Java

If not for C, this would be the most successful language in programming history

Hello World

```java
public class Intro {
   public static void main(String[] args) {
      System.out.println("Hello World!");
   }
}
```

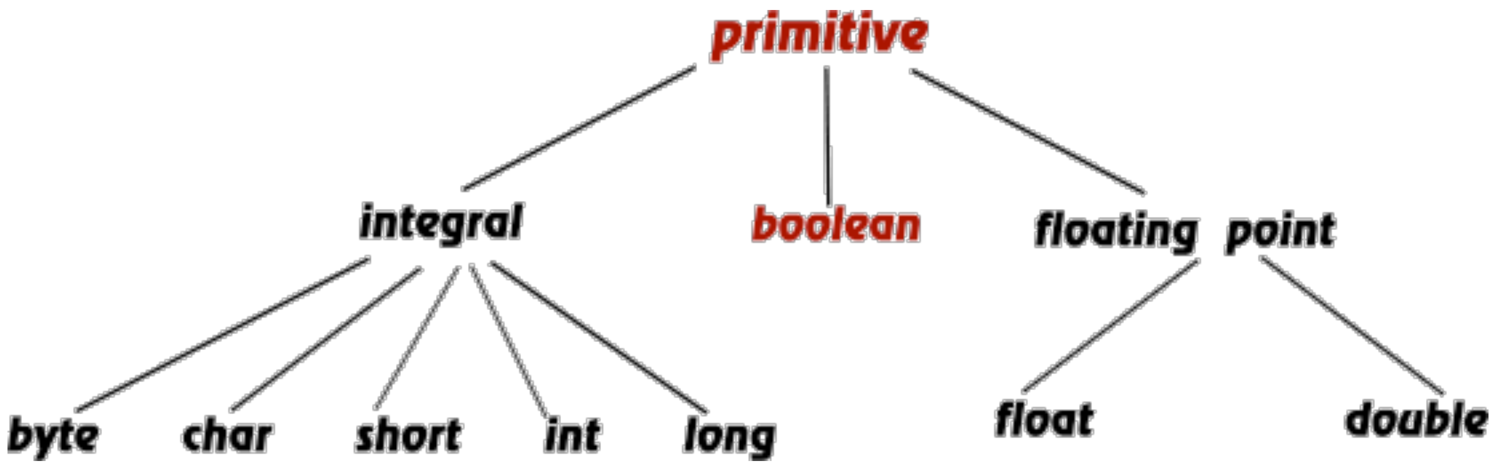
Features

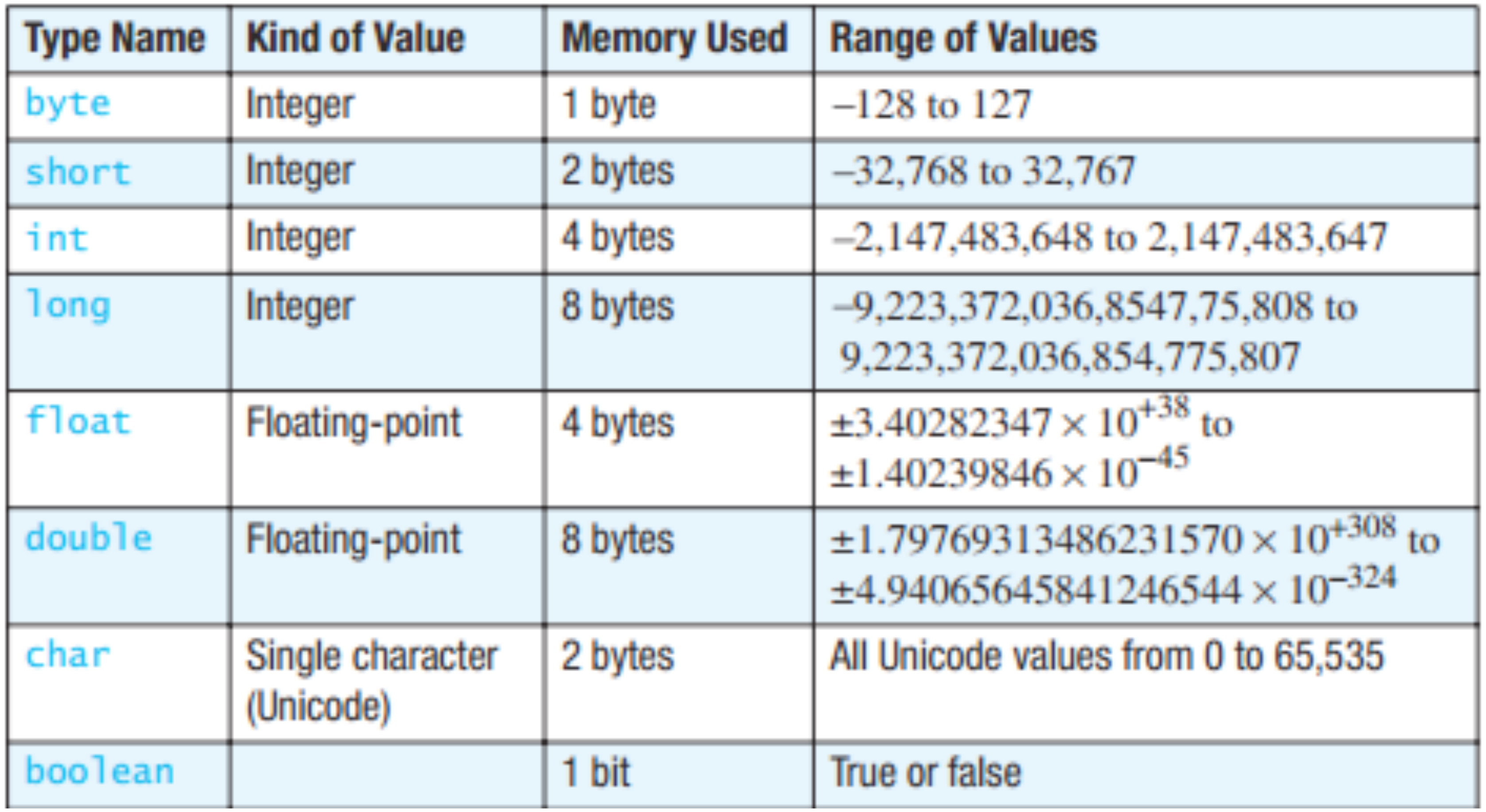
- Object Oriented: In Java, everything is an Object.
- Platform independent
- Secure: enables to develop virus-free, tamper-free systems. Authentication techniques are based on public-key encryption.
- Multithreaded: Java taps into operating system threads

