DigiPen
INSTITUTE OF TECHNOLOGY

Chapter 15 | Classes 2

Java AP

Inheritance

- Inheritance is a form of code reuse that allows programmers to develop new classes that are based on existing classes.
- The existing classes are often referred to as **base classes** or **superclasses**, while the new classes are usually called **subclasses**.
- A key advantage of inheritance is that it allows you to reuse code from a base class without modifying the existing code.
- Rather than modifying an existing class that may have been thoroughly tested or may already be in use, you can use inheritance to create a new class with the same functionality that you can also extend with additional data fields or methods.
- You use the **extends** keyword to indicate that a class inherits from another class.

Example

```
/* Base Class */
public class BaseClass
{
    public int fieldInBase;

    public void MethodInBase()
    {
        System.out.println("Method called");
    }
}
```

```
/* Sub Class */
public class SubClass extends BaseClass
{
    public int fieldInSub;
}
```

```
/* Main Class  */
public class Main
{
    public static void main(String[] args)
    {
        BaseClass base = new BaseClass();
        base.fieldInBase = 15;
        base.MethodInBase();
        SubClass sub = new SubClass();
        sub.fieldInSub = 20;
        sub.fieldInBase = 30;
        sub.MethodInBase();
    }
}
```

Output

```
Method called
Method called
```

Explanation

By specifying that the SubClass extends from the BaseClass, the SubClass inherited the data fields and methods found in the BaseClass. All instances of the SubClass contain the data fields and methods found in the SubClass class and the BaseClass.

Note: The Subclass inherited all data fields and methods only because they are all marked as **public**. Later in this chapter we will learn how a base class can prevent some data fields and methods from being inherited by sub classes.

Example 2

```
/* GameObject as Base Class */
public class GameObject
{
    public int positionX;
    public int positionY;
    public float scaleX;
    public float scaleY;
    public float rotation;
}
```

```
/* Knight Sub Class */
public class Knight extends GameObject
{
    public int health;
    public int lives;
    public int shield;

    public void Hit()
    {
        /* method code here */
    }
    public void Jump()
    {
        /* method code here */
    }
}
```

```
/* Dragon Sub Class */
public class Dragon extends GameObject
{
    public int damage;

    public void Fly()
    {
        /* method code here */
    }
}
```

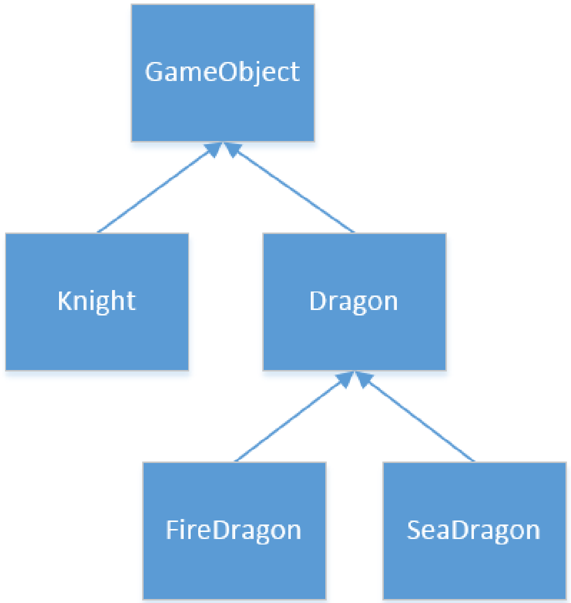
```
/* Main Class  */
public class Main
{
    public static void main(String[] args)
    {
        Knight player = new Knight();
        player.positionX = 100;
        player.positionY = 150;
        player.scaleX = 1.0f;
        player.scaleY = 1.0f;
        player.rotation = 0.0f;
        player.health = 100;
        player.lives = 3;
        player.shield = 75;
        player.Jump();

        Dragon enemy = new Dragon();
        enemy.positionX = 300;
        enemy.positionY = 450;
        enemy.scaleX = 2.0f;
        enemy.scaleY = 2.0f;
        enemy.rotation = 20.5f;
        enemy.damage = 30;
        enemy.Fly();
    }
}
```

Explanation	In this case we have a GameObject class as a parent for two classes, Knight and Dragon. In the GameObject class you will find the data fields that all objects on the screen should have (position, scale and rotation values). Instead of having the same variables in both the Knight and Dragon classes, inheritance allowed us to have all the common fields in one superclass which we use in all classes that are meant to be objects in our game. Note that each subclass also has extra specific data fields added to it like "**health"** in the **Knight** class and "**damage"** in the **Dragon** class.

