type of `CompoundInterestCalculator`'s `computeInterest` to, say, `Object`, code in `Account`'s `getInterest` will fail because an `Object` is not necessarily a `Number` and it won't be able to call `doubleValue` on an `Object`.

3. **parameter types** - This is kind of obvious. If you change the parameter types, the compiler will consider this a completely different method and not a replacement for the method in `InterestCalculator`. The JVM will never invoke this new method and `Account` will still be computing simple interest instead of compound interest.

4. **throws clause** - Since the `computeInterest` method of `InterestCalculator` says that it may throw an `Exception`, the caller of this method, i.e., `Account`'s `getInterest` is prepared to deal with this exception. It has a catch block that is meant to catch an `Exception`. The code in `getInterest` will work even if the `getInterest` method throws `IllegalArgu-mentException` because `IllegalArgumentException` is an `Exception` and it will be caught by `catch(Exception )` clause. It doesn't have any problem if `computeInterest` doesn't throw any exception at all either. But if the overriding method decides to throw a super

class exception such as Throwable, then the code in Account's getInterest will fail because it is not prepared to handle a Throwable. You will have to update the catch clause to catch(Throwable) and recompile Account.

I suggest that you write a similar example to validate the above rules in case of hiding of a static method.

13.4.3 Invalid overrides

Overriding a static method with an instance method and vice versa

Java does not allow a subclass to change the type (static/instance) of a non-private method defined in the superclass. Thus, the following code will not compile:

```java
class Person{
  static void message(){ }
  void initialize(){ }
}
class Employee extends Person{
  void message(){} // will not compile because overridden method is static

  static void initialize(){} // will not compile because overriding method is static
}
```

Overriding private methods

Since private methods are not inherited by a subclass, it is not possible to override them. But a subclass is allowed to have a method with the same signature as a private method of a superclass.

Overriding final methods

By marking a method as final, you prohibit subclasses from overriding that method. Therefore, if you try to have a method with the same signature as a final method of a superclass, it will not compile. The final keyword works similarly for static methods as well. It prevents you from hiding a static method in the subclass.

Note that final works very differently in case of fields. A subclass is allowed to have a field by the same name even if that field is declared final in the superclass. As discussed earlier, final just prevents you from the changing the value of that field.

13.4.4 The @Override annotation

The @Override annotation informs the compiler that the method on which it is applied is meant to override a method declared in a super type. Although **optional**, it is a good idea to apply this annotation while overriding a method for two reasons:

1. It makes your intention of overriding the method well documented. That makes the code more maintainable.
2. It allows the compiler to check whether a method actually does override a super type's method or not and generate an error at compile time if it does not. This is a great benefit because while coding a method, a programmer may mistype the method name or have a slightly different parameter list. While the programmer may believe that they have successfully overridden a method, in reality, they would have created a new or an overloaded method inadvertently. This will introduce hard to trace bugs in the code. But if the programmer applies `@Override` to the method, the compiler will prevent them from making such mistakes.

The exam does not ask questions on this annotation directly but you will see code annotated with `@Override`.

13.5 Utilize polymorphism to cast and call methods

13.5.1 Type of a reference and type of an object

In the 'Kickstarter for Beginners'chapter I discussed the difference between a reference and an object in detail. I used the analogy of a remote and a TV to explain that a reference is a handle with which you access an object. Just as the remote and the TV it controls are two different things, the reference and the object to which the reference refers are also two different things.

The type of the variable is specified at the time of defining the variable and can never change (which makes Java a "statically typed" language, as opposed to JavaScript, which is a dynamically typed language). For example, `String str;` defines a variable `str` of type `String`. Specifying the type of the variable is important because that is how the compiler is able to determine the variables and methods that you are allowed to access through that variable. A Java compiler will not allow you to invoke methods that are not available in the type of the reference that you are using (which makes Java a "strongly typed" language). It is important to understand this point because it drives how and why the compiler generates compilation errors in situations involving overloading, overriding, generics, and what not.

An object, on the other hand, is created at run time. Although the compiler may sometimes be able to guess the type of an object that will be created at run time (for example, when you instantiate a class using the `new` keyword), it is not always the case. For example, while compiling the following method, the compiler has no way of knowing the type of actual object that is referred to by the method parameter `obj`:

```
public class Test{
   public void dump(Object obj){
      //do something with obj
   }
}
```

The compiler does not know who may be calling `dump` and what they may be passing to it as an argument. All it knows is that `obj` will refer to an `Object` but it has no idea about the exact type of that object. For example, it is possible for another developer to write a method in another class that calls `Test`'s `dump` with a `String` or an `Integer` as an argument:

```
public class Other{
   public void testDump(){
      new Test().dump("hello");
      new Test().dump(new Integer(1));
   }
}
```

The above example actually shows polymorphism in action. The variable `obj` in `dump` could potentially behave like a `String`, an `Integer`, or like any other subclass of `Object` depending on the actual class of the object that is passed to `dump`.

