Core Java

A general-purpose programming language that is class-based, object-oriented, and designed to have as few implementation dependencies as possible.

Cognixia™

Outline

Prerequisites

1. JDK Version 11
   - If on Windows, make sure **java is on your path**: How to set up Path
   - At least Java 11
2. Eclipse Download **(Windows Users)**
   - Select option for **“Eclipse IDE for Enterprise Java Developers”**
   - Create a folder in root directory called **“Java_Workspace”** and set your workspace to this folder when asked
   - Once opened, click **Help** -> **Eclipse Marketplace** -> **Search for “Spring Tools 4”** -> **Install**
3. STS Download **(Mac/Linux Users)**
   - Click on download for STS on Eclipse

Introduction to Java

What is Java?

- ➔ Java is a **programming language** and a platform
- ➔ **Platform** – any hardware or software environment in which a program runs
- ➔ Used in 9 billion devices around the world
- ➔ Used in 4 types of Applications:
  - ◆ Standalone Application
  - ◆ Web Application
  - ◆ Enterprise Application
  - ◆ Mobile Application

Why Java?

According to Sun, the **Java** language is simple because:

- ➔ **Simple syntax**, based in C++ (easier to learn it after C++)
- ➔ Removed confusing and/or rarely-used features (explicit pointer, operator overloading etc.)
- ➔ No need to remove unreferenced objects, there is **Automatic Garbage Collection**
- ➔ **Object-oriented**
- ➔ **Architecture neutral**

Java Virtual Machine (JVM)

→ What is a Virtual Machine?

JVM

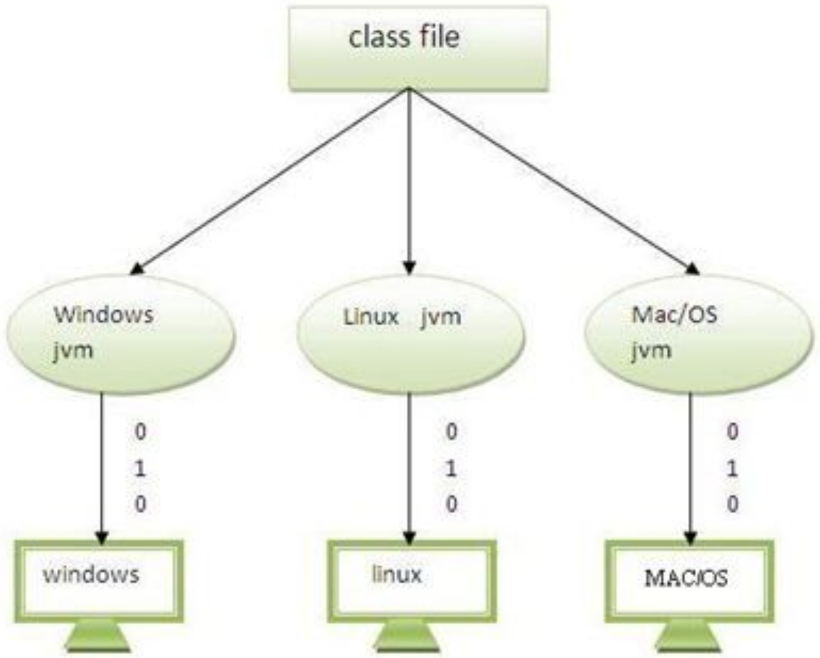
Java Virtual Machine (JVM)

- ➔ Abstract machine
- ➔ **Provides runtime environment** for java bytecode to be executed
- ➔ Main tasks:
  - ◆ *Loads code*
  - ◆ *Verifies code*
  - ◆ *Executes code*
  - ◆ *Provides runtime environment*

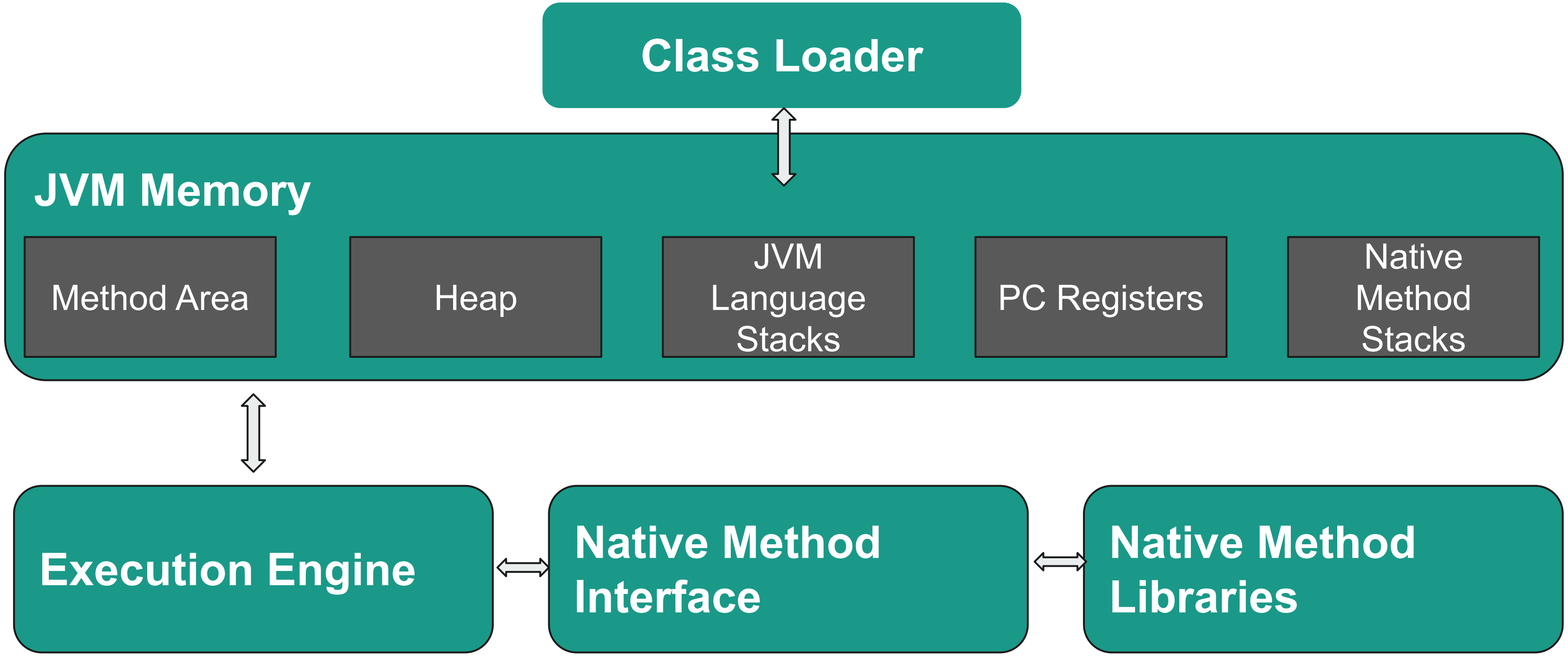
JVM - Platform Independence

- Java code can be run on **multiple platforms**
    - Windows, Linux, Sun Solaris, Mac/OS
- Java code is compiled by the compiler and converted into **bytecode**
- This **bytecode is a platform independent** code because it can be run on multiple platforms
    - Write Once and Run Anywhere (WORA)

