Java is a popular programming language, created in 1995.

It is owned by Oracle, and more than 3 billion devices run Java.

It is used for:

- Mobile applications
- Desktop applications
- Web applications
- Web servers and application servers
- Games
- Database connection

Why Use Java?

- Java works on different platforms (Windows, Mac, Linux, Raspberry Pi, etc.)
- It is one of the most popular programming languages in the world
- It is open-source and free
- It is secure, fast and powerful
- It has huge community support (tens of millions of developers)
- Java is an object oriented language which gives a clear structure to programs and allows code to be reused, lowering development costs

Main.java

```java
public class Main {

  public static void main(String[] args) {

    System.out.println("Hello World");

  }

}

System.out.print("Hello World! ");

System.out.println("This sentence will work!");
System.out.println(This sentence will produce an error);
```

Java Comments

```
// This is a comment

/* The code below will print the words Hello World

to the screen, and it is amazing */
```

Java Variables

- Variables are containers for storing data values.
- The general rules for naming variables are:
- Names can contain letters, digits, underscores, and dollar signs
- Names must begin with a letter
- Names should start with a lowercase letter, and cannot contain whitespace
- Names can also begin with $ and _ (but we will not use it in this tutorial)
- Names are case-sensitive ("myVar" and "myvar" are different variables)
- Reserved words (like Java keywords, such as int or boolean) cannot be used as names
- Data types are divided into two groups:
- Primitive data types - includes byte, short, int, long, float, double, boolean and char
- Non-primitive data types - such as String, Arrays and Classes (you will learn more about these in a later chapter)
- In Java, there are different types of variables, for example:

Syntax

```
type variableName = value;
```

```
int myNum = 5;
float myFloatNum = 5.99f;
char myLetter = 'D';
boolean myBool = true;
String myText = "Hello";
```

Final Variables

```
final int myNum = 15;
myNum = 20;  // will generate an error: cannot assign a value to a
final variable
```

Java Type Casting

Type casting is when you assign a value of one primitive data type to another type.

Widening Casting

Widening casting is done automatically when passing a smaller size type to a larger size type:

Example

```
public class Main {

  public static void main(String[] args) {
    int myInt = 9;
    double myDouble = myInt; // Automatic casting: int to double

    System.out.println(myInt);      // Outputs 9
    System.out.println(myDouble);   // Outputs 9.0
  }

}
```

Narrowing Casting

Narrowing casting must be done manually by placing the type in parentheses in front of the value:

Example

```
public class Main {
  public static void main(String[] args) {
    double myDouble = 9.78d;
    int myInt = (int) myDouble; // Manual casting: double to int

    System.out.println(myDouble);   // Outputs 9.78
    System.out.println(myInt);      // Outputs 9
  }

}
```

Java Strings

A `String` variable contains a collection of characters surrounded by double quotes:

String Length

```
String txt = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";

System.out.println("The length of the text string is: " +
text.length());

String txt = "Hello World";
System.out.println(txt.toUpperCase());   // Outputs "HELLO WORLD"

System.out.println(txt.toLowerCase());   // Outputs "hello world"

String txt = "Please locate where 'locate' occurs!";

System.out.println(txt.indexOf("locate")); // Outputs 7
```

String Concatenation

```
String firstName = "John";
String lastName = "Doe";

System.out.println(firstName + " " + lastName);

System.out.println(firstName.concat(lastName));
```

Java Math

Math.max(*x*,*y*)

The `Math.max(x,y)` method can be used to find the highest value of *x* and *y*:

Math.min(*x*,*y*)

The `Math.min(x,y)` method can be used to find the lowest value of *x* and *y*:

Math.sqrt(*x*)

The `Math.sqrt(x)` method returns the square root of *x*:

Math.abs(*x*)

The `Math.abs(x)` method returns the absolute (positive) value of *x*:

Random Numbers

`Math.random()` returns a random number between 0.0 (inclusive), and 1.0 (exclusive):

Example

```
int randomNum = (int)(Math.random() * 101);  // 0 to 100
```

Java If ... Else

```
int time = 20;
if (time < 18) {
  System.out.println("Good day.");
} else {
  System.out.println("Good evening.");
}
// Outputs "Good evening."
```

Ternary operator

```
variable = (condition) ? expressionTrue : expressionFalse;
```

```
int time = 20;
String result = (time < 18) ? "Good day." : "Good evening.";
System.out.println(result);
```

Java Switch

```
int day = 4;
switch (day) {
  case 6:
    System.out.println("Today is Saturday");
    break;
  case 7:
    System.out.println("Today is Sunday");
    break;
  default:
    System.out.println("Looking forward to the Weekend");
}
// Outputs "Looking forward to the Weekend"
```

Java While Loop

The `while` loop loops through a block of code as long as a specified condition is `true`:

```
int i = 0;
while (i < 5) {
  System.out.println(i);
  i++;
}
```

The Do/While Loop

The loop will always be executed at least once, even if the condition is false, because the code block is executed before the condition is tested:

```
int i = 0;
do {
  System.out.println(i);
  i++;
}
while (i < 5);
```

Java For Loop

When you know exactly how many times you want to loop through a block of code, use the `for` loop instead of a `while` loop:

```
for (int i = 0; i < 5; i++) {
  System.out.println(i);
}
```

For-Each Loop

There is also a "for-each" loop, which is used exclusively to loop through elements in an array:

```
String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};

for (String i : cars) {

  System.out.println(i);

}
```

Java Break

It was used to "jump out" of a `switch` statement.

The `break` statement can also be used to jump out of a loop.

Java Continue

The `continue` statement breaks one iteration (in the loop), if a specified condition occurs, and continues with the next iteration in the loop.

Java Arrays

Arrays are used to store multiple values in a single variable, instead of declaring separate variables for each value.

To declare an array, define the variable type with square brackets:

```
String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
int[] myNum = {10, 20, 30, 40};
String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
cars[0] = "Opel";
System.out.println(cars[0]);
// Now outputs Opel instead of Volvo
```

Array Length

To find out how many elements an array has, use the `length` property:

```
String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
System.out.println(cars.length);
// Outputs 4
```

Loop Through an Array

You can loop through the array elements with the for loop, and use the length property to specify how many times the loop should run.

