**Classes and Objects**

* **Classes**
* **Declaring Classes**
  + This is a class declaration

Class MyClass extends MySuperClass implements YourInterface {

// fields

// constructor

// method declarations

}

* + In general, class declarations can include these components, in order
    - Modifiers such as public, private and number of others
    - The class name, with the initial letter capitalized be conventions
    - The name of the class’s parent, if any, preceded by the keyword extends
    - A class can only extend one parent.
    - Comma separated list of interfaces implemented by the class, is any, preceded by the keyword implements.
    - A class can implement more than one interface
    - The class body, surrounded by braces {}.
* **Declaring Member Variables**
  + There are several kinds of variable
    - Member variable in a class – these are called fields
    - Variables in a method or block of code – these are called local variables
    - Variables in method declaration – these are called parameters
  + Field declaration is composed of three components
    - Zero or more modifiers, such as private or public
    - The field’s type
    - The field’s name
  + **Access Modifiers**
    - The Modifiers used lets you control what other class have access to member field
      * Public modifier – the Field is accessible from all the classes
      * Private modifier – the Field is accessible only within its own class
  + **Types**
    - All the variables must have a type
    - You can use primitive types, such as int, float , Boolean, etc
    - Or you can use reference types, such as strings, arrays or objects
  + **Variable Names**
    - All the variables whether they are fields, local variables or parameters follow the same naming rules and conventions.
    - The first letter of a class name should be capitalized
    - The first word in a method name should be verb
* **Defining Methods**
  + More generally method declarations have six components.
    - Modifiers
    - The return type
    - The method name
    - The parameter list in parenthesis
    - An exceptional list
    - The method body
  + Two of the components of a method declaration compromise the method signature – The method’s name and the parameter types.
  + Example – calculateAnswer(double, int, double, double)
* **Overloading Methods**
  + The java programming language supports the overloading methods.
  + Java can distinguish between methods with different method signature.
  + This means methods within a class can have same name, if they have different parameter list.
  + The compiler does not consider the return type when differentiating methods.
  + So you cannot declare two methods with same signature even if they have a different return type.
* **Providing Constructor for Your Classes**
  + A class contain constructors that are invoked to create the object of the class blue print.
  + A constructor declaration looks like method declarations except that they use the name of the class and have no return type.
  + It will create the space in the memory and initialize its fields.
  + The compiler automatically provides a no-argument default constructor for any class without constructors.
  + The default constructor will call the no-argument constructor of the super class.
  + If your class has no explicit super class, then it has an implicit super class of Object, which does not have no-argument constructor.
  + You can use the access modifiers in a constructor’s declaration to control which other classes can call the constructor.
  + If any class cannot call a MyClass constructor, it cannot directly create the MyClass object.
* **Passing Information to a Method or a Constructor**
  + Parameter refers to the list of variables in a method declaration.
  + Arguments are actual values that are passed in when the method is invoked.
  + **Parameter Types**
    - You can use any data type for a parameter of a method or constructor.
    - This includes primitive data type and reference data type.
    - The java programming language does not let you pass methods into methods
    - But can pass object into method and then invoke objects methods.
  + **Arbitrary Number of Arguments**
    - You can use a construct called varargs to pass an arbitrary number of values to a method.
    - You can use varargs when you don’t know how many of a particular type of argument will be passed to the method.
    - To use varargs, you follow the type of the last parameter by an ellipse(three dots …) then a space and parameter name.
    - Example

public PrintStream printf(String format, Object... args)

System.out.printf("%s: %d, %s%n", name, idnum, address);

* **Objects**
* **Creating Objects**
  + A class provides the blue print for objects, you can create object from class.
  + Program creates an object and assigns it to a variable.

*Point* ***originOne*** *= new Point (60, 90);*

* + Each of these statement has three parts
    - Declaration – The code set in bold are variable declaration that associate a variable name and its object type.
    - Instantiation – The new keyword is a java operator that creates an object.
    - Initialization – The new operator is followed by call to a constructor, which initializes the new object.
* **Declaring a variable to Refer to an Object**
  + When we declare the primitive variable, compiler reserves the proper amount of memory for that variable.
  + For reference variable, its values will be undetermined until object is created and assigned to it.
  + Simply declare a reference variable does not create an object.
  + For that you need to use new operator.
  + You must assign the object to reference variable before to use, otherwise it will get a compile error.
* **Instantiating a Class**
  + The new operator instantiating a class by allocating memory for a new object and returning reference to that memory.
  + The new operator also invokes the object constructor.
  + The reference is usually assigned to a variable of the appropriate type.