JVM - Platform Independence

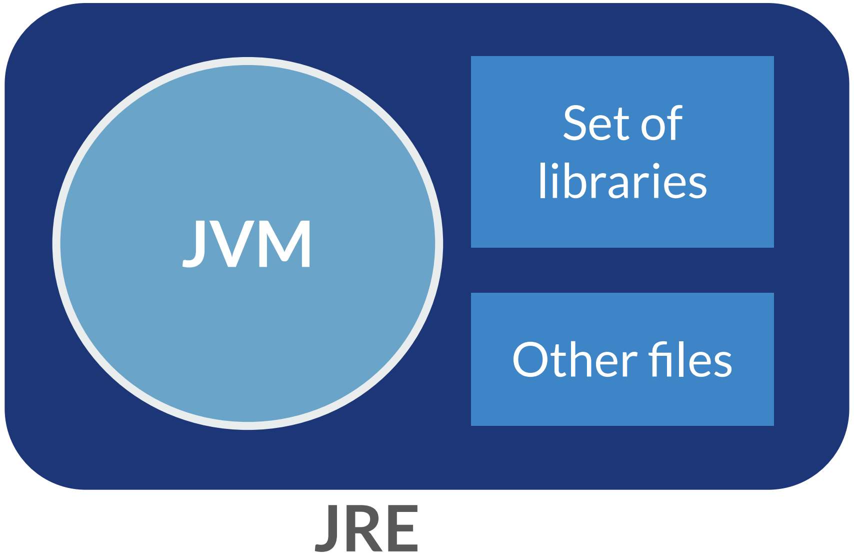


JVM - Architecture



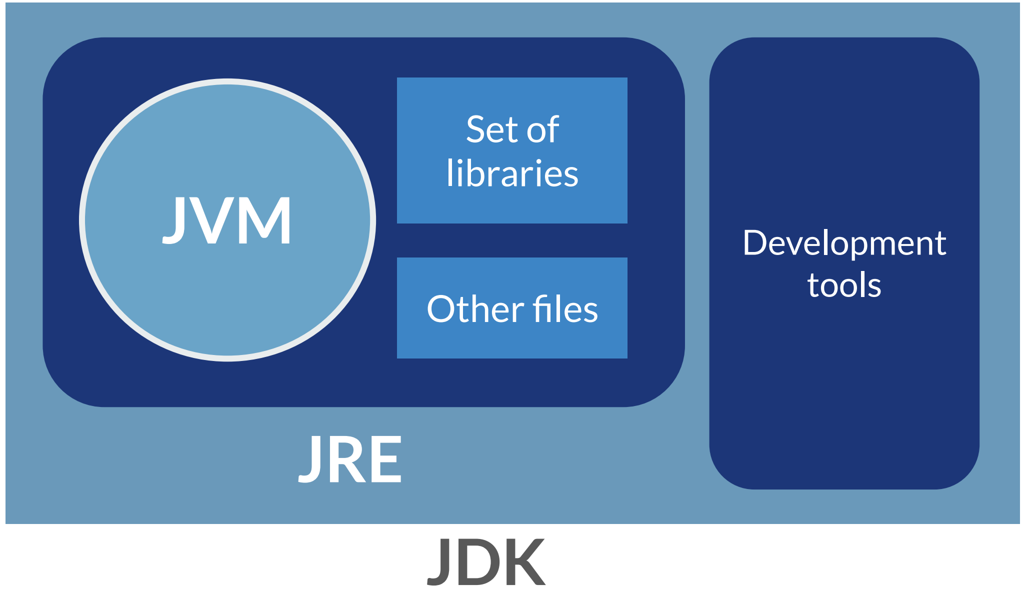
Java Runtime Environment (JRE)

- ➔ Provides runtime environment
- ➔ Implementation of JVM
- ➔ Physically exists
- ➔ **Contains libraries + other files that JVM uses at runtime**



Java Development Kit (JDK)

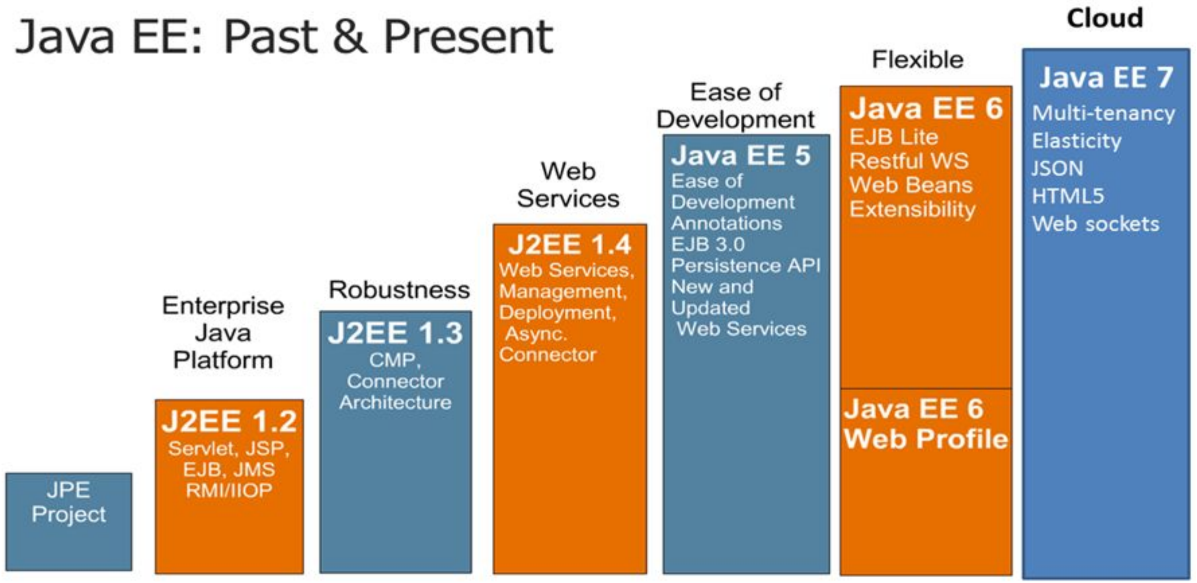
- ➔ Physically exists
- ➔ Contains JRE + development tools



Java Editions

- Java Standard Edition (J2SE)
  - Core Java Platform targeting applications running on workstations
- Java Enterprise Edition (J2EE / JEE)
  - Component-based approach to developing distributed, multi-tier enterprise applications
- Java Micro Edition (J2ME)
  - Targeted at small, stand-alone or connectable consumer and embedded devices

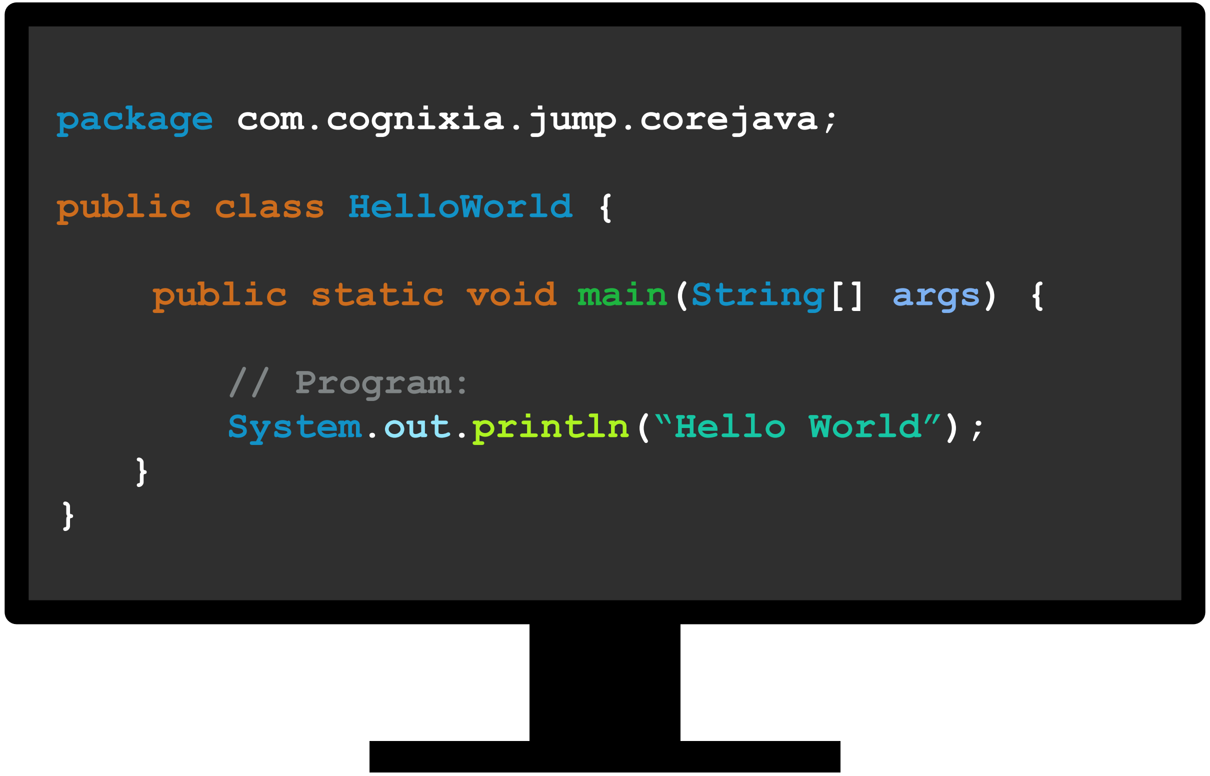
Java - Continued Evolution



Java - JAR , WAR, and EAR

JAR:
EJB modules which contains enterprise java beans class files and EJB deployment descriptor are packed as JAR files with .jar extension
WAR:
Web modules which contains Servlet class files, JSP Files, supporting files, GIF and HTML files are packaged as JAR file with .war(web archive) extension
EAR:
All above files(.jar and .war) are packaged as JAR file with .   ear (enterprise archive) extension and deployed into Application Server