$(a+b)^2 = a^2+b^2+2ab$

- Prove the above relationship for a=10 and b=20
- Use datatype int. Define it as int a=10;
- Evaluate and print LHS and RHS values
- Java is strongly typed.. so all types must be defined and used

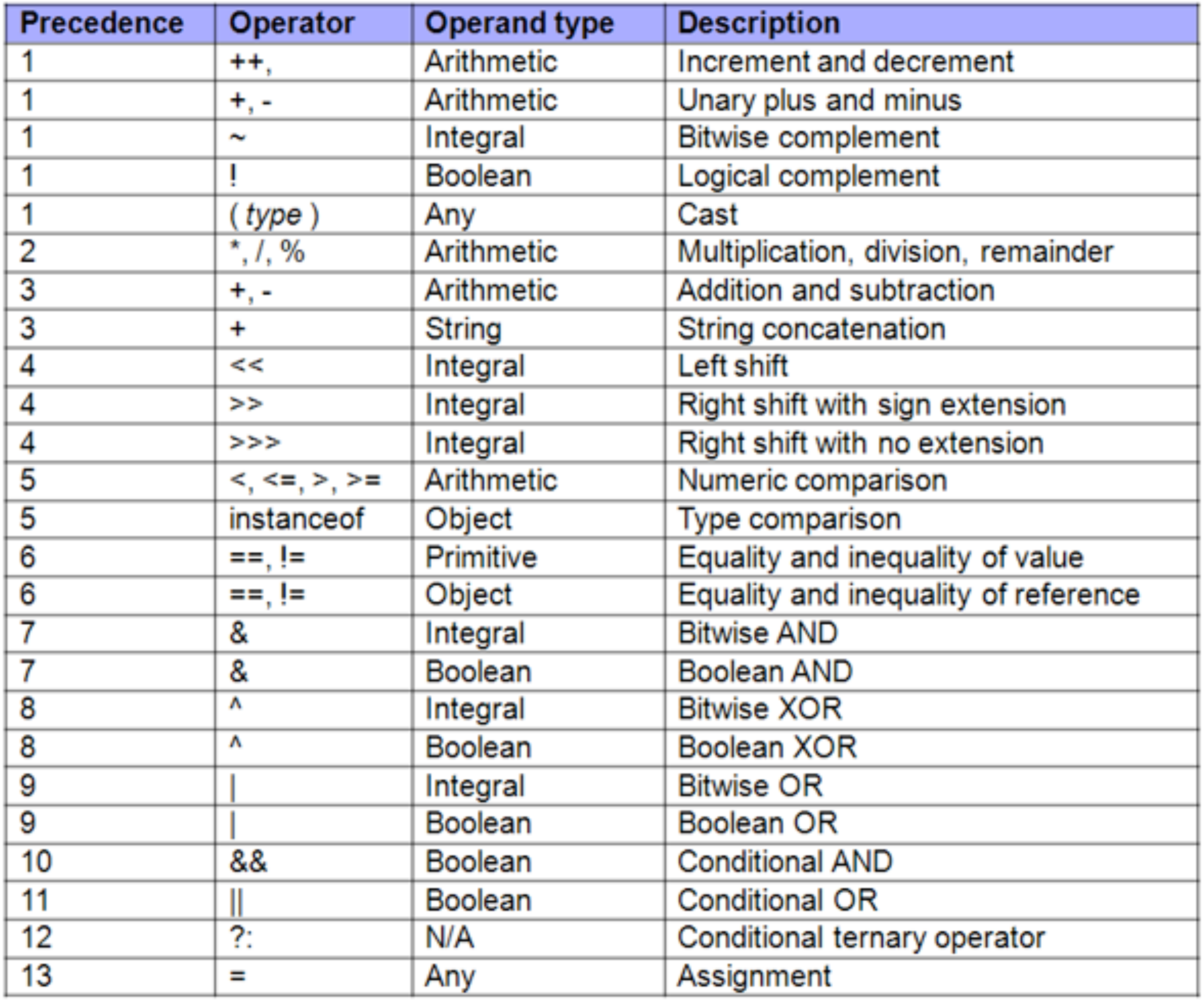
Datatypes





Type Name	Kind of Value	Memory Used	Range of Values
byte	Integer	1 byte	–128 to 127
short	Integer	2 bytes	–32,768 to 32,767
int	Integer	4 bytes	–2,147,483,648 to 2,147,483,647
long	Integer	8 bytes	–9,223,372,036,8547,75,808 to 9,223,372,036,854,775,807
float	Floating-point	4 bytes	±3.40282347 × 10^{+38} to ±1.40239846 × 10^{-45}
double	Floating-point	8 bytes	±1.79769313486231570 × 10^{+308} to ±4.94065645841246544 × 10^{-324}
char	Single character (Unicode)	2 bytes	All Unicode values from 0 to 65,535
boolean		1 bit	True or false