*Point orginOne = new Point (30, 20);*

* + The reference returned by the new operator does not assign to a variable.
  + It can also be used directly in an expression.

*int height = new Rectangle().height();*

* **Using Objects**
  + You can use object to use the value of one of its fields
  + Change one of its fields
  + Or Call one of its method to perform an action
  + **Referencing an Object’s fields**
    - Object fields are accessed by their name.
    - Code that’s outside the object class must use an object reference or expression, followed by the dot operator followed by a simple field name as in

objectReference.fieldName

int height = new Rectangle ().height;

* + **Calling an Object’s Methods**
    - You also can use object reference to invoke an object’s method.
    - You append the method’s simple name to the object reference, with an intervening dot (.) operator.
    - Also you provide within enclosing parenthesis, any arguments to the method.
      * objectReference.methodName (argumentList);
      * objectReference.methodName();
      * int areaOfRectangle = new Rectangle(20, 30).getArea();
  + **The Garbage Collector**
    - The Java platform allows you to create as many Objects as you want(limited, of course, by what your system can handle)
    - And you don’t have to worry about destroying them.
    - The Java run time environment deletes objects when it determines that they are no longer being used.
    - This process is called Garbage collection.
    - You can explicitly drops an object reference by setting the variable to the special value null.
* **More on Classes**
* **Returning a Value from a method**
  + A method returns to the code that invoked it, when it
    - Completes all the statement in the method
    - Reaches a return statement
    - Throws an exception
  + Whichever occurs first
  + You declare a method’s return type in its method declaration, within the body of the method, you use the return statement to return the value.
  + A method declared void does not return a value. It does not need to contain a return statement. But it may do so, In such cases, a return statement can be used to branch out of a control flow block and exit the method and is simply used like this

*return;*

* + The data type of the return value must match the method’s declared return type.
  + A method can return primitive type or reference type.
  + **Returning a Class or Interface**
    - When a method uses a class name as its return type, the class of the type of the returned object must be either a sub class of, or the exact class of the return type.
    - This technique is called covariant return type, means the return type is allowed to vary in the same direction as the subclass.
    - You can use interface name as return types, In this case, the object returned must be implement the specified interface.
* **Using the this Keyword**
  + Within an instance method or a constructor, *this* is a reference to the current object.
  + The object whose method or constructor is being called.
  + You can refer any member of current object from within an instance method or constructor by using this.
  + **Using this With a Field**
    - The most common reason for using *this* keyword is because a field is shadowed by a method or a constructor parameter.

public class Point {

public int x = 0;

public int y = 0;

**//constructor**

**public Point(int x, int y) {**

**this.x = x;**

**this.y = y;**

**}**

}

* + - Each argument of the constructor shadows one of the object’s fields
    - Inside the constructor x is local copy of the constructor’s first argument.
    - To refer to the Point field x, the constructor must use *this.x*
  + **Using this With a Constructor**
    - From within a constructor you can use this keyword to call another constructor in the same class.
    - Doing so is called explicit constructor invocation.
    - If present invocation of another constructor must be the first line of the constructor.

public class Rectangle {

private int x, y;

private int width, height;

public Rectangle() {

**this(0, 0, 0, 0);**

}

public Rectangle(int width, int height) {

**this(0, 0, width, height);**

}

public Rectangle(int x, int y, int width, int height) {

this.x = x;

this.y = y;

this.width = width;

this.height = height;

}

...

}

* **Controlling Access to Member of a Class**
  + Access level modifiers determine whether other classes can use particular field or invoke a particular method.
  + There are two levels of access control.
    - At the top level – public or package-private (no explicit modifier)
    - At the member level – public, private, protected or package-private (no explicit modifier)
  + A class may be declared with the modifier public, in which case the class is visible to all the classes everywhere.
  + If a class has no modifier, its visible only within its own package (the default also known as package-private)
  + At the member level you can also use public modifier or no modifier (package-private) just as with top level classes.
  + For members there are two additional modifiers private and protected.
  + The private modifier specifies that member can only be accessed within its own class.
  + The protected modifier specifies that member can only be accessed within its own package and in addition, by a subclass of its class in another package.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Access Levels** | | | | |
| **Modifier** | **Class** | **Package** | **Subclass** | **World** |
| Public | Y | Y | Y | Y |
| protected | Y | Y | Y | N |
| *no modifier* | Y | Y | N | N |
| Private | Y | N | N | N |

* **Understanding Instance and class members**
  + **Class Variables**
    - When a number of objects are created from the same class blue print, they each have their own distinct copies of instant variables.
    - Each object has own values for these variables, stored in different memory locations.
    - Sometimes you want to have variables that are common to all objects. This is accomplished with static modifier.
    - Fields that have the static modifier in their declaration are called static fields or class variables.
    - They are associated with class rather than with any object.
    - Every instance of the class shares the class variable, which is one fixed location in memory.
    - Any object can change the value of a class variable.
    - Class variable can also be manipulated without creating an instance of the class.