The following example outputs all elements in the cars array:

```
String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
for (int i = 0; i < cars.length; i++) {
  System.out.println(cars[i]);
}
```

The following example outputs all elements in the cars array, using a "for-each" loop:

Example

```
String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};

for (String i : cars) {

  System.out.println(i);

}
```

Multidimensional Arrays

A multidimensional array is an array of arrays.

Multidimensional arrays are useful when you want to store data as a tabular form, like a table with rows and columns.

To create a two-dimensional array, add each array within its own set of curly braces:

```
int[][] myNumbers = { {1, 2, 3, 4}, {5, 6, 7} };
System.out.println(myNumbers[1][2]); // Outputs 7
```

Change Element Values

You can also change the value of an element:

Example

```
int[][] myNumbers = { {1, 2, 3, 4}, {5, 6, 7} };

myNumbers[1][2] = 9;

System.out.println(myNumbers[1][2]); // Outputs 9 instead of 7
```

Loop Through a Multidimensional Array

You can also use a for loop inside another for loop to get the elements of a two-dimensional array (we still have to point to the two indexes):

Example

```
int[][] myNumbers = { {1, 2, 3, 4}, {5, 6, 7} };

for (int i = 0; i < myNumbers.length; ++i) {

  for (int j = 0; j < myNumbers[i].length; ++j) {

    System.out.println(myNumbers[i][j]);

  }

}
```

Java Methods

A method is a block of code which only runs when it is called.

You can pass data, known as parameters, into a method.

Methods are used to perform certain actions, and they are also known as functions.

Why use methods?

To reuse code, define the code once, and use it many times.

Create a Method

A method must be declared within a class. It is defined with the name of the method, followed by parentheses (). Java provides some predefined methods, such as System.out.println(), but you can also create your own methods to perform certain actions:

Create a method inside Main:

```
public class Main {

  ublic void myMethod() {

    // code to be executed

  }

}
```

`myMethod()` is the name of the method

- `void` means that this method does not have a return value. You will learn more about return values later in this chapter

Inside `main`, call the `myMethod()` method:

```
public class Main {
  public void myMethod() {
    System.out.println("I just got executed!");
  }
  public static void main(String[] args) {
    myMethod();
  }
}
// Outputs "I just got executed!"
```

A method can also be called multiple times

Java Method Parameters

Parameters and Arguments

Information can be passed to methods as parameter. Parameters act as variables inside the method.

Parameters are specified after the method name, inside the parentheses. You can add as many parameters as you want, just separate them with a comma.

The following example has a method that takes a String called fname as parameter. When the method is called, we pass along a first name, which is used inside the method to print the full name:

```
public class Main {
 public void myMethod(String fname) {
    System.out.println(fname + " Refsnes");
  }
  public static void main(String[] args) {
    myMethod("Liam");
    myMethod("Jenny");
    myMethod("Anja");
  }
}
// Liam Refsnes
// Jenny Refsnes
// Anja Refsnes
```

Multiple Parameters

You can have as many parameters as you like:

Example

```
public class Main {

  public void myMethod(String fname, int age) {

    System.out.println(fname + " is " + age);

  }


  public static void main(String[] args) {

    myMethod("Liam", 5);

    myMethod("Jenny", 8);

    myMethod("Anja", 31);

  }

}


// Liam is 5

// Jenny is 8

// Anja is 31
```

Return Values

The `void` keyword, used in the examples above, indicates that the method should not return a value. If you want the method to return a value, you can use a primitive data type (such as `int`, `char`, etc.) instead of `void`, and use the `return` keyword inside the method:

Example

```
public class Main {

  public int myMethod(int x) {

    return 5 + x;

  }

  public static void main(String[] args) {

    System.out.println(myMethod(3));

  }

}

// Outputs 8 (5 + 3)
```

A Method with If...Else

It is common to use if...else statements inside methods:

Example

```
public class Main {
  // Create a checkAge() method with an integer variable called age
public void checkAge(int age) {
    // If age is less than 18, print "access denied"
    if (age < 18) {
      System.out.println("Access denied - You are not old enough!");
    // If age is greater than, or equal to, 18, print "access granted"
    } else {
      System.out.println("Access granted - You are old enough!");
}
}



  public static void main(String[] args) {
     checkAge(20); // Call the checkAge method and pass along an age of 20
  }
```

```
}

// Outputs "Access granted - You are old enough!"
```

Java Method Overloading

Method Overloading

With method overloading, multiple methods can have the same name with different parameters:

```
int myMethod(int x)

float myMethod(float x)

double myMethod(double x, double y)
```

Consider the following example, which has two methods that add numbers of different type:

Example

```
public int plusMethodInt(int x, int y) {

  return x + y;

}

public double plusMethodDouble(double x, double y) {
 return x + y;

}



public static void main(String[] args) {

  int myNum1 = plusMethodInt(8, 5);

  double myNum2 = plusMethodDouble(4.3, 6.26);

  System.out.println("int: " + myNum1);

  System.out.println("double: " + myNum2);

}
```

Instead of defining two methods that should do the same thing, it is better to overload one.

In the example below, we overload the `plusMethod` method to work for both `int` and `double`:

Example

```
Public int plusMethod(int x, int y) {

  return x + y;
}

Public double plusMethod(double x, double y) {

  return x + y;

}

public static void main(String[] args) {

  int myNum1 = plusMethod(8, 5);

  double myNum2 = plusMethod(4.3, 6.26);

  System.out.println("int: " + myNum1);

  System.out.println("double: " + myNum2);

}
```

Java - What is OOP?

OOP stands for Object-Oriented Programming.

object-oriented programming is about creating objects that contain both data and methods.

Object-oriented programming has several advantages

OOP is faster and easier to execute

- OOP provides a clear structure for the programs
- OOP helps to keep the Java code DRY "Don't Repeat Yourself", and makes the code easier to maintain, modify and debug
- OOP makes it possible to create full reusable applications with less code and shorter development time

Tip: The "Don't Repeat Yourself" (DRY) principle is about reducing the repetition of code. You should extract out the codes that are common for the application, and place them at a single place and reuse them instead of repeating it.