Operators



Precedence	Operator	Operand type	Description
1	++,	Arithmetic	Increment and decrement
1	+, -	Arithmetic	Unary plus and minus
1	~	Integral	Bitwise complement
1	!	Boolean	Logical complement
1	(*type*)	Any	Cast
2	*, /, %	Arithmetic	Multiplication, division, remainder
3	+, -	Arithmetic	Addition and subtraction
3	+	String	String concatenation
4	<<	Integral	Left shift
4	>>	Integral	Right shift with sign extension
4	>>>	Integral	Right shift with no extension
5	<, <=, >, >=	Arithmetic	Numeric comparison
5	instanceof	Object	Type comparison
6	==, !=	Primitive	Equality and inequality of value
6	==, !=	Object	Equality and inequality of reference
7	&	Integral	Bitwise AND
7	&	Boolean	Boolean AND
8	^	Integral	Bitwise XOR
8	^	Boolean	Boolean XOR
9	|	Integral	Bitwise OR
9	|	Boolean	Boolean OR
10	&&	Boolean	Conditional AND
11	||	Boolean	Conditional OR
12	?:	N/A	Conditional ternary operator
13	=	Any	Assignment

Flow Control Statements

- If condition

```
if(a>b){

}else if(a==b){

}else{

}
```

- for loop

```
for(int i =0 ; i< 100; i++){

}
for( ;canbreak();){

}
```

- While loop

```
while(a<b){

}
```

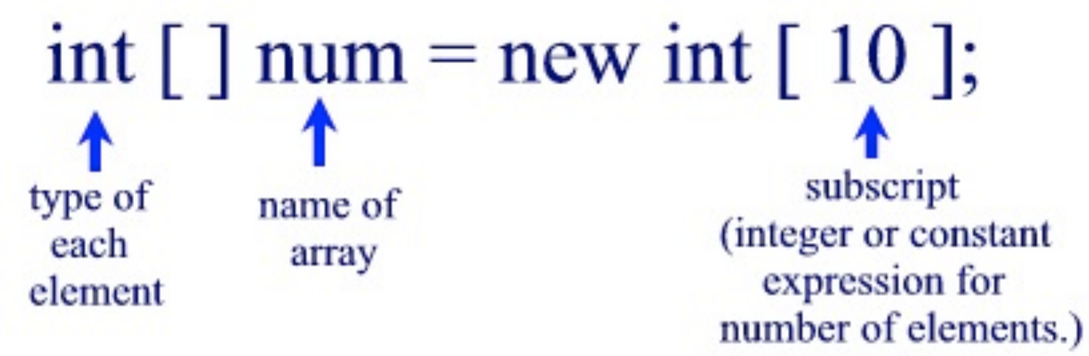
- do…while

```
do{

}while(a<b);
```

- switch case

```
switch(i){
    case 2:
        break;
    case 3:
        break;
    default:
        break;
}
```

