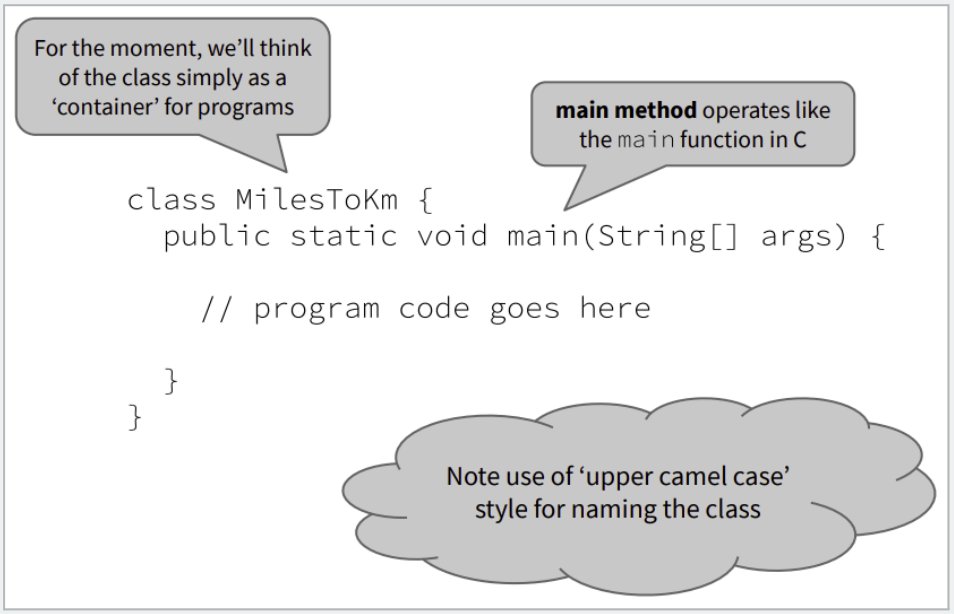
Java

Basics

Classes



Input - reading from standardinput

- Create a Scanner object, associated with System.in.
- Call one of its methods to parse the input stream, returning a value of the desired type.

```
System.out.print("Enter distance in miles: ");
Scanner input = new Scanner(System.in);
double distanceMiles = input.nextDouble();
```

input = name of Scanner object
nextDouble = name of method belonging to Scanner class

Input types

`nextBoolean()`	Boolean
`nextByte()`	Byte
`nextFloat()`	Double
`nextInt()`	Int
`nextLine()`	String

`nextLong()`	Long
`nextShort()`	Short
`nextFloat()`	Float

Output

Call methods on System.out:

`print`	Print text
`println`	Print text followed by newline
`printf`	Do formatted printing of values, embedded in surrounding text if necessary (similar to C)

```
System.out.printf("%.1f miles = %.1f km\n", distanceMiles, distanceKm);
```

Objects

- Values of the **primitive types** (char, int, double, etc) are always created on the stack.
- Objects are always created on the heap, using new.
- Compare with C - where data normally resides on the stack unless we use a pointer and allocate structure using Malloc.
- Objects are created using the new operator.
- Objects are manipulated indirectly, via references.

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

Input is a reference - similar to pointer in C but does not need to be dereferenced.

If statement

```
if (condition) {
  // block of code to be executed if the condition is true
}
```

Example: basic program

Task

To create a small program that converts a distance in miles into a distance in kilometres

The program should behave like this:

```
$ java MilesToKm
Enter a distance in miles: 42
42.0 miles = 67.2 km
```

Program

```
public class MilesToKm {
  public static void main(String[] args) {
    System.out.print("Enter distance in miles: ");
    Scanner input = new Scanner(System.in);
    double distanceMiles = input.nextDouble();

    double distanceKm = distanceMiles * 1.6;
    System.out.printf("%.1f miles = %.1f km\n", distanceMiles, distanceKm);
  }
}
```

Variables

Boolean

```
boolean check = true;
Boolean isPositive = n > 0;
```

Java will not treat 0 as false and non-zero as true.

Strings

Strings are implemented as a class called string. Text is represented using unicode characters.

Strings don't need to use new.

```
String greeting = "Hello";
String alert = "Temperature is 95\u00b0"; // unicode escape sequence
String greeting = "Hello " + name + "!"; // concatenation using +
```

Strings are immutable; they never change after creation. String methods return a new version of the string they are called on. This allows strings to be managed in a more memory efficient way.

Reading in strings

```
Scanner input = new Scanner(System.in);
String word = input.next();
String line = input.nextLine();
```

String length

```
String txt = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
System.out.println("The length of the txt string is: " + txt.length());
```

String index starts from 0.

String handling

`toUpperCase()`	Makes string upper case.	`System.out.println(txt.toUpperCase());`
`toLowerCase()`	Makes string lower case.	`System.out.println(txt.toLowerCase());`
`indexOf()`	Returns the index (the position) of the first occurrence of a specified text in a string (including whitespace).	`String txt = "Please locate where 'locate' occurs!";` `System.out.println(txt.indexOf("locate")); // Outputs 7`
`+`	String concatenation.	`System.out.println(firstName + " " + lastName);`
`concat()`	String concatenation.	`System.out.println(firstName.concat(lastName));`
`isEmpty`	Returns true if length is 0, false otherwise.	
`charAt`	Returns character at given index.	
`substring`	Returns substring, given two indices and returns characters from index 1 up to but not including index 4.	`s.substring(1, 4)`
`codePointAt(int index)`	Returns character (unicode code point) at index.	
`compareTo(String str)`	Compares two strings lexicographically.	
`concat(string str)`	Concatenates the	

	specified string to the end of this string.	
`contains(CharSequence s)`	Returns true if string contains the specified sequence of char values.	
`contentEquals(CharSequence cs)` `contentEquals(StringBuffer sb)`	Compares string to specified CharSequence/StringBuffer.	