Java - What are Classes and Objects?

Classes and objects are the two main aspects of object-oriented programming.

Look at the following illustration to see the difference between class and objects:

Class	objects
Fruits	Apple
	Banana
	Mango

So, a class is a template for objects, and an object is an instance of a class.

When the individual objects are created, they inherit all the variables and methods from the class.

Java Classes/Objects

Java is an object-oriented programming language.

Everything in Java is associated with classes and objects, along with its attributes and methods. For example: in real life, a car is an object. The car has attributes, such as weight and color, and methods, such as drive and brake.

A Class is like an object constructor, or a "blueprint" for creating objects.

Create a Class

To create a class, use the keyword `class`:

Create a class named "Main" with a variable x:

```
public class Main {

  int x = 5;

}
```

Create an Object

In Java, an object is created from a class. We have already created the class named Main, so now we can use this to create objects.

To create an object of Main, specify the class name, followed by the object name, and use the keyword new:

Example

Create an object called "myObj" and print the value of x:

```
public class Main {

  int x = 5;



  public static void main(String[] args) {

    Main myObj = new Main();

    System.out.println(myObj.x);

  }

}
```

Multiple Objects

You can create multiple objects of one class:

Example

Create two objects of `Main`:

```
public class Main {

  int x = 5;

  public static void main(String[] args) {

    Main myObj1 = new Main();  // Object 1

    Main myObj2 = new Main();  // Object 2

    System.out.println(myObj1.x);

    System.out.println(myObj2.x);

  }

}
```

Using Multiple Classes

You can also create an object of a class and access it in another class. This is often used for better organization of classes (one class has all the attributes and methods, while the other class holds the `main()` method (code to be executed)).

Remember that the name of the java file should match the class name. In this example, we have created two files in the same directory/folder:

- Main.java

- Second.java

Main.java

```
public class Main {

  int x = 5;

}
```

Second.java

```
class Second {

  public static void main(String[] args) {

    Main myObj = new Main();

    System.out.println(myObj.x);

  }

}
```

If you don't want the ability to override existing values, declare the attribute as `final`:

Example

```
public class Main {
  final int x = 10;

  public static void main(String[] args) {
    Main myObj = new Main();
    myObj.x = 25; // will generate an error: cannot assign a value to a final variable
    System.out.println(myObj.x);
  }
}
```

Static vs. Public

In the example above, we created a `static` method, which means that it can be accessed without creating an object of the class, unlike `public`, which can only be accessed by objects:

```
public class Main {

  // Static method

  static void myStaticMethod() {

    System.out.println("Static methods can be called without creating objects");

  }



  // Public method

  public void myPublicMethod() {

     System.out.println("Public methods must be called by creating objects");

  }



  // Main method

  public static void main(String[] args) {

   myStaticMethod(); // Call the static method

```

```
    // myPublicMethod(); This would compile an error

    Main myObj = new Main(); // Create an object of Main

    myObj.myPublicMethod(); // Call the public method on the object

  }

}
```

Access Methods With an Object

Example

Create a Car object named `myCar`. Call the `fullThrottle()` and `speed()` methods on the `myCar` object, and run the program:

```
// Create a Main class

public class Main {

  // Create a fullThrottle() method

  public void fullThrottle() {
    System.out.println("The car is going as fast as it can!");
  }

  // Create a speed() method and add a parameter
  public void speed(int maxSpeed) {
    System.out.println("Max speed is: " + maxSpeed);
  }

  // Inside main, call the methods on the myCar object
  public static void main(String[] args) {
    Main myCar = new Main();   // Create a myCar object
    myCar.fullThrottle();      // Call the fullThrottle() method
    myCar.speed(200);          // Call the speed() method
  }
}
```

```
// The car is going as fast as it can!

// Max speed is: 200
```

Example explained

1) We created a custom Main class with the class keyword.

2) We created the fullThrottle() and speed() methods in the Main class.

3) The fullThrottle() method and the speed() method will print out some text, when they are called.

4) The speed() method accepts an int parameter called maxSpeed - we will use this in 8).

5) In order to use the Main class and its methods, we need to create an object of the Main Class.

6) Then, go to the main() method, which you know by now is a built-in Java method that runs your program (any code inside main is executed).

7) By using the new keyword we created an object with the name myCar.

8) Then, we call the fullThrottle() and speed() methods on the myCar object, and run the program using the name of the object (myCar), followed by a dot (.), followed by the name of the method (fullThrottle(); and speed(200);). Notice that we add an int parameter of 200 inside the speed() method.’

Java Constructors

A constructor in Java is a special method that is used to initialize objects. The constructor is called when an object of a class is created. It can be used to set initial values for object attributes:

Create a constructor:

```java
// Create a Main class

public class Main {

  int x;  // Create a class attribute

  // Create a class constructor for the Main class

  public Main() {

    x = 5;  // Set the initial value for the class attribute x

  }

  public static void main(String[] args) {

     Main myObj = new Main(); // Create an object of class Main (This will call the constructor)

    System.out.println(myObj.x); // Print the value of x

  }

}

// Outputs 5
```

Constructor Parameters

Constructors can also take parameters, which is used to initialize attributes.

The following example adds an int y parameter to the constructor. Inside the constructor we set x to y (x=y). When we call the constructor, we pass a parameter to the constructor (5), which will set the value of x to 5:

Example

```
public class Main {
  int x;


  public Main(int y) {
    x = y;
  }


  public static void main(String[] args) {
    Main myObj = new Main(5);
    System.out.println(myObj.x);
  }
}

// Outputs 5
```

Example

```
public class Main {
  int modelYear;
  String modelName;

  public Main(int year, String name) {
    modelYear = year;
    modelName = name;
  }
  public static void main(String[] args) {
    Main myCar = new Main(1969, "Mustang");
    System.out.println(myCar.modelYear + " " + myCar.modelName);
  }
}

// Outputs 1969 Mustang
```

Modifiers

Access Modifiers

For classes, you can use either `public` or default:

Modifier	Description	
public	The class is accessible by any other class	
default	The class is only accessible by classes in the same package. This is used when you don't specify a modifier.	