Arrays

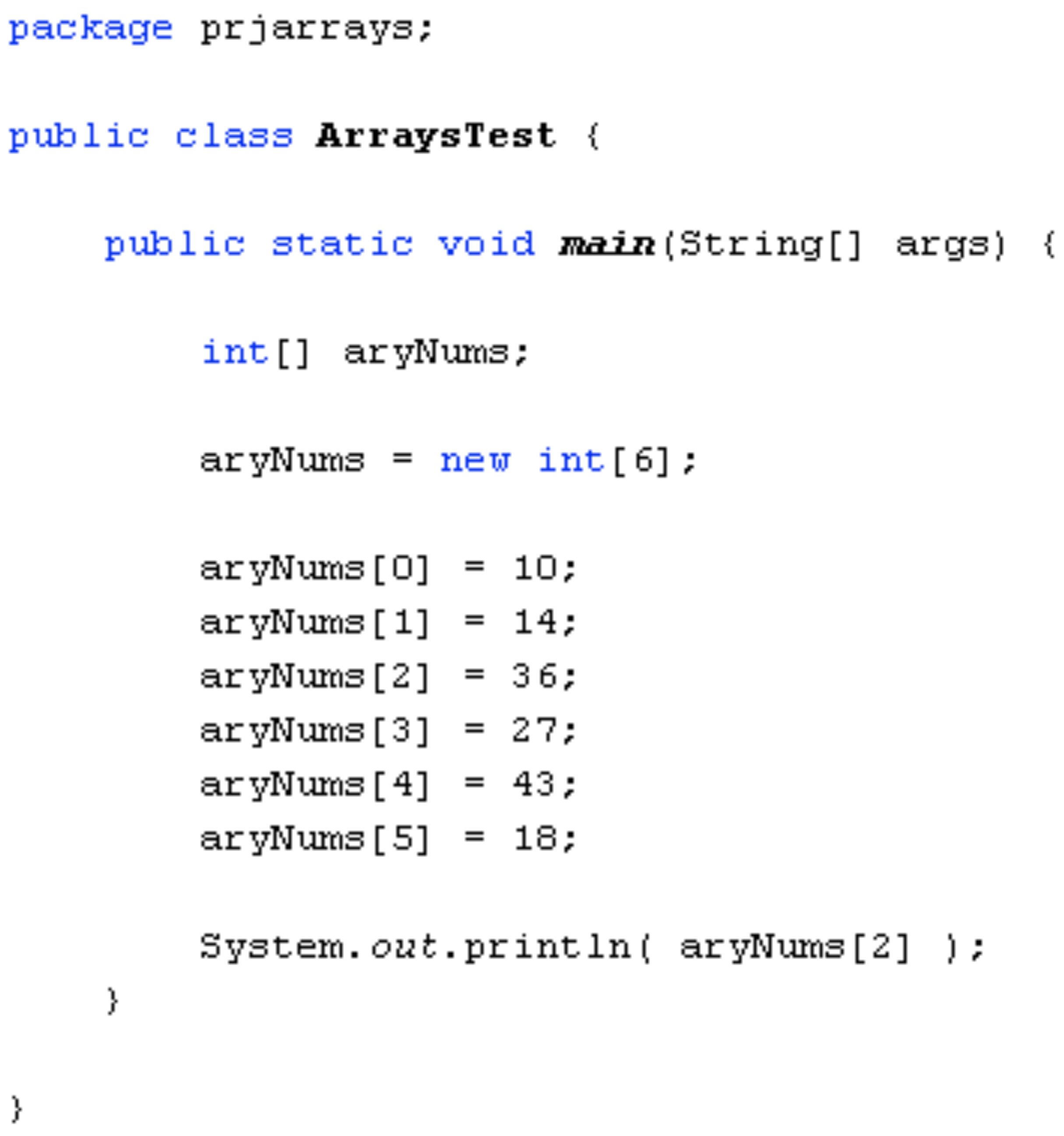


int [ ] num = new int [10];

type of each element

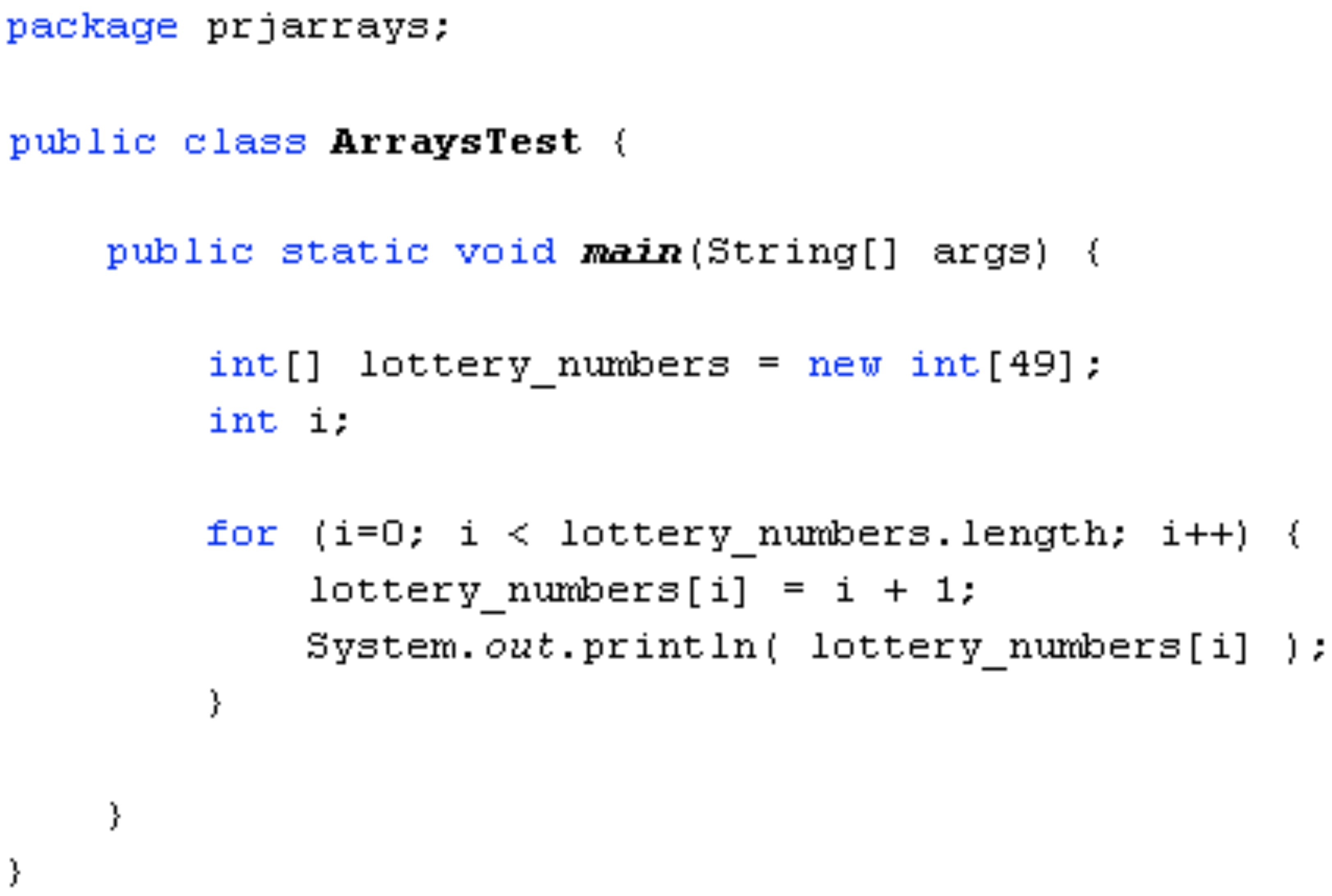
name of array

subscript (integer or constant expression for number of elements.)



```
package prjarrays;

public class ArraysTest {

    public static void main(String[] args) {

        int[] aryNums;

        aryNums = new int[6];

        aryNums[0] = 10;
        aryNums[1] = 14;
        aryNums[2] = 36;
        aryNums[3] = 27;
        aryNums[4] = 43;
        aryNums[5] = 18;

        System.out.println( aryNums[2] );
    }

}
```