*private static int numberOfBicycles = 0;*

* + - Class Variables are referenced by class name itself.

*className.variableName;*

* + - You can also refer to static fields with an object reference link.
  + **Class Methods**
    - Static methods which have static modifier in their declarations
    - Should be invoked with class name, without the need for creating an instance of the class.

*ClassName.methodName(args)*

* + - You can also refer to static methods with an object reference like

*instanceName.methodName(args);*

* + - A common use for static method is to access static fields.

public **static** int getNumberOfBicycles() {

return numberOfBicycles;

}

* + - Not all the combination of instance and class variables and methods are allowed.
      * Instance methods can access instance variables and instance methods directly.
      * Instance methods can access class variables and class methods directly.
      * Class methods can access class variables and class methods directly.
      * Class methods cannot access instance variables or instance methods directly- they must use an object reference.
    - Class methods cannot use *this* keyword as there is no instance for *this* to refer to.
  + **Constants**
    - The static modifier in combination with final modifier, is also used to define constant.
    - The final modifier indicates that the value of this field cannot change.

static final double PI = 3.141592653589793;

* + - Constant defined by this way cannot be reassigned.
    - By convention the names of the constant value are spelled in upper case letters.
    - If the name is composed of more than one word, the words are separated by an underscore.
* **Initializing fields**
  + You can often provide an initial value for a field in its declaration.

public static int capacity = 10;

* + **Static Initialization Blocks**
    - Static initialization block is a normal block of code enclosed in braces {} and preceded by the static keyword.

static {

// whatever code is needed for initialization goes here

}

* + - A class can have any number of static initialization blocks.
    - They can appear in anywhere in class body.
  + **Initializing Instance Members**
    - Normally you would put code to initialize an instance variable in constructor.
    - There are two alternatives to using a constructor to initialize instance variable.
      * Initializer blocks
      * Final methods
    - Initializer blocks for instance variables looks just like static initializer blocks, but without the static keyword.

{

// whatever code is needed for initialization goes here

}

* + - The java compiler copies initializer blocks into every constructor.
    - A final method cannot be overridden in sub classes.
* **Nested Classes**
  + The Java Programming language allows you to define a class within another class. Such a class is called nested class.

class OuterClass {

...

class NestedClass {

...

}

}

* + Nested classes are divided into two categories.
    - Static nested classes
    - Non-static nested classes
  + Nested classes that are declared static are simply called Static nested classes.
  + Non-static nested classes are called inner classes.

class OuterClass {

...

static class StaticNestedClass {

...

}

class InnerClass {

...

}

}

* + Nested class is a member of its enclosing class.
  + Non-static nested classes have access to other member of the enclosing class, even if they are declared private.
  + Static nested classes do not have access to other member of the enclosing class.
  + As a member of outer class, a nested class can be declared private, public, protected or package private.
  + Recall that outer class can only be declared public or package private.
* **Why Use Nested Classes**
  + There are several compelling reasons for using nested classes.
    - It’s a way of logically grouping classes that are only used in one place.
    - Its increasing encapsulation
    - Nested classes can lead to more readable and maintainable code.
* **Static Nested Classes**
  + As with class methods and variables, a static nested class is associated with its outer class.
  + And like static class methods, static nested class cannot refer directly to instance variables or methods defined in its enclosing class.
  + It can use them only through object reference.
  + Static nested classes are accessed using the enclosing class name.

OuterClass.StaticNestedClass

* + For example to create an object for static nested class, use this syntax

OuterClass.StaticNestedClass nestedObject =

new OuterClass.StaticNestedClass();

* **Inner Classes** 
  + As with instance methods and variables, an inner class is associated with an instance of its enclosing class and has direct access to object methods and fields.
  + Objects that are instance of an inner class exist within an instance of an outer class.

class OuterClass {

...

class InnerClass {

...