13.5.2 Bridging the gap between compile time and run time

To write polymorphic code, it is critical to understand that polymorphism happens at **run time** i.e. at the time of execution of the code, but the code that makes it happen is written at **compile time**. There is a difference between the amount of information that the compiler has about the program and the amount of information that the JVM has while it is executing that program. For example, when you define a variable, the compiler only knows the type of that variable, but it cannot know the exact object to which it refers because objects are created only when the program is run, i.e., at run time. Now, Java is a strongly typed language, which means that the type of the variable is defined at compile time itself and it cannot be changed once defined. This means, the compiler must make sure that the code does not assign an object of one type to a variable of another type. It must not allow the code to invoke methods that are not supported by the type of the variable. However, relying only on the type information available at compile time will impose too many restrictions on the code. For example, if you define a reference variable of type `java.util.List`, the compiler must not allow the code to invoke methods defined in some other class using that variable. However, this rigidity will make polymorphism impossible because the whole premise of polymorphism is that an object may behave differently at run time than what was promised at compile time.

You can see that there are two contradictory requirements that the compiler needs to fulfill. On one hand, you want the compiler to type check the code to make sure the code doesn't invoke random methods using a reference and on the other hand you want the compiler to let code violate type checking in some cases. Java tries to meet both the objectives by letting the compiler make certain assumptions and by letting the programmer give guarantees to the compiler that it is not trying to do anything fishy.

Let us now see how these two things are done in practice.

The "is a" test

The **is-a** test is an intuitive test to determine the relationship between two reference types. It allows you to check whether there is a parent child relationship between two reference types and if there is, which one of the two types is the parent class, and which one is the child class. For example, it makes sense if you say that a Dog is a Pet or a Cat is a Pet but not if you say that a Dog is a Cat. A similar relationship exists between an Apple and a Fruit or a Mango and a Fruit. An Apple is a Fruit, A Mango is a Fruit, but an Apple is not a Mango. If you were to model Apple, Mango, and Fruit as classes, you would put all common features of fruits in a `Fruit` class and you would make `Apple` and `Mango` classes extend `Fruit`. The importance of establishing this relationship is that it allows you to use a subclass object anywhere in place of a superclass object. Indeed, if you promised to give someone a fruit, you can certainly give them an apple because an apple is a fruit. The reverse is obviously not true. If you promised to give someone an apple, you cannot just give them any kind fruit. It has to be an apple. But if you had a Macintosh apple, you could give them that because a Macintosh apple is an apple.

The Java compiler recognizes if an **is-a** relationship exists between two types and allows you to assign a subclass object to a reference of a superclass type. For example:

```
class Fruit{   }
class Apple extends Fruit{ }
class Mango extends Fruit{ }
...
Apple a = new Apple();
Mango m =  new Mango();
Fruit f1 = a; //ok, because Apple is a Fruit

m = a; //will NOT compile because an apple is not a mango
Fruit f2 = m; //ok, because Mango is a Fruit

m = f1; //will NOT compile because all fruits are not mangoes
m = f2; //will NOT compile because all fruits are not mangoes
```

Note that the actual objects are created only at run time but the compiler knows that the variable `a` will always refer to an `Apple` (i.e. to an object of a class that satisfies the is-a test with `Apple`) at run time because it would not compile the code that tries to assign anything else to `a`. This is proven by the fact that the line `Mango m = a;` does not compile. Similarly, the compiler knows that `f1` and `f2` will always point to a `Fruit` at run time because it will not allow you to write code that tries to assign anything else to `f1` and `f2`.

The last line, i.e., `m = f2;` is important. Even though we know that `f2` really does point to a `Mango`, the compiler refuses to compile this line. Remember that the compiler does not execute any code and therefore, the compiler doesn't know that `f2` really does point to a `Mango`. It only knows that `f2` may refer to any kind of `Fruit`. That fruit could be a `Mango` or an `Apple`. Since the compiler cannot know for sure what `f2` will point to at run time, it does not let you assign `f2` to `m`. If it allowed this line to compile, there would be a violation of type safety during execution if `f2` pointed to an `Apple` instead of a `Mango`.

The same logic discussed above applies to interfaces as well. If a class implements an interface, then an object of that class **is-a** that interface. For example, if `Fruit` implements an interface named `Edible`, then any fruit will satisfy the is-a test for `Edible`. In other words, a `Fruit` **is-a** `Edible`.

The cast Operator `( )`

You saw above that the compiler does not let the lines `Mango m = f1;` and `Mango m = f2;`compile because the compiler cannot know for sure that `f1` and `f2` will point to `Mango` objects at run time. Instead of letting an `Apple` get assigned to `m` and thereby violating type safety, it rejects the code altogether. However, the programmer knows what she expects `f2` to refer to at run time (because she wrote the code after all!). Java language allows a programmer to use their knowledge about their program to assure the compiler that a reference will point to an object of the correct type at run time using the cast operator. This assurance convinces the compiler to let them do the assignment. Here is how this works:

```
m = (Mango) f1;
m = (Mango) f2;
```

By casting `f1` and `f2` to `Mango`, the programmer basically guarantees to the compiler that `f1` and `f2` will point to `Mango` objects at run time. The compiler accepts the guarantee because it knows that `Mango` **is a** `Fruit` and it is possible for a variable of type `Fruit` to refer to a `Mango`. The compiler rejected the code earlier because the variables could point to `Apple` objects as well but with the explicit guarantee given by the programmer that they will point to a `Mango` objects at run time, the compiler accepts the code.