Character handling

`Character.isDigit();`	Checks if character is digit.	`Character.isDigit(text.charAt(1));`
`Character.isLowerCase();`	Checks if character is lower case.	`Character.isLowerCase(text.charAt(1));`
`Character.isUpperCase();`	Checks if character is upper case.	`Character.isUpperCase(text.charAt(1));`

Special characters

Result	Escape character
‘	\’
“	\”
\\	\
New line	\n
Carriage return	\r
Tab	\t
Backspace	\b
Form feed	\f

StringBuilder

```
StringBuilder lineBuilder = new StringBuilder();

for (int i = 0; i < lineSize; ++i) {
    lineBuilder.append('*');
}

String line = lineBuilder.toString();
```

Loops

For

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

While

```
int n = 0;
while (n < 10) {
    System.out.println(n);
    ++n;
}
```

Advanced loop control

Break

Terminates a loop prematurely if a stopping condition suddenly becomes true.

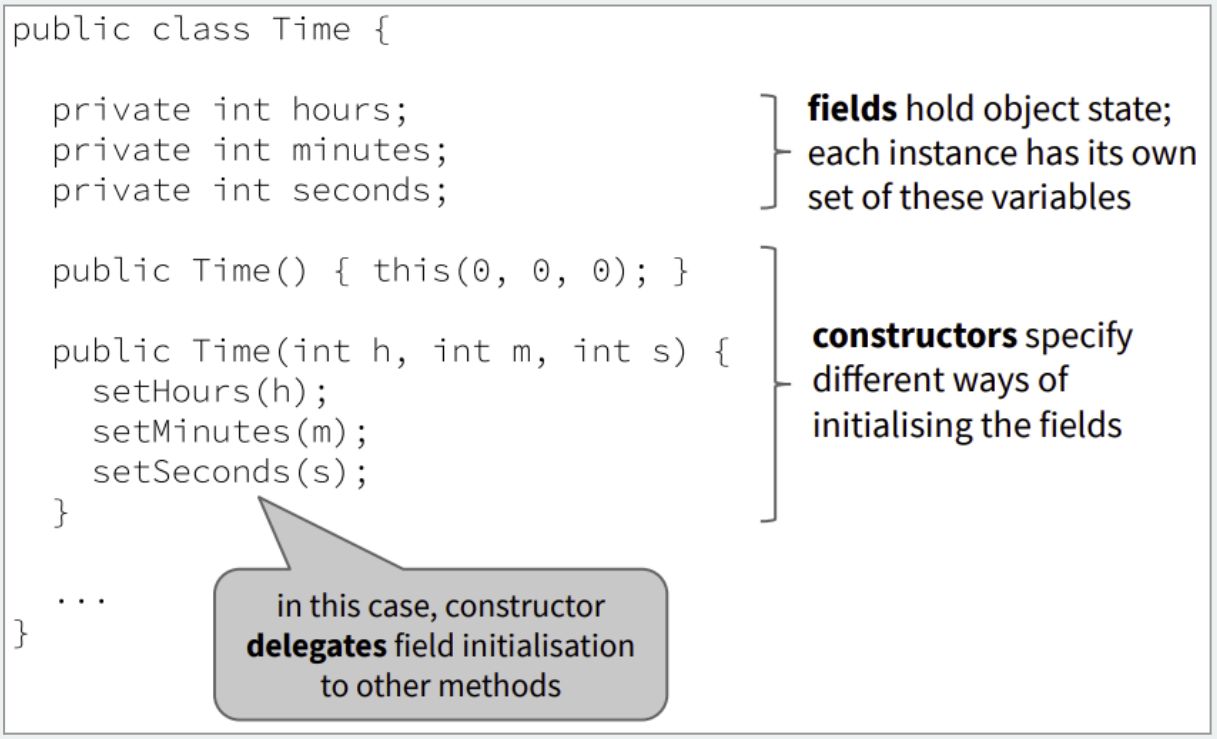
```
Scanner input = new Scanner(System.in);

System.out.print("Enter a value > 0: ");

while (true) {
  int value = input.nextInt();
  if (value > 0) {
    break;
  }
  else {
    System.out.print("Nope, try again: ");
  }
}
```

Continue

Skips the remainder of a loop body if a particular condition becomes true, jumping to the start of the next iteration.

Classes

Software classes are abstractions that represent the attributes of a class as a set of fields and its behaviour as a set of methods.



Constructors

- Constructors are called to create an instance of the class.
- They look like methods but have no return type and must have exactly the same name as the class. They must also initialise the fields directly, or do so indirectly by calling other methods.
- Can call other constructors by using the 'this' keyword.

Constructors are used to initialise fields:

```
public class Time {
   private int hours;
   private int minutes;
   private int seconds;
   ...
   Public Time() {
          hours = 0;
          minutes = 0;
          seconds = 0;
   }
   ...
}
```

Constructors can be overloaded with multiple versions, provided that the parameter lists are different, providing different ways to make an object.

```
public class Time {
   ...
   public Time() { ... }

   public Time(int h, int m, int s) {
         hours = h;
         minutes = m;
         seconds = s;
   }
   ...
}
```

Adding validation

```
public class Time {
  ...
  public Time(int h, int m, int s) {
    if (h < 0 || h >= 24) {
      throw new IllegalArgumentException("invalid hours");
    }
    if (m < 0 || m >= 60) {
      throw new IllegalArgumentException("invalid minutes");
    }
    if (s < 0 || s >= 60) {
      throw new IllegalArgumentException("invalid seconds");
    }
    hours = h;
    minutes = m;
    seconds = s;
  }
}
```

an exception will halt normal execution unless it is intercepted and dealt with…

Put code to validate and set each field in its own method.

