

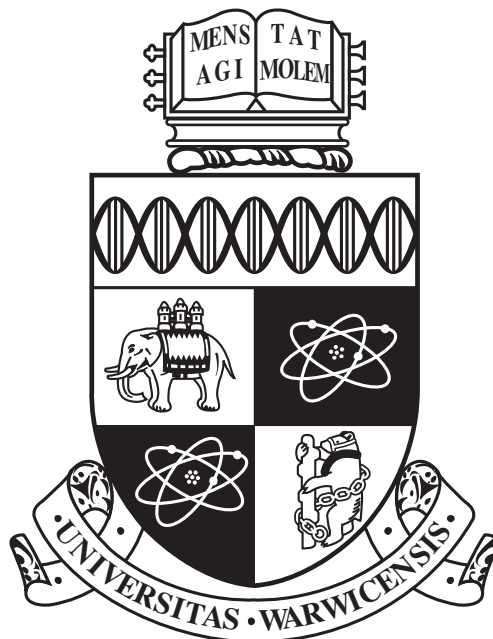
University of Warwick  
Department of Computer Science

---

# CS118

Programming for Computer Scientists

---



Cem Yilmaz

December 23, 2021

# Contents

<b>1</b>	<b>Definitions</b>	<b>3</b>
<b>2</b>	<b>Base <math>n</math> counting</b>	<b>5</b>
2.1	Binary	5
2.1.1	The connection between between Base 2 and Base 10	5
2.1.2	Base $n$	6
<b>3</b>	<b>Understanding Java</b>	<b>7</b>
3.1	Errors	8
3.1.1	Syntax Error	8
3.1.2	Runtime Error	8
3.1.3	Logic Error	8
3.2	Variables	8
3.3	Taking in input from console	9
3.4	Identifiers	10
3.5	Variables and Data Types	11
3.6	Math operators	12
3.7	Number Literals	13
3.8	Augmented Assignment Operators	13
3.9	Increment and Decrement operators	13
3.10	Numeric Type Conversions	13
3.11	Software Development Process	14
<b>4</b>	<b>Selections</b>	<b>17</b>
4.1	Boolean Data Types	17
4.2	if Statements	17
4.2.1	one-way if statements	17

4.2.2	Two-Way if statements . . . . .	18
4.2.3	Nested and Multi-way if and if-else statements . . . . .	18
4.2.4	Common and crucial errors . . . . .	19
4.2.5	Random Number Generator . . . . .	20
4.2.6	Logical Operators . . . . .	20
4.2.7	switch Statements . . . . .	22
4.2.8	Conditional Operators . . . . .	22
4.2.9	Operator Precedence and Associativity . . . . .	23
4.2.10	Debugging . . . . .	23
4.3	Mathematical Functions, Characters, and String . . . . .	24
4.3.1	Mathematical Functions . . . . .	24
4.3.2	Data Type and Operations . . . . .	24
4.4	Arrays and Loops . . . . .	27
4.4.1	Bounded and Unbounded Repetition . . . . .	27
4.4.2	Break and Continue . . . . .	28
4.4.3	Arrays . . . . .	28
4.5	Methods . . . . .	30
4.5.1	The Main method . . . . .	30
4.5.2	Other Methods . . . . .	30
4.5.3	Advanced Methods . . . . .	31
4.5.4	Scope . . . . .	31
4.5.5	Recursion . . . . .	32
4.6	Objects . . . . .	32
4.6.1	Classes . . . . .	32
4.6.2	Alternative Way of Thinking . . . . .	33
4.6.3	Constructor methods . . . . .	33
4.6.4	Creating Objects . . . . .	33
4.6.5	Point Objects . . . . .	34
4.7	Modifiers . . . . .	35
4.7.1	Access Modifiers . . . . .	35
4.7.2	Class variables . . . . .	35
4.7.3	Static . . . . .	35
4.7.4	Enumeration . . . . .	35

## CONTENTS

4.8	Inheritance and Polymorphism . . . . .	35
4.8.1	Inheritance . . . . .	35
4.8.2	Polymorphism . . . . .	37
4.9	Abstract Classes and Inheritance . . . . .	38
4.9.1	Abstract Classes . . . . .	38
4.9.2	Multi-inheritance . . . . .	39
4.10	Exceptions . . . . .	39
4.10.1	Checked Exceptions . . . . .	40
4.10.2	Unchecked Exceptions . . . . .	40
4.10.3	Chained Exceptions . . . . .	40
4.11	Generics . . . . .	40





# Chapter 1

## Definitions

**Definition 1.0.1.** Binary is the representation used by computers in a base 2 number system. It is represented by two numbers, 0 and 1.

**Definition 1.0.2.** Bits is the digit size for the binary system.

**Definition 1.0.3.** Byte is 8 bits, aka 8 digits in base 2.

**Definition 1.0.4.** Function in computer science is the command which tells the computer to execute something. It is an action or a verb.

**Definition 1.0.5.** Conditions in computer science gives the choice to the computer to do an action depending on what is present. e.g. if or else statement

**Definition 1.0.6.** Loop in computer science is the command that returns the computer to a specific line to repeat programs

**Definition 1.0.7.** A boolean question is a question whose answer is yes or no. They are usually paired with conditions.

**Definition 1.0.8.** A string is characters sandwiched between quotation marks.

**Definition 1.0.9.** A statement terminator is a semi-colon (;), and it ends statements. Every statement in Java must be terminated.

**Definition 1.0.10.** An interpreter reads one statement from the source code, translates it to the machine code or virtual machine code, then executes it.

**Definition 1.0.11.** A compiler fully translates all of the source code into machine code for execution.

**Definition 1.0.12.** Casting is an operation that converts the value of one data type into a value of another data type.





# Chapter 2

## Base $n$ counting

### 2.1 Binary

#### 2.1.1 The connection between Base 2 and Base 10

Counting is the same no matter base system one uses, just represented differently. In Base 10, when we reach from 0 to 9 and add +1, we add a tens digit to represent higher numbers. In Base 2, a similar system follows, however, instead for counting from 0 to 9, we do it with 0 and 1. For example, let us consider the numbers 123. It can also be represented as

$$\begin{array}{ccc} \underbrace{1} & \underbrace{2} & \underbrace{3} \\ 1 \times 10^2 & 2 \times 10^1 & 3 \times 10^0 \end{array} \quad (2.1)$$

$$\implies 1 \times 10^2 + 2 \times 10^1 + 3 \times 10^0 \quad (2.2)$$

Generalisation for a number of  $n$  digits

$$abc, def, \dots \zeta \quad (2.3)$$

$$\begin{array}{ccccccc} \underbrace{a} & \underbrace{b} & \underbrace{c} & , & \underbrace{d} & \underbrace{e} & \underbrace{f} & , \dots & \underbrace{\zeta} \\ a \times 10^n & b \times 10^{n-1} & c \times 10^{n-2} & & d \times 10^{n-3} & e \times 10^{n-4} & f \times 10^{n-5} & & \zeta \times 10^0 \end{array} \quad (2.4)$$