Iterating Arrays



```
package prjarrays;

public class ArraysTest {

    public static void main(String[] args) {

        int[] lottery_numbers = new int[49];
        int i;

        for (i=0; i < lottery_numbers.length; i++) {
            lottery_numbers[i] = i + 1;
            System.out.println( lottery_numbers[i] );
        }

    }
}
```

For Loop Type 2

- Can be used on arrays or any other iteratable collection

```
int[] arr = new int[10];

for (int i : arr) {

}

ArrayList<String> names = new ArrayList<String>();

for (String name : names) {

}
```

Lets Try It

- Create an array of 10 ints and 10 Strings. Write a program to create a clone array thats a reverse of the main array.
    - Understand memory allocation and deallocation

Object Oriented Programming

- Real life interactions are with objects
- Objects are classified based on their characteristics
- Objects have properties and behaviour
- An encapsulated object is a stable object

Defining A Class, its properties and methods

```
class Vehicle{
   int engineSize;
   String color;

   public void start(){
      System.out.println("Starting the vehicle with engine size: 
"+this.engineSize);
   }
}
```

Constructors & Destructors

```
class Vehicle{
   int engineSize;
   String color;

   public Vehicle() {
      System.out.println("Building vehicle");
   }
   protected void finalize() throws Throwable {
      System.out.println("Cleaning up...");
   }
}
```

Parameters and Overloading

```
public Vehicle(int engineSize, String color){
    this.engineSize=engineSize;
    this.color=color;
}

public Vehicle() {
    this(1000,"Red");
    System.out.println("Building vehicle");
}

Vehicle v = new Vehicle();
Vehicle v2 = new Vehicle(2000,"Pink");
```

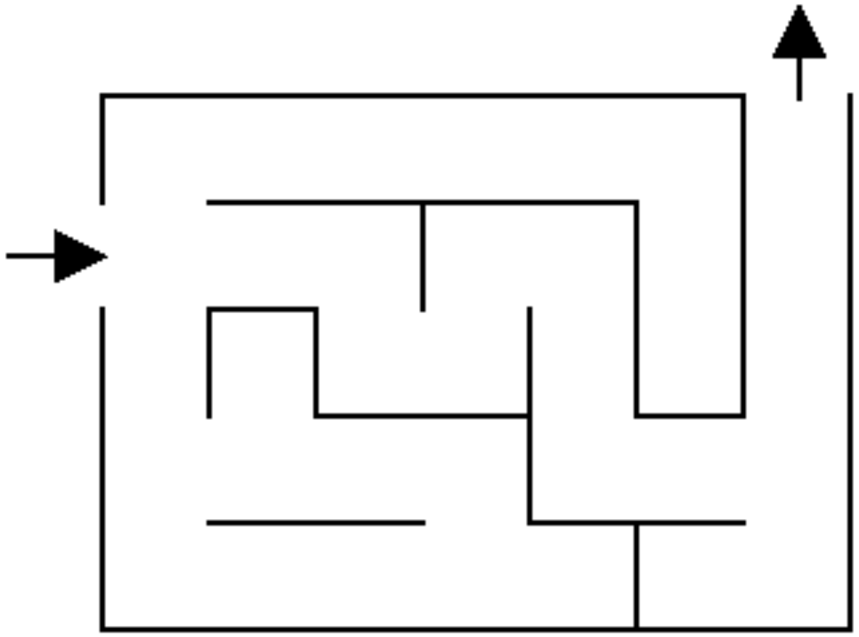
Lets Try It

- Create 100 Vehicles in a loop and put them in an array of vehicles
    - … and try variations on this to understand Java Memory Management

Try This

- Create a “Car” class with functions turnRight(), turnLeft(), moveStraight()

- Create a “Maize” class that represents the below maize. Add a method navigateMaize(Car c) that takes the car from entry to exit



Try This

- Create Class “Truck” with functions turnRight(), turnLeft(), moveStraight()
- Change the navigateMaize method to take a truck thru

Inheritance

- Generalisation is about abstracting out a class of behaviours/properties
- Specialisation is the reverse and is about getting specific
- Java does not support multiple inheritance - avoids the classic C++ diamond problem