For attributes, methods and constructors, you can use the one of the following:

Modifier	Description	
`public`	The code is accessible for all classes	
`private`	The code is only accessible within the declared class	
`protected`	The code is accessible in the same package and subclasses.	

Non-Access Modifiers

For classes, you can use either `final` or `abstract`:

Modifier	Description	
`final`	The class cannot be inherited by other classes	
`abstract`	The class cannot be used to create objects (To access an abstract class, it must be inherited from another class.	

For attributes and methods, you can use the one of the following:

Modifier	Description
`final`	Attributes and methods cannot be overridden/modified
`static`	Attributes and methods belongs to the class, rather than an object

abstract	Can only be used in an abstract class, and can only be used on methods. The method does not have a body, for example abstract void run();.

Final

If you don't want the ability to override existing attribute values, declare attributes as final:

Static

A static method means that it can be accessed without creating an object of the class, unlike public:

Example

An example to demonstrate the differences between static and public methods:

```
public class Main {

  // Static method

  static void myStaticMethod() {

    System.out.println("Static methods can be called without creating objects");

  }
  // Public method

  public void myPublicMethod() {

      System.out.println("Public methods must be called by creating objects");

  }



  // Main method

  public static void main(String[ ] args) {

    myStaticMethod(); // Call the static method

    // myPublicMethod(); This would output an error



    Main myObj = new Main(); // Create an object of Main

    myObj.myPublicMethod(); // Call the public method

  }

}
```

Abstract

An abstract method belongs to an abstract class, and it does not have a body. The body is provided by the subclass:

Example

```java
// Code from filename: Main.java

// abstract class

abstract class Main {

  public String fname = "John";

  public int age = 24;

  public abstract void study(); // abstract method

}



// Subclass (inherit from Main)

class Student extends Main {
  public int graduationYear = 2018;

   public void study() {  // the body of the abstract method is provided here
     System.out.println("Studying all day long");
   }
}
// End code from filename: Main.java


// Code from filename: Second.java
class Second {
  public static void main(String[] args) {
     // create an object of the Student class (which inherits attributes and methods from Main)
     Student myObj = new Student();


     System.out.println("Name: " + myObj.fname);
```

```
    System.out.println("Age: " + myObj.age);

    System.out.println("Graduation Year: " + myObj.graduationYear);

    myObj.study(); // call abstract method

  }
}
```

Encapsulation

The meaning of Encapsulation, is to make sure that "sensitive" data is hidden from users. To achieve this, you must:

- declare class variables/attributes as `private`
- provide public get and set methods to access and update the value of a `private` variable

Why Encapsulation?

- Better control of class attributes and methods
- Class attributes can be made read-only (if you only use the `get` method), or write-only (if you only use the `set` method)
- Flexible: the programmer can change one part of the code without affecting other parts
- Increased security of data

Get and Set

You learned from the previous chapter that `private` variables can only be accessed within the same class (an outside class has no access to it). However, it is possible to access them if we provide public get and set methods.

The `get` method returns the variable value, and the `set` method sets the value.

Syntax for both is that they start with either `get` or `set`, followed by the name of the variable, with the first letter in upper case:

Example

```
public class Person {
  private String name; // private = restricted access

  // Getter
  public String getName() {
    return name;
  }

  // Setter
  public void setName(String newName) {
    this.name = newName;

  }

}
```

Example explained

The `get` method returns the value of the variable `name`.

The `set` method takes a parameter (`newName`) and assigns it to the `name` variable. The `this` keyword is used to refer to the current object.

However, as the `name` variable is declared as `private`, we cannot access it from outside this class:

Java Inheritance (Subclass and Superclass)

In Java, it is possible to inherit attributes and methods from one class to another. We group the "inheritance concept" into two categories:

- subclass (child) - the class that inherits from another class
- superclass (parent) - the class being inherited from

To inherit from a class, use the `extends` keyword.

In the example below, the `Car` class (subclass) inherits the attributes and methods from the `Vehicle` class (superclass):

Example

```
class Vehicle {
  protected String brand = "Ford";        // Vehicle attribute
  public void honk() {                    // Vehicle method
    System.out.println("Tuut, tuut!");
  }
}


class Car extends Vehicle {
  private String modelName = "Mustang";    // Car attribute
  public static void main(String[] args) {

    // Create a myCar object
    Car myCar = new Car();
    // Call the honk() method (from the Vehicle class) on the myCar object
    myCar.honk();

    // Display the value of the brand attribute (from the Vehicle class) and the value of the modelName from the Car class
    System.out.println(myCar.brand + " " + myCar.modelName);
  }
}
```

The final Keyword

If you don't want other classes to inherit from a class, use the `final` keyword:

If you try to access a `final` class, Java will generate an error:

```
final class Vehicle {
  ...
}
```

```
class Car extends Vehicle {

  ...

}
```

Java Polymorphism

Polymorphism means "many forms", and it occurs when we have many classes that are related to each other by inheritance.

Like we specified in the previous chapter; Inheritance lets us inherit attributes and methods from another class. Polymorphism uses those methods to perform different tasks. This allows us to perform a single action in different ways.

For example, think of a superclass called `Animal` that has a method called `animalSound()`. Subclasses of Animals could be Pigs, Cats, Dogs, Birds - And they also have their own implementation of an animal sound (the pig oinks, and the cat meows, etc.):

Example

```
class Animal {

  public void animalSound() {

    System.out.println("The animal makes a sound");

  }

}
```

```java
class Pig extends Animal {
  public void animalSound() {
    System.out.println("The pig says: wee wee");
  }
}

class Dog extends Animal {
  public void animalSound() {
    System.out.println("The dog says: bow wow");
  }
}
```

Now we can create `Pig` and `Dog` objects and call the `animalSound()` method on both of them:

Example

```
class Animal {
  public void animalSound() {
    System.out.println("The animal makes a sound");
  }
}

class Pig extends Animal {
  public void animalSound() {
    System.out.println("The pig says: wee wee");
  }
}

class Dog extends Animal {
  public void animalSound() {
    System.out.println("The dog says: bow wow");
  }
}

class Main {
  public static void main(String[] args) {
    Animal myAnimal = new Animal();  // Create a Animal object
    Animal myPig = new Pig();  // Create a Pig object
    Animal myDog = new Dog();  // Create a Dog object
    myAnimal.animalSound();
    myPig.animalSound();
    myDog.animalSound();
  }
}
```

Abstract Classes and Methods

Data abstraction is the process of hiding certain details and showing only essential information to the user.
Abstraction can be achieved with either abstract classes or interfaces (which you will learn more about in the next chapter).