"is a" relationship and classes hierarchy

- Any class that can serve as base class (unless the class is **final** which we will explain later in the chapter). Even sub classes can be extended and become base classes for a sub class.
- Let's take the below hierarchy as example:



- The **Knight** and **Dragon** classes **extend** from the **GameObject** class. The **FireDragon** and **SeaDragon** classes extend from the **Dragon** class. In other words the relationship between these classes is:
    - **Knight is a GameObject**
    - **Dragon is a GameObject**
    - **FireDragon is a Dragon**
    - **SeaDragon is a Dragon**

Note: **The opposites of the above relationships are *not* true (a GameObject is not a Knight).**

- The **"is a"** relationship doesn't stop there. Since a **Dragon is a GameObject** and a **FireDragon is a Dragon**, it means a **FireDragon is a GameObject.**

Constructors and inheritance

Example

```
/* Base Class */
public class BaseClass
{
  public int fieldInBase;

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

  public void MethodInBase()
  {
    System.out.println("Method called");
  }
}
```

```
/* Sub Class */
public class SubClass extends BaseClass
{
  public int fieldInSub;

  public SubClass()
  {
    System.out.println("SubClass constructor");
  }
}
```

```
/* Main Class  */
public class Main
{
      public static void main(String[] args)
      {
            SubClass sub = new SubClass();
      }
}
```

Output

```
BaseClass constructor
SubClass constructor
```

Explanation

When creating an instance of the **SubClass**, the **BaseClass** constructor gets called in order to initialize the base class' data fields.
This is a form of **code reuse**.

Example 2

```
/* Base Class */
public class BaseClass
{
  public int fieldInBase;

  public BaseClass(int fieldInBase_)
  {
    System.out.println("BaseClass constructor");
    fieldInBase = fieldInBase_;
  }

  public void MethodInBase()
  {
    System.out.println("Method called");
  }
}
```

```
/* Sub Class */
public class SubClass extends BaseClass
{
  public int fieldInSub;

  public SubClass(int fieldInSub_)
  {
    System.out.println("SubClass constructor");
    fieldInSub = fieldInSub_;
  }
}
```

```
/* Main Class  */
public class Main
{
  public static void main(String[] args)
  {
    SubClass sub = new SubClass(15);
  }
}
```

Output

```
Exception in thread "main" java.lang.Error: Unresolved compilation
problem:
      Implicit super constructor BaseClass() is undefined. Must
explicitly invoke another constructor

      at SubClass.<init>(SubClass.java:5)
      at Main.main(Main.java:5)
```

Explanation

The above error is telling us that the **SubClass constructor** tried to call the **default constructor** in the base class but couldn't find it since we created our own **custom constructor**.

When no **default constructor** is found, it is our job to call the **BaseClass' constructor** inside the **SubClass constructor**.

We call the parent's custom constructor using the **super** keyword.

Note: When you call the the parent's constructor, it should be the first statement in the child's constructor.

Solution

Below is the **SubClass'** new constructor:

```
public SubClass(int fieldInSub_)
{
  super(20);
  System.out.println("SubClass constructor");
  fieldInSub = fieldInSub_;
}
```

By calling the **BaseClass constructor** we made sure that the "**fieldInBase**" variable got initialized with a proper value and is ready to be used by the **SubClass instance**.

Access Level Modifiers

- Access Level Modifiers are used to define the accessibility of a class' field.
- So far we've been using "public" which means that the field or method is accessible through the instance of a class and inherited by all subclasses (as seen in all the above examples).
- Now let us learn how a class can control the accessibility of its fields and methods.

Private Access Modifier

- The **private** access modifier makes a field or method visible only to callers within the method's defining class.
- Fields and methods that are marked as **private** are unavailable outside the class. They cannot be accessed through instances of the class.
- As far as inheritance goes, **private** fields and methods are not inherited by subclasses.