Classification

Vehicle

Animal

```
class Car extends Vehicle{
   String brand;
}
```

Try This

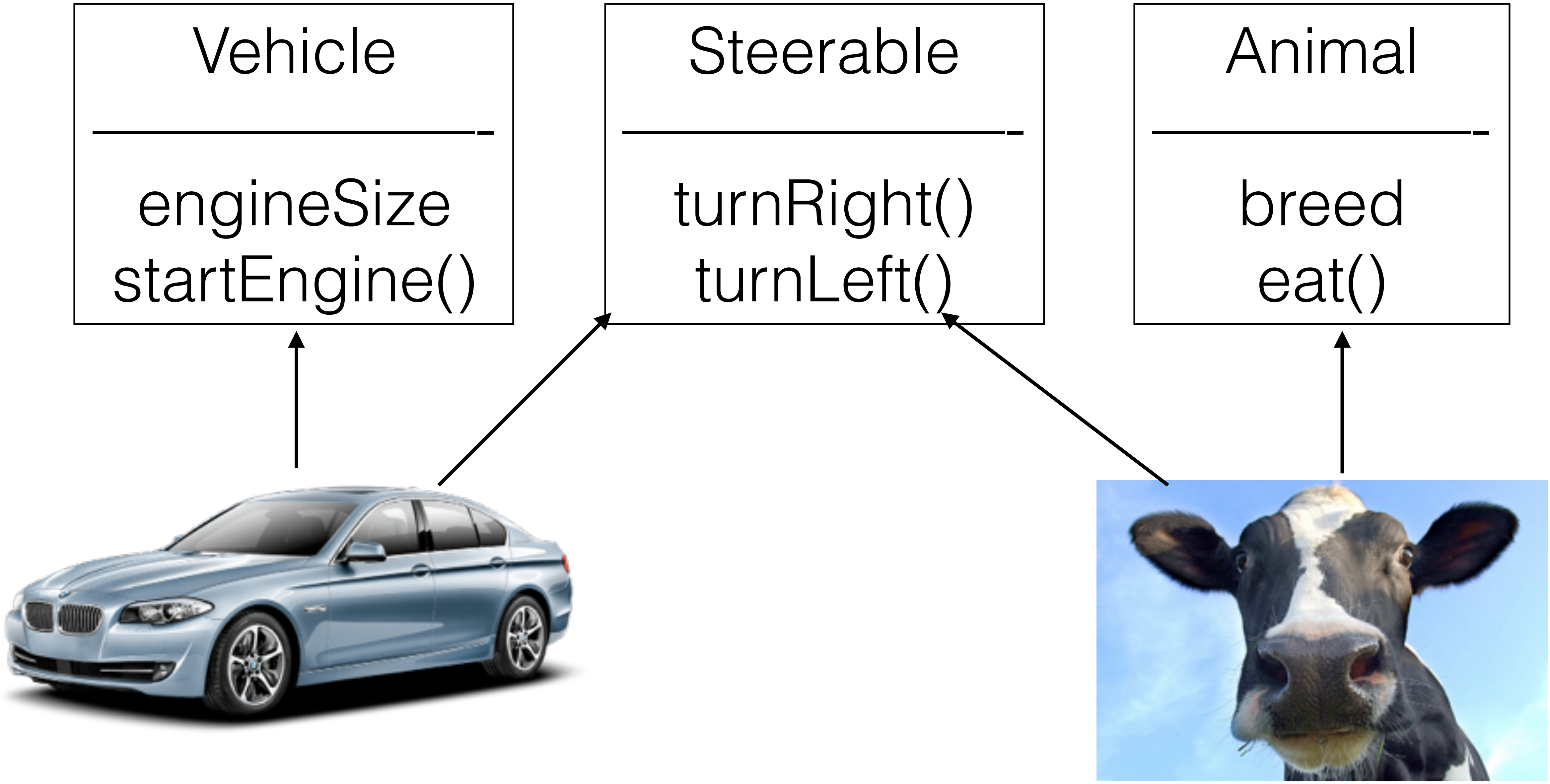
- Create Class “Cow” with functions turnRight(), turnLeft(), moveStraight()
- Change the navigateMaize method to take a cow thru

Behaviour Inheritance

engineSize
startEngine()
turnRight()
turnLeft()

breed
eat()
turnRight()
turnLeft()

Interfaces



Vehicle

engineSize
startEngine()

Steerable

turnRight()
turnLeft()

Animal

breed
eat()

More Syntax

```
class vehicle{
    int engineSize;
    public void startEngine(){

    }
}
interface steerable{
    void turnRight();
    void turnLeft();
}

class car extends vehicle implements steerable{
    public void startEngine(){

    }
    public void turnRight(){

    }
    public void turnLeft(){

    }
}
```

Abstraction and Overloading

```
abstract class vehicle{
    int engineSize;
    public void startEngine();
}
interface steerable{
    void turnRight();
    void turnLeft();
}

class car extends vehicle implements steerable{
    public void startEngine(){

    }
    public void turnRight(){

    }
    public void turnLeft(){

    }
}
```

Method Overloading and Overriding

- IS_A relationship

```
Vehicle v = new Car();
```

- Two methods can have the same name but different method parameters

```
void turnRight();
void turnRight(int degrees);
```

Static

- If you prefix any declaration with “static”, it is known static variable.
  - The static variable can be used to refer the common property of all objects (that is not unique for each object) e.g. company name of employees,college name of students etc.
  - The static variable gets memory only once in class area at the time of class loading.
- Static variables initialised using static blocks.

Packages

- Packages are used in Java to prevent naming conflicts, to control access, to make searching/locating and usage of classes, interfaces, etc

- Some of the existing packages in Java are:

  - java.lang - bundles the fundamental classes

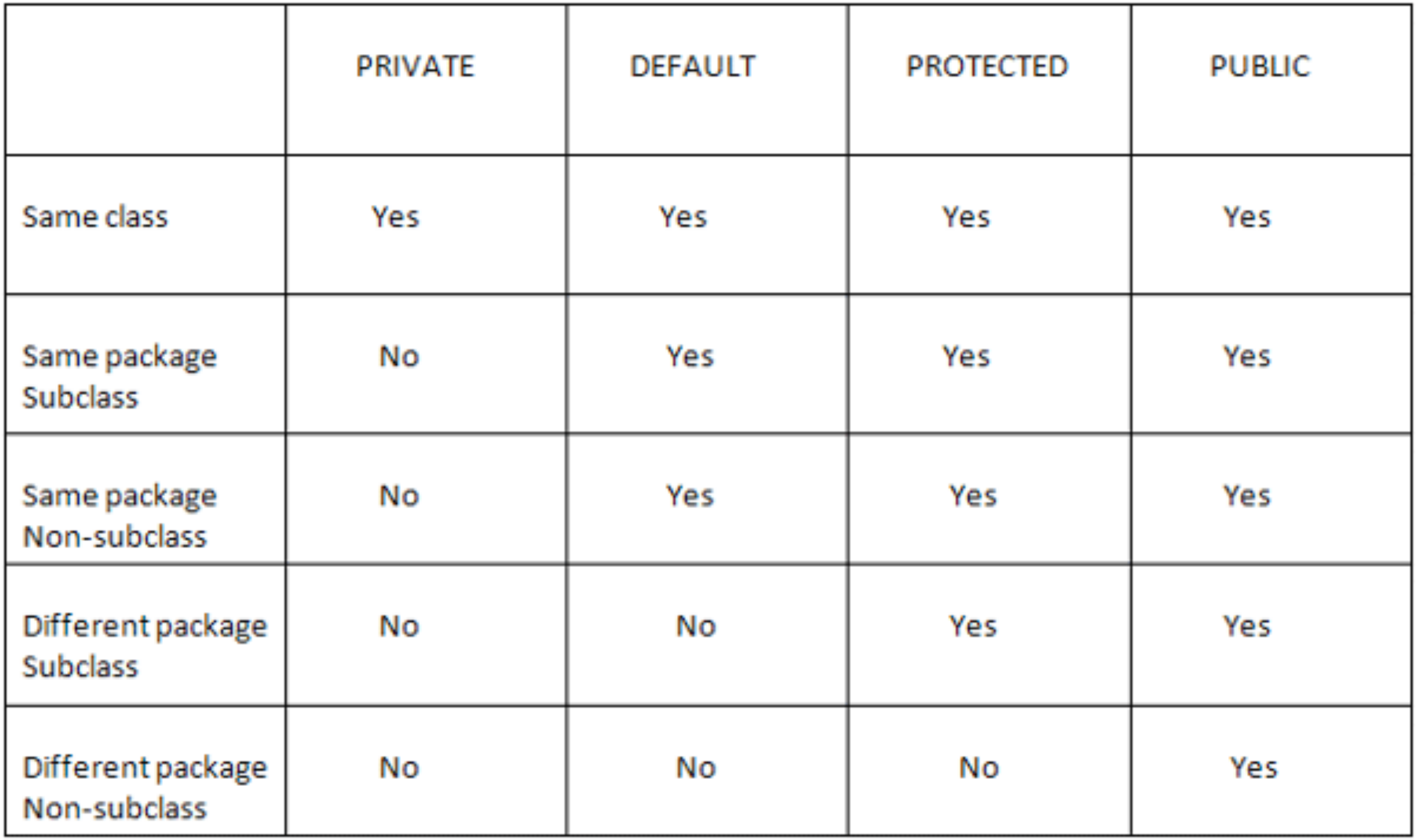
  - java.io - classes for input , output functions are bundled in this package

Create Our Packages

- A Package Declaration
- A Folder Structure
- Using classes in other packages using Import
- Classpaths and jars

```
package models;
class vehicle{
   int engineSize;
   public vehicle(int engineSize) {
      this.engineSize = engineSize;
   }
}
```

Access Control



	PRIVATE	DEFAULT	PROTECTED	PUBLIC
Same class	Yes	Yes	Yes	Yes
Same package Subclass	No	Yes	Yes	Yes
Same package Non-subclass	No	Yes	Yes	Yes
Different package Subclass	No	No	Yes	Yes
Different package Non-subclass	No	No	No	Yes

OO Principles

- Single Responsibility
- Dependency Inversion
- Interface Segregation
- Open close principle
- Liskov’s Substitution principle

- Build a calculator that accepts two numbers and an operation. Does the calculation and prints the result with the problem as follows: 2 x4 = 8. (Hint: use a class called calculator that does computation)
- Create a specialised Scientific calculator to calculate sin and cos on a number using the same operator function. Takes only one operand

Substitution Principle

- Substitution: If S is a subtype of T, then objects of type T may be replaced with objects of type S without altering any of the desirable properties of the program.
- Inheritance: If S is a subtype of T, then any term of type S can be safely used in a context where a term of type T is expected.
- The key difference - desirable, safely. Liskov Substitution Principle should be considered more constricting than sub-typing.

- Interface segregation example
- Liskov’s substitution principle example

Singly Rooted Hierarchy

- Object is the base class of all classes that dont extend some other class
- Object class defines some basic methods that are needed for all objects. Lets introduce only two now
    - toString()
    - equals()
- Use the instanceof operator

Consider this code

```
openAFile();
readFromFile();
openNetworkConnection();
sendDataOverNetwork();
```

Same code with C style error handling

```
if(openAFile()==1){
    if(readFromFile()==1){
        if(openNetworkConnection()==1){
            if(sendDataOverNetwork()==1){

            }else{
                print("Data could not be sent");
            }
        }else{
            print("Network connection cant be opened");
        }
    }else{
        print("Cant read from file");
    }
}else{
    print("Cant open file");
}
```