Getters and setters

```
public class Time {
  ...

  public int getHours() {
    return hours;
  }

  public void setHours(int h) {
    if (h < 0 || h >= HOURS_IN_A_DAY) {
      throw new IllegalArgumentException("Invalid hours");
    }
    hours = h;
  }

  // + similar code for minutes & seconds
}
```

notice the parameter validity checking done here!

Getters

Getters (/accessor methods) provide information on object state.

- Getters can return field values.
- Allow users of class to access object state.

```
public class Time {
    private int hours;
    Private int minutes;
    private int seconds;

    public int getHours() { return hours; }
    ...
}
```

Fields usually have getters, but don't alway need them.

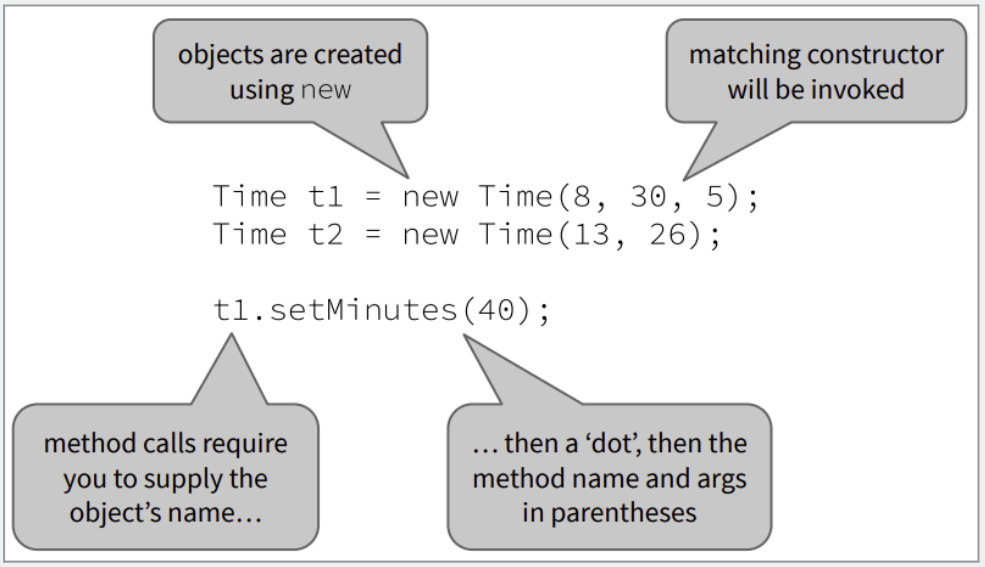
Naming convention:
For 'return the value' getters, get+field name.

Setters

Setters (/mutator methods) modify object state by changing fields to given values.

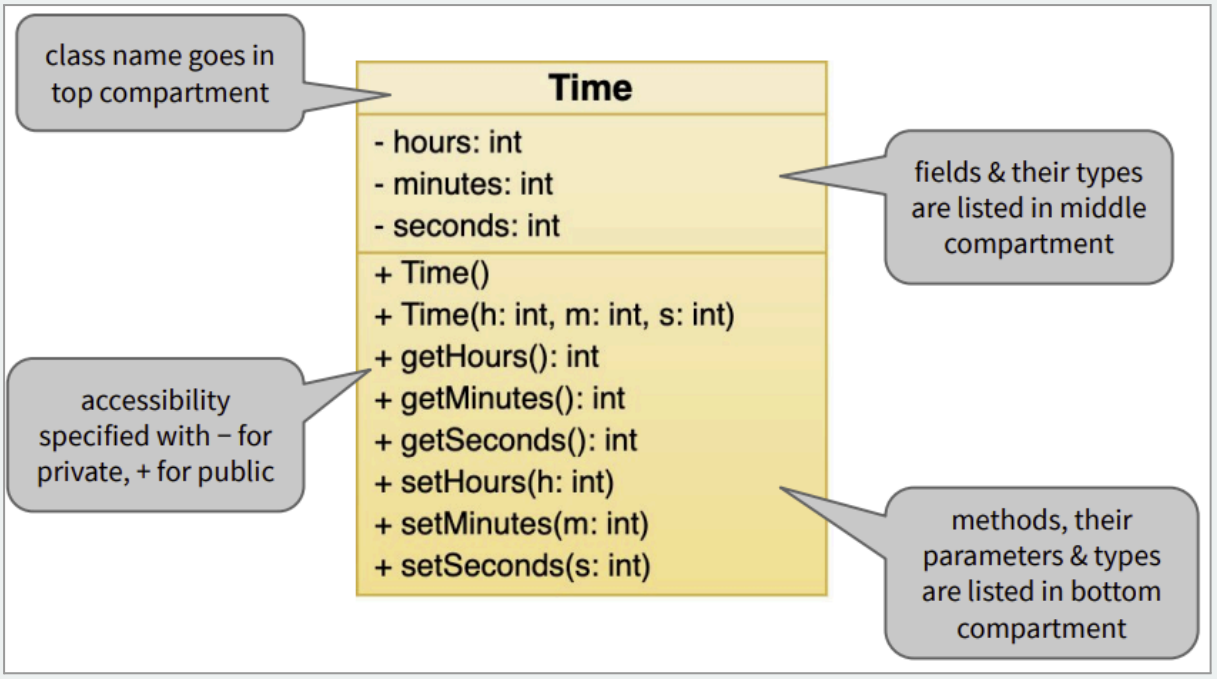
- Setters should check the validity of the given values before modifying a field; they can throw an exception to signal that a value is not valid.

