More on Classes

Encapsulation

There are 4 core concepts in OOP: encapsulation, inheritance, polymorphism, and **abstraction**.

The idea behind encapsulation is to ensure that implementation details are not visible to users. The variables of one class will be hidden from the other classes, accessible only through the methods of the current class. This is called **data hiding**.
To achieve encapsulation in Java, declare the class' variables as **private** and provide public setter and getter methods to modify and view the variables' values.

For example:

```
class BankAccount {
  private double balance=0;
  public void deposit(double x) {
    if(x > 0) {
      balance += x;
    }
  }
}
```

This implementation hides the **balance** variable, enabling access to it only through the **deposit** method, which validates the amount to be deposited before modifying the variable.

> In summary, **encapsulation** provides the following benefits:
> - Control of the way data is accessed or modified
> - More flexible and easily changed code
> - Ability to change one part of the code without affecting other parts

Inheritance

Inheritance is the process that enables one class to acquire the properties (methods and variables) of another. With inheritance, the information is placed in a more manageable, hierarchical order.

The class inheriting the properties of another is the **subclass** (also called derived class, or child class); the class whose properties are inherited is the **superclass** (base class, or parent class).

To inherit from a class, use the extends keyword.
This example shows how to have the class **Dog** to inherit from the class **Animal**.

```
class Dog extends Animal {
 // some code
}
```

> Here, Dog is the **subclass**, and Animal is the **superclass**.

Inheritance

When one class is inherited from another class, it inherits all of the superclass' **non-private** variables and methods.
Example:

```
class Animal {
  protected int legs;
  public void eat() {
    System.out.println("Animal eats");
  }
}

class Dog extends Animal {
  Dog() {
    legs = 4;
  }
}
```

As you can see, the Dog class inherits the legs variable from the Animal class.
We can now declare a Dog object and call the **eat** method of its superclass:

```
class MyClass {
  public static void main(String[ ] args) {
    Dog d = new Dog();
    d.eat();
  }
}
```

Try It Yourself

Recall the **protected** access modifier, which makes the members visible only to the subclasses.

Inheritance

Constructors are not member methods, and so are not inherited by subclasses.
However, the constructor of the superclass is called when the subclass is instantiated.
Example:

```
class A {
  public A() {
    System.out.println("New A");
  }
}
class B extends A {
  public B() {
    System.out.println("New B");
  }
}

class Program {
  public static void main(String[ ] args) {
     B obj = new B();
  }
}
```

```
/*Outputs
"New A"
"New B"
*/
```

Try It Yourself

You can access the superclass from the subclass using the **super** keyword.
For example, **super.var** accesses the var member of the superclass.

Polymorphism

Polymorphism, which refers to the idea of "having many forms", occurs when there is a hierarchy of classes related to each other through inheritance.
A call to a member method will cause a different implementation to be executed, depending on the type of the object invoking the method.
Here is an example: **Dog** and **Cat** are classes that inherit from the **Animal** class. Each class has its own implementation of the **makeSound**() method.

```
class Animal {
  public void makeSound() {
    System.out.println("Grr...");
  }
}
class Cat extends Animal {
  public void makeSound() {
    System.out.println("Meow");
  }
}
class Dog extends Animal {
  public void makeSound() {
    System.out.println("Woof");
  }
}
```

As all **Cat** and **Dog** objects are **Animal** objects, we can do the following in **main**:

```
public static void main(String[ ] args) {
  Animal a = new Dog();
  Animal b = new Cat();
}
```

We've created two reference variables of type Animal, and pointed them to the **Cat** and **Dog** objects.
Now, we can call the makeSound() methods.

```
a.makeSound();
//Outputs "Woof"

b.makeSound();
//Outputs "Meow"
```

Try It Yourself

As the reference variable **a** contains a Dog object, the makeSound() method of the Dog class will be called.
The same applies to the **b** variable.

This demonstrates that you can use the **Animal** variable without actually knowing that it contains an object of the subclass.
This is very useful when you have multiple subclasses of the superclass.