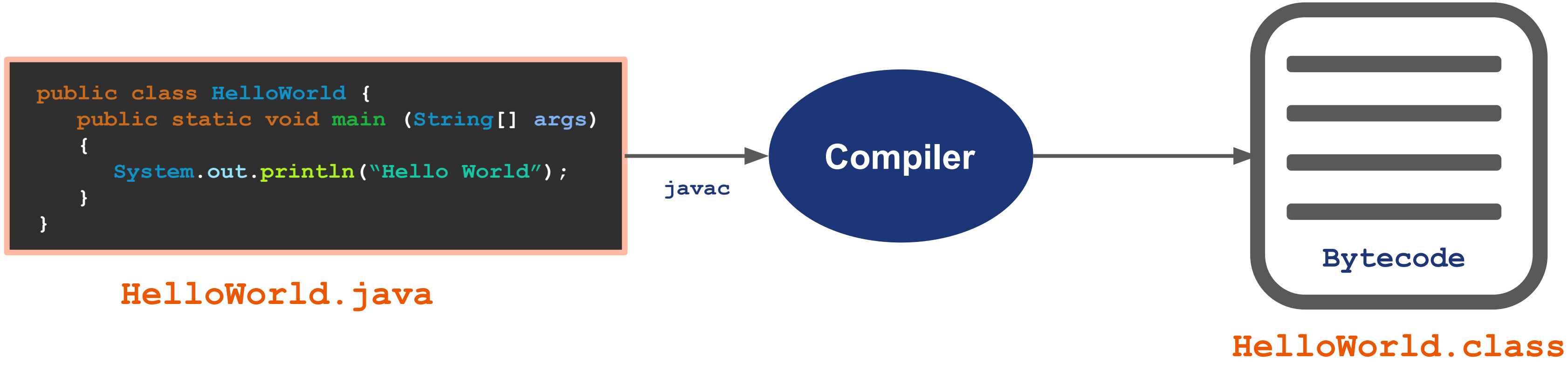
Hello World: First Program



```
package com.cognixia.jump.corejava;

public class HelloWorld {

    public static void main(String[] args) {

        // Program:
        System.out.println(“Hello World”);
   }
}
```

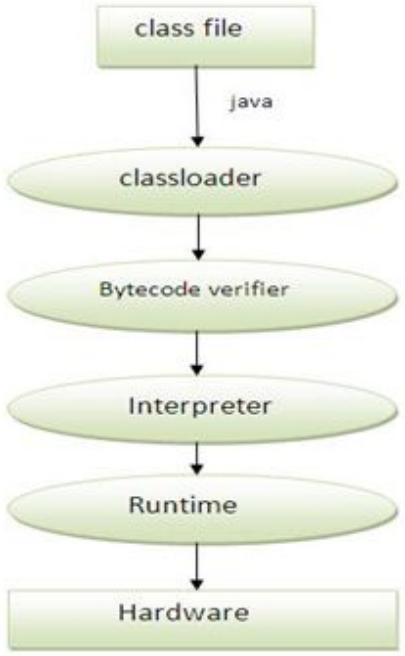
What Happens at Compile Time?

At **compile time**, the java file is compiled by the Java Compiler (does not interact with the OS) and converts the java code into bytecode.



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

What Happens at Runtime?



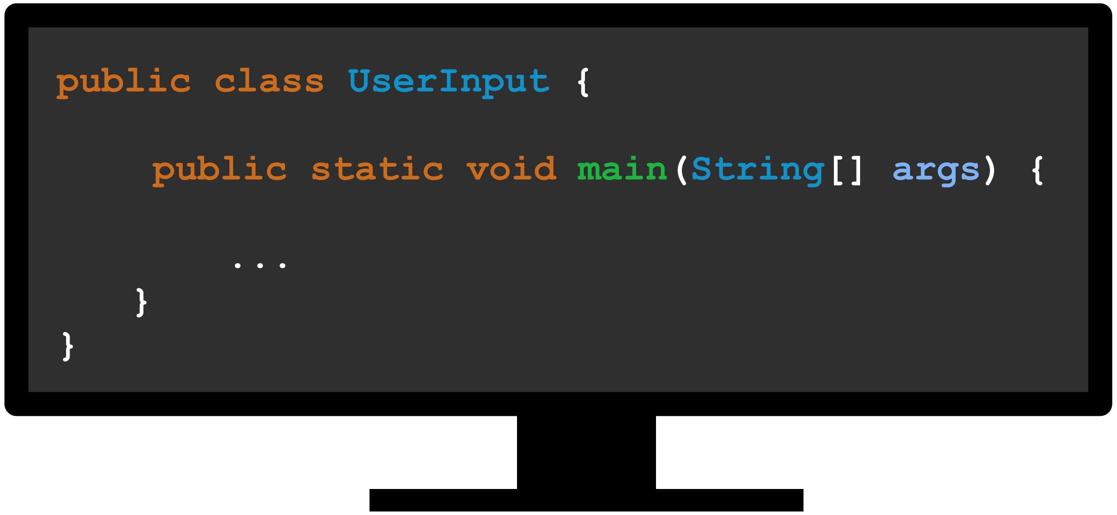
Basic Programming

Variable Types

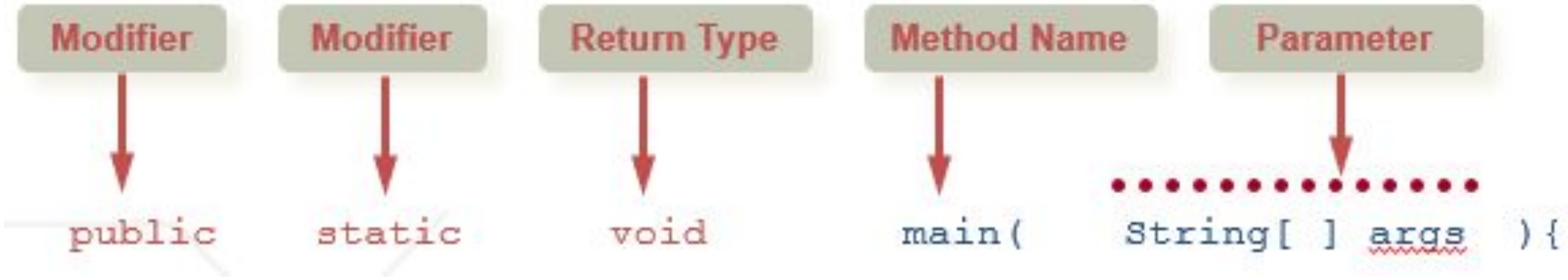
Primitives - most basic data type in Java, hold pure, simple values of a kind

Name	**byte**	**short**	**int**	**long**	**float**	**double**	**char**	**boolean**
Value	number	number	number	number	float number	float number	character	true or false
Size	1 byte	2 byte	4 byte	8 byte	4 byte	8 byte	2 byte	1 byte

The main method



```
public class UserInput {

    public static void main(String[] args) {

            ...
    }
}
```