Using the class



Example

A class in UML



Initial implementation steps:

I. Start with class containing field definitions.
II. Add getter methods for those fields (if necessary).
III. Add one or more constructors.

Implementation

```
public class Time {

  private int hours;
  private int minutes;
  private int seconds;

  public Time() { this(0, 0, 0); }

  public Time(int h, int m, int s) {
    setHours(h);
    setMinutes(m);
    setSeconds(s);
  }

  ...
}
```

fields hold object state; each instance has its own set of these variables

constructors specify different ways of initialising the fields

in this case, constructor **delegates** field initialisation to other methods

Do not do this:

```
public class Time {
   public int hours;
   public int minutes;
   public int seconds;
}
```

It allows other code to freely modify Time objects and to make them invalid.
Instead, fields should be made private. This gives us controlled access via methods.

Using class

```
Time t1 = new time();
Time t2 = new Time(8, 45, 30);

System.out.println(t2.getHours());
```

Class constants

We can make a variable available to all methods by defining it as a constant field, using the final modifier.
We also use static – which associates the field with the class rather than giving every instance its own copy.

Naming convention:
Upper case, with words separated by underscores.

```
private static final int HOURS_IN_A_DAY = 24;
```

Overriding a method

Classes get a default version inherited from parent class object which isn't very useful. You can override it to create a version that yields meaningful string representation of the object.

toString

```
@Override
public String toString() {
    return String.format("%02d:%02d:%02d", hours, minutes, seconds);
}
```

Will be automatically called to generate a string when needed.

equals

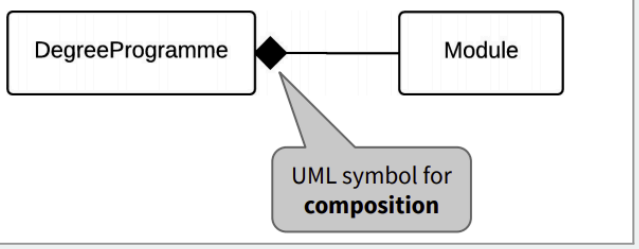
We need the equals method because the == operator tests whether two object references point to the same object.
Classes get a default equals method from Object that does the same as ==, but we can override it with our own version.

```
@Override
public boolean equals(Object other) {
  if (other == this) {
    // Object is being compared with itself!
    return true;
  }
  else if (other instanceof Time) {
    Time otherTime = (Time) other;
    return hours == otherTime.hours &&
           minutes == otherTime.minutes &&
           seconds == otherTime.seconds;
  }
  else {
    // other is not a Time object
    return false;
  }
}
```

Use of collections

When modelling a problem we might find that one class represents the component part of some other class. This implies that the 'container' class holds a collection of instances of the component class.

Example: `Module` is a part of `DegreeProgramme`




```
public class Module {
    private String code;
    private String title;
    private int credits;
    ...
}

public class DegreeProgramme {
    private String name;
    private Module[] modules;
    ...
}
```

Or...

```
public class DegreeProgramme {
    private String name;
    private List<Module> modules;
    ...
}
```

Collection types

Arrays

- Created on the heap using `new`.
- Size can be computed at run time but remains fixed.
- Size provided as a property called `length`.
- Primitive types initialised to a default value automatically.
    - ➢ 0 for numbers.
- Bounds checking done on element access, resulting in an exception if bounds are exceeded.

Arrays of primitives

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

System.out.println(x.length); // prints 10

System.out.println(x[0]); // prints 0
System.out.println(x[1]); // prints 0
...
System.out.println(x[9]); // prints 0
System.out.println(x[10]); // exception!!
```

Arrays of objects

```
Time[] t = new Time[10];

System.out.println(t.length); // prints 10

System.out.println(t[0]); // prints null
System.out.println(t[1]); // prints null

t[0] = new Time(7, 30, 15);
System.out.println(t[0]); // prints 07:30:15
```

Memory management

In java, you allocate storage for objects and arrays using `new` and do not need to free it.

Static initialisation

```
double[] x = { 0.1, 0.3, 0.5, 0.7 };
```

Multidimensional arrays

```
int[][] x = new int[3][4];

int[][] y = { { 1, 2, 3, 4 },
              { 2, 4, 6, 8 },
              { 7, 5, 3, 1 } };
```

Iteration

```
for (int i = 0; i < data.length; ++i) {
   double number = data[i];
   ...
}

for (double number: data) {
   ...
}
```

Lists

- Represents an ordered sequence of values.
- More flexible than arrays; can grow or shrink.
- Implemented as classes, so methods can manipulate list contents in useful ways.
- Two different implementations in `java.util`:
  - `ArrayList`
  - `LinkedList`

Creating a list

List element type must be specified in angle brackets.

```
ArrayList<String> lines = new ArrayList<String>();

LinkedList<String> words = new LinkedList<String>();

LinkedList<String> words = new LinkedList<>();
```

Or...