Example

```
public class MyClass
{
        private int privateField;

        public MyClass(int privateField_)
        {
                privateField = privateField_;
        }

        public void PrintPrivateField()
        {
                System.out.println(privateField);
        }
}
```

```
public class Main
{
        public static void main(String[] args)
        {
                MyClass mc = new MyClass(15);
                System.out.println(mc.privateField);
        }
}
```

Output

```
Exception in thread "main" java.lang.Error: Unresolved compilation
problem:
	The field MyClass.privateField is not visible

	at Main.main(Main.java:6)
```

Explanation

The above compiler error is telling us that the **privateField** variable is not visible outside of the class.
As explained above, it is not possible to access a class' **private** fields through an instance of the class. The **private** fields are only accessible inside the class' methods.

Solution

```java
public class Main
{
	public static void main(String[] args)
	{
		MyClass mc = new MyClass(15);
		mc. PrintPrivateField();
	}
}
```

Output

```
15
```

Example 2

```java
public class BaseClass
{
	private int privateFieldInBase;
}
```

```java
public class SubClass extends BaseClass
{
	public SubClass()
	{
		privateFieldInBase = 15;
	}
}
```

Output

```
Exception in thread "main" java.lang.Error: Unresolved compilation
problem: 
	The field BaseClass.privateFieldInBase is not visible

	at SubClass.<init>(SubClass.java:5)
	at Main.main(Main.java:5)
```

Explanation

The above compiler error is telling us that the **privateFieldInBase** variable is not visible in the **SubClass**. In other words, the **SubClass** class did not inherit that **private** field.

Note: We would get a similar error when trying to access private methods.

Protected Access Modifier

- The **protected** access modifier makes a field or method visible to callers within the method's defining class and all its subclasses. **protected** data fields and methods are inherited by subclasses.
- Data fields and methods that are marked as **protected** are unavailable outside the class and subclasses unless the classes are sharing the same package. They cannot be accessed through instances of the class or subclass unless the instance is created in a class that shares the same package with the owner class.
- The **protected** attribute is useful when you have a field or method that your subclasses need but that you want to hide from code that is outside the inheritance chain and package.

Example

```
public class BaseClass
{
	protected int protectedFieldInBase;
}
```

```
public class SubClass extends BaseClass
{
	public SubClass()
	{
		protectedFieldInBase = 15;
	}
}
```

```java
public class Main
{
	public static void main(String[] args)
	{
		SubClass sub = new SubClass();
		System.out.println(sub.protectedFieldInBase);
	}
}
```

Output

```
15
```

Explanation

The **SubClass** inherited the protected field found in the **BaseClass**, was able to access it in its constructor and set it to 15.
Since all classes belong to the same package, the **SubClass** instance created in the **Main** class was able to access the **protected** field.

Example 2

```java
package defined_package;

public class BaseClass
{
	protected int protectedFieldInBase;
}
```

```java
package defined_package;

public class SubClass extends BaseClass
{
	public SubClass()
	{
		protectedFieldInBase = 15;
	}
}
```

```
import defined_package.*;

public class Main
{
	public static void main(String[] args)
	{
		SubClass sub = new SubClass();
		System.out.println(sub.protectedFieldInBase);
	}
}
```

Output

```
Exception in thread "main" java.lang.Error: Unresolved compilation 
problem: 
	The field BaseClass.protectedFieldInBase is not visible

	at Main.main(Main.java:8)
```

Explanation

The **SubClass** inherited the **protected** field found in the **BaseClass** and was able to access it in its constructor and set it to 15.
But, when attempting to access the **protected** field in the **Main** class through an instance of the **SubClass** we got a **compiler error**.
The compiler error is telling us that the **protected** field is not visible in **Main** (cannot be accessed through the **SubClass** instance). Even though code that was nearly the same worked in the previous example, accessing the **SubClass' protected** variables through an instance is no longer possible because **Main** doesn't belong to the same package as the **SubClass**.

Package-Private Access Modifier (no modifier)

- The **package-private** access modifier is used when the developer doesn't specify any access modifier. It is the ***default modifier***.
- The **package-private** access modifier makes a field or method visible to callers within the method's defining class but not its subclasses unless the subclass belongs to the same package. So **package-private** fields and methods are only inherited by subclasses in the same package.
- Fields and methods without a modifier are unavailable outside the class unless the classes are sharing the same package. In other words, they cannot be accessed through instances of the class unless the instance is created in a class that shares the same package with the owner class.