The coefficient  $b$  in a base 10 for a digit  $a10^{n+1} + b10^n$  can be any of the numbers  $a, b \in \{0, 1, 2, \dots, 9\}$ , as when it goes above 9, it will add an extra 1 to  $a$ , i.e. it will become

$$a10^{n+1} + (b + 10)10^n \iff (a + 1)10^{n+1} + b10^n \quad (2.5)$$

In base 2, a similar principle follows. Let us consider a few numbers

$$0 \implies 0 \text{ base 10} \quad (2.6)$$

$$1 \implies 1 \text{ base 10} \quad (2.7)$$

$$10 \implies 2 \text{ base 10} \quad (2.8)$$

$$11 \implies 3 \text{ base 10} \quad (2.9)$$

$$100 \implies 4 \text{ base 10} \quad (2.10)$$

$$101 \implies 5 \text{ base 10} \quad (2.11)$$

It is possible to see that for each new digit added, instead of growing with  $10^n$ , we grow as  $2^n$  for  $n$  binaries digits (bits). Furthermore, our coefficient for each digit must either be 0 or 1, meaning that the power  $2^n$  exists or it doesn't. Hence, we actually count for  $a, b, \zeta \in \{0, 1\}$

$$a2^n + b2^{n-1} + \dots + \zeta 2^0 \quad (2.12)$$

We can convert a number from base 2 to base 10 using the formula above. Similarly,

$$a2^{n+1} + (b+2)2^n \iff (a+1)2^{n+1} + b2^n \quad (2.13)$$

### 2.1.2 Base $n$

Hence we can generalise further, for base  $n$  counting we have the following rules with coefficients

$a, b, \zeta \in \{0, 1, \dots, n-1\}$  and an  $\alpha$  digit number in base  $n$ :

$$an^\alpha + bn^{\alpha-1} + \dots + \zeta n^0 \quad (2.14)$$

and

$$an^{\alpha+1} + (b+n)n^\alpha \iff (a+1)n^{\alpha+1} + bn^\alpha \quad (2.15)$$

## Chapter 3

# Understanding Java

Java is a programming language that can be run in many OS thanks to JVM (Java Virtual Machine). This allows for it to run "bytecode" within any operating system. When a .java program is launched, the java file is compiled into bytecode with a .class file extension. This bytecode is then interpreted by JVM and is finally translated into binary. The figures below represent the hierarchy of the execution of Java code.

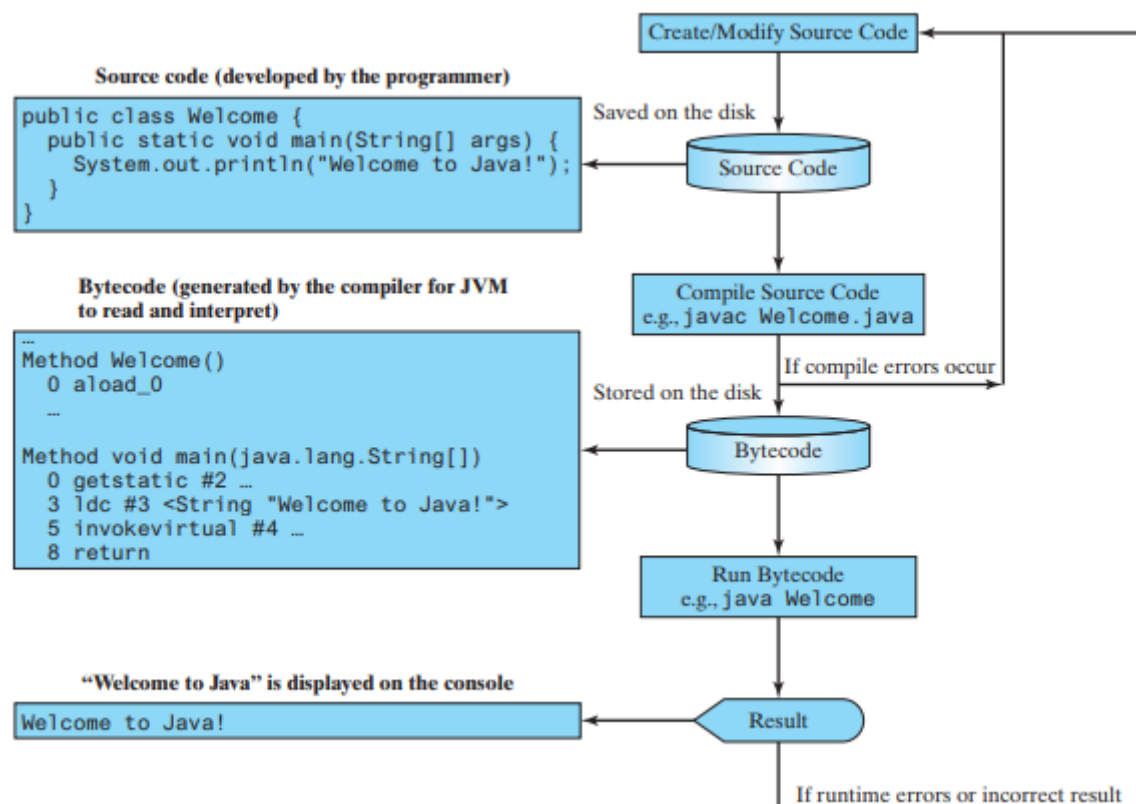
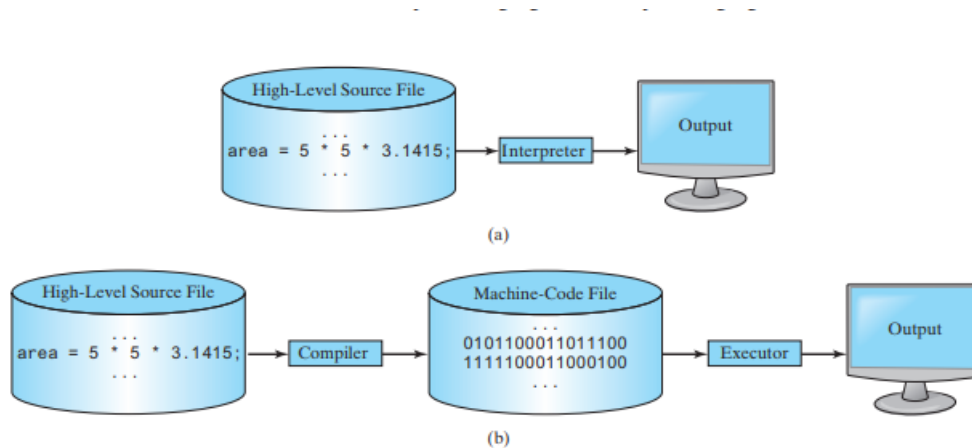


Figure 3.1: The execution cycle of a java program

Furthermore, there is a difference between a compiler and an interpreter for an IDE



**FIGURE 1.4** (a) An interpreter translates and executes a program one statement at a time. (b) A compiler translates the entire source program into a machine-language file for execution.

## 3.1 Errors

### 3.1.1 Syntax Error

Errors which are detectable by the compiler are called syntax or compile errors. These are caused when a code is constructed wrongly, e.g. mistyping, missing punctuation, brace etc.

### 3.1.2 Runtime Error

In runtime errors, a program terminates abnormally. This happens because the program expects a certain value but the user inserts a value that the program is not able to handle. E.g. if there is an expectation of a number but there is a string.

### 3.1.3 Logic Error

Occurs when a program does not output what it is intended to output. E.g. integer division using floor function to output the final answer

## 3.2 Variables

A variable represents a value stored in the computer's memory. Variables which are defined by their data type, on example of which is *double*.