```
List<String> lines = new ArrayList<>();
List<String> words = new LinkedList<>();
```

List is an interface implemented by the `ArrayList` and `LinkedList` classes. This helps to decouple code from implementation choices, so it is easier to change the chosen implementation later.

Wrapper classes

This is wrong:

```
List<int> data = new ArrayList<>();
```

This is because collections hold objects, not instances of primitive types. So, instead we use the `Integer` wrapper class.

Prepopulated lists

Lists returned by List.of are unmodifiable.

```
List<Integer> numbers = List.of(1, 2, 3);
List<String> words = List.of("One", "Two", "Three");
```

List methods

`size`	Returns number of elements in list.
`isEmpty`	Returns true if list is empty, false otherwise.
`get`	Returns element at the given integer index.
`indexOf`	Returns position of the first occurrence of the specified item, or -1 if item is not present.
`add`	Adds item to the end of the list, or at given index.
`clear`	Removes all items.
`toArray`	Returns an array containing the list items.

Iteration

```
for (int i = 0; i < data.size(); ++i) {
    double value = data.get(i);
    ...
}

for (double value: data) {
    ...
}
```

Stack and queues

Stack:

- LIFO (Last In, First Out).
- Items are pushed onto the top of the stack.
- Then popped off the top, in reverse order.

Queue:

- FIFO (First In, First Out).
- Items are added to the tail of the queue.
- And removed from the head.

Implementation

- Stacks and queues are both implemented using a deque (pronounced “deck”) – a double-ended queue.
- Two implementations of deque provided in Java; ArrayDeque and LinkedList.
- Use Deque interface to manipulate a deque as a stack (since this specifies push and pop methods).
- Use the more restrictive Queue interface if you want it to act strictly as a queue.

```
Deque<String> stack = new ArrayDeque<>();
stack.push("Alice");
stack.push("Bob");
stack.push("Charlie");
while (!stack.isEmpty()) {
  System.out.println(stack.pop());
}

Queue<String> queue = new ArrayDeque<>();
queue.add("Alice");
queue.add("Bob");
queue.add("Charlie");
while (!queue.isEmpty()) {
  System.out.println(queue.remove());
}
```

could use `LinkedList` here instead

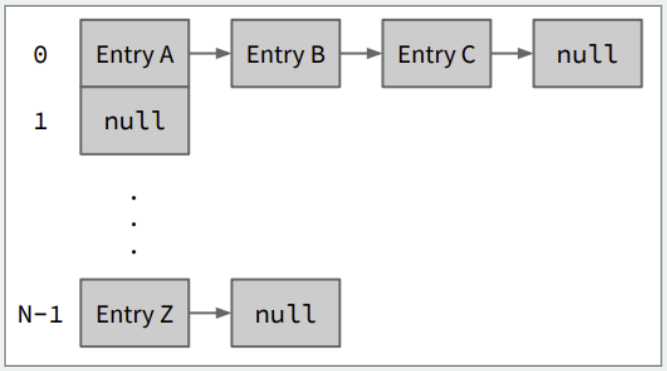
Associative collections

- Not sequences; items are not referenced by position.
- Instead, we have the idea of a value being looked up using a piece of associated information: the key.
- In effect, we have collections of key-value pairs.
- Keys are unique and there is only one value associated with each key (although that value could be another kind of collection).

Maps

- Most useful type of associative collection.
- Represented by Map interface.
- Three standard implementations:
  - HashMap : fast, but keys not ordered.
  - LinkedHashMap: less efficient than HashMap, but preserves order of key-value insertion.
  - TreeMap: keeps keys sorted into their ‘natural order’, or according to the provided comparator.

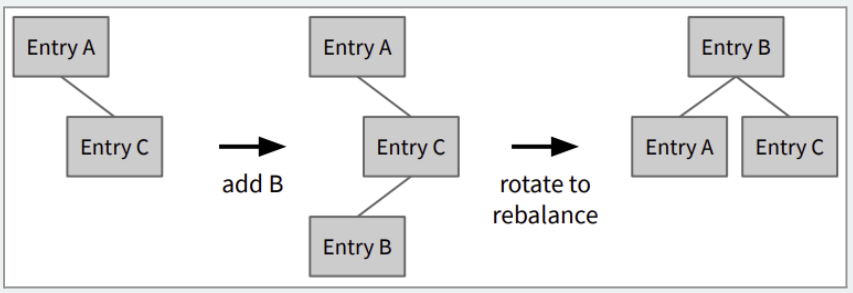
Hashmaps



- Implemented as an array of N buckets - linked lists of Entry objects.
- Each Entry object holds a key, a value and a hashcode of the key.
- N is a power of two, initially 16; when 75% of buckets are non-null, array size is double and entries are redistributed, to keep lists small.

If we are fetching from the map and an Entry with the matching key is found, we return the value; otherwise we return null. If we are storing in the map and a matching Entry is found, we update it with the value we want to store; otherwise, we created a new Entry at the head of the list.

TreeMaps



- Implemented as a self-balancing binary search tree.
- Goal is to optimise search by maintaining a tree of keys where keys that come before the parent's key are to the left and keys that come after are to the right.
- Tree rebalances itself after insertions.

Creating a map

```
Map<String, Integer> map1 = new HashMap<>();
Map<String, Integer> map2 = new LinkedHashMap<>();
Map<String, Integer> map3 = new TreeMap<>();
```

Here, String is the key's type and Integer is the value's type. The key should always be an immutable type.

Map methods

`size`	Returns number of key-value pairs in map.
`isEmpty`	Returns true if map is empty.
`containsKey`	Returns true if map contains given key.
`containsValue`	Returns true if map contains given value.
`get`	Returns value associated with given key, or null.
`getOrDefault`	Returns value for given key or the given default.
`put`	Inserts a new key-value pair or updates a key.
`remove`	Removes a key-value pair, given the key.
`clear`	Removes all key-value pairs from map.

Iterating over a map