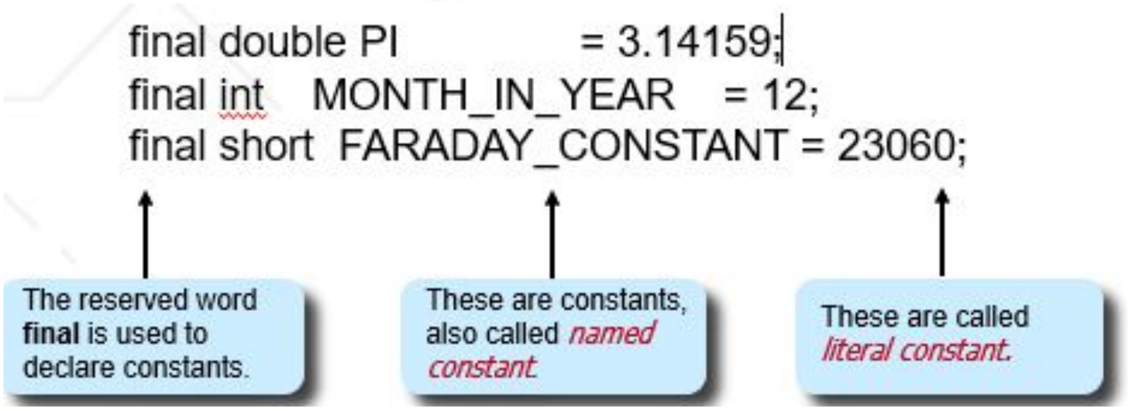


Final Keyword

Final - a constant in Java

Final can apply to:

- ➔ **Variables**
- ➔ Methods and Classes (covered later)
- ➔ Are Immutable - *constant*



```
final double PI              = 3.14159;
final int    MONTH_IN_YEAR   = 12;
final short  FARADAY_CONSTANT = 23060;
```

The reserved word **final** is used to declare constants.

These are constants, also called *named constant*.

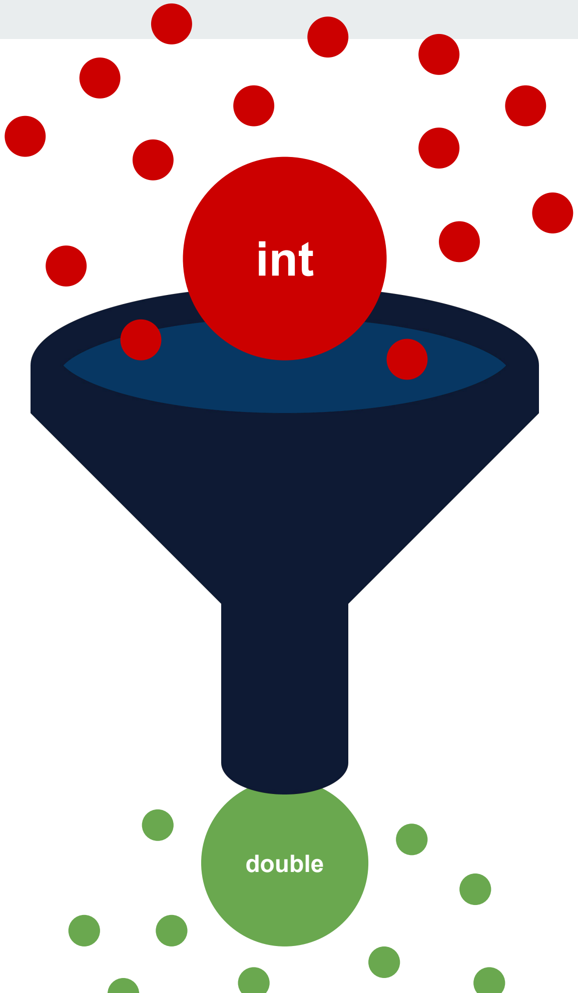
These are called *literal constant.*

Casting Variables

Casting variables is explicitly convert one type to another.

Primitives and **Objects** can be converted between one another using casting.

```java
// converts from double to int
double dubs = 5.0;
int num = (int) dubs;
```



Order of Precedence (High to Low)

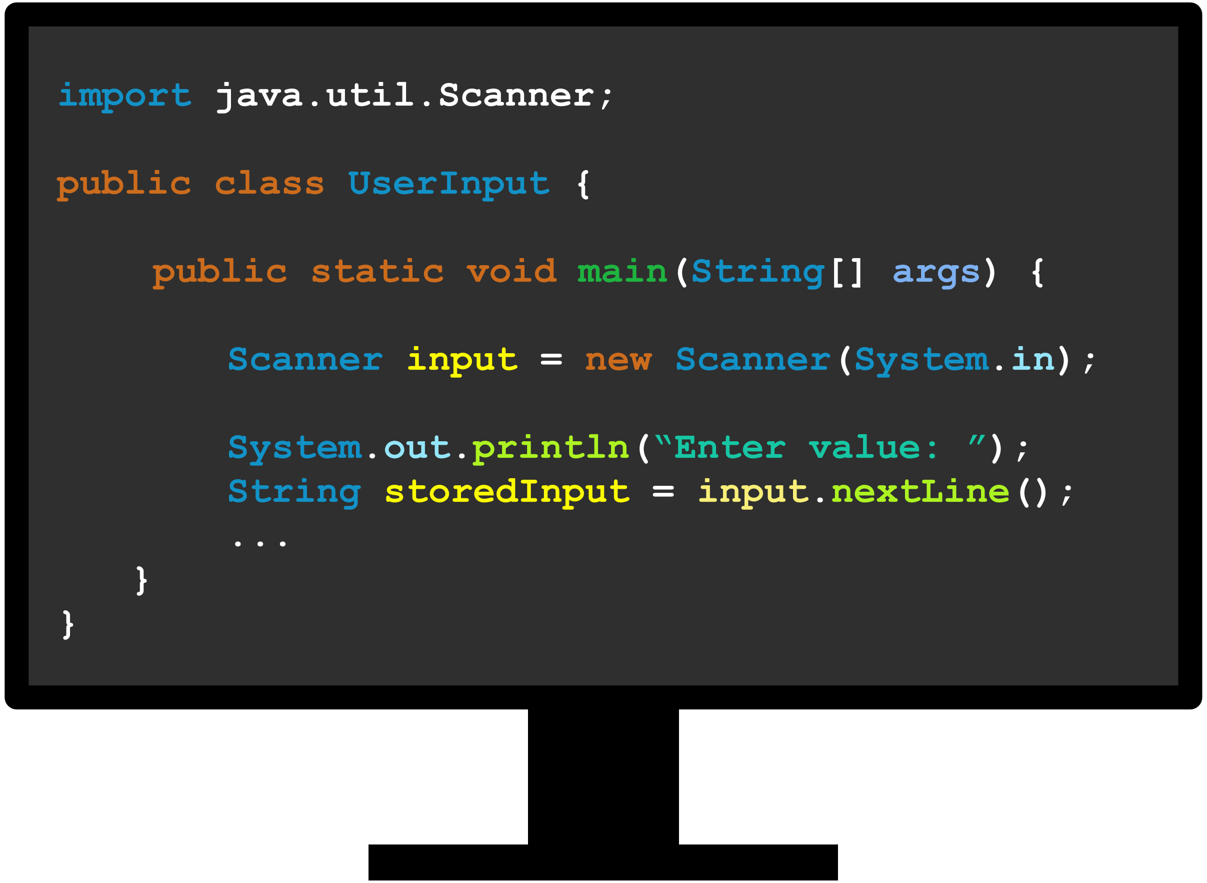
Order of Precedence - order in which operators in an expression are evaluated

Oracle Documentation:

https://docs.oracle.com/cd/E19253-01/817-6223/chp-typeopexpr-12/index.html

Important to note	
1	`!, +, - (unary)`
2	`*, /, %`
3	`+, -`
4	`<, <=, >=, >`
5	`==, !=`
6	`&&`
7	`||`
8	`= (assignment)`

Read From the Console



```
import java.util.Scanner;

public class UserInput {

    public static void main(String[] args) {

        Scanner input = new Scanner(System.in);

        System.out.println(“Enter value: ”);
        String storedInput = input.nextLine();
        ...
    }
}
```