**LISTING 2.1** ComputeArea.java

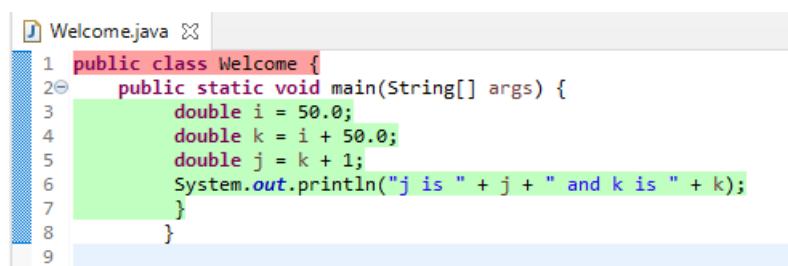
```

1 public class ComputeArea {
2     public static void main(String[] args) {
3         double radius; // Declare radius
4         double area; // Declare area
5
6         // Assign a radius
7         radius = 20; // radius is now 20

```

Figure 3.2: Use of double to define radius and area

This can be used to create shortcuts. Furthermore, in strings, the + sign can also be used as the string concatenation operator. In other words, it can also be used to combine strings. For example,



```

Welcome.java
1 public class Welcome {
2     public static void main(String[] args) {
3         double i = 50.0;
4         double k = i + 50.0;
5         double j = k + 1;
6         System.out.println("j is " + j + " and k is " + k);
7     }
8 }
9

```

Figure 3.3: Use of double and + as a concatenation operator

### 3.3 Taking in input from console

Suppose that now we want to change the definition of radius to another number. For an input, the way we have defined print is to be *System.out*. however, for input, our new command is *System.in*. To perform console input, you need to use the Scanner class to create an object to read input from System.in, as follows:

```
Scanner input = new Scanner(System.in);
```

The syntax new Scanner(System.in) creates an object of the Scanner type. The syntax Scanner input declares that input is a variable whose type is Scanner. The whole line Scanner input = new Scanner(System.in) creates a Scanner object and assigns its reference to the variable input. An object may invoke its methods. To invoke a method on an object is to ask the object to perform a task. You can invoke the nextDouble() method to read a double value as follows:

```
double radius = input.nextDouble();
```

This statement reads a number from the keyboard and assigns the number to radius. We could also have several of these statements together separated by a line, and we would be able input all of these in a single line. That is:

```

10     double number1 = input.nextDouble();           read a double
11     double number2 = input.nextDouble();
12     double number3 = input.nextDouble();
13
14     // Compute average
15     double average = (number1 + number2 + number3) / 3;
16
17     // Display results
18     System.out.println("The average of " + number1 + " " + number2
19         + " " + number3 + " is " + average);
20 }
21 }

```

```

Enter three numbers: 1 2 3 ↵
The average of 1.0 2.0 3.0 is 2.0

```



Figure 3.4: Three inputs demonstration

However, for this to work, we require the package Scanner which is obtained by writing

```
import java.util.Scanner; // Scanner is in the java.util package
```

The code above only imports Scanner from the java.util package. This is a specific imports. We could've also replaced .Scanner with .\* which would then turn it into a wildcard import, that is, it would import everything in the package. There is no performance difference between the two. For now, also simply accept that is how objects are declared to create inputs.

### 3.4 Identifiers

Identifiers are the names that identify the elements such as classes, methods, and variables in a program. For example, the program name "Welcome" is an identifier. radius and area are identifiers. Main and input are also identifiers. All identifiers must obey the following rules:

1. An identifier is a sequence of characters that consists of letters, digits, underscores (\_), and dollar signs (\$).
2. An identifier must start with a letter, an underscore (\_), or a dollar sign (\$). It cannot start with a digit.
3. An identifier cannot be a reserved word.
4. An identifier cannot be true, false, or null.
5. An identifier can be of any length.

## 3.5 Variables and Data Types

Variables are used to store certain information using an identifier. They can be declared by doing

```
datatype VariableName ;
```

The datatypes that exist are the following:

```
double //used to denote a real number to 15 decimal points
int //used to denote an integer from -2,147,483,648 to 2,147,483,647
boolean // used to denote true or false
byte // used to store integer numbers from -128 to 127
short // stores whole numbers from -32,768 to 32,767
long // Stores whole numbers from -9,223,372,036,854,775,808
// to 9,223,372,036,854,775,807
float // Stores a real number to 7 decimal points
```

Name	Range	Storage Size	
<b>byte</b>	$-2^7$ to $2^7 - 1$ (-128 to 127)	8-bit signed	byte type
<b>short</b>	$-2^{15}$ to $2^{15} - 1$ (-32768 to 32767)	16-bit signed	short type
<b>int</b>	$-2^{31}$ to $2^{31} - 1$ (-2147483648 to 2147483647)	32-bit signed	int type
<b>long</b>	$-2^{63}$ to $2^{63} - 1$ (i.e., -9223372036854775808 to 9223372036854775807)	64-bit signed	long type
<b>float</b>	Negative range: $-3.4028235E + 38$ to $-1.4E - 45$ Positive range: $1.4E - 45$ to $3.4028235E + 38$	32-bit IEEE 754	float type
<b>double</b>	Negative range: $-1.7976931348623157E + 308$ to $-4.9E - 324$ Positive range: $4.9E - 324$ to $1.7976931348623157E + 308$	64-bit IEEE 754	double type



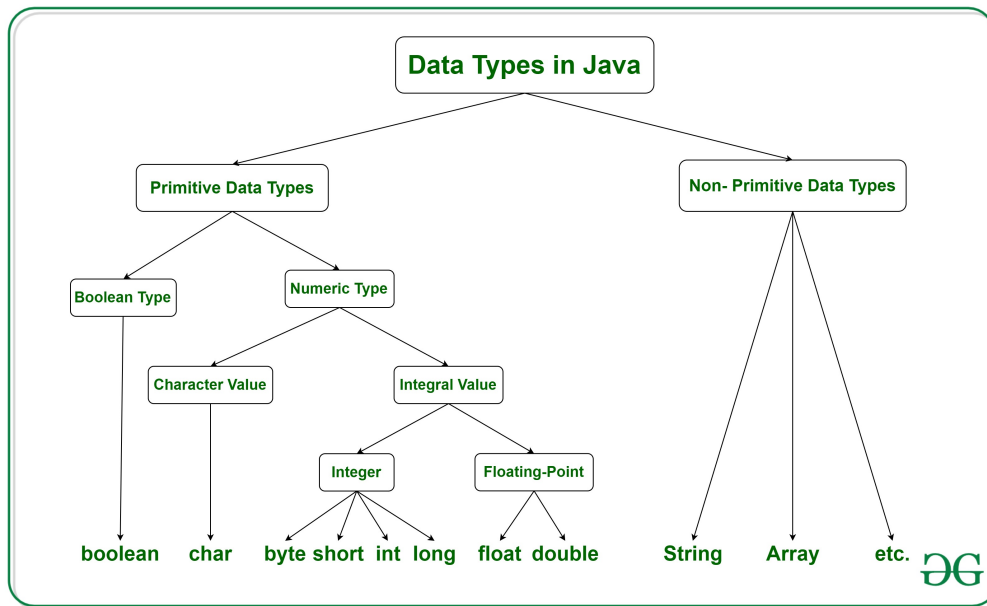


Figure 3.5: Data types

**Definition 3.5.1.** How Number range is determined

It is important to understand why, for example, in a byte it counts from  $-2^7$  to  $2^7 - 1$ . From definition, we know that a byte is 8 bits, that is, 00000000. The biggest number, we can get, is therefore 11111111, which translates to 255. However, we also need to define the negative numbers, and this was done by definition by setting the most left number 1. That is, 10000000 is in fact  $-128$ . The rest of the zeroes are computed as  $-128 + a$ , where  $a$  is determined by the 0000000. For example, 10000001 is  $-127$ . This means the highest positive integer we can achieve is 01111111, which is 127.