Example

```
package defined_package;

public class BaseClass
{
  int fieldInBase;
}
```

```
package defined_package;

public class SubClassSamePackage extends BaseClass
{
  public SubClassSamePackage()
  {
    fieldInBase = 15;
  }
}
```

```
import defined_package.*;

public class Main
{
  public static void main(String[] args)
  {
    SubClassSamePackage sub = new SubClassSamePackage();
    sub.fieldInBase = 20;
  }
}
```

Output

```
Exception in thread "main" java.lang.Error: Unresolved compilation
problem:
	fieldInBase cannot be resolved or is not a field

	at Main.main(Main.java:11)
```

Explanation

The **SubClassSamePackage** class is in the same package as the **BaseClass,** so it inherited the **package-private** field "**fieldInBase**".
However, trying to access the **package-private** field " **fieldInBase** " through the instance of **SubClassSamePackage** is illegal because **Main** is in a different package than the **BaseClass**.

Example 2

```
package defined_package;

public class BaseClass
{
  int fieldInBase;
}
```

```
import defined_package.BaseClass;

public class SubClassDiffPackage extends BaseClass
{
      public SubClassDiffPackage()
      {
            fieldInBase = 15; //Not inherited
      }
}
```

```
public class Main
{
  public static void main(String[] args)
  {
    SubClassDiffPackage sub = new SubClassDiffPackage();
  }
}
```

Output

```
Exception in thread "main" java.lang.Error: Unresolved compilation
problem: 
      The field BaseClass.fieldInBase is not visible

      at SubClassDiffPackage.<init>(BaseClassDifferentPackage.java:7)
      at Main.main(Main.java:7)
```

Explanation

The **SubClassDiffPackage** class is not in the same package as the **BaseClass** so it did not inherit the **package-private** field "**fieldInBase**", which lead to the above compiler error.

Access Modifier Recap

Modifier	Class	Subclass (in a different package)	Subclass (in the same package)	Package	World
Public	Yes	Yes	Yes	Yes	Yes
Private	Yes	No	No	No	No
Protected	Yes	Yes	Yes	Yes	No
None	Yes	No	Yes	Yes	No

Note: **Always choose the most restrictive modifier that makes sense for the method or member. In other words default to private unless you have a reason not to.**

Getters and Setters (a.k.a Accessors and Mutators)

- **Getters** and **Setters** allow you to control access to private fields in your class.
- A class instance will now have a way to access private fields through function calls.

Example

```
public class Hero
{
    private int health;

    public int GetHealth()
    {
        return health;
    }

    public void SetHealth(int health_)
    {
        if(health_ < 0)
        {
            health = 0;
        }
        else if (health_ > 100)
        {
            health = 100;
        }
        else
        {
            health = health_;
        }
    }
}
```

```
public class Main
{
  public static void main(String[] args)
  {
    Hero h = new Hero();
    h.SetHealth(-10);
    System.out.println("Health Value = " + h.GetHealth());
    h.SetHealth(170);
    System.out.println("Health Value = " + h.GetHealth());
    h.SetHealth(90);
    System.out.println("Health Value = " + h.GetHealth());
  }
}
```

Output

```
Health Value = 0
Health Value = 100
Health Value = 90
```

Explanation

The **health** field inside the Hero class is private which makes it inaccessible through a **Hero** class instance.

By providing a getter and setter function, the **Hero** Class developer gave access to the health field while maintaining control over how it can be changed.

As you can see, the **SetHealth** function inside the **Hero** class will change the **health** field value but makes sure that it stays between 0 and 100.

The class developer can write the logic inside the setters and getters to control what the class user can do.

Static fields and methods

- The **static** modifier allows you to attach a field or method to the class rather than to instances of the class.
- All instances of the class will share the **static** fields and methods.
- The memory of all **static** members is allocated when the application starts.
- Code external to the class should call **static** fields and methods by using **the class name**.
- The **static** modifier can be combined with any other access modifier.

Example

```java
public class GameObject
{
      public static int numberOfObjects = 0;
      /* More data fields here */

      public GameObject()
      {
           ++numberOfObjects;
      }
}
```

```java
public class Main
{
  public static void main(String[] args)
  {
    GameObject go1 = new GameObject();
    System.out.print("Number of objects: ");
    System.out.println(GameObject.numberOfObjects);

    GameObject go2 = new GameObject();
    System.out.print("Number of objects: ")
    System.out.println(GameObject.numberOfObjects);

    GameObject go3 = new GameObject();
    System.out.print("Number of objects: ");
    System.out.println(GameObject.numberOfObjects);

    GameObject go4 = new GameObject();
    System.out.print("Number of objects: ");
    System.out.println(GameObject.numberOfObjects);
  }
}
```

Output

```
Number of objects: 1
Number of objects: 2
Number of objects: 3
Number of objects: 4
```

Explanation

All the **GameObject** instances are sharing the **numberOfObjects static int**. The **GameObject's** constructor is being called every time we create a new instance which leads to incrementing the numberOfObject's value by one. That is a nice way to keep track of how many game objects we created so far in the game.