Flow Control

Conditionals: If/Else

- Conditionals are a core part of nearly every programming language.
- Java uses **if / else if/ else** syntax to control the flow of a program

```
if (condition1){
     // this code will execute if condition1 a strict true boolean
}
else if (condition2){
     // this code will execute if condition2 a strict true boolean
     // and condition1 is a strict false boolean
}
else {
     // this code will execute if neither condition1 or condition2
     // are strict true booleans
}
```

Conditionals: Nested If Statements

- ➔ Nesting if statements can test conditions that are reliant on the state of other conditions

```
if (condition1) {
    if (condition2) {
        // this code will execute if condition1 and condition2 are
        // strict true booleans
    } else {
        // this code will execute if condition1 is a strict true
        // boolean, and condition2 is a strict false boolean
    }
} else {
    // this code will execute if condition1 is a strict false boolean,
    // but has no relationship to condition2
}
```

Conditionals: Logical Operators

Logical operators can be used to check the conditions on primitive data types

```
<  ➙ Less than
>  ➙ Greater than
<= ➙ Less than or equal to
>= ➙ Greater than or equal to
== ➙ Equal to (NOTE: This compares memory locations)
!= ➙ Not equal to
!  ➙ Not (reverses a boolean)
&& ➙ And (True if Both booleans are true)
|| ➙ Or (True if at least one boolean is true)
^  ➙ XOR (True if one boolean is true and the other false)
```

Important note: Strings are objects, not primitives, these operators with not work as properly on Strings.

Conditionals: Switch

- Switch statements are a more compact syntax for conditionals.
- Switch statements can be used to direct the flow of a program based on the value of an int, char or (as of Java 7) an enum value

```
switch (condition){
      case condition1:
            // code
            break;
      case condition2:
            // code
            break;
}
```

- Note the break statements. Without them, the switch will execute all code following the matching condition

Conditionals: Switch

➔ A default case can be added to a switch expression which will be executed when no case is matched

```
switch (condition){
      case condition1:
            // code
            break;
      case condition2:
            // code
            break;
     default:
           // This code will run if neither
           // condition1 or condition2 is met
           break;
}
```

Loops: While Loop

➔ **Loops** executes block of code a number of times until condition is met
➔ **While loop** repeat a code block until condition is a strict boolean false

```
int counter = 1;
while (counter < 10){
      System.out.println(counter);
      counter++;
}
```

➔ Above, the loop will print numbers from 1 to 9
➔ While loops have no internal means of keeping track of the number of loops
➔ Developer must be careful to ensure that infinite loops don’t occur

Loops: While Loop

➔ The conditional in a while loop does not have to be a counter

```
boolean condition1 = true;
while (condition1){
       // code
       if (condition2) {
           condition1 = false;
       }
}
```

➔ Above, loop executes code block indefinitely because it is looping on a true boolean
➔ Once **condition2** is met, will swap the loop boolean **condition1** to false
➔ Last loop is executed

Loops: Do While

➔ The do/while loop similar to while loop, except it executes its code block at least once before checking the condition

```
boolean condition = false;
do {
     // code
} while (condition);
```

➔ Above, even though condition is immediately set to false, the loop will execute once

Loops: For Loop

- ➔ For loops are more complex loop that have terminating and increment conditions built in
- ➔ Consist of an ***initialization block***, a ***condition block,*** and an ***increment block***

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

- ➔ Above, an int i is set to zero, it is incremented by one for each iteration of the loop, and once i is greater than ten, the loop terminates.
- ➔ Initialized variable is block-scoped to the for loop, i cannot be accessed outside the loop

Loops: For Loop

➔ Standard form of for loop is the most common, but there are a few variations possible

```
boolean condition = true;
for (int i = 1; condition; i *= 5){
     if (i % 3 == 0){
          condition = false;
     }
     System.out.println(i);
}
```

➔ Above, an int i is set to one, it is incremented by multiplying it by five each time through the loop, and the loop is broken through some outside condition

➔ Some specific use cases for unusual for loops like this, but most standard cases require a loop initialized to 1 that increments by one, and ends when a number is reached

Loops: Nested Loops

➔ Nesting loops can be used to generate two dimensional arrays or tables

```
for (int length = 1; length < 4; length++){
    for (int width = 1; width < 4; width++){
        area = length * width;
        System.out.println("area: " + area);
    }
    System.out.println("");
}
```

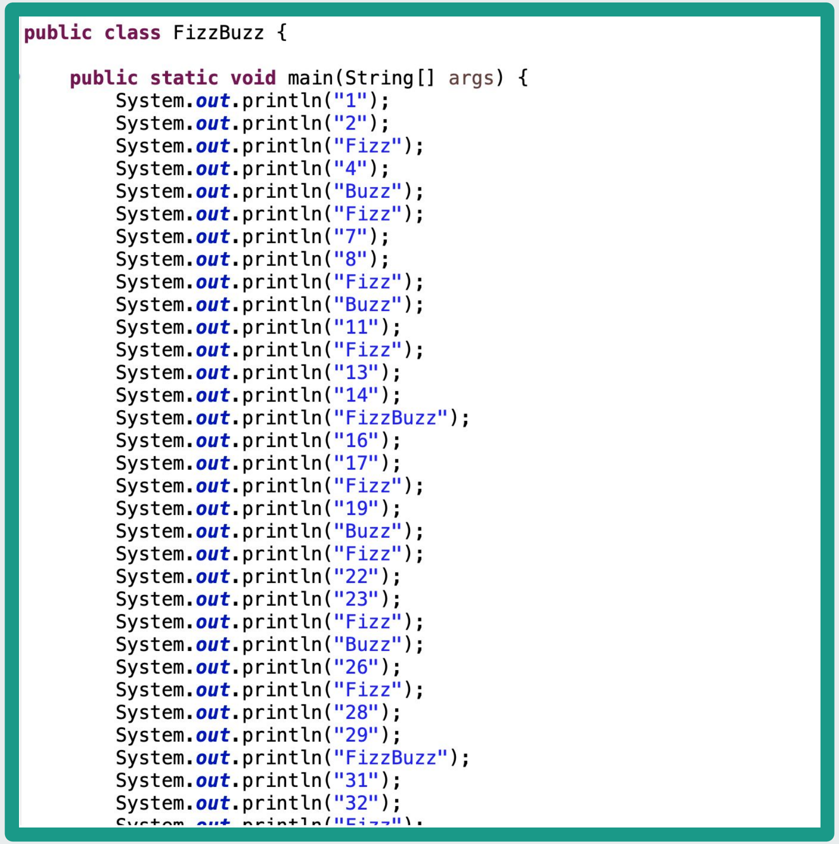
➔ Code above will print a grid that labels the area of rectangles of the given length and width

➔ Nesting loops is significantly more memory and processor intensive than a single for loop, so be careful with implementations that require them

Loops: Break and Continue

- ➔ **Loop-and-a-half** conditions implemented for more precise control of code execution within a loop
- ➔ **Break**
  - ◆ Will immediately end all repetitions of a loop and return to normal flow of a program
- ➔ **Continue**
  - ◆ Will end the current iteration of a loop, and move on to the next iteration
  - ◆ In ***for loop***, will still trigger the increment block
- ➔ **Return**
  - ◆ Within a method, return statement can be used to end a loop and return a value; similar to a break statement.
  - ◆ Will end any resources associated with a method, including any further iterations of loop

White Board Exercise



```java
public class FizzBuzz {

    public static void main(String[] args) {
        System.out.println("1");
        System.out.println("2");
        System.out.println("Fizz");
        System.out.println("4");
        System.out.println("Buzz");
        System.out.println("Fizz");
        System.out.println("7");
        System.out.println("8");
        System.out.println("Fizz");
        System.out.println("Buzz");
        System.out.println("11");
        System.out.println("Fizz");
        System.out.println("13");
        System.out.println("14");
        System.out.println("FizzBuzz");
        System.out.println("16");
        System.out.println("17");
        System.out.println("Fizz");
        System.out.println("19");
        System.out.println("Buzz");
        System.out.println("Fizz");
        System.out.println("22");
        System.out.println("23");
        System.out.println("Fizz");
        System.out.println("Buzz");
        System.out.println("26");
        System.out.println("Fizz");
        System.out.println("28");
        System.out.println("29");
        System.out.println("FizzBuzz");
        System.out.println("31");
        System.out.println("32");
        System.out.println("Fizz");
```