NOTE: it is important that for float values and that for long values, we must put L at the end of a number if it the number goes beyond integer's limits. Similarly, to declare a float value, it is recommended to put an F. The biggest difference between datatypes are their data that takes it to store. For example, double is 8 bytes whereas float is 4 bytes. Furthermore, if a datatype is a constant, then you can add the *final* text beforehand declaring the variable. This declares that it is a constant with unchanging data. By convention, constants are also written with capital letters. For example

```
final double PI = 3.14159;
```

### 3.6 Math operators

Other than adding, subtracting, multiplying and dividing, we also have the remainder and the exponent math operators in Java. They are declared as the following:

*% // This is the remainder operator. For example,  $5 \% 3$  would give out 2.*

`Math.pow(a, b)` // This is the power operator. This would show  $a^b$ .

## 3.7 Number Literals

## 3.8 Augmented Assignment Operators

The operators `+`, `-`, `*`, `/`, and `%` can be combined with the assignment operator to form augmented operators. For example, we can define  $x = x + 1$ , this way, the variable  $x$  increases by 1. These augmented assignments also have shortcuts, which are shown in the table below:

**TABLE 2.4** Augmented Assignment Operators

Operator	Name	Example	Equivalent
<code>+=</code>	Addition assignment	<code>i += 8</code>	<code>i = i + 8</code>
<code>-=</code>	Subtraction assignment	<code>i -= 8</code>	<code>i = i - 8</code>
<code>*=</code>	Multiplication assignment	<code>i *= 8</code>	<code>i = i * 8</code>
<code>/=</code>	Division assignment	<code>i /= 8</code>	<code>i = i / 8</code>
<code>%=</code>	Remainder assignment	<code>i %= 8</code>	<code>i = i % 8</code>

## 3.9 Increment and Decrement operators

Operator	Name	Description	Example (assume i = 1)
<code>++var</code>	preincrement	Increment <code>var</code> by 1, and use the new <code>var</code> value in the statement	<code>int j = ++i;</code> // j is 2, i is 2
<code>var++</code>	postincrement	Increment <code>var</code> by 1, but use the original <code>var</code> value in the statement	<code>int j = i++;</code> // j is 1, i is 2
<code>--var</code>	predecrement	Decrement <code>var</code> by 1, and use the new <code>var</code> value in the statement	<code>int j = --i;</code> // j is 0, i is 0
<code>var--</code>	postdecrement	Decrement <code>var</code> by 1, and use the original <code>var</code> value in the statement	<code>int j = i--;</code> // j is 1, i is 0

## 3.10 Numeric Type Conversions

In Java, widening cast type will be automatically done. However, narrowing cast will require manual work.

The type of data type for a number can be expressed by adding

`(Data type)Number`

That is, for example

```

public class Example {
    public static void main(String[] args) {
        System.out.println(1/2);
        // this will display zero because the output will originally
        // display 0.5 with floor function 0.
        System.out.println((double)1 / 2)
        // this will display 0.5 due to the double operator.
    }
}

```

### 3.11 Software Development Process

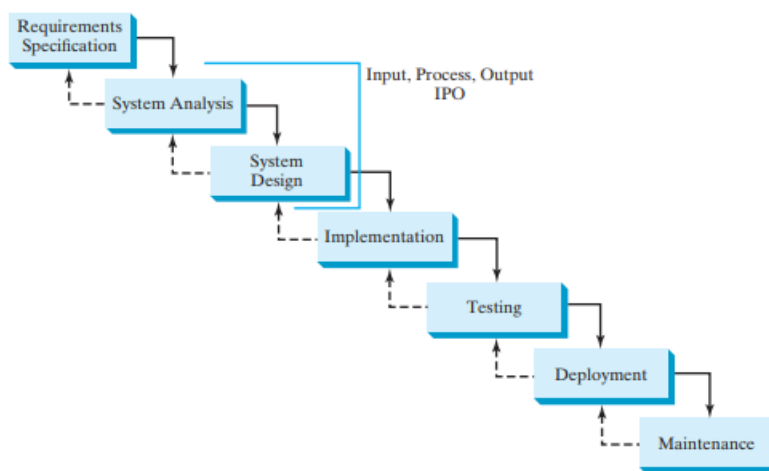


Figure 3.6: Software Development Process visualised

1. Requirements specification - seek and understand the problem the software will address. It will also require documentation of what the software system needs to do.
2. System analysis - identify the system's input and output. It is easier to find the output first, and then base off inputs depending on the output.
3. System Design - a process for obtaining the output from the input. This phase involves the use of many levels of abstraction to break down the problem into manageable components and design strategies for implementing each component.
4. Implementation - bringing the project to life by writing code as planned in system design. Separate

programs are written for each component then integrated together.

5. Testing - ensure that the code written is functional in different scenarios.
6. Deployment - how the software will be distributed to the public, e.g. installation or server online
7. Maintenance - updating and improving the product after launch. If the software is planned to be lasting, then it must be periodically upgraded.



# Chapter 4

## Selections

In this chapter we will look into choosing alternative courses depending on the input.

### 4.1 Boolean Data Types

Boolean data types allow us to compare the numerical input to desired limits. The table below shows an example of these data types. Assume that the radius of the circle is 10.

**TABLE 3.1** Relational Operators

Java Operator	Mathematics Symbol	Name	Example (radius is 5)	Result
<	<	Less than	<code>radius &lt; 0</code>	<code>false</code>
<=	≤	Less than or equal to	<code>radius &lt;= 0</code>	<code>false</code>
>	>	Greater than	<code>radius &gt; 0</code>	<code>true</code>
>=	≥	Greater than or equal to	<code>radius &gt;= 0</code>	<code>true</code>
==	=	Equal to	<code>radius == 0</code>	<code>false</code>
!=	≠	Not equal to	<code>radius != 0</code>	<code>true</code>

Figure 4.1: Table of Boolean Data Types

A boolean value is always true or false. We could, for example, assign a true or false value a variable.

```
boolean lightsOn = true;
```

### 4.2 if Statements

#### 4.2.1 one-way if statements

Java has several types of selection statements: one-way if statements, two-way if-else statements, nested if statements, multi-way if-else statements, switch statements, and conditional operators. A one-way if

statement executes an action if and only if the condition is true. The syntax for a one-way if statement is as follows:

```
if ( boolean-expression ) {  
    statement ( s );  
}
```

In other words,

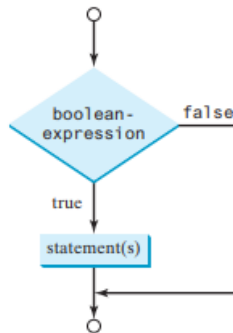


Figure 4.2: If expression flowchart

However, note that the brackets `{}` are not necessarily required if the statement is 1 line only.