Code outside the class should call the static field by using the class name instead of the instance name: **GameObject.numberOfObjects**.

Notes: **It is possible to access the public static fields and methods through instances but this is not recommended because it does not make it clear that the field is shared by all instances.**

- **Static** methods can only access the class' static members.

Example

```
public class GameObject
{
        static public int numberOfObjects = 0;
        public float positionX;
        public float positionY;

        public GameObject()
        {
                ++numberOfObjects;
        }

        static public void PrintAllFields()
        {
                System.out.println(numberOfObjects);
                System.out.println(positionX);
                System.out.println(positionY);
        }
}
```

```
public class Main
{
  public static void main(String[] args)
  {
    GameObject go1 = new GameObject();
    GameObject.PrintAllFields();
  }
}
```

Output

```
Exception in thread "main" java.lang.Error: Unresolved compilation
problems:
	Cannot make a static reference to the non-static field positionX
	Cannot make a static reference to the non-static field positionY

	at GameObject.PrintAllFields(GameObject.java:16)
	at Main.main(Main.java:6)
```

Explanation

The above error states that the static **PrintAllFields** function cannot access the **positionX** and **positionY** fields because they are not **static**.

Static and inheritance

- Static field's and method's inheritance depends on the second attribute that is given to the field or method (**public** static / **private** static / **protected** static / **(package-private)**static).

Example

```java
public class GameObject
{
    public static int numberOfObjects = 0;
    /* More data fields here */

    public GameObject()
    {
        ++numberOfObjects;
    }
}
```

```java
public class Knight extends GameObject
{
    /* Data fields here */

    public Knight()
    {
    }
}
```

```java
public class Dragon extends GameObject
{
    /* Data fields here */
```

```
      public Dragon()
      {

      }
}
```

```
public class Main
{
  public static void main(String[] args)
  {
    GameObject go1 = new GameObject();
    System.out.print("Number of objects: ");
    System.out.println(GameObject.numberOfObjects);

    GameObject go2 = new GameObject();
    System.out.print("Number of objects: ");
    System.out.println(GameObject.numberOfObjects);

    Knight player = new Knight();
    System.out.print("Number of objects: ");
    System.out.println(GameObject.numberOfObjects);

    Dragon boss = new Dragon();
    System.out.print("Number of objects: ");
    System.out.println(GameObject.numberOfObjects);
  }
}
```

Output

```
Number of objects: 1
Number of objects: 2
Number of objects: 3
Number of objects: 4
```

Explanation

All the **GameObject** and **GameObject** subclass' instances share the **numberOfObjects** static int. The **GameObject's** constructor is being called everytime we create a new instance which leads to incrementing the **numberOfObject's** value by one. As mentioned earlier, this is a good way to keep track of how many game objects we created so far in the game.

Code outside the class can access the **static** field by using the baseclass name, subclass name or through any instance of the baseclass / subclass:

GameObject.numberOfObjects	**ACCESSIBLE**
Knight.numberOfObjects	**ACCESSIBLE**
Dragon.numberOfObjects	**ACCESSIBLE**
go1.numberOfObjects	**ACCESSIBLE (WITH WARNING*)**
player.numberOfObjects	**ACCESSIBLE (WITH WARNING*)**
boss.numberOfObjects	**ACCESSIBLE (WITH WARNING*)**

**** The following warning will be shown:***
The static field GameObject.numberOfObjects should be accessed in a static way

- Static fields are useful to share data between all instances of a class. Static methods are very useful for creating utility classes that encapsulate related methods, such as the **Math** library, i.e. **Math.abs(double x).** Math is the name of a class, and abs is the name of a method in it.

Virtual Methods

- In Java all methods are by default **virtual**. This means that they can be **overridden** by an inherited class.
- **Overriding** an inherited method redefines its behavior.
- Only inherited methods can be overridden.
- A subclass overrides a method by defining a method that matches the signature and return type of the baseclass method.

