# Imperative programming

# Data types: Primitive types

- byte — 1 byte signed integer
- short — 2 byte signed integer
- int — 4 byte signed integer
- long — 8 byte signed integer
- float — 32 bit floating point
- double — 64 bit floating point
- char — 2 byte unsigned integer
- boolean
- There are 4 integer types, 2 float types, 1 character type, and 1 boolean type
- Range of an integer is $-2^{n-1}$ to $2^{n-1}-1$
- Array (and hence String (which is really a type alias for char[])) are objects but are often used like primitives

# Data types

- Explicit casts are required when losing numerical precision or converting to a subclass but implicit casts occur when gaining numerical precision or converting to a superclass
- Explicit cast: `int` **`a = (int) b;`**
  - In `float ans = a/b;` where a and b are ints, the implicit cast happens after the division is carried out so you only get the integer part
    - Explicitly cast at least one of a and b instead e.g. `float ans = (float) a/b` or `float ans = a/(float) b`
- Boxed versions of the primitive types exist for use with object oriented code: Integer, Float, Double, Long, Short, Byte, Boolean, Character
- Java will implicitly cast a primitive to/from the boxed version

# Converting to/from float

- **Floats are stored in binary as SEE…MM… — mantissa is in sign and magnitude but they are separated by the exponent**

1. **Set S (1 for negative, 0 for positive)**
2. **Write in fixed point binary as a positive number**
3. **Get number into the form …001·… adjusting the exponent from an initial value of 0 to reflect how the binary point has been moved**
4. **Mantisaa = everything after binary point rounded to have correct number of bits**
5. **Add the bias to the exponent and represent as an unsigned int**

1. **Convert biased exponent to raw exponent by subtracting bias**
2. **Add 1. to front of mantissa**
3. **Move binary point using exponent**
4. **Interpret in fixed point binary using sign bit**

# IO

- **`Scanner input = new Scanner(System.in);`**
- int a = input.nextInt();
- **`String b = input.nextLine();`**
- Apart from line for string, its **`input.next`*`Primitive`*`();`**
  - Can guard with **`input.hasNext`*`Primitive`*`();`** if want to be safe
- `System.out.print`
  - Can print any primitive type
  - If passed an object calls toString method and prints the returned value
- `System.out.println` — same as print but with a new line at the end

# Evaluation

- If one argument is a String (even if the other is numeric) the + operator carries out concatenation (by first calling toString)
- **A double logical operator is lazy whereas a single logical operator is strict** — important if side effects on RHS
- ++x preincremnets x (increments x then evaluates the rest of the expression) whereas x++ postincrements x (evaluates as much of rest of the expression as it can then increments x)

```
int a = 5; int b = 5;
System.out.println(a++ == 5);System.out.println(a); // True 6
System.out.println(++b == 5);System.out.println(b); // False 6
```

- **Variables must be initialised in all code paths, the compiler will throw an error even if it is initialised in all paths that can actually occur**
- **Variables only exist in the scope of the {} they are initialised within**

# Order of operations

- **BPUMASRESLA — be pumas wrestler**
  **Brackets — (,)**
  (unary) **Postfix — a++, a--**
  **Unary (prefix) — ++a, --a**
  **Multiplicative — *, /, %**
  **Additive — +, -**
  **Shift — <<, >>**
  **Relational — <, >, <=, >=**
  **Equality — ==, !=**
  **Strict operators — &, |, ^**
  **Lazy operators — &&, ||**
  **Assignments — =, +=, -=, *=, /=**

# Selection

- **`if (predicate) {}`**
  - **If an if statement evaluates to false, execution jumps to after the closing curly bracket (same as break)**
- **Ternary operator — `a = (predicate) ? valueIfTrue : valueIfFalse;`**
  - James says is hard to read
- `switch (variable) {case …: …, case …: …., default: …}`
  - **A switch acts as a goto the first case that variable that is being switched on is equal to and then continues execution line by line** from there (ignoring the case statements) — almost always want a **break;** at the end of each case
  - default is optional

# Iteration

- Bounded repetition is for loop
- Unbounded repetition is while
- **`for (initialisation; continuationCondition; iterativeStep){}`**
  1. Execute initialization
  2. Evaluate continuationCondition
  3. If true, execute loop body, otherwise jump to after }
  4. Execute iterativeStep
  5. Go to step 2
- **continue jumps** to the next execution of iterativeStep (and then continuation check etc) **skipping over the rest of the loop body for the current iteration**
- `while (predicate) {}`
- **`do {} while (predicate);`**