The java.lang.ClassCastException

If you observe the above code carefully, you will know that `f1` will not really point to a `Mango` at run time. The line `Fruit f1 = a;` actually makes `f1` point to the same object as `a` and `a` points to an `Apple`. Therefore, `f1` will actually point to an `Apple` and not a `Mango` at run time. So basically, we just fooled the compiler into accepting the code by making a false guarantee. Well, we did fool the compiler but we can't fool the JVM. The JVM knows what kind of object `f1` actually points to and it will not let the cast of `f1` to `Mango` succeed. The JVM will throw a `ClassCastException` when it executes this line.

The JVM is essentially the second line of defence against violation of Java's type safety. It checks at run time what the compiler is unable to check at compile time. If the JVM sees that a reference variable is being cast to a type that does not satisfy the is-a test for the class of the object to which the variable is referring, it will throw a `ClassCastException`. Thus, the type safety of the program is never compromised.

Fooling the compiler in this case doesn't mean that you can fool the compiler by giving a false guarantee every time. Take a look at this code:

```
Mango m = new Mango();
Apple a = (Apple) m;
```

The compiler rejects the above code in spite of you guaranteeing that `m` will point to an `Apple` at run time. Well, it turns out that compiler is not completely clueless. It knows that there is absolutely no way `m` can point to an `Apple` because the declared class of `m` is `Mango` and there is no is-a relationship between `Mango` and `Apple`. It knows that anything that is a `Mango` can never be an `Apple` and so it calls your bluff. Thus, for a cast to pass compilation, the cast must at least be plausible!

Casting a reference to an interface

It is easy for a compiler to determine whether a reference of one class can ever point to an object of another class because a class can only extend one class at the most. Therefore, if the class mentioned in the cast is a subclass of the declared class of the reference, the compiler knows that the cast may succeed at run time.

It is not so easy with interfaces. Since a class can extend one class and can also implement any number of interfaces at the same time, the compiler cannot rule out cases where a reference can never point to an instance of a class that implements the interface mentioned in the cast. Here is what I mean:

```
interface Poisonous{ }
class TestClass{
   public static void main(String[] args){
       Fruit f = new Mango(); //ok, because Mango is-a Fruit
       Poisonous p = (Poisonous) f; //compiles fine but throws a ClassCastException at
    run time
   }
}
ENDRC
```

The compiler accepts the casting of `f` to `Poisonous` even though it knows that `Fruit` does not implement `Poisonous`. The reason is that the compiler also knows that even though `Fruit` does not implement `Poisonous`, there could be a subclass of `Fruit` that implements `Poisonous` and since that subclass would be a `Fruit`, `f` could potentially point to an object of a class that implements `Poisonous`. Something like this:

```
class StarFruit extends Fruit implements Poisonous{ }
...
Fruit f = new StarFruit();
Poisonous p = (Poisonous) f; //compiles and runs fine
```

The above reasoning implies that you can cast any type of reference to any interface. That's true except in one case. If the declared class of the variable is `final` and if that class does not implement the interface given in the cast, the compiler knows that this class can never have any subclass and therefore, it knows that there is no way the reference can point to an object of a class that extends this class and also implements the given interface. For example: `String` is a final class and the following code will not compile for the same reason:

```
String s = new String();
Poisonous p = (Poisonous) s; //will not compile
```

Downcast vs Upcast

Casting a variable of a type to a subtype (e.g. casting the variable `f` of type `Fruit` to `Apple`) is called " **downcasting**". It is called downcasting because in a UML diagram a subclass is always drawn below the superclass. Also, a variable of type the `Fruit` may refer to any kind of fruit but when you cast it to `Apple`, you have reduced the possibilities for the kind of fruits that the variable could be referring to. So, Fruit is a **wider** type and Apple is a **narrower** type in relation to each other. In other words, when you cast a variable to a subtype, you are narrowing the type of the object to which this variable is pointing, down to a more specific type. Hence, the term "narrowing". It means the same as downcasting. The opposite of downcasting is "**upcasting**" or "widening".

As explained above, downcasting always requires a check by the JVM to make sure that the variable is really pointing to an object of type that the programmer has claimed it is pointing to. Upcasting, on the other hand, requires no such check and is, in fact, almost always redundant (I will show you the one situation where it is not redundant soon). Since an Apple is-a Fruit, you can always assign a variable of type Apple to a variable of type Fruit without any explicit cast.

13.5.3 When is casting necessary

In the previous section I used the analogy of a remote and a TV to show you that the type of a variable and the type of the actual object referred to by that variable are two different things and may not necessarily be the same.

Let me continue with the same analogy to show you how you can deal with the difference between the type of a variable and type of the object referred to by the variable. Imagine you have the remote for an old model that has a limited number of buttons and you are using this remote to control a TV from a new model that has a lot of functions. Obviously, you will only be able to use those functions for which there are buttons in the remote. Even though the TV supports many more functions, you cannot use them with the old remote because the old remote has no knowledge of the new functions.