Create a method that follows the following rules that :

The method should print out a list of length n, with each index i following these rules

If a number n is divisible by 3, print “Fizz”

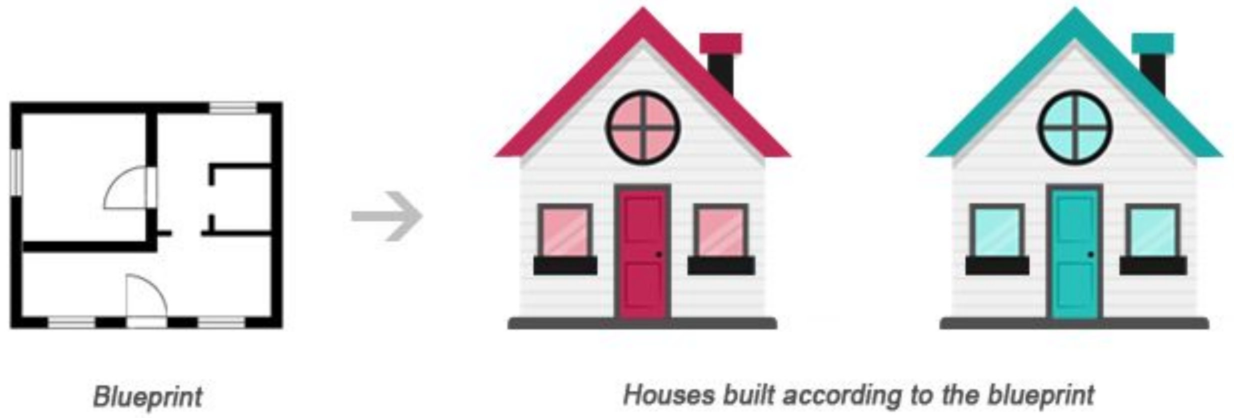
If a number n is divisible by 5, print “Buzz”

If a number n is divisible by 3 and 5, print “Fizzbuzz”

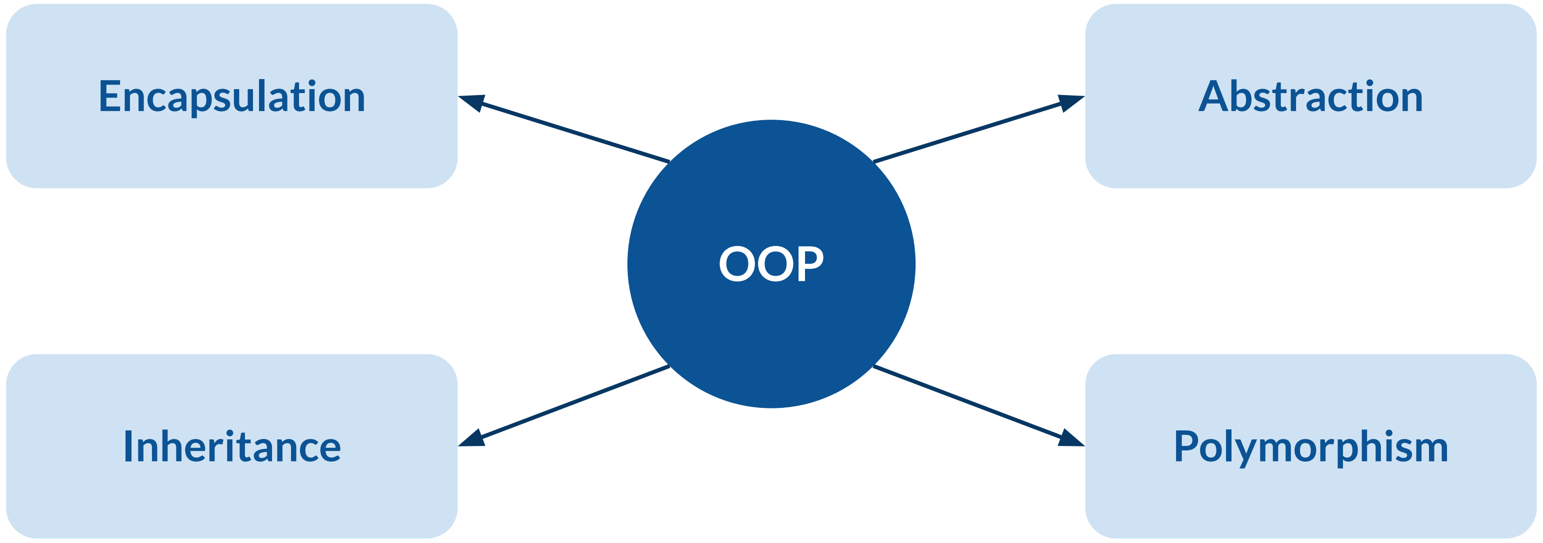
Bonus :

If a number is prime, do not print it.

Classes & Objects

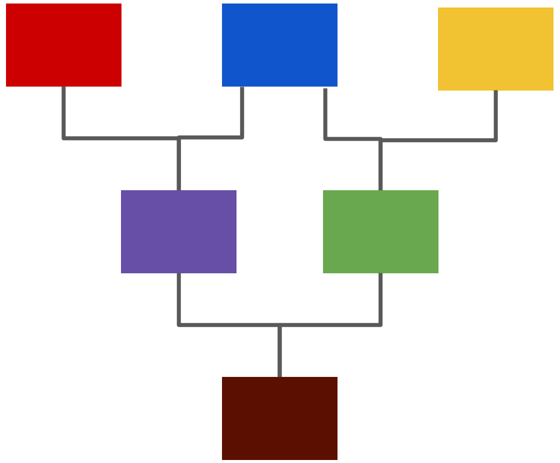


Object Oriented Programming (OOP)

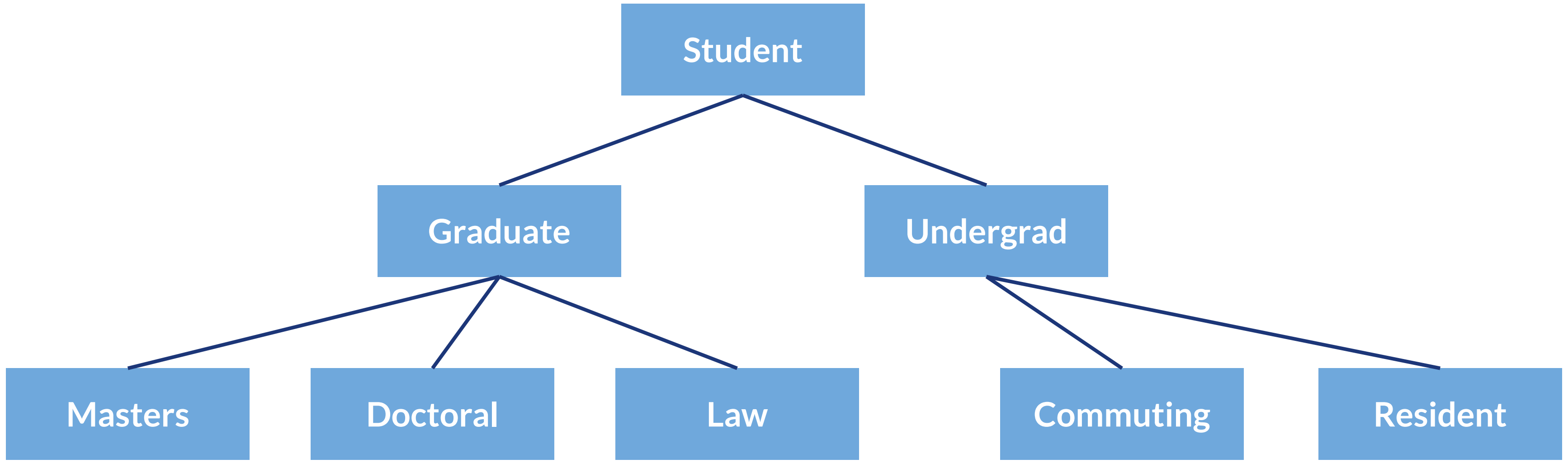


OOP - Inheritance

- ➔ ***Inheritance*** is a mechanism in OOP to design two or more entities that are different but share many common features
- ➔ Features common to all classes are defined in the ***superclass***
- ➔ The classes that inherit common features from the superclass are called ***subclasses***
- ➔ We also call the superclass an ***ancestor*** and the subclass a ***descendant***

