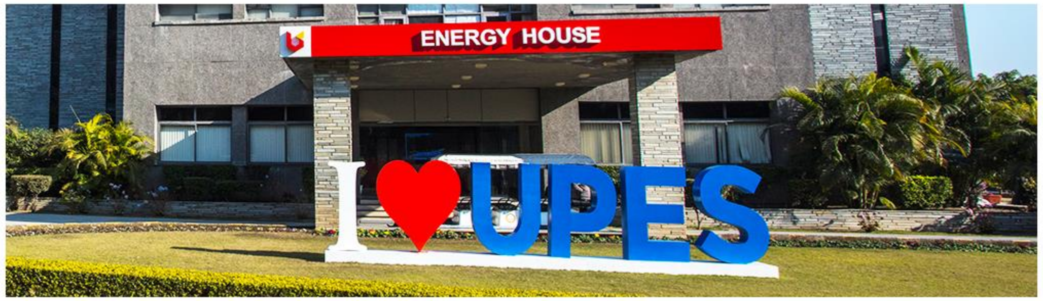

UPES

UNIVERSITY WITH A PURPOSE

Object Oriented Programming

Hello World Program

```
class HelloWorld {
public static void main(String[] args) {
System.out.println("Hello, world");
     }
}
```

Compile: javac HelloWorld.java
Execute: java HelloWorld
Output: Hello, world

Hello World Program contd..

main:

- The program declares a class called `HelloWorld` with a single member: a method called `main`.
- The `main` method of a class, if declared exactly as shown, is executed when you run the class as an application. When run, a `main` method can create objects, evaluate expressions, invoke other methods, and do anything else needed to define an application's behavior.
- The `main` method is declared public so that anyone can invoke it (in this case the Java virtual machine) and static, meaning that the method belongs to the class and is not associated with a particular instance of the class.
- The `main` method is declared void because it doesn't return a value and so has no return type.

Hello World Program contd..

main:

- The main method's only parameter is an array of String objects, referred to by the name `args`. Arrays of objects are denoted by the square brackets [] that follow the type name.
- In this example, the body of main contains a single statement that invokes the `println` method the semicolon ends the statement. A method is invoked by supplying an object reference (in this case `System.out` the out field of the System class) and a method name (`println`) separated by a dot (.).
- `HelloWorld` uses the out object's `println` method to print a newline-terminated string on the standard output stream. The string printed is the string literal "`Hello,world`" , which is passed as an argument to `println`. A string literal is a sequence of characters contained within double-quotes " and ".

Variables

```
class Fibonacci {
      /** Print out the Fibonacci sequence for values < 50
*/
      public static void main(String[] args) {
      int lo = 1;
      int hi = 1;
      System.out.println(lo);
      while (hi < 50) {
      System.out.println(hi);
      hi = lo + hi; // new hi
      lo = hi - lo; /* new lo is (sum - old lo)
                  that is, the old hi */
}     }     }
```

Variables contd..

- The Java programming language has built-in "primitive" data types to support integer, floating-point, boolean, and character values. The primitive data types are:
  - boolean: either True or false
  - char: 16-bit Unicode character (unsigned)
  - byte: 8-bit integer (signed)
  - short: 16-bit integer (signed)
  - int: 32-bit integer (signed)
  - long: 64-bit integer (signed)
  - float: 32-bit floating-point (IEEE 754)
  - double: 64-bit floating-point (IEEE 754)
- For each primitive type there is also a corresponding object type, generally termed a "wrapper" class. For example, the class Integer is the wrapper class for int.

Comments in Code

There are three styles of comments:

1. Text that occurs between /* and */ is ignored by the compiler. This style of comment can be used on part of a line, a whole line, or more commonly (as in the example) to define a multiline comment.
2. For single line and part line comments you can use // which tells the compiler to ignore everything after it on that line.
3. The third kind of comment appears at the very top, between /** and */. A comment starting with two asterisks is a documentation comment ("doc comment" for short). Documentation comments are intended to describe declarations that follow them.