# Arrays

- To declare an array put `[]` after either the type or after the variable name
  - Putting `[]` after both creates a 2D array
- To declare a 3D array put `[][][]`
- To initialise array `new primitive[size]`
- To initialize a 2D array `new primitive[rows][columns]`
- To initialise can specify all the values — `{v1, v2, …}`
  - Only works if done at same time as declaration
- The new keyword allocates memory
- **`.length` — its an attribute not a method**

# Methods

- **`public static void main(String[] args) {}`**
- **Method signature is: `accessibility return type name parameters`**
- **Function overloading — can define several methods with same name but different signatures and Java will pick the correct one**
- Primitive types are passed by value but objects and arrays are passed by reference
  - `System.arraycopy(src, srcPos, dest, destPos, length)` //reading from src starting at index srcPos, add the first length elements to dest starting at destPos (overwriting current values)
    - Most of the time we want System.arraycopy(src, 0, dest, 0, src.length)

# OOP

# Classes

- **Public classes must be in their own file whose filename matches the class name**
- Class names conventionally start with a capital letter
- **`new` is used to instantiate an object from a class**
- **Constructor has return type of class and no name** (or same name as class and no return type depending on how you want to think about it)
- Java creates a default constructor with no arguments
    - Calls the constructor with no arguments of the superclass if there is a superclass, else does nothing

class AName {

//declare attributes (object attributes and static attributes) — only initialize static ones

//constructors, initialize non-static attributes inside

//other methods

}

- Can use name of an attribute as parameter name (e.g. in constructor) as long as attribute is referred to using this. when wanted in that scope (as parameter name shadows it)

# Access modifiers

- **If an access modifier isn't specified, is package-private — can be accessed from outside its class but not outside its package**
  - **In same file ⇒ in same package**
- **`public` — can be accessed anywhere**
- **`private` — can only be accessed inside its class**
- **`protected` — can be accessed by its class and its subclasses**
- **`static` — belongs to the class rather than an object** — e.g. `Math.round(x);` not `math = new Math(); math.round(x);`
- **`final` — cannot be redefined** — if a variable cannot be mutated, if a method cannot be overridden

# Inheritance

- … **`class SomeChild extends SomeParent {...`**
- **Method overloading = defining a method with the same name but different signature**
  - Static polymorphism — which method to use is determined at compile time
- **Method overriding = defining a method with the same signature in a subclass**
  - Dynamic polymorphism — which method to use is determined at run time
- Unless the **first line of the constructor of a sub class calls the constructor of the superclass `super(...);`** Java will call super(); which probably isn't correct
- Every class (and hence everything apart from primitive types) in Java descends from the Object class
- `someObject instanceof someClass` is a boolean expression
  - Will return false if `someObject` is null

# Abstract classes

- **Abstract classes are like normal classes but can contain abstract methods and cannot be instantiated**

```
… abstract class AClass {
    …
    abstract public int someMethod();
    …
```

- **Note a abstract method declaration ends with a semicolon**
- **An interface is effectively a class with no attributes and no concrete methods**

```
… interface SomeClass {
    …
    public int someMethod();
    …
```

  - **Abstract keyword is not used**
  - An abstract or concrete class **`implements`** **an interface instead of `extends`**
  - An interface can `extends` another interface
- Cannot extend more than one class but can implement multiple interfaces (and extend up to one class at the same time)

# Misc

- Strings are (immutable) objects
  - Passed by reference
  - When changed a new string is created
- **== compares memory addresses of objects**
  - **Use .equals to compare values**
- **Enums (enumerated types) can take values from a specified list of named values**
  - `public enum Day {SUN, MON, TUE, WED, THU, FRI, SAT}; Day aDay = Day.MON;`