### 4.2.2 Two-Way if statements

A two-way if statement considers another statement if the answer to the if statement is false.

```
if ( boolean-expression ) {  
    statement ( s )—for—the—true—case ;  
}  
else {  
    statement ( s )—for—the—false—case ;  
}
```

### 4.2.3 Nested and Multi-way if and if-else statements

A nested if statement is defined that if the statement for either the true case or the false case contains another if or if else statement. A good example of a nested and multi-way statement is the following:

```
if ( score >= 90 )  
    System.out.print ( "A" );  
else if ( score >= 80 )
```

```

    System.out.print("B");
else if (score >= 70)
    System.out.print("C");
else if (score >= 60)
    System.out.print("D");
else
    System.out.print("F");

```

However, it is important to see that this code works because it keeps introducing a new range for checking.

Consider the following code, which is blatantly wrong:

```

if (score >= 60)
    System.out.println("D");
else if (score >= 70)
    System.out.println("C");
else if (score >= 80)
    System.out.println("B");
else if (score >= 90)
    System.out.println("A");
else
    System.out.println("F");

```

#### 4.2.4 Common and crucial errors

##### Math float points

Unfortunately, because numbers in computer science definitions aren't precise, things can get messy.

Consider the following code:

```

double x = 1.0 - 0.1 - 0.1 - 0.1 - 0.1 - 0.1;
System.out.println(x == 0.5);

```

And this is in fact false. The final value of  $x$  is around  $x = 0.5000000000000001$ . One way to avoid this, is to introduce the variable  $\varepsilon$ , a small number that will allow us to create comparisons. That is, we can check if it is sufficiently close enough by computing  $|x - y| < \varepsilon$ . Normally, you set  $\varepsilon$  to  $1E-14$  for comparing two values of the double type, and to  $1E-7$  for comparing two values of the float type. For example:

```

final double EPSILON = 1E-14;

```



```

double x = 1.0 - 0.1 - 0.1 - 0.1 - 0.1 - 0.1;
if (Math.abs(x - 0.5) < EPSILON)
    System.out.println(x + " is approximately 0.5");

```

### Boolean operator assignment

Do not use the if statement to create assignment values for boolean when it can be simplified by denoting the boolean variable in the first place. Consider the following example:

```

if (number % 2 == 0)
    even = true;
else
    even = false;

```

It could instead be written better as

```

boolean even = number % 2 == 0;

```

## 4.2.5 Random Number Generator

A good approach to generate randoms is to utilise the random from Math class. It generates a random number  $d$  such that  $0 \leq d < 1$ .

## 4.2.6 Logical Operators

**TABLE 3.3** Boolean Operators

Operator	Name	Description
!	not	Logical negation
&&	and	Logical conjunction
	or	Logical disjunction
^	exclusive or	Logical exclusion

Figure 4.3: Logical operators

These apply operators work as the following:

Table 4.1: ! NOT operator

Value 1	Output
T	F
F	T

Table 4.2: AND operator

Value 1	Value 2	Output
T	T	T
T	F	F
F	T	F
F	F	F

Table 4.3: OR operator

Value 1	Value 2	Output
T	T	T
T	F	T
F	T	T
F	F	F

Table 4.4: XOR operator

Value 1	Value 2	Output
T	T	F
T	F	T
F	T	T
F	F	F

However, it is important to note that for the AND and the OR statement listed above, if the first is true, it will NOT check the second case.

#### Theorem 4.2.1. *De Morgan's Law*

De Morgan's Law states specific rules for boolean logic algebra. Before the math is invoked below, it is important to know that these are the notations:

NOT:  $\neg = !$

AND:  $\wedge = \&\&$

OR:  $\vee = ||$

XOR (Exclusive):  $\oplus = \wedge$

Logical Equivalence:  $\equiv$

Material Conditional:  $\rightarrow$

$$\neg(A \wedge B) \equiv \neg A \vee \neg B \quad (4.1)$$

$$\neg(A \vee B) \equiv \neg A \wedge \neg B \quad (4.2)$$

Other rules of Boolean algebra include but are not limited to the following:

$$\neg(A \oplus B) = A \equiv B$$

Material conditional is always true except when  $A$  is false and  $B$  is true.

#### 4.2.7 switch Statements

A cumulation of if statements can be avoided with the creation of a switch statement. A switch statements will run onto a specific line depending on the case that is in the input. An example can be seen below:

```
switch ( status ) {
    case 0: statement(s)—for—case —0;
        break ;
    case 1: statement(s)—for—case —1;
        break ;
    case 2: statement(s)—for—case —2;
        break ;
    case 3: statement(s)—for—case —3;
        break ;
    default : statement(s)—for—default ;
        System.exit(1);
}
```

The only thing to keep note here is that the case number  $n$  must be the same data type.

#### 4.2.8 Conditional Operators

Conditional operator is used to assign a variable depending on whether the boolean expression is true or not. Namely:

```
y = (x > 0) ? 1 : -1;
```

This is equivalent to

```
if (x > 0)
    y=1;
else
    y=-1;
```

### 4.2.9 Operator Precedence and Associativity

Table 4.5: Precedence and Associativity

0	<i>var</i> ++ and <i>var</i> --
1	+, − (Unary plus and minus), ++ <i>var</i> , − − <i>var</i>
2	! (Not)
3	*, / , %
4	+, − (Binary addition and subtraction)
5	<, <=, >, >=
6	==, !=
7	^
8	&&
9	
10	=, +=, − =, .* =, / =, % =

Where the smaller the number, the higher the priority. Another important thing to note is the associativity of these. It follows that for regular operators the associativity goes from left to right. That is, if the priority is the same. The output is calculated by going from left to right. However, assignment operators, e.g. +=, = etc. are right associative.

### 4.2.10 Debugging

Logical errors are called bugs. Debugging is the process of finding these logical errors. Methods of debugging, which are usually integrated to the idea, are the following:

1. Executing a single statement at a time - The debugger allows you to execute one statement at a time to see the effect of each statement
2. Tracing into or stepping over a method - You can skip over some methods. For example, you would always step over system-applied methods e.g. `System.out.println()`;
3. Setting breakpoints - Your program pauses at this breakpoint. You can set as many of these as you want.
4. Displaying variables - The debugger allows you to select variables and display their values.
5. Displaying call stacks - The debugger allows you to trace all of the method calls.
6. Modifying variables - Some debuggers enable you to modify the value of a variable when debugging for testing.

## 4.3 Mathematical Functions, Characters, and String

### 4.3.1 Mathematical Functions

All math functions in Java are executed by first writing *Math.*, followed by the function itself. There are many such functions, including, but not limited to:

```
Math.sin ();  
Math.cos ();  
Math.tan ();  
Math.toRadians ();  
Math.toDegrees ();  
Math.asin ();  
Math.acos ();  
Math.atan ();
```

However, note that this is all done in Radians by default. Other useful constants include *PI* and *E* which are  $\pi$  and  $e$  respectively. For exponents, the list is the following:

```
Math.exp ();  
log ();  
log10 ();  
pow(a, b);  
sqrt ();
```

Finally, we can also use math to round numbers

```
ceil ();  
floor ();  
rint ();  
round (); // Returns (int)Math.floor(x + 0.5) if x is float and returns  
// (long)Math.floor(x+0.5) if x is double.
```

### 4.3.2 Data Type and Operations

#### Instance method vs static method

There are types of methods are used in Java, that is the instance method and the static method. The instance method is when the variable is used at the beginning, that is

```
referenceVariable.methodName(arguments) // instance method
```

```
ClassName.methodName(arguments) // static method
```

In strings, for example, they are instance methods therefore

```
string s = "Welcome to java";
```

```
char t = s.charAt(3);
```

```
System.out.println(t); // prints out an "c". Note that it counts from 0.
```

The relevance of these methods is discussed later in this section.