The reference and the object behave the same way. The type of the reference is like the model of the remote and type of the object is like the model of the TV. The following code makes this clear:

```
Object obj = "hello";
int h = obj.hashCode(); //ok because hashCode is defined in Object
int i  = obj.length(); //will not compile
String str = "hello";
int j  = str.length(); //OK
```

In the above code, the declared type of the reference variable `obj` is `Object` and the type of the object to which it refers to at run time is `String`. Since `String` is a subclass of `Object`, `String` is-a `Object` and therefore, it is ok to assign a `String` object to `obj`.

You can say that `String` is a kind of new model of `Object` and it has several new features in addition to all the features of `Object`. However, since the declared type of `obj` is `Object`, the compiler will only let you use the functionality that is supported by `Object` because the compiler does not know that `obj` will actually point to a string at run time.

If you want to use the new features of String class using `obj`, you will need to cast it to `String` using the cast operator:

```
Object obj = "hello";
String str =  (String) obj; //cast obj to String
int i  = str.length();

int j  = ((String) obj).length(); //casting and accessing at the same time
```

Thus, the answer to the question when is casting necessary is simple. You need casting when you want to use the features (i.e. instance variables and methods) defined in a subclass using a reference whose declared type is of a superclass.

Remember that casting doesn't change the actual object. The purpose of casting is to provide the compiler with the type information of the actual object to which a variable will be pointing to at run time. Thus, casting just changes the perspective from which the compiler views the object.

Impact of casting on static members

Casting is essentially an aspect of object-oriented programming while "static" is not. Nevertheless, due to the peculiarity of the Java language, it is possible to access static members of a type through a variable of that type. Recall from the "Working with Data types" chapter that static members of a type "shadow" the static members of the same name in the super type. Casting a variable to the super type is how you "unshadow" those members. Here is an example:

```java
class Fruit{
   static int count = 5;
   static int getCount(){ return count; }
}
class Apple extends Fruit{
   static int count = 10; //shadows Fruit's count
   static int getCount(){ //shadows Fruit's getCount
       return count;
    }
}
class TestClass{
   public static void main(String[] args){
      Apple a = new Apple();
      System.out.println(a.count); //prints 10;
      System.out.println(a.getCount()); //prints 10;

      System.out.println(  ((Fruit) a).count ); //prints 5;
      System.out.println(  ((Fruit) a).getCount() ); //prints 5;

      Fruit f = a; //observe that no cast is needed here because we are widening
      System.out.println(  f.count); //prints 5;
      System.out.println(  f.getCount() ); //prints 5;

   }
}
```

As promised, the above code shows that an upcast is not always redundant. But I cannot stress enough that the above code is completely unprofessional. You should never access static members through a variable.

Note

While understanding Casting is important, it is used sparingly in professionally designed code because it is, after all, a means of avoiding the type safety mechanism of the compiler. When you cast a reference to another type, you are basically saying that the program does something that is not evident from the code itself and you are browbeating the compiler into accepting that code. This reflects a bad design. Ideally, you should almost never need to use casting.

13.5.4 The instanceof operator

Since the actual type of the object which a variable is pointing to is not always evident from the code, Java has the `instanceof` operator for the programmer to check whether the actual object is an instance of a particular reference type (i.e. class, interface, or enum) that the programmer is interested in. It takes two arguments - a reference on the left-hand side and a type name on the right-hand side. It returns a boolean value of `true` if the object pointed to by the variable satisfies the is-a test with the type name given and `false` otherwise. For example, if you are passed a `Fruit` as an argument to a method, here is how you can use `instanceof` to know whether you have actually been given a `Mango`:

```java
class Fruit{ }

class Mango extends Fruit{ }

class Apple extends Fruit{ }

public class Juicer{
   public void crush(Fruit f){
      if(f instanceof Mango){
         System.out.println("crushing mango...");
      } else {
         System.out.println("crushing some other fruit...");
      }
   }

   public static void main(String[] args){
       Mango m = new Mango();
       new Juicer().crush(m);
   }
}
```

In the above code, the method `crush` doesn't know the type of the actual object that is referred to by `f`, but using the `instanceof` operator you can check if that object **is-a**`Mango`.

The `instanceof` operator is handy when you want to treat an object of a particular kind differently. For example, if `Mango` had a method named `removeSeed` and if you want to invoke that method before crushing it, you would want to know whether you have actually been given a `Mango` or not before you cast your `Fruit` reference `f` to `Mango`, otherwise, you will get a `ClassCastException` at run time if `f` does not refer to a `Mango`. Here is how this can be done:

```
class Mango extends Fruit{
   public void removeSeed(){ }
}

class Juicer{
   public void crush(Fruit f){

      if(f instanceof Mango){
         Mango m = (Mango) f;
         m.removeSeed();
      }
      System.out.println("crushing fruit...");
  }

  ...
}
```

Remember that you can't invoke `removeSeed` on `f` because the type of `f` is `Fruit` and `Fruit` doesn't have `removeSeed` method. `Mango` does. Therefore, as discussed before, you must cast `f` to `Mango` before you can invoke a method that is specific to a `Mango`. There are a couple of things that you should understand clearly about `instanceof`:

1. The `instanceof` operator cannot tell you the exact type of the object being pointed to by a variable. It can only tell you whether that object is-a something. For example, if you do `f instanceof Fruit`, it will return `true` if the object referred to by `f` **is-a** `Fruit`, which means it will return `true` even if `f` points to a `Mango` because a `Mango` **is-a** `Fruit`. In the case of interfaces, it will return `true` if the class of the object pointed to by the reference implements the given interface (directly or indirectly).