The abstract keyword is a non-access modifier, used for classes and methods:

- Abstract class: is a restricted class that cannot be used to create objects (to access it, it must be inherited from another class).
- Abstract method: can only be used in an abstract class, and it does not have a body. The body is provided by the subclass (inherited from).

An abstract class can have both abstract and regular methods:

```
abstract class Animal {

  public abstract void animalSound();

  public void sleep() {

    System.out.println("Zzz");

  }

}
```

From the example above, it is not possible to create an object of the Animal class:

```
Animal myObj = new Animal(); // will generate an error
```

To access the abstract class, it must be inherited from another class. Let's convert the Animal class we used in the Polymorphism chapter to an abstract class:

Example

```
// Abstract class

abstract class Animal {

  // Abstract method (does not have a body)

  public abstract void animalSound();

  // Regular method

  public void sleep() {
    System.out.println("Zzz");
  }
}

// Subclass (inherit from Animal)
class Pig extends Animal {
  public void animalSound() {
    // The body of animalSound() is provided here
    System.out.println("The pig says: wee wee");
  }
}

class Main {
  public static void main(String[] args) {
    Pig myPig = new Pig(); // Create a Pig object
    myPig.animalSound();

    myPig.sleep();

  }

}
```

Why And When To Use Abstract Classes and Methods?

To achieve security - hide certain details and only show the important details of an object.

Note: Abstraction can also be achieved with Interfaces, which you will learn more about in the next chapter.

Interfaces

Another way to achieve abstraction in Java, is with interfaces.

An `interface` is a completely "abstract class" that is used to group related methods with empty bodies:

Example

```
// interface

interface Animal {

  public void animalSound(); // interface method (does not have a body)

  public void run(); // interface method (does not have a body)

}
```

To access the interface methods, the interface must be "implemented" (kinda like inherited) by another class with the `implements` keyword (instead of `extends`). The body of the interface method is provided by the "implement" class:

Example

```
// Interface

interface Animal {

  public void animalSound(); // interface method (does not have a body)

  public void sleep(); // interface method (does not have a body)

}



// Pig "implements" the Animal interface
```

```
class Pig implements Animal {
  public void animalSound() {
    // The body of animalSound() is provided here
    System.out.println("The pig says: wee wee");
  }
  public void sleep() {
    // The body of sleep() is provided here
    System.out.println("Zzz");
  }
}

class Main {
  public static void main(String[] args) {
    Pig myPig = new Pig();  // Create a Pig object
    myPig.animalSound();
    myPig.sleep();

  }

}
```

Notes on Interfaces:

- Like abstract classes, interfaces cannot be used to create objects (in the example above, it is not possible to create an "Animal" object in the MyMainClass)
- Interface methods do not have a body - the body is provided by the "implement" class
- On implementation of an interface, you must override all of its methods
- Interface methods are by default `abstract` and `public`
- Interface attributes are by default `public`, `static` and `final`
- An interface cannot contain a constructor (as it cannot be used to create objects)

Multiple Interfaces

To implement multiple interfaces, separate them with a comma:

Example

```
interface FirstInterface {

  public void myMethod(); // interface method
}

interface SecondInterface {
  public void myOtherMethod(); // interface method
}

class DemoClass implements FirstInterface, SecondInterface {
  public void myMethod() {
    System.out.println("Some text..");
  }
  public void myOtherMethod() {
    System.out.println("Some other text...");
  }
}
```

```
class Main {
  public static void main(String[] args) {
    DemoClass myObj = new DemoClass();
    myObj.myMethod();
    myObj.myOtherMethod();
  }
}
```

Enums

An enum is a special "class" that represents a group of constants (unchangeable variables, like final variables).

To create an enum, use the enum keyword (instead of class or interface), and separate the constants with a comma. Note that they should be in uppercase letters:

Example

```
enum Level {
  LOW,
  MEDIUM,
  HIGH
}
```

You can access `enum` constants with the dot syntax:

```
Level myVar = Level.MEDIUM;
```

Difference between Enums and Classes

An `enum` can, just like a `class`, have attributes and methods. The only difference is that enum constants are `public`, `static` and `final` (unchangeable - cannot be overridden).

An `enum` cannot be used to create objects, and it cannot extend other classes (but it can implement interfaces).

Why And When To Use Enums?

Use enums when you have values that you know aren't going to change, like month days, days, colors, deck of cards, etc.

Input Types

In the example above, we used the `nextLine()` method, which is used to read Strings. To read other types, look at the table below:

Method	**Description**
`nextBoolean()`	Reads a `boolean` value from the user
`nextByte()`	Reads a `byte` value from the user

`nextDouble()`	Reads a `double` value from the user
`nextFloat()`	Reads a `float` value from the user
`nextInt()`	Reads a `int` value from the user
`nextLine()`	Reads a `String` value from the user
`nextLong()`	Reads a `long` value from the user
`nextShort()`	Reads a `short` value from the user

In the example below, we use different methods to read data of various types:

Example

```
import java.util.Scanner;

class Main {
  public static void main(String[] args) {
    Scanner myObj = new Scanner(System.in);

    System.out.println("Enter name, age and salary:");
    // String input
    String name = myObj.nextLine();

    // Numerical input
    int age = myObj.nextInt();
    double salary = myObj.nextDouble();

    // Output input by user
    System.out.println("Name: " + name);
    System.out.println("Age: " + age);
    System.out.println("Salary: " + salary);
  }
}
```