# Advanced Java

# Generics

- ```
  public class AClass<T> {
          private T value
          public AClass(T val) {
              value = val;
          }
      }
      public static <U> U get(U x){
          return x
      }
  AClass<String> aObject = new AClass<String>("Hello world!");
  ```
- **To use multiple type placeholders (type parameters) use a comma separated list in the angle brackets**
- **<T extends C> restricts T to C and its subclasses** — upper bounded type parameter
- <T super U> restricts T to C and its superclasses — lower bounded type parameter
  - ? acts as a wildcard in a type parameter
  - Using `<? extends U>` facilitates *co*variance (preserves the subtyping relation (*co*-operates with the subtyping relationship)) — can pass in any subtype of U and will treat it as U inside the function
  - Using `<? super U>` facilitates *contra*variance (reverses the subtyping relation (is *contra* to the subtyping relationship) — can pass in any supertype of U and will treat it as Object inside the function
  ```

# Throwable

- **Error and Exception are both subclasses of Throwable**
- **Errors are problems that are unlikely to be able to be handled — bad practice to catch**
- **Exceptions that might be able to be handled**
- **Exceptions are either checked or unchecked**
    - Checked is typically used for issues when accessing resources whereas unchecked is typically used for errors in the program itself
- **A method that calls a method that could throw a checked exception must catch or rethrow — enforced (checked) by the compiler**
    - **Rethrow — add `throws` *ExceptionType* to the method signature between the parameter list and the {**
    - **Catch — try {...} catch (*ExceptionType* e) {...}**
        - **First catch block to match top to bottom is the one that is executed — <u>superclasses should be after subclasses</u>**
        - Can put throw e if things get messy as long as throws is in method signature
    - Rethrowing means methods that call this method must also catch or rethrow
- **An unchecked exception can self-propagate outside the scope the exception was thrown in — can catch if you want but don't need to explicitly rethrow**
- The `finally` in a `try catch finally` contains code that is executed when the try (and any relevant catches) block finishes executing whatever happens during their execution — useful for closing resource handles etc

# Custom exceptions

```
public class CustomCheckedException extends Exception {
    public CustomCheckedException(String msg, Throwable e) {
        super(msg, e);
    }
}
public class CustomUncheckedException extends RuntimeException {
    public CustomUncheckedException(String msg, Throwable e) {
        super(msg, e);
    }
}
…
    throw new CustomCheckedException("...", null);
    throw new CustomUncheckedException("...", null);
```

# Reflection

- Reflection allows information about a class to be accessed at run time — class does not need to be accessible at compile time as long as it will be accessible at run time
- **`Class.forName("className").getDeclaredMethods()`** returns an array of Method objects
  - **`getDeclaredConstructors`** gives an array of Constructor objects
  - **`getDeclaredFields`** gives an array of Field objects
- **To use a `Method m`, `m.invoke(object, arg1, …)`**
- **To use a `Constructor c`, `c.newInstance(arg1, …)`**
- **To use a `Field f`**
  - **`f.get(object)`**
  - **`f.set(object, value)`**
- Use `null` as `object` for static methods/fields — can use anything that exists but null is standard

# Concurrency

```
public class … implements Runnable {
    …
    public void run(){
        …
    }
}
c = new …();
Thread t = new Thread(c);
t.start();
```

**or**

```
public class … extends Thread {
    …
    public void run(){
        …
    }
}
c = new …();
c.start();
```

# Concurrency: Synchronization

- **A Synchronized (v){} block only allows a thread to enter the block if no one owns the lock for the object v**
  - **v.notify() resumes a random thread that is waiting to enter the block — the random thread will only actually acquire the lock and start executing if the lock has been released** (otherwise it can't enter the block)
  - v.notifyAll() resumes all threads that are waiting to enter the block — they will compete for the lock
  - **v.wait(); causes the current thread to pause execution and release the lock**
- **Can only synchronize on objects**
- No point in synchronizing on immutable objects (e.g. String, Integer etc) because when changed they return a new object which we won't be synchronizing on
- **Concurrency with primitive types can be made safe with `volatile` — only guarantees safety for atomic operations e.g. a=a+5; is unsafe but b=a;b+=5;a=b; is safe**

# *Common interfaces*

- *Classes that implement Iterable can be iterated over*
  - *Can be implemented by simply creating and managing an iterator*
  - *Can use a for-each loop on an iterable object*

    ```
    for (type x: aList) {System.out.println(x);}
    ```
- *Classes that implement Iterator can be used to iterate — e.g. iterator = aList.iterator(); first = iterator.next();*
- *Classes that implement Comparable can be compared*
- *Classes that implement Comparator can be used to compare*
  - *Allows objects to be compared even if they don't implement Comparable (or they do and you don't want to use their ordering)*

# *Functional Interfaces*

- *These provide types for lambda expressions, method references, and use in signatures of higher order functions*

```
public interface Supplier<T> {
    T get();
}
public interface Consumer<T> {
    void accept(T t);
}
public interface Function<T, R> {
    R apply(T t);
}
public interface BiFunction<T, U, R> {
    R apply(T t, U u);
}
```

# *Functional programming*

- *Lambda expressions — `(parameter1, parameter2, …) -> {...}`*
- *Method references — `Cls::mthd` is syntactic sugar for a lambda function that calls the mthd method of an object of class Cls*
- *Streams are lazily evaluated and do not mutate the underlying data structure*
  - *`forEach` is used insead of `map` if you are only interested in side effects not in the returned values*
- *All collections have a .stream() method*
- *There is a method Arrays.stream(...)*