```
Map<String, Integer> map = new HashMap<>();
map.put("apple", 42);
map.put("orange", 34);
map.put("mango", 100);

for (String key: map.keySet()) {
   int value = map.get(key);
   System.out.printf("%s: %d\n", key, value);
}
```

Sets

Set = a collection of items that cannot contain duplicates (same as in maths).

- Can be thought of as a map containing only keys, no values.
- Manipulated via the Set interface.
- With similar implementations to those for maps.
  - HashSet
  - LinkedHashSet
  - TreeSet

Collection-friendly classes

- To work properly as key of a hash-based collection, a class should define a proper hashCode method and a proper equals method.
- To work properly as key of a tree-based collection, a class must implement the Comparable<> interface and provide a compareTo method.
  - This also is required to sort instances of the class in a list.

hashCode():

- Must return the same int value consistently for the same object, if that object hasn't been modified.
- Must return the same int value for two objects that are equal.
- Should return different values if possible for two objects that are not equal (to get good performance in hash-based collections).

IDEs can generate hashCode() and equals() automatically.

compareTo():

To make your class implement the Comparable<> interface (from java.util), it must provide a compareTo method that:

- Has another object of the class as the parameter.
- Returns a positive integer if the object you are calling the method is 'greater than' the other object.
- Returns a negative integer if the object you are calling the method on is 'less than' the other.
- Returns 0 if the two objects are equal.

```
public class Time implements Comparable<Time> {
  ...
  public int compareTo(Time other) {
    int comp = Integer.compare(hours, other.hours);
    if (comp == 0) {
      comp = Integer.compare(minutes, other.minutes);
      if (comp == 0) {
        comp = Integer.compare(seconds, other.seconds);
      }
    }
    return comp;
  }
}
```

Maths

Symbols

You do not need an import statement to use things defined in the Math class.

Pi	`math.pi`
Power	`Math.pow(base,exponent)`
Square root	`Math.sqrt(double a)`

Decimal places

```
System.out.printf("%.2f", val);
```

Prints to two decimal places. Must use printf.

Static methods

Defining a method with the static keyword means that it is associated with the class and not with any particular object of that class. So, the class mainly acts as a container or namespace for the method.

```
public class MilesToKm {
  private static double getDistanceInMiles() {
    Scanner input = new Scanner(System.in);
    System.out.print("Enter distance in miles: ");
    return input.nextDouble();
  }

  private static double milesToKm(double distMiles) {
    return distMiles * 1.6;
  }

  public static void main(String[] args) {
    double distMiles = getDistanceInMiles();
    double distKm = milesToKm(distMiles);
    System.out.printf("%.1f miles = %.1f km\n",
      distMiles, distKm);
  }
}
```

Example

Unit conversion utility class

```
public class Conversion {
  public static final double KM_PER_MILE = 1.6;
  public static final double CM_PER_INCH = 2.54;

  public static double milesToKm(double miles) {
    return miles * KM_PER_MILE;
  }

  public static double inchestoCm(double inches) {
    return inches * CM_PER_INCH;
  }

  ...
}
```

Input utility class

```
public class Input {
  public static double getDouble(String prompt) {
    Scanner input = new Scanner(System.in);
    System.out.print(prompt);
    return input.nextDouble();
  }

  public static int getInteger(String prompt) {
    Scanner input = new Scanner(System.in);
    System.out.print(prompt);
    return input.nextInt();
  }
  ...
}
```

Final result

Call static methods from the Input and Conversion classes by using the class name as a prefix:

```
public class MilesToKm {
  public static void main(String[] args) {
    double distMiles = Input.getDouble("How many miles? ");
    double distKm = Conversion.milesToKm(distMiles);
    System.out.printf("%.1f miles = %.1f km\n",
      distMiles, distKm);
  }
}
```

Static imports

Fields and methods of a class that are public and static can be imported, allowing us to omit the class name.

```
import static java.lang.Math.cos;
import static java.lang.Math.sin;
import static java.lang.Math.toRadians;

public class PolarToCartesian {
  public static void main(String[] args) {
    ...
    double theta = toRadians(input.nextDouble());
    double x = r * cos(theta);
    double y = r * sin(theta);
    ...
  }
}
```

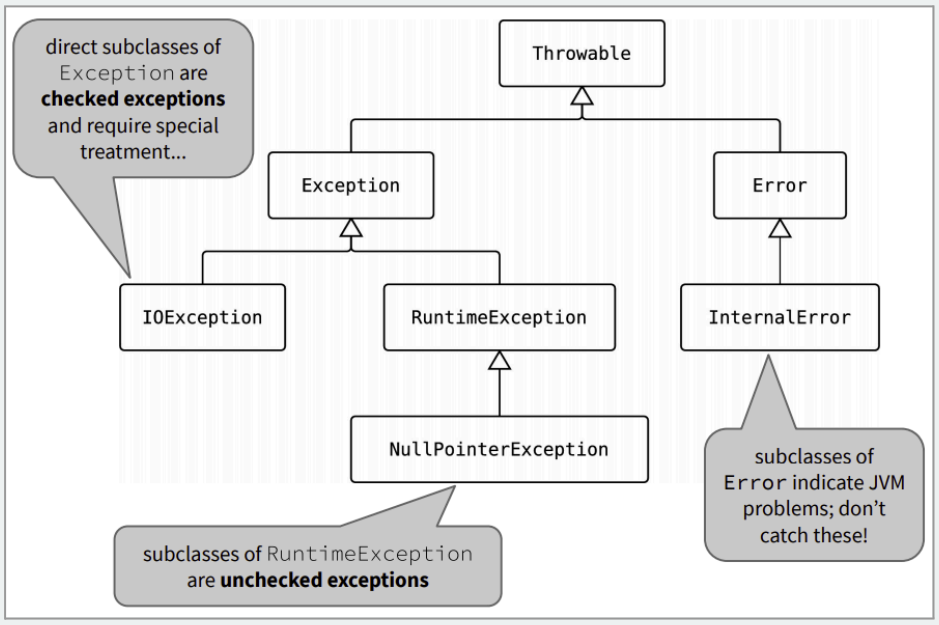
no `Math.` prefix here!

Exceptions

Exceptions are how errors are dealt with in object-oriented languages. They are messages that cannot be ignored and will halt execution until dealt with.

Exception object = a signal that something has gone wrong during program execution.

Exception class conveys information about the type of error, and the object can store additional information in its fields.



Throwing your own exceptions

Throwing an exception signals to code elsewhere that a problem has occurred, and that you are unable (or unwilling) to deal with it yourself.