Java Dates

Java does not have a built-in Date class, but we can import the `java.time` package to work with the date and time API. The package includes many date and time classes. For example:

Class	**Description**
`LocalDate`	Represents a date (year, month, day (yyyy-MM-dd))
`LocalTime`	Represents a time (hour, minute, second and nanoseconds (HH-mm-ss-ns))
`LocalDateTime`	Represents both a date and a time (yyyy-MM-dd-HH-mm-ss-ns)
`DateTimeFormatter`	Formatter for displaying and parsing date-time objects

Display Current Date

To display the current date, import the `java.time.LocalDate` class, and use its `now()` method:

Example

```
import java.time.LocalDate; // import the LocalDate class

public class Main {

  public static void main(String[] args) {
    LocalDate myObj = LocalDate.now(); // Create a date object

    System.out.println(myObj); // Display the current date

  }

}
```

The output will be:

[redacted]

Display Current Time

To display the current time (hour, minute, second, and nanoseconds), import the `java.time.LocalTime` class, and use its `now()` method:

Example

```
import java.time.LocalTime; // import the LocalTime class



public class Main {
  public static void main(String[] args) {
    LocalTime myObj = LocalTime.now();
    System.out.println(myObj);
  }
}
```

The output will be:

Display Current Date and Time

To display the current date and time, import the `java.time.LocalDateTime` class, and use its `now()` method:

Example

```
import java.time.LocalDateTime; // import the LocalDateTime class


public class Main {
  public static void main(String[] args) {
    LocalDateTime myObj = LocalDateTime.now();
    System.out.println(myObj);
  }
}
```

The output will be:

[redacted]

Formatting Date and Time

The "T" in the example above is used to separate the date from the time. You can use the `DateTimeFormatter` class with the `ofPattern()` method in the same package to format or parse date-time objects. The following example will remove both the "T" and nanoseconds from the date-time:

Example

```
import java.time.LocalDateTime; // Import the LocalDateTime class
import java.time.format.DateTimeFormatter; // Import the DateTimeFormatter class
```

```
public class Main {

  public static void main(String[] args) {

    LocalDateTime myDateObj = LocalDateTime.now();

    System.out.println("Before formatting: " + myDateObj);

    DateTimeFormatter myFormatObj = DateTimeFormatter.ofPattern("dd-MM-yyyy HH:mm:ss");


    String formattedDate = myDateObj.format(myFormatObj);

    System.out.println("After formatting: " + formattedDate);

  }

}
```

The output will be:

[redacted]

Java ArrayList

The `ArrayList` class is a resizable array, which can be found in the `java.util` package.

The difference between a built-in array and an `ArrayList` in Java, is that the size of an array cannot be modified (if you want to add or remove elements to/from an array, you have to create a new one). While elements can be added and removed from an `ArrayList` whenever you want. The syntax is also slightly different:

Example

Get your own Java Server

Create an `ArrayList` object called cars that will store strings:

```
import java.util.ArrayList; // import the ArrayList class

ArrayList<String> cars = new ArrayList<String>(); // Create an ArrayList object
```

Add Items

The `ArrayList` class has many useful methods. For example, to add elements to the `ArrayList`, use the `add()` method:

Example

```
import java.util.ArrayList;

public class Main {
  public static void main(String[] args) {
    ArrayList<String> cars = new ArrayList<String>();
    cars.add("Volvo");
    cars.add("BMW");
    cars.add("Ford");
    cars.add("Mazda");
    System.out.println(cars);
  }
}
```

Access an Item

To access an element in the `ArrayList`, use the `get()` method and refer to the index number:

Example

```
cars.get(0);
```

Remember: Array indexes start with 0: [0] is the first element. [1] is the second element, etc.

Change an Item

To modify an element, use the `set()` method and refer to the index number:

Example

```
cars.set(0, "Opel");
```

Remove an Item

To remove an element, use the `remove()` method and refer to the index number:

Example

```
cars.remove(0);
```

To remove all the elements in the `ArrayList`, use the `clear()` method:

Example

```
cars.clear();
```

ArrayList Size

To find out how many elements an ArrayList have, use the `size` method:

Example

```
cars.size();
```

Loop Through an ArrayList

Loop through the elements of an `ArrayList` with a `for` loop, and use the `size()` method to specify how many times the loop should run:

Example

```
public class Main {
  public static void main(String[] args) {
    ArrayList<String> cars = new ArrayList<String>();
    cars.add("Volvo");
    cars.add("BMW");
    cars.add("Ford");
    cars.add("Mazda");
    for (int i = 0; i < cars.size(); i++) {
      System.out.println(cars.get(i));
    }
  }
}
```

You can also loop through an `ArrayList` with the for-each loop:

Example

```java
public class Main {
  public static void main(String[] args) {
    ArrayList<String> cars = new ArrayList<String>();
    cars.add("Volvo");
    cars.add("BMW");
    cars.add("Ford");
    cars.add("Mazda");
    for (String i : cars) {
      System.out.println(i);
    }
  }
}
```

Sort an ArrayList

Another useful class in the `java.util` package is the `Collections` class, which include the `sort()` method for sorting lists alphabetically or numerically:

Example

Sort an ArrayList of Strings:

```
import java.util.ArrayList;
import java.util.Collections;  // Import the Collections class

public class Main {
  public static void main(String[] args) {
    ArrayList<String> cars = new ArrayList<String>();
    cars.add("Volvo");
    cars.add("BMW");
    cars.add("Ford");
    cars.add("Mazda");
    Collections.sort(cars);  // Sort cars
    for (String i : cars) {
      System.out.println(i);
    }
  }
}
```

Java LinkedList

In the previous chapter, you learned about the `ArrayList` class. The `LinkedList` class is almost identical to the `ArrayList`:

Example

Get your own Java Server

```
// Import the LinkedList class
import java.util.LinkedList;


public class Main {
  public static void main(String[] args) {
    LinkedList<String> cars = new LinkedList<String>();
    cars.add("Volvo");
    cars.add("BMW");
    cars.add("Ford");
    cars.add("Mazda");
    System.out.println(cars);
  }
}
```

Try it Yourself »

ArrayList vs. LinkedList

The `LinkedList` class is a collection which can contain many objects of the same type, just like the `ArrayList`.