## Char

Similarly how numeric values are defined, you can define character by the character data type *char*.

```
char letter = 'A';
```

```
char numChar = '4';
```

NOTE: String literals are to be enclosed in `""` whereas character literals for a single character must be in `"`.

You can, similarly, define characters using their unicode value. You can, similarly use the decrement and increment operators (`--a`, `++a`) to increase the number in unicode, e.g. from letter *a* to letter *b*.

You can furthermore use the escape sequence `\` before a character.

Another important thing to note that is `char` is also 16-bit however it is UNSIGNED. What this means that instead of representing numbers from  $-2^{15}$  to  $2^{15} - 1$ , it represents from 0 to  $2^{16}$ . `Char` can also be invoked for ASCII numbers. That is, it'll apply the floor function to find the relevant ASCII letter. When declaring unicode, it is important to begin the expression with `\u`. For example:

```
int a = (int)'A' // assigns the unicode value (65) to the integer a.
```

```
char b = '\uFF75'
```

Furthermore, there are more methods in the character class which can be useful:

```
isDigit() // Returns true if the char is a digit
```

```
isLetter() // Returns true if the char is a letter
```

```
isLetterOrDigit()
```

```
isLowerCase() // returns true if char is lowercase
```

```
isUpperCase() // returns true if char is uppercase
```

```
toLowerCase() // returns lowercase of the char
```

```
toUpperCase() // returns uppercase of the char
```

```
/* E.g. Character.toUpperCase(); */
```

## String

The string type is a type of data in Java, and it is a reference type. The following are different methods used to invoke in the String class:

```
length() // returns the number of characters in a string
charAt() // returns the char at the specified index in the string
concat() // returns a new string that concates this string with string input
toUpperCase() // returns a new string all letters in uppercase.
toLowerCase() // returns a new string all letters in lowercase.
trim() // Returns a new string with whitespace characters trimmed.
```

In order to insert a string using scanner, one would execute the following code:

```
Scanner sc = new Scanner(System.in);
string message = sc.nextLine();
```

You can also compare your strings using the following methods for your instance methods:

```
equals() // Returns true if the strings are equiv
equalsIgnoreCase() // returns if the strings are equiv ignoring case sensitivity
compareTo() // finds the unicode/hex number difference between first letters.
// NOTE: "First letter" is the letter with the first difference when compared.
compareToIgnoreCase() // same as above code but ignores upper/lowercases.
startsWith() // returns true if the string begins with the specified arg.
endsWith() // returns true if the string ends with the specified arg.
contains() // returns true if it is a substring.
```

You can also extract a substring from a string using the following command:

```
string s1 = "Welcome_to_Java"
string s2 = s1.substring(0,11); // substring(firstIndex ,lastIndex)
//Further note that firstIndex is inclusive whereas lastIndex is exclusive.
System.out.println(s2); // prints out "Welcome to"
```

Lastly, we can also locate specific characters in a string. All of the commands below share the location of the first occurrence of the string or letter, i.e. "index". It returns -1 if not matched.

```
indexOf(ch/s, fromIndex) // returns an index int of a char/string entered.
lastIndexOf (ch/s, fromIndex) // returns last index occurence of a char/string ent
```

A good example of code would be this:

```
Scanner sc = new Scanner(System.in);
string name = sc.nextLine();
int k = name.indexOf(' '); // finds the first space
string firstname = name.substring(0,k);
string lastname = name.substring(k+1);
```

## 4.4 Arrays and Loops

### 4.4.1 Bounded and Unbounded Repetition

#### Bounded Repetition

##### For Loop

There is a loop for a fixed amount of times, guaranteed to end.

If we know

1. Where we wish to begin
2. Where we wish to end
3. What iterative step to take each repetition

We use bounded repetition and when appropriate we use for loop that is like this:

```
for (initilisation; booleanExpression; iteration) {
// Loop body
}
```

#### Unbounded Repetition

##### While Loop

We are unsure how many times something will repeat.

```
while (booleanExpression) {
// Loop body
}
```



## Do While Loop

The syntax of this loop is similar to that of while.

```
do {
    // Loop body
} while (booleanExpression);
```

This will ensure that its executed at least once.

### 4.4.2 Break and Continue

#### Break

A break statement will cause the loop to stop at a specific boolean value. For example

```
for (int i = 0; i < 5; i++) {
    System.out.println("*");
    if (i &= 3) break;
}
```

#### Continue

The continue function will make it skip a specific iteration. Its syntax is similar to that of the break statement.

### 4.4.3 Arrays

Arrays are used if we want to store a lot of values that are all related. Arrays allow lists of monomorphic data (data of the same type) to be stored. Arrays are declared using the following method

```
[ type ][] variableName ;
// e.g.
int [] variableName ;
```

Arrays can be multidimensional. That is, you can have a 2D array would be declared by

```
[ type ][][] variableName ;
```

This makes it like a table. You can in fact go to  $n$  dimensional arrays. The squares can also be attached to the type or the variable name, that is, you can choose `[]` to be added after variableName.

```
int [] a, b, c [];
// is equivalent to
```

```
int a[], b[], c[][];
```

When an array is declared, it is given the null value. However, it is important that an array must always have the same amount of data as when you declare it, you tell the programme how much memory is to be allocated.

```
[type][] variableName = new [type][size];
```

```
// e.g. an array that stores 5 integers
```

```
int[] arrayOfInt = new int[5];
```

You can then refer to particular integers by putting the name and the number of column. Know that, however, arrays begin with the number 0, and therefore makes up to  $n - 1$ . Furthermore, unlike primitive variables, when we *new* an array, default values are set. More specifically,

1. Numerical values are set to 0
2. Boolean values are set to false
3. Arrays of objects are set to null

You can also have different default values using

```
int[] arrayOfData = {value1, value2, \ldots, valueN};
```

For this, the size of the array is not required as it is figured automatically. An example of a code of arrays that finds the average of `Math.random` is as follows:

```
public class arraySum {
    public static void main(String[] args) {
        double[] numbers = new double[1000];
        for (int i = 0; i < 1000; i++){
            numbers[i] = Math.random();
        }
        double total = 0.0;
        for (int i = 0; i < 1000; i++){
            total += numbers[i];
        }
        System.out.println(total/1000.0);
    }
}
```

We also have the command

`nameOfArray.length`

Which prints out  $n$ , that is, the 0 to  $n - 1$  count. Therefore, we could actually modify our top code to be  $i < \text{numbers.length}$  instead of  $i < 1000$ . When we set up multidimensional arrays, we can have them uniform. However, this is not a necessity and each array can have different number of elements. You can also initialise 2D braces as before, but that would require double braces

```
int [][] arrayOfInts = { { 1, 2, 3 }, {4, 5, 6}, { 7, 8, 9 } };
// And we can call the value by
arrayOfInts [0][0] = 1;
arrayOfInts [0][1] = 2;
// etc .
```

## 4.5 Methods

### 4.5.1 The Main method

The main method is a special method is a special method that is the entry point for a Java application. All methods have a signature that defines the name of the method, the return type, the access privilege and parameters. In particular,

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

Access privileges - public, private or protected

Return type - int, double, String etc.

Name

(Comma separated) List of function parameters

The main method has 1 parameter which is an array of Strings. This is used to take input from console.

Suppose our Java program is called *ABC.java*, then, the className as its main method *ABC*. To compile this we write *javacABC.java*, and after compilation will generate *ABC.class*. When we run this, if we pass some value it will go to args.