Method Overriding

As we saw in the previous lesson, a subclass can define a behavior that's specific to the subclass type, meaning that a subclass can implement a parent class method based on its requirement.
This feature is known as method **overriding**.
Example:

```
class Animal {
  public void makeSound() {
    System.out.println("Grr...");
  }
}
class Cat extends Animal {
  public void makeSound() {
    System.out.println("Meow");
  }
}
```

Try It Yourself

In the code above, the Cat class overrides the **makeSound**() method of its superclass Animal.

Rules for Method Overriding:
- Should have the **same** return type and arguments
- The **access level** cannot be more restrictive than the overridden method's access level
(Example: If the superclass method is declared public, the overriding method in the sub class can be neither private nor protected)
- A method declared **final** or **static** cannot be overridden
- If a method cannot be inherited, it cannot be overridden
- Constructors cannot be overridden

> Method overriding is also known as **runtime polymorphism**.

Method Overloading

When methods have the same name, but different parameters, it is known as method **overloading**.
This can be very useful when you need the same method functionality for different types of parameters.
The following example illustrates a method that returns the maximum of its two parameters.

```
int max(int a, int b) {
  if(a > b) {
    return a;
  }
  else {
    return b;
  }
}
```

The method shown above will only work for parameters of type **integer**.
However, we might want to use it for **doubles**, as well. For that, you need to overload the **max** method:

```
double max(double a, double b) {
  if(a > b) {
    return a;
  }
  else {
    return b;
  }
}
```

Try It Yourself

Now, our **max** method will also work with **doubles**.
An overloaded method **must** have a different argument list; the parameters should differ in their type, number, or both.

> Another name for method overloading is **compile-time polymorphism**.

Abstraction

Data **abstraction** provides the outside world with only essential information, in a process of representing essential features without including implementation details.
A good real-world example is a *book*. When you hear the term book, you don't know the exact specifics, such as the page count, the color, or the size, but you understand the idea, or abstraction, of a book.
The concept of **abstraction** is that we focus on essential qualities, rather than the specific characteristics of one particular example.

In Java, abstraction is achieved using **abstract classes** and **interfaces**.
An abstract class is defined using the **abstract** keyword.
- If a class is declared abstract it cannot be instantiated (you cannot create objects of that type).
- To use an abstract class, you have to inherit it from another class.
- Any class that contains an abstract method should be defined as abstract.

> An abstract method is a method that is declared without an implementation (without braces, and followed by a semicolon): **abstract void walk();**

Abstract Class

For example, we can define our Animal class as abstract:

```
abstract class Animal {
  int legs = 0;
  abstract void makeSound();
}
```

The makeSound method is also abstract, as it has no implementation in the superclass.
We can inherit from the Animal class and define the makeSound() method for the subclass:

```
class Cat extends Animal {
  public void makeSound() {
    System.out.println("Meow");
  }
}
```

Try It Yourself

> Every Animal makes a sound, but each has a different way to do it. That's why we define an abstract class Animal, and leave the implementation of how they make sounds to the subclasses.
> This is used when there is no meaningful definition for the method in the superclass.

Interfaces

An **interface** is a completely abstract class that contains only abstract methods.
Some specifications for interfaces:
- Defined using the **interface** keyword.
- May contain only static final variables.
- Cannot contain a constructor because interfaces cannot be instantiated.
- Interfaces can extend other interfaces.
- A class can implement any number of interfaces.

An example of a simple interface:

```
interface Animal {
  public void eat();
  public void makeSound();
}
```

Interfaces have the following properties:
- An interface is implicitly abstract. You do not need to use the abstract keyword while declaring an interface.
- Each method in an interface is also implicitly abstract, so the abstract keyword is not needed.
- Methods in an interface are implicitly public.

> A class can inherit from just **one** superclass, but can implement **multiple** interfaces!

Interfaces

Use the **implements** keyword to use an interface with your class.

```
interface Animal {
  public void eat();
  public void makeSound();
}

class Cat implements Animal {
  public void makeSound() {
    System.out.println("Meow");
  }
  public void eat() {
    System.out.println("omnomnom");
  }
}
```

Try It Yourself

> When you implement an interface, you need to override all of its methods.