The `LinkedList` class has all of the same methods as the `ArrayList` class because they both implement the `List` interface. This means that you can add items, change items, remove items and clear the list in the same way.

However, while the ArrayList class and the LinkedList class can be used in the same way, they are built very differently.

How the ArrayList works

The ArrayList class has a regular array inside it. When an element is added, it is placed into the array. If the array is not big enough, a new, larger array is created to replace the old one and the old one is removed.

How the LinkedList works

The LinkedList stores its items in "containers." The list has a link to the first container and each container has a link to the next container in the list. To add an element to the list, the element is placed into a new container and that container is linked to one of the other containers in the list.

When To Use

Use an ArrayList for storing and accessing data, and LinkedList to manipulate data.

LinkedList Methods

For many cases, the ArrayList is more efficient as it is common to need access to random items in the list, but the LinkedList provides several methods to do certain operations more efficiently:

Method	Description	Try it

addFirst()	Adds an item to the beginning of the list.	Try it »
addLast()	Add an item to the end of the list	Try it »
removeFirst()	Remove an item from the beginning of the list.	Try it »
removeLast()	Remove an item from the end of the list	Try it »
getFirst()	Get the item at the beginning of the list	Try it »
getLast()	Get the item at the end of the list	Try it »

Java HashMap

In the `ArrayList` chapter, you learned that Arrays store items as an ordered collection, and you have to access them with an index number (`int` type). A `HashMap` however, store items in "key/value" pairs, and you can access them by an index of another type (e.g. a `String`).

One object is used as a key (index) to another object (value). It can store different types: `String` keys and `Integer` values, or the same type, like: `String` keys and `String` values:

Example

Get your own Java Server

Create a HashMap object called capitalCities that will store String keys and String values:

```
import java.util.HashMap; // import the HashMap class



HashMap<String, String> capitalCities = new HashMap<String, String>();
```

Add Items

The HashMap class has many useful methods. For example, to add items to it, use the put() method:

Example

```
// Import the HashMap class

import java.util.HashMap;
```

```
public class Main {
  public static void main(String[] args) {
    // Create a HashMap object called capitalCities
    HashMap<String, String> capitalCities = new HashMap<String, String>();


    // Add keys and values (Country, City)
    capitalCities.put("England", "London");
    capitalCities.put("Germany", "Berlin");
    capitalCities.put("Norway", "Oslo");
    capitalCities.put("USA", "Washington DC");
    System.out.println(capitalCities);
  }
}
```

Access an Item

To access a value in the `HashMap`, use the `get()` method and refer to its key:

Example

```
capitalCities.get("England");
```

Remove an Item

To remove an item, use the `remove()` method and refer to the key:

Example

```
capitalCities.remove("England");
```

To remove all items, use the `clear()` method:

Example

```
capitalCities.clear();
```

HashMap Size

To find out how many items there are, use the `size()` method:

Example

```
capitalCities.size();
```

Loop Through a HashMap

Loop through the items of a `HashMap` with a for-each loop.

Note: Use the `keySet()` method if you only want the keys, and use the `values()` method if you only want the values:

Example

```
// Print keys
for (String i : capitalCities.keySet()) {
  System.out.println(i);
}
```

Example

```
// Print values
for (String i : capitalCities.values()) {
  System.out.println(i);
}
```

Example

```
// Print keys and values
for (String i : capitalCities.keySet()) {
  System.out.println("key: " + i + " value: " + capitalCities.get(i));
}
```

Other Types

Keys and values in a HashMap are actually objects. In the examples above, we used objects of type "String". Remember that a String in Java is an object (not a primitive type). To use other types, such as int, you must specify an equivalent wrapper class: `Integer`. For other primitive types, use: `Boolean` for boolean, `Character` for char, `Double` for double, etc:

Example

Create a HashMap object called people that will store String keys and Integer values:

```
// Import the HashMap class

import java.util.HashMap;



public class Main {

  public static void main(String[] args) {



    // Create a HashMap object called people

    HashMap<String, Integer> people = new HashMap<String, Integer>();





    // Add keys and values (Name, Age)

    people.put("John", 32);
    people.put("Steve", 30);
    people.put("Angie", 33);

    for (String i : people.keySet()) {
      System.out.println("key: " + i + " value: " + people.get(i));
    }
  }
}
```

Java HashSet

A HashSet is a collection of items where every item is unique, and it is found in the `java.util` package:

Example

Get your own Java Server

Create a `HashSet` object called cars that will store strings:

```
import java.util.HashSet; // Import the HashSet class
```

```
HashSet<String> cars = new HashSet<String>();
```

Add Items

The `HashSet` class has many useful methods. For example, to add items to it, use the `add()` method:

Example

```
// Import the HashSet class
import java.util.HashSet;

public class Main {
  public static void main(String[] args) {
    HashSet<String> cars = new HashSet<String>();
    cars.add("Volvo");
    cars.add("BMW");
    cars.add("Ford");
    cars.add("BMW");
    cars.add("Mazda");
    System.out.println(cars);
  }
}
```

Try it Yourself »

Note: In the example above, even though BMW is added twice it only appears once in the set because every item in a set has to be unique.