2. The compiler will let you use `instanceof` operator only if it is possible for the variable to refer to an object of the type given on the right-hand side. For example, `f instanceof Mango` is valid because the compiler knows that the declared type of `f` is `Fruit` and since `Mango` is a `Fruit`, it is possible for `f` to point to a `Mango`. But `f instanceof String` will not compile because the compiler knows that there is no way `f` can ever point to a `String`. The `instanceof` operator behaves the same way as the cast operator in this respect.

Note

Just like the cast operator, the `instanceof` operator is also used only sparingly in professionally written code. Usage of `instanceof` reflects a bad design and if you feel the need to use instanceof operator in your code too often, you should think about redesigning your application.

13.5.5 Invoking overridden methods

If I haven't hammered it in enough yet, let me say this again - polymorphism is all about the ability to replace one object with another without the need to recompile existing code as long as the objects stick to an agreed upon contract. For example, if a method requires an object of a class as an argument, then it should work well with an object of its subclass. If a method expects an interface as an argument, the class of the object that you pass to it shouldn't matter as long as that class implements the interface. This flexibility makes it easier, and thereby cheaper, to develop and maintain an application.

In technical terms, you must remember that polymorphism works only because of **dynamic binding** of methods calls. When you invoke an instance method using a reference variable, it is not the compiler but the JVM that determines which code to execute based on the class of the actual object referenced by the variable.

Some languages let the programmer decide whether they want to let the compiler bind a method call to the version provided by the declared class of the variable or to let the JVM bind the call at run time based on the class of the object referred to by the variable. In such languages, methods that are not bound by the compiler are called **"virtual methods"**. Java does not give the programmer the ability to customize this behavior. In Java, calls to **non-private** and **non-final instance** methods are bound **dynamically** by the JVM and are therefore, always **"virtual"**. Everything else is bound **statically** at compile time by the compiler.

To take advantage of polymorphism, it is advisable to use interfaces and non-final classes as method parameter types. For example, the interest calculator example that I showed earlier could be redesigned as follows:

```
interface InterestCalculator{

   //interface methods are public by default
    double computeInterest(double p, double r, double t);
}

class Account{
    double balance, rate;


    double getInterest(InterestCalculator ic, double time){
        return ic.computeInterest(balance, rate, time);
    }
}

class SimpleInterestCalculator implements InterestCalculator{

   //must be public because an overriding method must not reduce accessibility
   public double computeInterest(double principle, double yrs, double rate) {
       return principle*yrs*rate;
   }
}
```

```
class CompoundInterestCalculator implements InterestCalculator{
   public double computeInterest(double principle, double yrs, double rate) {
       return principle*Math.pow(1 + rate, yrs) - principle;
   }
}
```

Observe that the `getInterest` method now takes an `InterestCalculator` as a parameter. This makes it very easy for any other class to compute any kind of interest on an account object. As of now, there are two classes that compute interest - `SimpleInterestCalculator` and `CompoundInterestCalculator`, but in future if you want to compute interest with different compounding, all you have to do is create a new class that implements `InterestCalculator` interface and pass an object of this class to the `getInterest` method. The Account class will not know the difference. The JVM will bind the call to `computeInterest` to the code provided by your new class automatically.

Dynamic binding of method calls may cause unexpected results if you are not careful. Consider the following code:

```
class Account{
   double balance = 0.0;

   Account(double balance){
     this.balance = balance;
     this.printBalance();
   }

   void printBalance(){
      System.out.println(balance);
   }

}

class DummyAccount extends Account{

   DummyAccount(double b ){
     super(b);
   }

   public void printBalance(){
     System.out.println("No balance in dummy account");
   }
}

public class TestClass{
   public static void main(String[] args){
      Account a =  new DummyAccount(100.0);
   }
}
```

Can you guess what it prints?
You may expect the call to `this.printBalance();` in `Account`'s constructor to be bound to `Account`'s `printBalance` method but observe that this method is overridden by `DummyAccount`. Since the class of the actual object is `DummyAccount`, the JVM will bind the call to `DummyAccount`'s `printBalance` instead of `Account`'s `printBalance`. Therefore, it will print `"No balance in DummyAccount"`.

This example shows that you need to be very careful about calling non-private methods from a constructor. And by careful, I mean, don't do it :) A constructor is meant to initialize the object's state variables to their appropriate values but if you invoke a non-private method from a constructor, a subclass can easily mess with your constructor's logic by overriding that method.

13.5.6 Impact of polymorphism on == and equals method

You know that when used on references the `==` operator checks whether the two operands point to the same object in memory or not. So, can you guess what the following code will print?

```
String s = "hello";
Integer n = 10;
System.out.println(n == s);
```

`false`, you say? Well, the code won't even compile. The compiler applies the same logic that it applies to the cast and the instanceof operators to check whether it is even possible for the two reference variables to point to the same object. It rejects the comparison if it is not. In the above code, there is no way `s` and `n` can point to the same object because `s` and `n` are variables of two different unrelated types. Thus, the compiler knows that this comparison is pointless and is most probably a mistake by the programmer. Here is the same code but with a small change:

```
Object s = "hello";
Integer n = 10;
System.out.println(n == s);
```

The above code indeed prints `false`. The compiler cannot reject the comparison now because the type of `s` is `Object` and therefore, it is possible for `s`to point to an `Integer` object (because Integer is-a Object).