Named Constants

- Constants are values like 12, 17.9, and "StringsLike This". Constants, or literals as they are also known, are the way you specify values that are not computed and recomputed but remain, well, constant for the life of a program.
- A named constant is a constant value that is referred to by a name. For example, we may choose the name MAX to refer to the constant 50 in the Fibonacci example.
- To make the value a constant we declare the field as **final**. A final field or variable is one that once initialized can never have its value changed. Further, because we don't want the named constant field to be associated with instances of the class, we also declare it as **static**.

Named Constants contd..

```
class Fibonacci2 {
     static final int MAX = 50;
     /** Print the Fibonacci sequence for values < MAX */
     public static void main(String[] args) {
      int lo = 1;
      int hi = 1;
      System.out.println(lo);
      while (hi < MAX) {
           System.out.println(hi);
           hi = lo + hi;
           lo = hi - lo;      }
      }
}
```

Named Constants contd..

- We can group related constants within a class. For example, a card game might use these constants:

```
class Suit {
final static int CLUBS = 1;
final static int DIAMONDS = 2;
final static int HEARTS = 3;
final static int SPADES = 4;
}
```

- To refer to a static member of a class we use the name of the class followed by dot and the name of the member. With the above declaration, suits in a program would be accessed as Suit.HEARTS, Suit.SPADES, and so on, thus grouping all the suit names within the single name Suit.
- Notice that the order of the modifiers `final` and `static` makes no difference though you should use a consistent order.

Classes and Objects

Every object has a class that defines its data and behavior. Each class has three kinds of members:

1. **Fields** are data variables associated with a class and its objects. Fields store results of computations performed by the class.

2. **Methods** contain the executable code of a class. Methods are built from statements. The way in which methods are invoked, and the statements contained within those methods, are what ultimately directs program execution.

3. **Classes** and **interfaces** can be members of other classes or interfaces.

Example:

```
class Point {
public double x, y;
}
```

Creating Objects

- Objects are created by expressions containing the “**new**” keyword. Creating an object from a class definition is also known as instantiation; thus, objects are often called instances.
- Newly created objects are allocated within an area of system memory known as the **heap**. All objects are accessed via object references, any variable that may appear to hold an object actually contains a reference to that object. Object references are **null** when they do not reference any object.
- Most of the time, you can use "object" and "object reference" interchangeably.

Creating Objects contd..

In the Point class, suppose you are building a graphics application in which you need to track lots of points. You represent each point by its own concrete Point object. Here is how you might create and initialize Point objects:

```
Point lowerLeft = new Point();
Point upperRight = new Point();
Point middlePoint = new Point();
lowerLeft.x = 0.0;
lowerLeft.y = 0.0;
upperRight.x = 1280.0;
upperRight.y = 1024.0;
middlePoint.x = 640.0;
middlePoint.y = 512.0;
```

- Each Point object is unique and has its own copy of the x and y fields. Changing x in the object lowerLeft, for example, does not affect the value of x in the object upperRight.

- The fields in objects are known as **instance variables**, because there is a unique copy of the field in each object (instance) of the class.

UPES
UNIVERSITY WITH A PURPOSE

Creating Objects contd..

- When we use new to create an object, a special piece of code, known as a **constructor**, is invoked to perform any initialization the object might need.
- A constructor has the same name as the class that it constructs and is similar to a method, including being able to accept arguments. If you don't declare a constructor in your class, the compiler creates one for you that takes no arguments and does nothing.
- When we say "new Point()“ we're asking that a Point object be allocated and because we passed in no arguments, the no-argument constructor be invoked to initialize it.

Static or Class Fields

- Per-object fields are usually what you need. You usually want a field in one object to be distinct from the field of the same name in every other object instantiated from that class.
- Sometimes, though, you want fields that are shared among all objects of that class. These shared variables are known as class variables: variables specific to the class as opposed to objects of the class.
- You obtain class-specific fields by declaring them static, and they are therefore commonly called static fields. For example, a Point object to represent the origin might be common enough that you should provide it as a static field in the Point class: **public static Point origin = new Point();**
- If this declaration appears inside the declaration of the `Point` class, there will be exactly one piece of data called `Point.origin` that always refers to an object at (0.0, 0.0).
- This static field is there no matter how many Point objects are created, even if none are created. The values of x and y are zero because that is the default for numeric fields that are not explicitly initialized to a different value.