Example

```java
public class GameObject
{
        public void Attack()
        {
                System.out.println("Run");
        }
}
```

```java
public class Knight extends GameObject
{
        public void Attack()
        {
                System.out.println("Swing the sword");
        }
}
```

```
public class Dragon extends GameObject
{
     public void Attack()
     {
          System.out.println("Breath fire");
     }
}
```

```
public class Main
{
     public static void main(String[] args)
     {
         GameObject go = new GameObject();
         go.Attack();

         Knight player = new Knight();
         player.Attack();

         Dragon boss = new Dragon();
         boss.Attack();
     }
}
```

Output

```
Run
Swing the sword
Breath fire
```

Explanation

Although both the **Knight** and **Dragon** classes inherited the **Attack** method, they were able to override it and give it their own behavior.

Using the super keyword in the overridden method

- When overriding a method, programmers often want to add to the behavior of the superclass method they are overriding instead of completely replacing the behavior. This requires a mechanism that allows a method in a subclass to call the superclass version of itself.
- The **super** keyword provides such mechanism. Using the **super** keyword you can access the methods defined in the base class inside the overridden method.

Example

```
public class GameObject
{
    public void Attack()
    {
        System.out.print("Run and ");
    }
}
```

```
public class Knight extends GameObject
{
    public void Attack()
    {
        super.Attack();
        System.out.println("swing the sword");
    }
}
```

```
public class Dragon extends GameObject
{
    public void Attack()
    {
        super.Attack();
        System.out.println("breath fire");
    }
}
```

```
public class Main
{
    public static void main(String[] args)
    {
      Knight player = new Knight();
      player.Attack();

      Dragon boss = new Dragon();
      boss.Attack();
    }
}
```

Output

```
Run and swing the sword
Run and breath fire
```

Explanation	The **Knight** and **Dragon** classes want to use the baseclass' **Attack** behavior but add some extra unique behavior in their inherited **Attack**. By using the **super** keyword, the subclasses were able to access the baseclass' **Attack** method and use it in their **overridden Attack** method.

The final Keyword

- In Java all methods are by default **virtual**. This means that they can be **overridden** by an inherited class.
- What if the base class doesn't want to allow any sub class to override a certain function?
- The **final** keyword helps with that. Adding the **final** keyword to a method prevents sub classes from overriding it.

Example

```
public class GameObject
{
    final public void Attack()
    {
        System.out.print("Run and ");
    }
}
```

```
public class Knight extends GameObject
{
    public void Attack()
    {
        super.Attack();
        System.out.println("swing the sword");
    }
}
```

```
public class Main
{
    public static void main(String[] args)
    {
       Knight player = new Knight();
       player.Attack();
    }
}
```

Output

```
Exception in thread "main" java.lang.VerifyError: class Knight overrides final method Attack.()V
	at java.lang.ClassLoader.defineClass1(Native Method)
	at java.lang.ClassLoader.defineClass(Unknown Source)
	at java.security.SecureClassLoader.defineClass(Unknown Source)
	at java.net.URLClassLoader.defineClass(Unknown Source)
	at java.net.URLClassLoader.access$100(Unknown Source)
	at java.net.URLClassLoader$1.run(Unknown Source)
	at java.net.URLClassLoader$1.run(Unknown Source)
	at java.security.AccessController.doPrivileged(Native Method)
	at java.net.URLClassLoader.findClass(Unknown Source)
	at java.lang.ClassLoader.loadClass(Unknown Source)
	at sun.misc.Launcher$AppClassLoader.loadClass(Unknown Source)
	at java.lang.ClassLoader.loadClass(Unknown Source)
	at Main.main(Main.java:5)
```

Explanation

Overriding a final method leads to the above compiler error.

Polymorphism

- Polymorphism is the ability to create an object that has more than one form.
- Its purpose is to implement a style of programming in which objects of various types define a common interface of operations for users.