The `equals` method, on the other hand, behaves differently. Remember that `equals` is defined in the `Object` class and its signature is `equals(Object )`. The type of the input parameter is `Object` and therefore, it must accept a reference of any type. Thus, the compiler has no option but to accept even the illogical invocations of the equals method such as `"1234".equals(n);`. It is for the same reason that when a class overrides the `equals` method, the first line of code in the method is usually an instanceof check:

```
class X{
  int val;
  public boolean equals(Object x){
    if( ! (x instanceof X) ) return false;
    //now compare the values of instance fields of this and x and return true/false
   accordingly
    return this.val == ((X) x).val;
  }
}
```

If you remember the rules of overriding, an overriding method is not allowed to change the type of the input parameter to a narrower type. Thus, if a class tries to override the equals method but changes the type of the input parameter from Object to a more specific type, it will not be a valid "override". It will be a valid "overload" though. For example, the following code will compile fine, but the equals method does not override the equals method that X inherits from Object:

```
class X{
    int val;
    //@Override //uncommenting this will cause compilation failure
    public boolean equals(X x){ //does not override but overloads the equals method
        return this.val == x.val;
    }
    public static void main(String[] args){
      X x1 = new X(); x1.val = 1;
      X x2 = new X(); x2.val = 1;
      System.out.println(x1.equals(x2)); //prints true
    }
}
```

On the face of it, the `equals` method written above doesn't seem to make much of a difference when you try to compare two `X` objects. But let's change the code inside the `main`method as follows and see what happens:

```
X x1 = new X(); x1.val = 1;
Object x2 = new X();  ((X) x2).val = 1;
System.out.println(x1.equals(x2)); //what will it print?
```

If you have understood the rules of method selection that we discussed in the "Creating and Using Methods" chapter, you should be able to figure out that the above code will print `false`. The compiler sees two versions of the equals method in class `X` to choose from when it tries to bind the `x1.equals(x2)` method call - the version that class `X` inherits from `java.lang.Object`, which takes `Object` as an argument and the version that class `X` implements itself, which takes `X` as an argument. Since the declared type of the variable `x2` is `Object`, the compiler binds the call to the `Object` version instead of the `X` version. The `Object` version returns `false` because `x1` and `x2` are pointing to two different objects.

This code also shows the importance of the `@Override` annotation. If it were present, the compiler would have alerted the programmer by generating an error saying that this method does not actually override anything.

13.6 Distinguish overloading, overriding, and hiding

13.6.1 Overriding and Hiding

You have seen how a class can inherit features by extending another class. These features include static and instance fields as well as static and instance methods. In the previous section, I explained how inheriting features is usually a good thing because it allows a class to get functionality without writing any code. But it could pose a problem if the subclass were not able to provide suitable behavior to a method that it inherited.

For example, what if there is a `SpecialInterestCalculator` that extends `InterestCalculator` and inherits a `computeInterest` method, but it wants to change how the interest is computed by this method? Or what if it wants to define a variable `interestRate`, but that variable is already defined in `InterestCalculator`?

Java has specific rules about what features can be tweaked and how they can be tweaked by a subclass. These rules are categorized into two categories: **Overriding** and **Hiding**. The rules of overriding are about **polymorphism** and therefore, only apply to **instance methods** and the rules of hiding apply to everything else, i.e., **static methods** as well as **static and instance variables**.

Remember that in both the cases, a member of a class has to be inherited in the subclass first. Since the private members of a class are not inherited by a subclass, the concepts of overriding and hiding are not applicable to them. Similarly, constructors of a class are not inherited either and are, therefore, out of the purview of overriding and hiding.

Overriding

A class is allowed to completely replace the behavior of an instance method that it inherited by providing its own implementation of that method. What this means is that the behavior provided by the subclass is what will be exhibited by any object of the subclass instead of the behavior provided by the super class. For example:

```
class InterestCalculator{
   public double computeInterest(double principle, int yrs, double rate){
         return principle*yrs*rate;
   }
}

class CompoundInterestCalculator extends InterestCalculator{

   public double computeInterest(double principle, int yrs, double rate){
         return principle*Math.pow(1 + rate, yrs) - principle; //don't worry about
    Math.pow()!
   }

}
```

In the above code, `CompoundInterestCalculator` has replaced the implementation of `computeInterest` method provided by its super class with its own implementation. If you call the `computeIn-`

terest method on a CompoundInterestCalculator object, CompoundInterestCalculator's version of the method will be called. The following code proves it.

```
class TestClass{
  public static void main(String[] args){
       InterestCalculator ic = new InterestCalculator();
       double interest  = ic.computeInterest(100, 2, 0.1);
       System.out.println(interest); //prints 20.0

       CompoundInterestCalculator cic = new CompoundInterestCalculator();
       interest  = cic.computeInterest(100, 2, 0.1);
       System.out.println(interest); //prints 21.0
  }
}
```