```
public void withdraw(int amount) {
    if (amount < 0 || amount > balance) {
           throw new IllegalArgumentException("Invalid amount");
    }

    balance -= amount;
}
```

Normal flow of control ceases immediately. Control is passed up the call stack, until an exception handler for the exception is found. If no matching exception handler is found by the time the method is reached, the program halts and a stack trace is displayed.

Catching exceptions

This is done in a different place from where the exception is thrown.
Wrap code that can cause the exception in a try block and follow this with one or more catch blocks. If one of the catch blocks matches the type of exception thrown, the code in that catch block runs.

```
try {
    // Code that can generate different kinds of exception
}
catch (FileNotFoundException error) {
    // This runs if code in try block produces a FileNotFoundException
}
catch (Exception error) {
    // This runs if code in try block produces any other kind of exception
}
finally {
    // Always runs
}

// Eventually, control passes to here.
```

Throw exceptions when you don't know how to deal with an error and want the caller to decide.
Don't micromanage exception handling — i.e., don't wrap every statement in its own try ... catch.

Files

Reading text files

Use:

`Files.readString()`
Or...
`Files.readAlllines()`

Use:
`BufferedReader`

Use:
`Scanner`

Approach 1

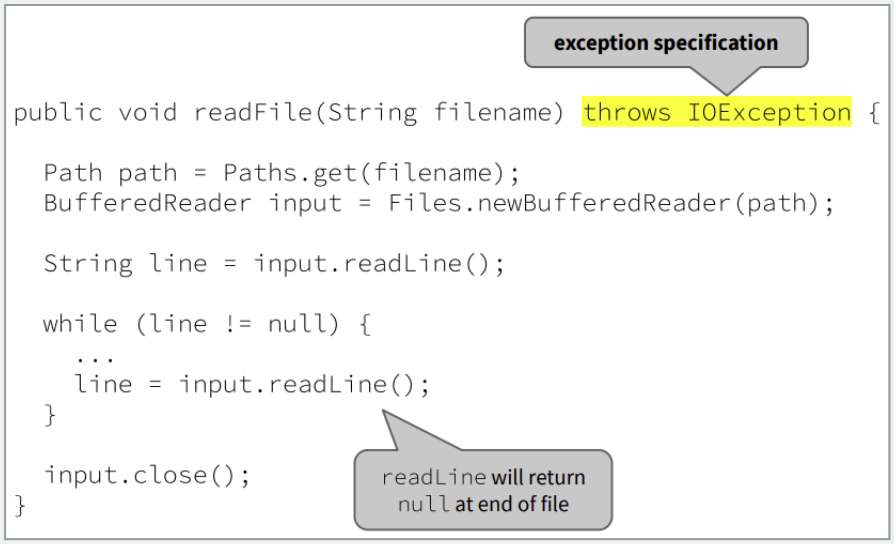
Using classes from the java.nio.file package.

Create a Path object for a file:
```
Path path = Paths.get("foo.txt");
```

Then read file into a string (>= JDK 11):
```
String text = Files.readString(path);
```

Or read lines into a list of strings:
```
List<String> lines = Files.readAllLines(path);
```

Approach 2: BufferedReader



If readFile doesn't catch a checked exception, it must declare that it can be thrown.
Do not write these for unchecked exceptions.

Approach 3: Scanner

```
public void readFile(String filename)
 throws IOException {

  Scanner input = new Scanner(Paths.get(filename));

  while (input.hasNextLine()) {
    String line = input.nextLine();
    ...
  }

  input.close();
}
```

note consistency between methods here; if you are reading lines, you **must** use the has method that checks for existence of more lines

For numeric data

```
public void readFile(String filename)
 throws IOException {

  Scanner input = new Scanner(Paths.get(filename));

  while (input.hasNextInt()) {
    int value = input.nextInt();
    ...
  }

  input.close();
}
```

... and similarly for `double` or other numeric types

Try with resources

Closes the file and ensures that allocated resources are released on leaving the try block – even if the code within generates an exception. This is useful when writing to files.