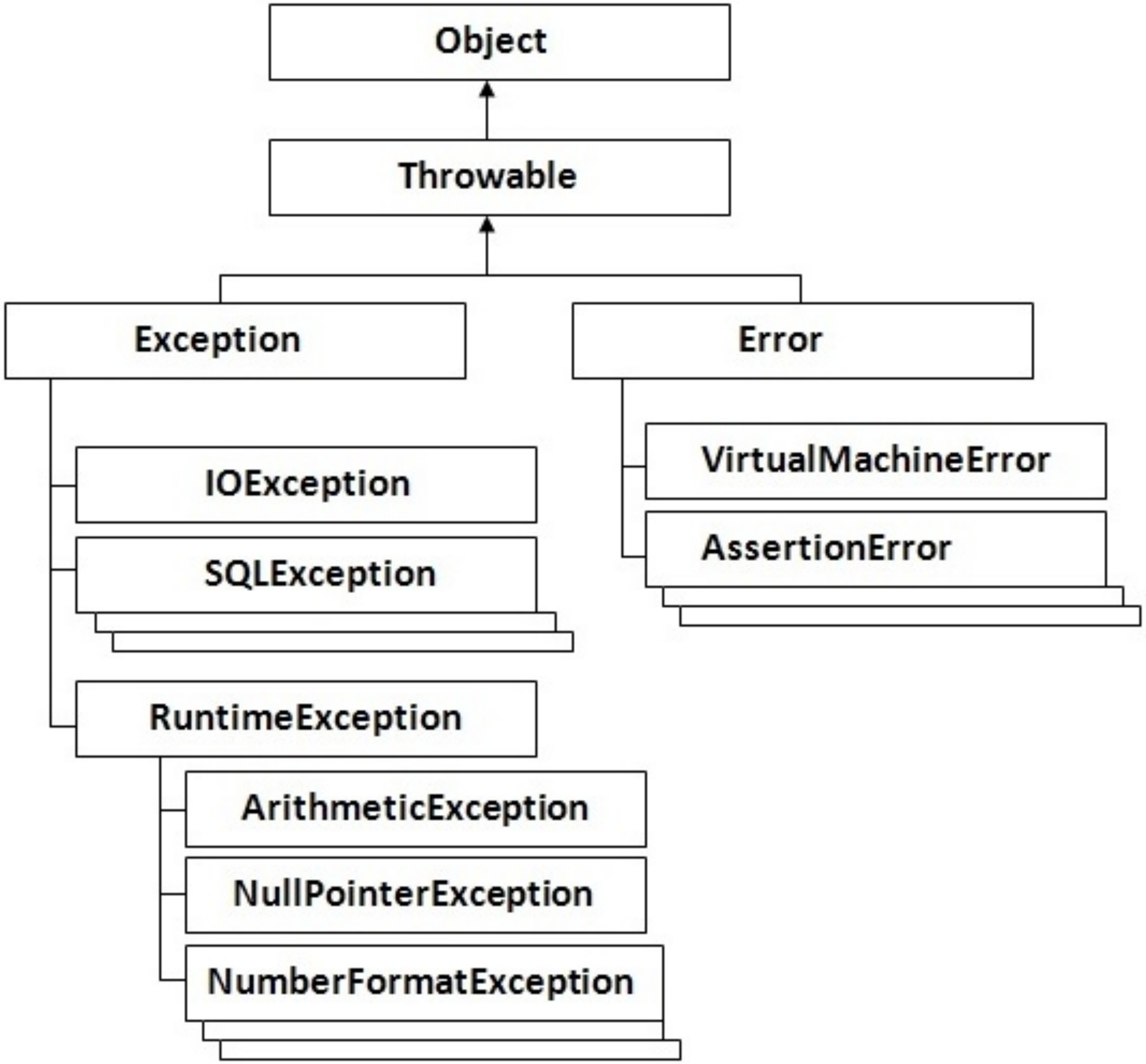
Error Handling In Java

```java
try{
   openAFile();
   readFromFile();
   openNetworkConnection()
   sendDataOverNetwork();
}catch(Exception e){
   print(e.getMessage());
}
```

Exceptions

- Exceptions are special conditions that happen during the course of execution of a program
  - Methods throw exceptions, callers catch or duck it
- All Exceptions have to be sub classes of java.lang.Throwable

Exception Hierarchy



Catching Exceptions

```
try
{
   //Protected code
}catch(ExceptionType1 e1)
{
   //Catch block
}catch(ExceptionType2 e2)
{
   //Catch block
}catch(ExceptionType3 e3)
{
   //Catch block
}
```

Throw/throws keywords

```
public void deposit(double amount) throws RemoteException{
           // Method implementation
       if(something went wrong)
           throw new RemoteException();
}
```

Finally Keyword

```
int a[] = new int[2];
try{
    System.out.println("Access element three :" + a[3]);
}catch(ArrayIndexOutOfBoundsException e){
    System.out.println("Exception thrown  :" + e);
}
finally{
    a[0] = 6;
    System.out.println("First element value: " +a[0]);
    System.out.println("The finally statement is executed");
}
```

Our Own Exceptions

```
public class InsufficientFundsException extends Exception{
      private double amount;
      public InsufficientFundsException(double amount)
      {
         this.amount = amount;
      }
      public double getAmount()
      {
         return amount;
      }
}
```

Using Custom Exceptions

```
public void withdraw(double amount) throws
   InsufficientFundsException{
     if(amount <= balance){
      balance -= amount;
     }else{
        double needs = amount - balance;
        throw new InsufficientFundsException(needs);
     }
}
```

Lets Try This

- Throw Exceptions from 3 levels of method calls.. each time catch the exception and re-throw it wrapped in another exception