Note that I am using the word replace to highlight the fact that it is not possible for any other class to see the behavior of computeInterest method as implemented by InterestCalculator class in a CompoundInterestCalculator object because the behavior of the super class has been replaced by the behavior provided by the subclass. Technically, we say that CompoundInterestCalculator has **"overridden"** computeInterest method of InterestCalculator. Let me make a small change to the above code to make this point clear:

```
class TestClass{
  public static void main(String[] args){
       InterestCalculator ic = new CompoundInterestCalculator();
       double interest  = ic.computeInterest(100, 2, 0.1);
       System.out.println(interest); //prints 21.0
  }
}
```

In the above code, the declared type of the variable ic is InterestCalculator but the actual object that it points to is of type CompoundInterestCalculator. Therefore, when you call computeInterest, CompoundInterestCalculator's version of this method is executed instead of InterestCalculator's version.

The point to understand here is that in the case of instance methods, it is always the method implemented by the class of the object that is invoked.

Hiding

Hiding is a less drastic version of overriding. Like overriding, hiding lets a class define its own version of the features implemented by its superclass, but unlike overriding, hiding does not completely replace them with the subclass' version. Thus, the subclass now has two versions of the same features and any unrelated class can access both the versions. The following code illustrates this point:

```
class InterestCalculator{
   public int yrs = 10;
   public static double rate = 0.1;
   public static String getClassName(){
         return "InterestCalculator";
   }
}

class CompoundInterestCalculator extends InterestCalculator{
   public int yrs = 20;
   public static double rate = 0.2;
   public static String getClassName(){
         return "CompoundInterestCalculator";
   }
}
```

In the above code, CompoundInterestCalculator inherits the instance variable yrs, the static variable rate and the static method getClassName from InterestCalculator. At the same time, CompoundInterestCalculator defines all of these on it own as well. So, now, CompoundInterestCalculator has two yrs variables, two rate variables, and two versions of the getClassName method. Both the versions can be accessed by an unrelated class as shown below:

```
class TestClass{
  public static void main(String[] args){
       CompoundInterestCalculator cic = new CompoundInterestCalculator();

       System.out.println(cic.yrs); //prints 20
       System.out.println( ((InterestCalculator) cic).yrs); //prints 10

       System.out.println(cic.rate); //prints 0.2
       System.out.println( ((InterestCalculator) cic).rate); //prints 0.1

       System.out.println(cic.getClassName()); //prints CompoundInterestCalculator
       System.out.println( ((InterestCalculator) cic).getClassName()); //prints
   InterestCalculator
  }
}
```

Observe that I used a cast to access the version provided by the superclass. The declared type of the variable cic is CompoundInterestCalculator but I cast it to InterestCalculator to go behind CompoundInterestCalculator and access the versions provided by InterestCalculator. The cast tells the compiler to treat a variable as if its declared type is the type mentioned in the cast.

As discussed in the rules of casting earlier, the point to understand here is that static methods, static variables, and instance variables are accessed as per the declared type of the variable through which they are accessed and not according to the actual type of the object to which the variable refers.

Contrast this with overriding, where type of the variable makes no difference to the version of the instance method that is invoked.

13.6.2 Overloading

Since we have already discussed overloading in great detail in Section 10.2, I will just recap it here by saying that overloading happens when two (or more) methods of a class have the same name but different parameter list. In other words, overloaded methods have different signatures but have the same name.

It is not necessary for the overloaded methods to be defined in the same class. As long as a class has more than one method with the same name, whether inherited or defined, it is considered an overloaded method in that class. For example, if you have one inheritable method in a superclass and another method by the same name defined in the subclass, it would still be considered a valid overload in the subclass because effectively, the subclass has two methods with the same name - one inherited and one defined.

Observe that I have used the word "inheritable" above. As you know, private methods can never be inherited and package private methods are not inherited by a subclass that belongs to another package. Thus, such methods will not constitute a valid overload in the subclass.

13.6.3 Method selection with overloading and overriding

Before going through this section, make sure you understand the rules of method selection discussed in Chapter 10 (Section 10.2.3). Let me recap the cardinal principle that you should always keep mind here:

While determining which method should be invoked, the compiler considers only the declared type of the variable on which the method is invoked and the declared type of the variables passed as arguments. At runtime, the JVM picks up the method signature that was selected by the compiler, finds out the actual object pointed to by the reference variable, and invokes the method on that object. If you remember the above rule, you will not have any problem answering questions involving overloaded polymorphic calls. The following example brings all the rules together:

```
class Fruit{
    public void mash(Collection fruits){
        System.out.println("In Fruit's mash(Collection )");
    }
}
class Apple extends Fruit{
    public void mash(Collection apples){
        System.out.println("In Apple's mash(Collection )");
    }
    public void mash(List apples){
        System.out.println("In Apple's mash(List )");
    }
    public void mash(Apple apple){
        System.out.println("In Apple's mash(Apple )");
    }
}
public class TestClass {
    public static void main(String[] args) {
        Collection c = new ArrayList();
        Fruit f = new Fruit();
        Fruit fa = new Apple();
        Apple a = new Apple();

        f.mash(c); //1
        fa.mash(c); //2
        a.mash(c); //3
        f.mash(a); //4

        List<Apple> la = new ArrayList<>();
        Collection<Apple> ca = la;

        a.mash(la); //5
        a.mash(ca); //6
    }
}
```