UNIVERSITY WITH A PURPOSE

Static or Class Fields

Important Points:

- **Static variables** can be created at class level only.
- A static variable is allotted memory once.
- **Static methods** cannot use this or super keywords.
- Abstract methods cannot be static.
- static methods cannot be overridden.
- static methods can only access static variables and other static methods.
- There can be multiple static blocks in a class.
- A static inner class can access all static variables and methods of the outer class.
- In Java, the outer class cannot be static.

The Garbage Collector

- After creating an object with new, how do you get rid of the object when you no longer want it?

  stop referring to it.

- When an object is no longer referenced, garbage collector can remove it from the storage allocation heap.

A Simple Class

```
class Body {
public long id;
public String name;
public static long nextID = 0;
}
Body mercury;
```

- This declaration states that mercury is a variable that can hold a reference to an object of type Body. The declaration does not create an object, it declares only a reference that is allowed to refer to a Body object. During its existence, the reference mercury may refer to any number of Body objects.

Class Modifiers

A class declaration can be preceded by class modifiers that give the class certain properties:

- **public** A public class is publicly accessible: Anyone can declare references to objects of the class or access its public members. Without a modifier a class is only accessible within its own package.
- **abstract** An abstract class is considered incomplete and no instances of the class may be created. Usually this is because the class contains abstract methods that must be implemented by a subclass.
- **final** A final class cannot be subclassed.
- **strict floating point** A class declared `strictfp` has all floating-point arithmetic in the class evaluated strictly.

NOTE: A class cannot be both final and abstract.

Fields

- A class's variables are called fields. The Body class's name and variables are examples.
- Field declarations can also be preceded by modifiers that control certain properties of the field:

-- access modifiers

-- Static

-- Final

-- Transient: This relates to object serialization

-- volatile: This relates to synchronization and memory model issues

Access Control

- If every member of every class and object were accessible to every other class and object then understanding, debugging, and maintaining programs would be an almost impossible task.
- One of the strengths of object-oriented programming is its support for encapsulation and data hiding.
- To control access from other classes, class members have four possible access modifiers:

1. **private** Members declared private are accessible only in the class itself.
2. **package** Members declared with no access modifier are accessible in classes in the same package, as well as in the class itself.
3. **protected** Members declared protected are accessible in subclasses of the class, in classes in the same package, and in the class itself.
4. **public** Members declared public are accessible anywhere the class is accessible.

NOTE: private and protected access modifiers apply only to members not to the classes or interfaces themselves (unless nested).

Field Initialization

- When a field is declared it can be initialized by assigning it a value of the corresponding type.
- For example, the following are all valid initializers:

```
double zero = 0.0; // constant
double sum = 4.5 + 3.7; // constant expression
double zeroCopy = zero; // field
double rootTwo = Math.sqrt(2); // method invocation
double someVal = sum + 2*Math.sqrt(rootTwo); // mixed
```

Field Initialization contd..

- If a field is not initialized, a default initial value is assigned to it depending on its type:

Type	**Initial Value**
Boolean	false
Char	'\u0000'
byte, short, int, long	0
float, double	+0.0
object reference	null

Static Fields

- Within its own class a static field can be referred to directly, but when accessed externally it must usually be accessed using the class name. For example, we could print the value of nextID as follows:

```
System.out.println(Body.nextID);
```

- A static member may also be accessed using a reference to an object of that class, such as:

```
System.out.println(mercury.nextID);
```

final Fields

- A final variable is one whose value cannot be changed after it has been initialized any attempt to assign to such a field will produce a compile-time error.
- When you decide whether a field should be final, consider three things:

1. Does the field represent an immutable property of the object?
2. Is the value of the field always known at the time the object is created?
3. Is it always practical and appropriate to set the value of the field when the object is created?

UPES
UNIVERSITY WITH A PURPOSE

Creating Objects

In this first version of Body, objects that represent particular celestial bodies are created and initialized like this:

```
Body sun = new Body();
sun.id = Body.nextID++;
sun.name = "Sol";
Body earth = new Body();
earth.id = Body.nextID++;
earth.name = "Earth";
```

References

Gosling, J., Holmes, D. C., & Arnold, K. (2005). The Java programming language.