```
try (Scanner input = new Scanner(Paths.get(name)} {
    // Use scanner object here
}

// File has been closed when we get to here
```

Writing text files

PrintWriter can be used to do formatted output:

```
Path path = Paths.get(filename);

try (PrintWriter out = new PrintWriter(
    Files.newBufferedWriter(Path))) {
          ...
    out.printf("%.3f\n", value);
}
```

Inheritance and other relationships

Composition / aggregation

```
class Band {

  private Set<Musician> members;
  ...

  public void join(Musician musician) {
    members.add(musician);
  }

  public void leave(Musician musician) {
    members.remove(musician);
  }

}
```

Inheritance

```
public class Person {

  private String givenName;
  private String familyName;
  private LocalDate dateOfBirth;
  ...
}

public class Student extends Person {

  private String degree;
  private YearMonth start;
  ...
}
```

Student inherits fields and methods from Person and can add its own

Constructors in the subclass need to ensure that both the inherited fields and added fields are initialised. However, they don't have direct access to inherited fields, so

we invoke a superclass constructor from within a subclass constructor, using `super`.

```
public class Student extends Person {

  private String degree;
  private YearMonth start;

  public Student(String given, String family,
    LocalDate birth, String degree, YearMonth start) {

    super(given, family, birth);

    this.degree = degree;
    this.start = start;
  }
  ...
}
```

invokes superclass constructor with matching parameters

Method overriding

- A method in a subclass can override the equivalent method in the superclass.
- Method signatures must match (return type, number and type of method's parameters).
- `@Override` annotation can be used to indicate your explicit intent to override a method.
- Version inherited from the superclass can still be called by using `super` as a prefix - e.g., `super.toString()`

Overriding vs overloading

In overloading, we define multiple methods with the same name, which are distinguished from each other by their parameter lists. Overloading can happen within a single class or across an inheritance hierarchy.

In overriding, we replace an inherited version of a method with a new version — so parameter lists (and return types) must be the same.

Protected access

- A subclass cannot access the private elements of its superclass.
- So subclasses generally interact with inherited fields via public getters and setters.
- Alternative approach is to declare superclass fields as `protected`, which grants privileged access to subclasses.

Preventing inheritance

Declare the class as final:

```
public final class Person {
    ...
}
```

Methods can be declared as final too, to prevent overriding:

```
public class Person {
    ...
    public final boolean isBirthday() { ... }
}
```

Overriding

```
class Person {
  public void speak() {
    System.out.println("I am a Person");
  }
}

class Student extends Person {
  @Override public void speak() {
    System.out.println("I am a Student");
  }
}
```

```
Person p1 = new Person();
p1.speak();

Student s = new Student();
s.speak();

Person p2 = new Student();
p2.speak();
```

'implicit casting', or **upcasting**

Upcasting doesn't change the type of the object being referenced, merely the way we refer to it. So the version of `speak` associated with the object actually being referenced is called; this is **dynamic binding.**

Polymorphism

Liskov Substitution Principle

```
class Picture {
  ...
  public void add(Shape shape) { ... }
}


Picture picture = new Picture();
picture.add(new Circle(...));
picture.add(new Rectangle(...));
```

In code that expects a Shape, we can substitute an instance of a subclass – such as Circle or Rectangle – without affecting the behaviour of the code.

```
public void draw(Graphics2D context) {
  for (Shape shape: shapes) {
    if (shape instanceof Circle) {
      Circle c = (Circle) shape;
      c.draw(context);
    }
    else if (shape instanceof Rectangle) {
      Rectangle r = (Rectangle) shape;
      r.draw(context);
    }
  }
}
```

'explicit casting', or **downcasting**

Shape has a draw method, which is overridden in subclasses.

```
public void draw(Graphics2D context) {
  for (Shape shape: shapes) {
    shape.draw(context);
  }
}
```

dynamic binding ensures that the appropriate version of `draw` is called in each case

A problem would be, there is nothing stopping us from creating a `shape` object and calling its `draw` method, even though it makes no sense.

```
public class Shape {
  ...

  public void draw(Graphics2D context) {
    // What goes in here?
  }
}
```

Fix 1

```
public class Shape {
  ...

  public void draw(Graphics2D context) {
    throw new UnsupportedOperationException(
      "Calling draw on a Shape is not permitted");
  }
}
```

This is one solution. However, is it even valid to create a `Shape` object?

Fix 2

```
public abstract class Shape {
  ...

  public abstract void draw(Graphics2D context);
}
```

no method body!

This is better, as it prevents the creation of `Shape` objects.

Abstract classes

- Act as 'partial classes'.
- May support code reuse by supplying subclasses with common fields and methods.
- Typically used to provide a polymorphic interface to certain methods that subclasses will implement.
- Cannot be instantiated themselves, but subclasses can implement the abstract methods to become concrete classes, capable of being instantiated.
- If a subclass doesn't implement all abstract methods then it, too, will be abstract.

Pure abstract classes

- Contain only abstract methods; supply no fields or method implementations to subclasses.
- Sole purpose to act as a polymorphic interface.
- But there are limitations, which is why Java provides a separate interface type.

Misc.

Library

Java's library of classes is organised into packages.
Scanner is defined in the java.util package, so its full name is java.util.Scanner.
You can either use the full name or use an import statement at the top of the .java file.

```
import java.util.Scanner;
```

Class	Package	Use
`Scanner`	`java.util`	Input

System exit

You can use the following code to exit a program prematurely, before the end of main.

```
System.exit();
```

This method expects an integer status code, and by convention a non-zero status is used to indicate that the program did not execute successfully.

Chaining of method calls

```
String word = “programming”;
System.out.println(word.toUpperCase().charAt(6));
System.out.println(word);
```

Examples

Reading numbers in a loop

```
Scanner input = new scanner(system.in);
while (input.hasNextDouble()) {
    Double value = input.nextDouble();
    …
}
```

Notes