Observe that `Apple` has three mash methods - `mash(Collection )`, `mash(List )`, and `mash(Apple )`. Thus, it has overridden the `mash(Collection )` method of its base class `Fruit` and it has also overloaded it with `mash(List )` and `mash(Apple )`. Let us now evaluate each of the six method calls:

1. `f.mash(c)`: The compiler notices that the declared type of `f` is `Fruit`. It will therefore, consider only the methods available in class `Fruit` to bind this invocation. It also notices that the declared type of the argument `c` is `Collection`. Since `Fruit` does have a mash method that that can accept a `Collection`, it binds the call to `Fruit`'s `mash(Collection c)` method.

   At run time, the JVM notices that `f` points to an object of class `Fruit` and so, it invokes `mash(c)` on that `Fruit` instance. Thus, it prints `In Fruit's mash(Collection )`.

2. `fa.mash(c)`: The compiler notices that the declared type of `fa` is `Fruit`. It will therefore, consider only the methods available in class `Fruit` to bind this invocation. It also notices that the declared type of the argument `c` is `Collection`. Since `Fruit` does have a mash method that can accept a `Collection`, it binds the call to `Fruit`'s `mash(Collection c)` method.

   At run time, the JVM notices that `fa` points to an object of class `Apple` and so, it invokes `mash(c)` on that `Apple` instance. Thus, it prints `In Apple's mash(Collection )`.

3. `a.mash(c)`: The compiler notices that the declared type of `a` is `Apple`. It will therefore, consider only the methods available in class `Apple` to bind this invocation. It also notices that the declared type of the argument `c` is `Collection`. Apple class has two methods that can accept a `Collection` - `mash(Collection )` and `mash(List )`. Since the declared type of argument `c` matches exactly with `mash(Collection )`, it binds the call to `Apple`'s `mash(Collection c)` method.

At run time, the JVM notices that `a` points to an object of class `Apple` and so, it invokes Apple's `mash(Collection ).` The fact that `c` actually points to an `ArrayList` object is irrelevant at this point. Thus, it prints `In Apple's mash(Collection )`.

4. `f.mash(a)`: The compiler notices that the declared type of `a` is `Fruit`. It will therefore, consider only the methods available in class `Fruit` to bind this invocation. It also notices that the declared type of the argument `a` is `Apple`. Since `Fruit` does not have a `mash` method that can accept an `Apple`, it refuses to compile the code. The fact that `f` will point to an `Apple` object at run time, which has `mash(Apple )` method is irrelevant to the compiler because the compiler cannot insure that `f` will always point to an `Apple` object at run time.

5. `a.mash(la)`: The compiler notices that the declared type of `a` is `Apple`. It will therefore, consider only the methods available in class `Apple` to bind this invocation. It also notices that the declared type of the argument `la` is `List`. Apple class has two methods that can accept a `List` - `mash(Collection )` and `mash(List )`. Since the declared type of argument `la` matches exactly with `mash(List )`, it binds the call to `Apple`'s `mash(List )` method.

   At run time, the JVM notices that `a` points to an object of class `Apple` and so, it invokes Apple's `mash(List )`. Thus, it prints `In Apple's mash(List )`.

6. `a.mash(ca)`: The compiler notices that the declared type of `a` is `Apple`. It will therefore, consider only the methods available in class `Apple` to bind this invocation. It also notices that the declared type of the argument `ca` is `Collection`. Apple class has only one method that can accept a `Collection` - `mash(Collection )`. Therefore, it binds the call to `Apple`'s `mash(Collection )` method.

   At run time, the JVM notices that `a` points to an object of class `Apple` and so, it invokes Apple's `mash(Collection )`. Thus, it prints In `In Apple's mash(Collection )`.

13.7 Exercise ✎

1. You are developing an application that allows a user to compare automobiles. Use abstract classes, classes, and interfaces to model `Car`, `Truck`, `Vehicle`, and `Drivable` entities. Declare and define a method named `drive()` in appropriate places.

2. Every vehicle must have a make and model. What can you do to ensure that a method named getMakeAndModel() can be invoked on every vehicle.

3. You need to be able to get the Vehicle Identification Number (VIN) of a vehicle by calling `getVIN()` on any vehicle. Furthermore, you don't want any subclass to change the behavior of the getVIN method. Where and how will you code the getVIN method?

4. Create an interface named `Drivable` with a default method `start()`. Invoke `start()` on instances of classes that implement `Drivable`. Override `start()` so that it prints a different message in each class.

5. Ensure that every vehicle is created with a VIN.

6. Create a class named `ToyCar` that extends `Car` but doesn't require any argument while creation.

7. You have a list of features such as height, width, length, power, and boot capacity, on which you want to compare any two vehicles. New feature names will be added to this list in future. Create a `getFeature(String featureName)` method such that it will return `"N.A"` for any feature that is not supported by a particular vehicle.

8. Create an interface named `VehicleHelper` with a static method `register(Vehicle v)` that prints the VIN of the vehicle. Ensure that VehicleHelper's register method is invoked whenever an instance of a vehicle is created.