Check If an Item Exists

To check whether an item exists in a HashSet, use the `contains()` method:

Example

```
cars.contains("Mazda");
```

Try it Yourself »

Remove an Item

To remove an item, use the `remove()` method:

Example

```
cars.remove("Volvo");
```

Try it Yourself »

To remove all items, use the `clear()` method:

Example

```
cars.clear();
```

Try it Yourself »

HashSet Size

To find out how many items there are, use the `size` method:

Example

```
cars.size();
```

Try it Yourself »

Loop Through a HashSet

Loop through the items of an `HashSet` with a for-each loop:

Example

```
for (String i : cars) {
  System.out.println(i);

}
```

Try it Yourself »

Other Types

Items in an HashSet are actually objects. In the examples above, we created items (objects) of type "String". Remember that a String in Java is an object (not a primitive type). To use other types, such as int, you must specify an equivalent wrapper class: `Integer`. For other primitive types, use: `Boolean` for boolean, `Character` for char, `Double` for double, etc:

Example

Use a `HashSet` that stores `Integer` objects:

```
import java.util.HashSet;



public class Main {

  public static void main(String[] args) {
```

```
// Create a HashSet object called numbers
HashSet<Integer> numbers = new HashSet<Integer>();

// Add values to the set
numbers.add(4);
numbers.add(7);
numbers.add(8);

// Show which numbers between 1 and 10 are in the set
for(int i = 1; i <= 10; i++) {
  if(numbers.contains(i)) {
    System.out.println(i + " was found in the set.");
  } else {
    System.out.println(i + " was not found in the set.");
  }
```

```
    }
  }
}
```

Java Iterator

An `Iterator` is an object that can be used to loop through collections, like ArrayList and HashSet. It is called an "iterator" because "iterating" is the technical term for looping.

To use an Iterator, you must import it from the `java.util` package.

Getting an Iterator

The `iterator()` method can be used to get an `Iterator` for any collection:

Example

```
// Import the ArrayList class and the Iterator class
import java.util.ArrayList;
import java.util.Iterator;
```

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

    // Make a collection
    ArrayList<String> cars = new ArrayList<String>();
    cars.add("Volvo");
    cars.add("BMW");
    cars.add("Ford");
    cars.add("Mazda");

    // Get the iterator
    Iterator<String> it = cars.iterator();

    // Print the first item
    System.out.println(it.next());
  }
}
```

Removing Items from a Collection

Iterators are designed to easily change the collections that they loop through. The `remove()` method can remove items from a collection while looping.

Example

Use an iterator to remove numbers less than 10 from a collection:

```
import java.util.ArrayList;
import java.util.Iterator;

public class Main {
  public static void main(String[] args) {
    ArrayList<Integer> numbers = new ArrayList<Integer>();
    numbers.add(12);
    numbers.add(8);
    numbers.add(2);
    numbers.add(23);
    Iterator<Integer> it = numbers.iterator();
    while(it.hasNext()) {
      Integer i = it.next();
      if(i < 10) {
        it.remove();
      }
    }
    System.out.println(numbers);
  }
}
```

Creating Wrapper Objects

To create a wrapper object, use the wrapper class instead of the primitive type. To get the value, you can just print the object:

Example

```
public class Main {

  public static void main(String[] args) {

    Integer myInt = 5;

    Double myDouble = 5.99;

    Character myChar = 'A';

    System.out.println(myInt);

    System.out.println(myDouble);

    System.out.println(myChar);

  }

}
```

Since you're now working with objects, you can use certain methods to get information about the specific object.

For example, the following methods are used to get the value associated with the corresponding wrapper object: `intValue()`, `byteValue()`, `shortValue()`, longValue(), floatValue(), doubleValue(), charValue(), booleanValue().

This example will output the same result as the example above:

Example

```
public class Main {

  public static void main(String[] args) {

    Integer myInt = 5;

    Double myDouble = 5.99;

    Character myChar = 'A';

    System.out.println(myInt.intValue());

    System.out.println(myDouble.doubleValue());

    System.out.println(myChar.charValue());

  }

}
```

Another useful method is the toString() method, which is used to convert wrapper objects to strings.

In the following example, we convert an `Integer` to a `String`, and use the `length()` method of the `String` class to output the length of the "string":

Example

```
public class Main {

  public static void main(String[] args) {

    Integer myInt = 100;

    String myString = myInt.toString();

    System.out.println(myString.length());

  }

}
```

Java Exceptions

When executing Java code, different errors can occur: coding errors made by the programmer, errors due to wrong input, or other unforeseeable things.

When an error occurs, Java will normally stop and generate an error message. The technical term for this is: Java will throw an exception (throw an error).

Java try and catch

The try statement allows you to define a block of code to be tested for errors while it is being executed.

The catch statement allows you to define a block of code to be executed, if an error occurs in the try block.

The try and catch keywords come in pairs:

Syntax

```
try {
  //  Block of code to try
}
catch(Exception e) {
  //  Block of code to handle errors
}
```

Consider the following example:

This will generate an error, because myNumbers[10] does not exist.

```
public class Main {
  public static void main(String[ ] args) {
    int[] myNumbers = {1, 2, 3};
    System.out.println(myNumbers[10]); // error!
  }
}
```

The output will be something like this:

[redacted]

If an error occurs, we can use `try...catch` to catch the error and execute some code to handle it:

Example

```
public class Main {
  public static void main(String[ ] args) {
    try {
      int[] myNumbers = {1, 2, 3};

      System.out.println(myNumbers[10]);

    } catch (Exception e) {

      System.out.println("Something went wrong.");

    }

  }

}
```

The output will be:

[redacted]

Finally

The `finally` statement lets you execute code, after `try...catch`, regardless of the result:

Example

```
public class Main {

  public static void main(String[] args) {

    try {

      int[] myNumbers = {1, 2, 3};

      System.out.println(myNumbers[10]);

    } catch (Exception e) {

      System.out.println("Something went wrong.");

    } finally {

      System.out.println("The 'try catch' is finished.");
```

```
        }

    }

}
```

The output will be:

[redacted]

The throw keyword

The `throw` statement allows you to create a custom error.

The throw statement is used together with an exception type. There are many exception types available in Java: ArithmeticException, FileNotFoundException, ArrayIndexOutOfBoundsException, SecurityException, etc:

Example

Throw an exception if age is below 18 (print "Access denied"). If age is 18 or older, print "Access granted":

```
public class Main {

  static void checkAge(int age) {

    if (age < 18) {

        throw new ArithmeticException("Access denied - You must be at least 18 years old.");

    }

    else {

      System.out.println("Access granted - You are old enough!");

    }

  }
  public static void main(String[] args) {

    checkAge(15); // Set age to 15 (which is below 18...)

  }

}
```

The output will be:

If age was 20, you would not get an exception:

Example

```
checkAge(20);
```

The output will be:

[redacted]