### 4.5.2 Other Methods

Methods are just a collection of statements that perform some action and return a result. You can tell when you're calling a method by the brackets afterwards. Even if it has no arguments, you put an empty set to differentiate it from a variable.

Unless declared *void*, methods must return a value with the *return* keyword. The returned value must match the declared type of the method.

### 4.5.3 Advanced Methods

Methods are defined by their name, return types and their parameter types. This means you can have methods with the same name but different return types, you can overload a function and Java will figure which one to run and will be optimal.

### 4.5.4 Scope

When we call a function, a new computation environment begins and all variables declared before we call a function do not have scope within the function. Sometimes we want a variable to be accessible within the entire program or class. For this we declare what we call *class variables*. E.g.

```
private static char c = '$';
```

Because of scope, it is also important to pass the parameter by value. If we have the code

```
public static void main( String [] args ) {  
    int n = 5;  
    n = A(n);  
    System.out.println(n);  
}
```

```
public static int A(int n) {  
    n = n + 1;  
    return n;  
}
```

Any changes to the variable only occur on the 'cloned' value and do not affect the original variable. If you want to change the original value, we would need to return the value in the function and assign it back to the original variable as the example above.

When changing parameters that are passed by value of reference, we need to be careful! One way to solve this issue is to clone the array. This problem occurs because they point to the same part in the memory. This issue also applies to Strings. Instead of the fact that it is copied, the address of the variable is used. One solution is the following

```
int [] n = { 5, 6, 8, 10, 23, 2, 65, 32 };
```

```

int [] m = new int[n.length];
for (int i = 0; i < n.length; i++) {
    m[i] = n[i];
}

```

Or we could special arraycopy method in System class.

```

System.arraycopy(Object src, int srcPos, Object dest, int destPos, int length)

```

Where src is the source array, srcPos is the starting position, dest is the destination array, destPos is the starting position in the destination data and length is the number of elements to be copied.

### 4.5.5 Recursion

Recursion involves a function calling itself. When a function is recursively defined, we need to consider when to stop the recursion (similar to how we must consider termination conditions for loops).

```

private static int factorial(int a) {
    if (a == 0) {
        return 1;
    }
    else if (a < 0) {
        return 0;
    }
    else {
        return factorial(a-1) * a;
    }
}

```

## 4.6 Objects

### 4.6.1 Classes

When working with objects, the first thing to do is to create a blueprint of the object. An object is a specific instance of this blueprint. The blueprint just describes how we create objects. In Java, we call these blueprints Classes. We define a new class with the class keyword. But there are rules:

1. Public classes must be in their own file

2. The convention is that class names begin with capital letter

### 4.6.2 Alternative Way of Thinking

You organise your code so that the data operations on that data are bundled together. Object types are just extended types. It is a collection of cooperating objects. Circle can be represented as data with radius and operations that are area, circumference etc.

### 4.6.3 Constructor methods

A constructor method is a special method that

1. has no return type
2. has the same name as the class

Its purpose is to set-up the object according to some rules. So, for a circle class

```
public class Circle {  
    // Object data  
    double radius;  
  
    public Circle(double radius) {  
        this.radius = radius;  
    }  
    ...  
}
```

WE can have methods variables with the same name as class variables, however, if we want to refer to the class variable, we must prefix it with *this*.

### 4.6.4 Creating Objects

Now our circle is complete we can start using circle objects. They require a small amount of contiguous memory to store all their properties, and to give them memory we use the new keyword. The new keyword reserves the memory and sets it to default values in arrays. In objects, it reserves enough memory and also calls the constructor function we just wrote.

```
[ObjectType] variableName = new [ObjectType]([parameters]);
```

For example,

```

public class CircleTest {
    public static void main(String[] args) {
        Circle c= new Circle(0.5);
        Circle c2 = new Circle(1.0);

        System.out.println("The area of c is: " + c.area());
        System.out.println("The circumference of c2 is: " + c2.circumference());
    }
}

```

Using methods that were defined from our other Java file which is

```

public class Circle {
    double radius;

    public Circle(double radius) {
        this.radius = radius;
    }

    public double area() {
        return radius * radius * Math.PI;
    }

    public double circumference() {
        return (radius + radius) * Math.PI;
    }
}

```

#### 4.6.5 Point Objects

```

Point(); // constructs and initialises a point in the origin (0,0) of cospace.
Point(int x, int y); // same but on x,y
point(point p) // initialises a point in the same location as point p.

```

## 4.7 Modifiers

### 4.7.1 Access Modifiers

These allow us to restrict access to an object properties and behaviours. Public can be accessed outside the class. Private can only be accessed inside the class. The point is to restrict data and properties are fundamental and we do not necessarily want them to be accessed. AS such, constructors are mostly public.

### 4.7.2 Class variables

You can use encapsulation where you make data private but methods public.

### 4.7.3 Static

Sometimes we want a variable to be shared between instances of a class. If we prefix or method declarations with the static keyword, they belong to the whole class, and not to any single instance. They are properties and methods of a class and not of an instance e.g. Math.PI and Math.round. nextInt, for example, is not a static method because we have to create a new object every time we input to an integer.

### 4.7.4 Enumeration

A enumerated type takes in variables and sets them a value in range. This allows them to build methods etc.

## 4.8 Inheritance and Polymorphism

### 4.8.1 Inheritance

Inheritance is a set of properties you get from a parent. In Java, classes can inherit from a parent class. That is, we have

1. Base classes called "Superclass"
2. Derived classes called "Subclass"

The subclass inherits some features from its parent class and may have some additional ones of its own. It is one of the biggest factors of object oriented programming.

A subclass inherits all the properties of its superclass and only needs to define any new properties. For example,



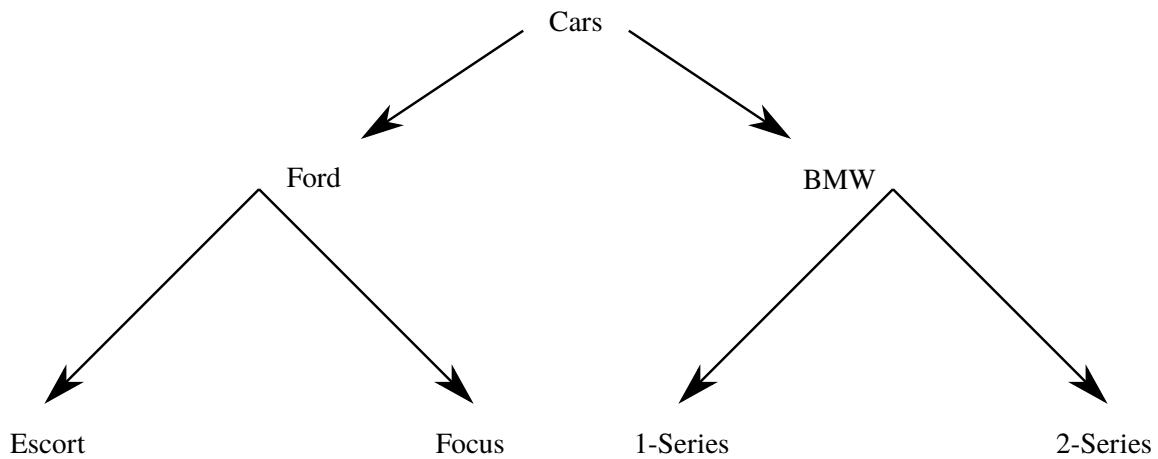


Figure 4.4: class

In this figure, we would have to define class properties in all different types of cars. Similarly, if things such as *Ford* has special properties to its cars, then we could just create that as a subclass of *Cars*, and create models of *Ford* class cars as subclasses of *Ford* itself. Inheritance is denoted using the *extends* keyword.

```

public class Lion extends Cat {
    ...
}
  