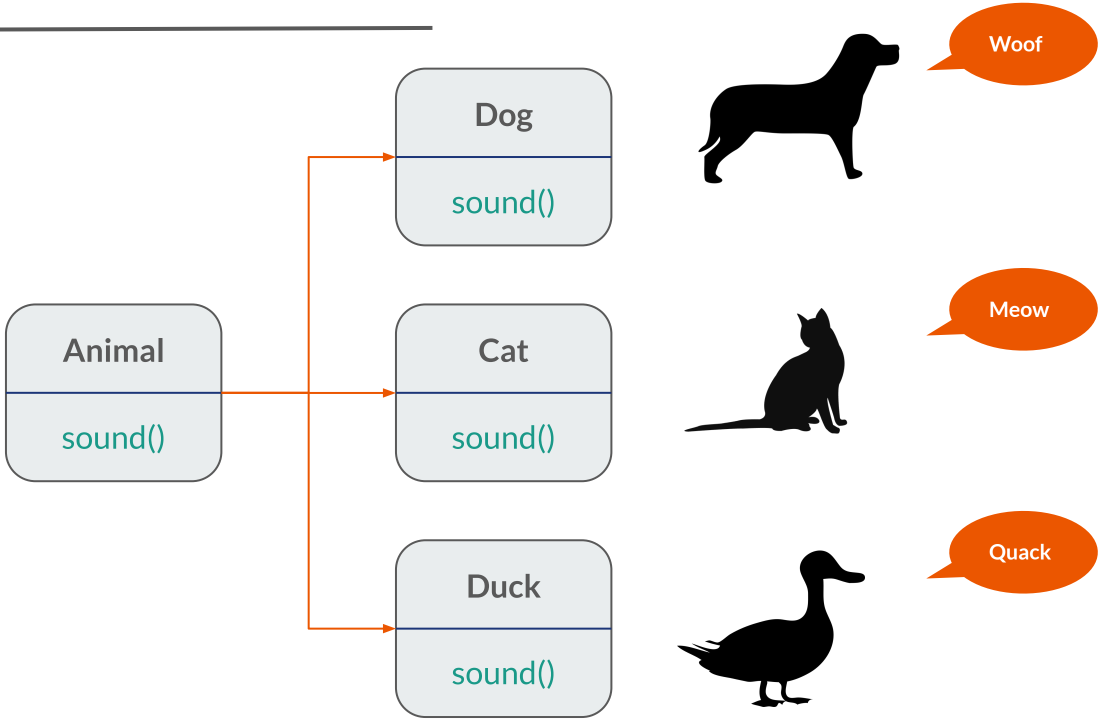


OOP - Inheritance



OOP - Polymorphism

- ➔ ***Polymorphism*** is a mechanism in OOP where one element of code can have many forms
  - ◆ Polymorphism can be implemented in such as **objects** and **classes**, and **class methods**

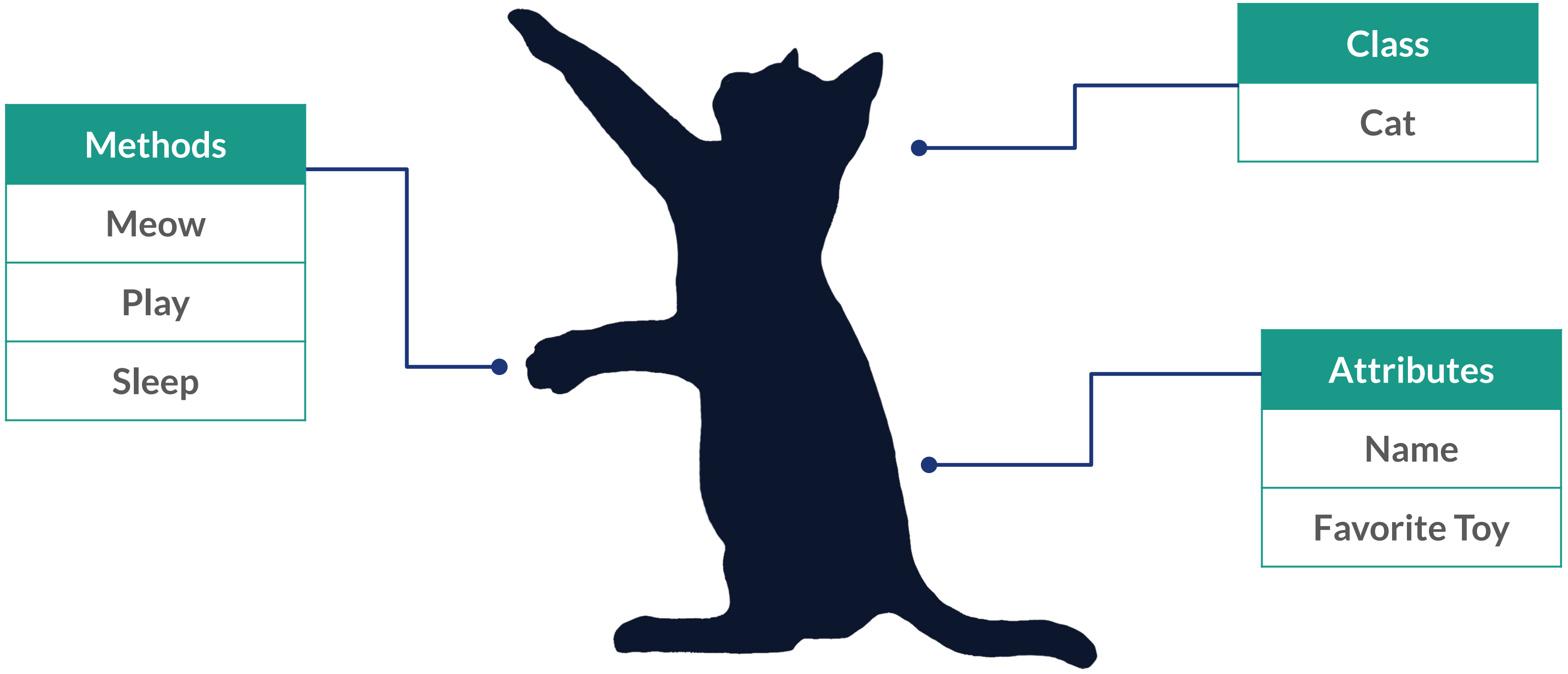


OOP - Abstraction and Encapsulation

Abstraction as a concept of OOP enforces "***data hiding***". That is, only relevant code is displayed, so that code is layered.

Encapsulation is a "***data grouping***". Think of this as a protective shield around code. An example would be grouping functions together in class.