}

}

* + To instantiate an inner class, you must first instantiate the outer class, Then create the inner object within the outer object with this syntax,

OuterClass.InnerClass innerObject = outerObject.new InnerClass();

* + Additionally there are two special kind of inner classes
    - Local classes
    - Anonymous classes
* **Enum Types**
  + An enum type is a type whose fields consist of fixed set of constants.
  + Common example is days of a week.
  + You can define the enum type using the enum keyword.

*Public enum Day {*

*SUNDAY, MONDAY, TUESDAY, WEDNESDAY,*

*THURSDAY, FRIDAY, SATURDAY*

*}*

* + All enums implicitly extend java.lang.Enum, enum cannot extend anything else.

public class EnumTest {

Day day;

public EnumTest(Day day) {

this.day = day;

}

public void tellItLikeItIs() {

switch (day) {

case MONDAY:

System.out.println("Mondays are bad.");

break;

case FRIDAY:

System.out.println("Fridays are better.");

break;

case SATURDAY: case SUNDAY:

System.out.println("Weekends are best.");

break;

default:

System.out.println("Midweek days are so-so.");

break;

}

}

public static void main(String[] args) {

EnumTest firstDay = new EnumTest(Day.MONDAY);

firstDay.tellItLikeItIs();

EnumTest thirdDay = new EnumTest(Day.WEDNESDAY);

thirdDay.tellItLikeItIs();

EnumTest fifthDay = new EnumTest(Day.FRIDAY);

fifthDay.tellItLikeItIs();

EnumTest sixthDay = new EnumTest(Day.SATURDAY);

sixthDay.tellItLikeItIs();

EnumTest seventhDay = new EnumTest(Day.SUNDAY);

seventhDay.tellItLikeItIs();

}

}

* **Annotations**
  + Annotation provides data about a program that’s not part of a program itself.
  + They have no direct effect on the operation of the code they annotate.
  + Annotations have a number of uses among them
    - Information for the compiler
      * Annotations can be used by the compiler to detect errors or suppress warnings.
    - Compile-time and deployment-time processing
      * Software tools can process annotation information to generate code, XML files and so forth.
    - Run time processing
      * Some annotations are used to be examined at run time.
  + Annotations can be applied to a program’s declarations of classes, fields, methods and other program elements.
  + The annotation appears first, often on its own line, and may include elements with named or unnamed values.

@Author(

name = "Benjamin Franklin",

date = "3/27/2003"

)

class MyClass() { }

or

@SuppressWarnings(value = "unchecked")

void myMethod() { }

* + If there is one element named “value” then the name may be omitted as in

@SuppressWarnings("unchecked")

void myMethod() { }

* + Also if an annotation has no elements, the parentheses may be omitted.

@Override

void mySuperMethod() { }

* **Documentation**
  + Many annotations replace what would otherwise have been comments in code.
  + The Syntax for define the annotation type.

@interface ClassPreamble {

String author();

String date();

int currentRevision() default 1;

String lastModified() default "N/A";

String lastModifiedBy() default "N/A";

// Note use of array

String[] reviewers();

}

* + Once the annotation type has been defined, you can use annotations of that type

@ClassPreamble (

author = "John Doe",

date = "3/17/2002",

currentRevision = 6,

lastModified = "4/12/2004",

lastModifiedBy = "Jane Doe",

// Note array notation

reviewers = {"Alice", "Bob", "Cindy"}

)

public class Generation3List extends Generation2List {

// class code goes here

}

* + Use @Documented annotation, information appears in javadoc generated documentation.
* **Annotations Used by the Compiler**
  + There are three annotation types they are predefined by language specification itself.
    - @Deprecated
    - @Override
    - @SuppressWarnings
  + @Deprecated
    - The @Deprecated annotation indicates that the marked element is deprecated and should no longer be used.
    - The compiler generates warning whenever a program uses this annotation.
  + @Override
    - It informs the compiler that the element is meant to override an element declared in a super class.
  + @SuppressWarnings
    - It tells the compiler to suppress specific warnings that it would otherwise generate.
    - Every compiler warnings belongs to two category
      * deprecation
      * unchecked

@SuppressWarnings({"unchecked", "deprecation"})

* **Annotation Processing**
  + The more advance use of annotation includes writing an annotation processor that can read java program and take actions based on its annotations.
  + To make annotation information available at run time, the annotation type itself must be annotated with @Retention(RetentionPolicy.RUNTIME)

import java.lang.annotation.\*;

@Retention(RetentionPolicy.RUNTIME)

@interface AnnotationForRuntime {

// Elements that give information

// for runtime processing