```

This command would extend the *Cat* class into *Lion* class. However, at the moment, if we have a mating defined at *Cat*, then if we have 2 lions mate, then we obtain a house cat. However, this a way to fix this, using *METHOD OVERRIDING*. We can write a method that is the same as the inherited method, and it will override from the inherited class. However, there is also another catch. The private variables in the *cat* class cannot be accessed by the *lion*. The solution to this problem, we call the constructor in the superclass.

```

super() // Denotes a superclass constructor
  
```

```

public Lion (String name, byte gender, Color[] colours) {
    super(name, gender, colours);
    // Other lion business
}
  
```

This would call the superclass variables for *name*, *gender*, *colours*. However, it does have its restrictions. The *super* keyword must be the first thing that is written in the subclass constructor. But what if we want to

augment the superclass's method with a few extra things?

We can call the overridden method with the `super` keyword as well. Let us say Lion sleeps twice as long as cats, however, we do not want to write the sleep method all over again.

```
public void sleep () {
    super.sleep ();
    super.sleep ();
}
```

### Protected

Everything in Java can see public variables and functions. Subclasses can see protected elements of superclasses. So properties such as age, name etc. so we do not require to create accessors methods.

## 4.8.2 Polymorphism

Some methods have a parameter such as

```
void someMethod (Object o) {
    ...
}
```

Because of how Java works, we can pass any `Object` into the method and it will work (because all objects are subclasses of `Object`). The question then becomes, how do we use the methods in our subclasses if Java thinks its just an `Object` object? The answer is, we must cast it.

### Definition 4.8.1. Polymorphism

Polymorphism describes its ability to process objects of different types through a single uniform interface.

In dynamic polymorphism, for a class such as `Cat` and two subclasses `Lion` and `Tiger`, both override the common method in the `Cat` class e.g. `purr()` method. For example, for the code

```
public class CatTest {
    public static void main (String [] args) {
        Cat tcat = new Tiger (...);
        Cat lcat = new Lion (...);

        tcat.purr ();
```

```

        lcat.purr();
    }
}

```

You will in fact get the correct printouts, Java is smart enough. But what if we try to use a method that doesn't exist in the Cat class? We then require to cast. In particular,

```

Cat lcat = new Lion (...);
((Lion) lcat).roar();

```

We can also ask Java if our object is of a particular type with an if statement. E.g., if we have a Cat, *c*

```

Cat c = new Tiger();
if (c instanceof Cat) // returns true
if (c instanceof Tiger) // returns true
if (c instanceof Object) // returns true
if (c instanceof Lion) // returns false

```

## 4.9 Abstract Classes and Inheritance

### 4.9.1 Abstract Classes

We know that each subclass becomes more specific than its parent class. Each superclass is therefore less specific. Eventually they become so general they become abstract. These are called abstract classes. Abstract classes are special classes that cannot be instantiated, but they allow us to capture common properties and behaviours. Abstract classes:

1. They cannot be initialised
2. They contain a mix of both abstract methods and concrete methods

To define one, we need to use the abstract keyword in the class declaration. In our cats example, we could define an abstract class of *felines*. Abstract methods are also a thing. These are methods that will be different for all felines (but all felines must provide). However, such methods and functions must be implemented into the classes that were extended from the abstract class.

### Interface

An interface is a situation in which abstract classes contain only abstract methods. The class is now simply describing the interface each implementing class should use. For such classes, we write interfaces. Interfaces

are defined like a class, but use the interface keyword. Interfaces can only contain methods and methods cannot be implemented. It is important to note that subclasses of interfaces do not extend the interface, they implement it. E.g.

```
public class TestImp1 implements Test {
    ...
}
```

### 4.9.2 Multi-inheritance

In Java, multi inheritance does not exist.

## 4.10 Exceptions

When running Java code, it is possible to run into an error that will crash the whole programme. However, we are able to catch these errors and ensure that the programme doesn't shut down. This is called catching errors, and is done using *try – catch* statements.

```
try {
    // Code that may generate as exception
} catch (TypeOfExceptionClass e) {
    // Code to handle particular exception
} catch (AnotherTypeOfExceptionClass e) {
    // Code to handle particular exception
} ... etc.
```

However, the ordering of catch blocks is important. If the first catch block is a superclass of any others, the catch blocks for the subclasses will never be used (since the exception type will match the superclass)

There is also the *try – catch – finally* statement. It is an extent to the try-catch block with a third block specifically for cleaning up. Regardless of whether any exceptions were generated, it is often the case that things need tidying up (such as streams need closing). We can encapsulate this in a finally block. The finally block will always be executed regardless of any return statements.

```
try {
    // Code that may generate as exception
}

catch (TypeOfExceptionClass e) {
```

```

        // Code to handle particular exception
    } finally {
        // Code that will always be executed at the end
    }

```

In Java, all errors and exceptions are subclasses of the Throwable class. The Throwable class has two direct subclasses, Error and Exception. An error is a subclass of throwable that indicates a serious problem that a reasonable application should not try to catch. The class Exception and its subclasses are a form of Throwable that indicates conditions that a reasonable application might want to catch.

#### 4.10.1 Checked Exceptions

Checked exceptions must be caught or re-thrown. Whilst we did deal with catching, the throws keyword is added to the method declaration and it lists all checked exceptions that may be thrown by the method.

#### 4.10.2 Unchecked Exceptions

Any exception that extends either Error or RuntimeException are unchecked. RuntimeException and its subclasses are unchecked exceptions. These do not need to be declared in a method or constructor's throws clause if they can be thrown by execution of the method or constructor and propagate outside the method or constructor boundary.

Not to be confused with the throws keyword, throw is used to throw an exception. The throw keyword requires that it is followed by an object that is an instance of the Throwable class.

#### 4.10.3 Chained Exceptions

Chained exceptions are used to pin down the cause of the issue even more. For example, a code that catches the error that requires an input of a file can catch the exception that the inserted file was not read by the programme because it is caused by an IO issue or that it does not exist. This can be done using `getCause()` method.

### 4.11 Generics

Generics in Java allows programmers to write "generic" code that enforces some stronger type checks at compile time. Such type checks are usually written with angled brackets. You can also define such generic classes.

A generic class definition looks exactly the same, however, it is a list of "type placeholders" is placed the class name e.g.

```
public class MyClass<A,B,C,D> {
    ...
}
```

The naming convention for these type parameters is a single upper-case letter. For example.

```
public class Box<T> {
    private T item;

    public Box(T o) {
        item = o;
    }
    public void set (T o) {
        ite = o;
    }
    public T get() {
        return item;
    }
}
```

```
public class BoxTest {
    public static void main( String [] args) {
        Box<String> stringBox = new Box<String >();
        ...
    }
}
```

From here, we can see that we have specified the type of primitive data that the box class will use for its inputs, to make error handling and controlling easier. It is also to note that every primitive type in Java also has an object equivalent e.g. Int to Integer etc.

You can also use the *extends* keyword to restrict type to subclasses of a particular type. For example,

```
public class Box<T extends Number> {
    private T item;
```

...  
}