Example

```java
public class GameObject
{
	public void Attack()
	{
		System.out.println("Run");
	}
}
```

```java
public class Knight extends GameObject
{
	public void Attack()
	{
		System.out.println("Swing the sword");
	}
}
```

```
public class Dragon extends GameObject
{
	public void Attack()
	{
		System.out.println("Breath fire");
	}
}
```

```
public class Main
{
	public static void main(String[] args)
	{
	    GameObject go = new GameObject();
	    go.Attack();

	    GameObject player = new Knight();
	    player.Attack();

	    GameObject boss = new Dragon();
	    boss.Attack();
	}
}
```

Output

```
Run
Swing the sword
Breath fire
```

Explanation

"go", **"player"** and **"boss"** all have the same type **GameObject**. The **Knight** and **Dragon** classes are also of type **GameObject** since they extend from the GameObject class. That allows us to do the following:

```
GameObject player = new Knight();
GameObject boss = new Dragon();
```

Now the question is, what happens when the **"player"** or the **"boss"** instance call their **"Attack"** method? As seen in the example above, each instance was able to access its own overridden "Attack" method. This is the essence of polymorphism.

Example 2

```
public class GameObject
{
        public void WhoAmI()
        {
        }
}
```

```
public class Knight extends GameObject
{
        public void WhoAmI()
        {
                System.out.println("I'm a knight");
        }
}
```

```
public class Dragon extends GameObject
{
        public void WhoAmI()
        {
                System.out.println("I'm a dragon");
        }
}
```

```
public class Main
{
        public static void main(String[] args)
        {
                GameObject objects[] = new GameObject [10];

                for(int i = 0; i < 10; ++i)
                {
                        if(Math.random() < 0.5)
                        {
                                objects[i] = new Knight();
                        }
                        else
                        {
                                objects[i] = new Dragon();
                        }
                }
        for(int i = 0; i < 10; ++i)
        {
                objects[i].WhoAmI();
        }
    }
}
```

Output

```
I'm a knight
I'm a dragon
I'm a knight
I'm a dragon
I'm a knight
I'm a knight
I'm a knight
I'm a dragon
I'm a dragon
I'm a knight
```

Explanation

Polymorphism is very handy in games. Assume we have an array that contains all the level's game objects. We don't know what type of game object we have in every element but, because of polymorphism, we can call the **WhoAmI** method and expect the right **WhoAmI** method to be called according to the game object type. Choosing at run-time which method to call according to the object's type is known as **dynamic binding**.

Note: The above example is randomly adding game objects in the array so the output can be different every run.

Downcasting

Example 2

```java
public class GameObject
{
    public GameObject()
    {

    }
}
```

```java
public class Knight extends GameObject
{
    public int health;

    public Knight()
    {
        health = 100;
    }
}
```

```java
public class Main
{
    public static void main(String[] args)
    {
        GameObject player = new Knight();
        System.out.println(player.health);
    }
}
```

Output

```
Exception in thread "main" java.lang.Error: Unresolved compilation
problem:
    health cannot be resolved or is not a field

    at Main.main(Main.java:6)
```

Explanation

The **player** instance is of type **GameObject**. Even though we instantiated it with **"new Knight",** the **"health"** field is not seen as part of a **GameObject** instance by the compiler. That leads to the above compiler error.

Solution

```
public class Main
{
      public static void main(String[] args)
      {
            GameObject player = new Knight();
            System.out.println(((Knight)player).health);
      }
}
```

Explanation

By replacing **"player.health"** with **"((Knight)player).health"** we are telling the compiler to treat the "**player"** instance as a **Knight** instead of a **GameObject**. This is called **downcasting**.

Note: You should only downcast when you are sure that the instance is actually of the downcasted type.

Override vs Hide

- Inherited static methods cannot be overridden. Instead, when we attempt to **override** them we actually **hide** them.

Example 2

```
public class GameObject
{
  public static void StaticMethod()
  {
    System.out.println("Static Method in GameObject");
  }
}
```

```
public class Knight extends GameObject
{
  public static void StaticMethod()
  {
    System.out.println("Static Method in Knight");
  }
}
```

```
public class Main
{
	public static void main(String[] args)
	{
		GameObject player = new Knight();
		GameObject.StaticMethod();
		player.StaticMethod();

		Knight player2 = new Knight();
		Knight.StaticMethod();
		player2.StaticMethod();
	}
}
```

Output

```
Static Method in GameObject
Static Method in GameObject
Static Method in Knight
Static Method in Knight
```

Explanation

With a non static public method, the statement **"player.StaticMethod();"** would have called the method in the **Knight** class (following the polymorphism rules). Instead, since the **StaticMethod** method is static, when attempting to override it we actually hid it, which means that the program will call the function depending on the caller's type.