Type Casting

Assigning a value of one type to a variable of another type is known as **Type Casting**.

To cast a value to a specific type, place the type in parentheses and position it in front of the value.
Example:

```
int a = (int) 3.14;
System.out.println(a);
//Outputs 3
```

Try It Yourself

The code above is casting the value 3.14 to an integer, with 3 as the resulting value.
Another example:

```
double a = 42.571;
int b = (int) a;
System.out.println(b);
//Outputs 42
```

Try It Yourself

Java supports automatic type casting of integers to floating points, since there is no loss of precision.
On the other hand, type casting is mandatory when assigning floating point values to integer variables.

Type Casting

For classes, there are two types of casting.

Upcasting

You can cast an instance of a subclass to its superclass.
Consider the following example, assuming that Cat is a subclass of Animal.

```
Animal a = new Cat();
```

Java automatically upcasted the Cat type variable to the Animal type.

Downcasting

Casting an object of a superclass to its subclass is called **downcasting**.
Example:

```
Animal a = new Cat();
((Cat)a).makeSound();
```

This will try to cast the variable a to the **Cat** type and call its makeSound() method.

Why is upcasting automatic, downcasting manual? Well, upcasting can never fail. But if you have a group of different Animals and want to downcast them all to a Cat, then there's a chance that some of these Animals are actually Dogs, so the process fails.

Anonymous Classes

Anonymous classes are a way to extend the existing classes on the fly.
For example, consider having a class Machine:

```
class Machine {
  public void start() {
    System.out.println("Starting...");
  }
}
```

When creating the Machine object, we can change the start method on the fly.

```
public static void main(String[ ] args) {
  Machine m = new Machine() {
    @Override public void start() {
      System.out.println("Wooooo");
    }
```

```
  };
  m.start();
}
//Outputs "Wooooo";
```

Try It Yourself

After the constructor call, we have opened the curly braces and have overridden the **start** method's implementation on the fly.

> The **@Override** annotation is used to make your code easier to understand, because it makes it more obvious when methods are overridden.

Anonymous Classes

The modification is applicable only to the current object, and not the class itself. So if we create another object of that class, the start method's implementation will be the one defined in the class.

```
class Machine {
  public void start() {
    System.out.println("Starting...");
  }
}
public static void main(String[ ] args) {
  Machine m1 = new Machine() {
    @Override public void start() {
      System.out.println("Wooooo");
    }
  };
  Machine m2 = new Machine();
  m2.start();
  }
//Outputs "Starting..."
```

Try It Yourself

> Tap **Try It Yourself** to play around with the code!

Inner Classes

Java supports **nesting** classes; a class can be a member of another class.
Creating an inner class is quite simple. Just write a class within a class. Unlike a class, an inner class can be private. Once you declare an inner class private, it cannot be accessed from an object outside the class.
Example:

```
class Robot {
  int id;
  Robot(int i) {
    id = i;
    Brain b = new Brain();
    b.think();
  }
```

```
  private class Brain {
    public void think() {
      System.out.println(id + " is thinking");
    }
  }

}
```

Try It Yourself

The class **Robot** has an inner class **Brain**. The inner class can access all of the member variables and methods of its outer class, but it cannot be accessed from any outside class.

Comparing Objects

Remember that when you create objects, the variables store references to the objects.
So, when you compare objects using the equality testing operator (==), it actually compares the references and not the object values.
Example:

```
class Animal {
  String name;
  Animal(String n) {
    name = n;
  }
}

class MyClass {
  public static void main(String[ ] args) {
    Animal a1 = new Animal("Robby");
    Animal a2 = new Animal("Robby");
    System.out.println(a1 == a2);
  }
}
//Outputs false
```

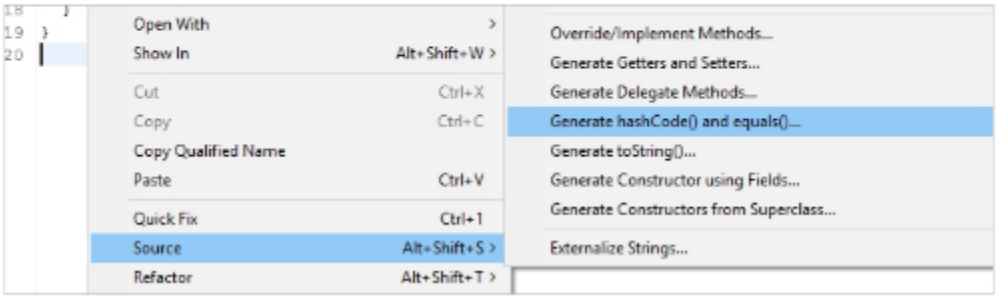
Try It Yourself

Despite having two objects with the same name, the equality testing returns false, because we have two different objects (two different references or memory locations).

---

equals()

Each object has a predefined **equals**() method that is used for semantical equality testing.
But, to make it work for our classes, we need to override it and check the conditions we need.
There is a simple and fast way of generating the equals() method, other than writing it manually.
Just right click in your class, go to **Source->Generate hashCode() and equals()...**



This will automatically create the necessary methods.

```
class Animal {
  String name;
  Animal(String n) {
    name = n;
  }
  @Override
  public int hashCode() {
    final int prime = 31;
    int result = 1;
    result = prime * result + ((name == null) ? 0 : name.hashCode());
    return result;
  }
  @Override
  public boolean equals(Object obj) {
    if (this == obj)
      return true;
    if (obj == null)
      return false;
    if (getClass() != obj.getClass())
      return false;
    Animal other = (Animal) obj;
    if (name == null) {
      if (other.name != null)
        return false;
    } else if (!name.equals(other.name))
      return false;
    return true;
  }
}
```

The automatically generated hashCode() method is used to determine where to store the object internally. Whenever you implement **equals**, you MUST also implement **hashCode**.
We can run the test again, using the **equals** method:

```
public static void main(String[ ] args) {
  Animal a1 = new Animal("Robby");
  Animal a2 = new Animal("Robby");
  System.out.println(a1.equals(a2));
}
//Outputs true
```

Try It Yourself

You can use the same menu to generate other useful methods, such as **getters** and **setters** for your class attributes.

Enums

An Enum is a special type used to define collections of constants.
Here is a simple Enum example:

```
enum Rank {
  SOLDIER,
  SERGEANT,
  CAPTAIN
}
```

Note that the values are **comma-separated**.
You can refer to the constants in the enum above with the **dot** syntax.

```
Rank a = Rank.SOLDIER;
```

Basically, Enums define variables that represent members of a fixed set.

Enums

After declaring an Enum, we can check for the corresponding values with, for example, a **switch** statement.

```
Rank a = Rank.SOLDIER;

switch(a) {
  case SOLDIER:
    System.out.println("Soldier says hi!");
    break;
  case SERGEANT:
    System.out.println("Sergeant says Hello!");
  break;
  case CAPTAIN:
    System.out.println("Captain says Welcome!");
    break;
}
//Outputs "Soldier says hi!"
```

Try It Yourself

Tap **Try It Yourself** to play around with the code!

Enums

You should always use Enums when a variable (especially a method parameter) can only take one out of a small set of possible values.
If you use Enums instead of integers (or String codes), you increase compile-time checking and avoid errors from passing in invalid constants, and you document which values are legal to use.

Some sample Enum uses include month names, days of the week, deck of cards, etc.

Java API

The Java API is a collection of classes and interfaces that have been written for you to use.
The Java API Documentation with all of the available APIs can be located on the Oracle website at **http://docs.oracle.com/javase/7/docs/api/**
Once you locate the package you want to use, you need to import it into your code.
The package can be imported using the import keyword.
For example:

```
import java.awt.*;
```

The **awt** package contains all of the classes for creating user interfaces and for painting graphics and images.

The wildcard character (*) is used to import all of the classes in the package.

End.