Classes


Methods
Meow
Play
Sleep
Class
Cat
Attributes
Name
Favorite Toy

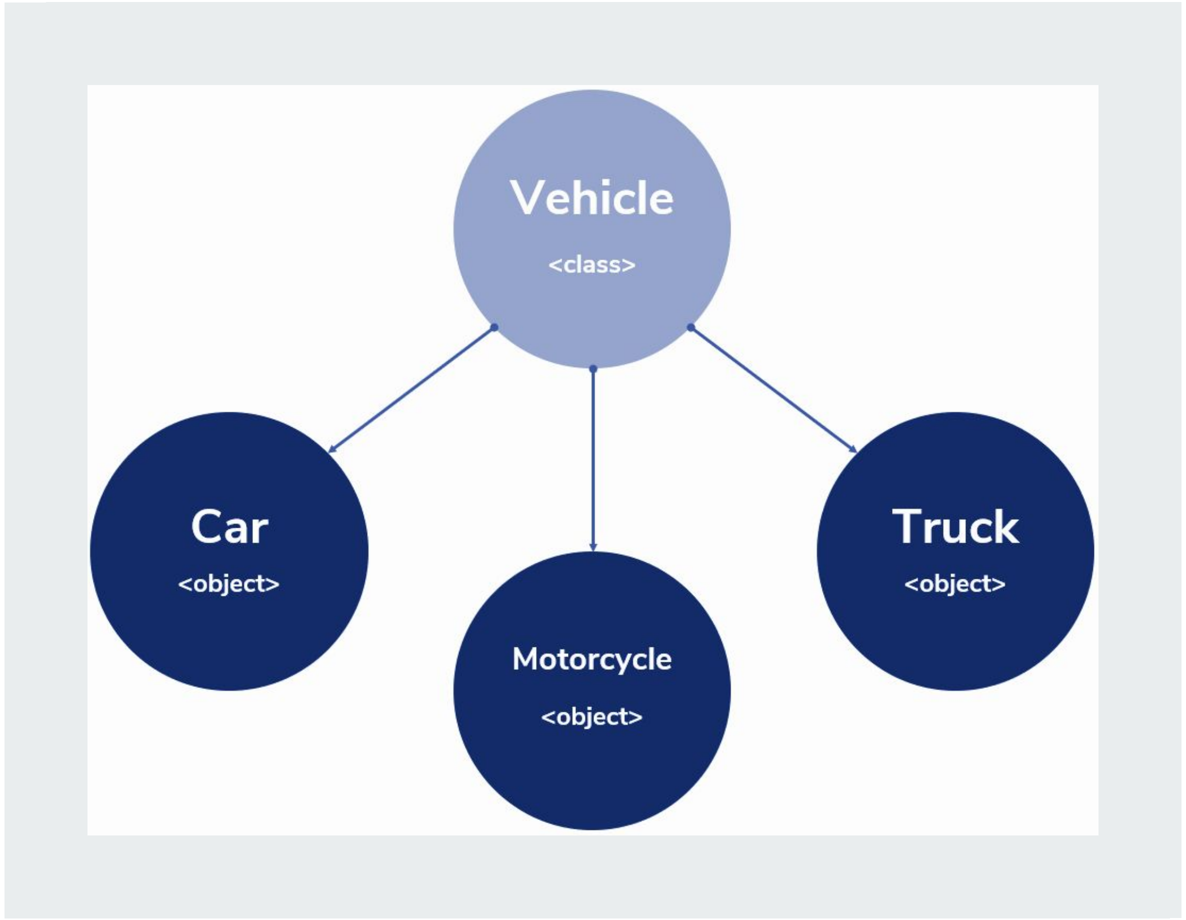
Classes and Objects

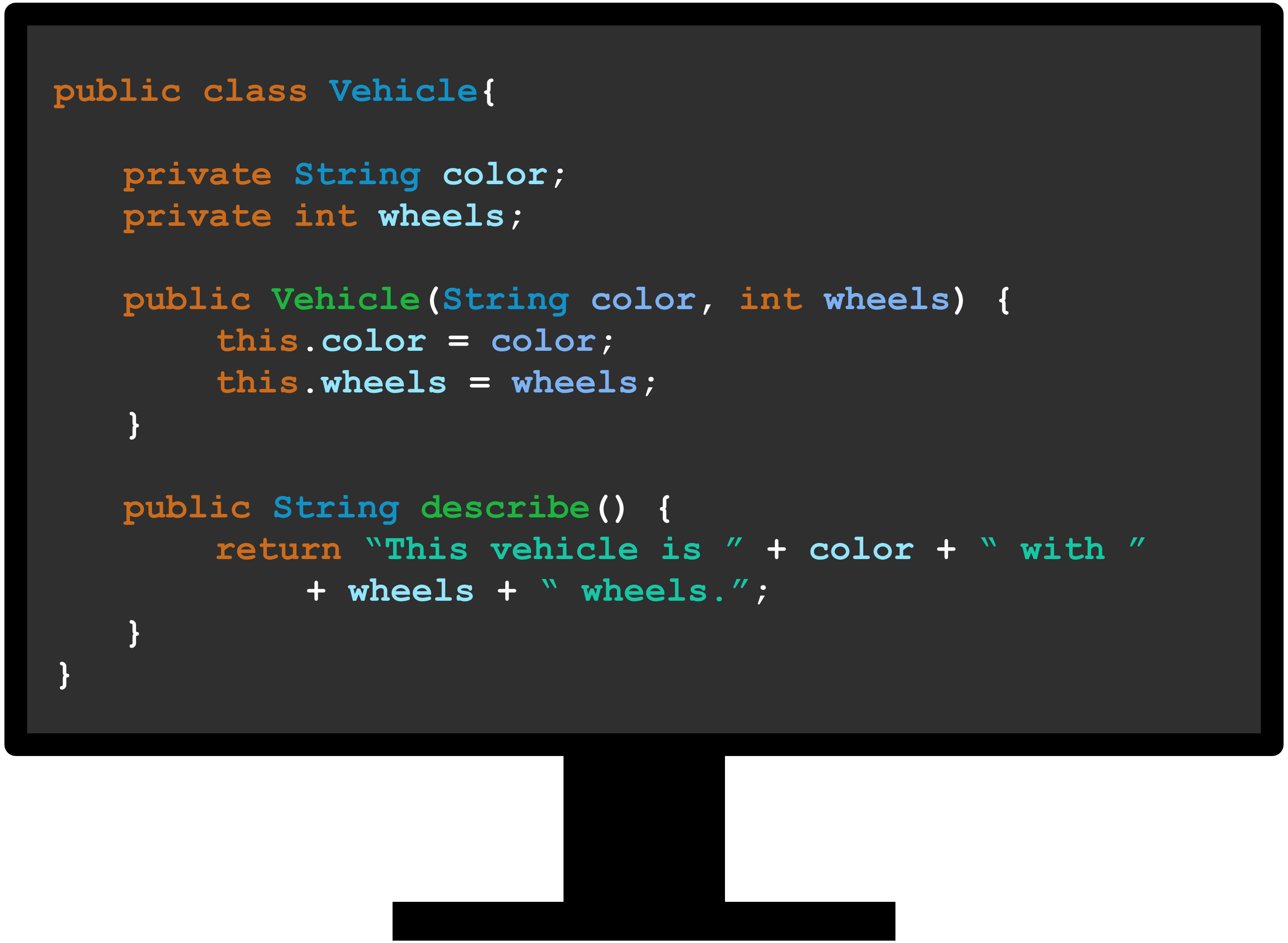
Object-oriented programs use **objects**.

An ***object*** is a thing, both tangible and intangible.

To create an object inside the computer program, we must provide a definition for objects—how they behave and what kinds of information they maintain —called a ***class***.

An object is called an ***instance*** of a class.


Vehicle
<class>
Car
<object>
Motorcycle
<object>
Truck
<object>



```java
public class Vehicle{

    private String color;
    private int wheels;

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

    public String describe() {
        return “This vehicle is ” + color + “ with ”
            + wheels + “ wheels.”;
    }
}
```

Messages and Methods

To instruct a class or an object to perform a task, we send a ***message*** to it.

You can send a message only to the classes and objects that understand the message you sent to them.

A class or an object must possess a matching ***method*** to be able to handle the received message.

Messages and Methods

A method defined for a class is called a ***class method***, and a method defined for an object is called an ***instance method***.

A value we pass to an object when sending a message is called an ***argument*** of the message.

Access Modifiers

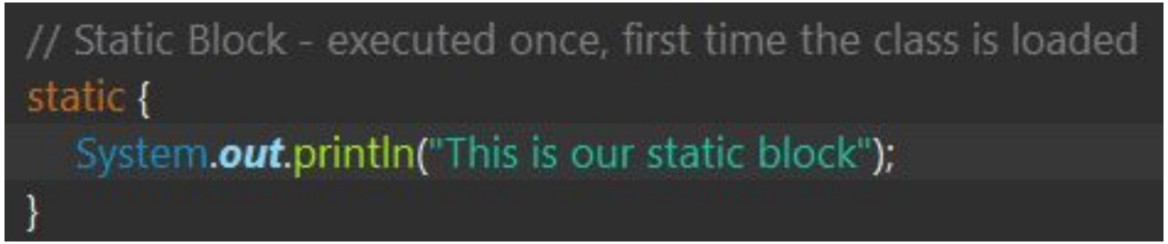
Modifier	Class	Package	Subclass	Global
Public	Allowed	Allowed	Allowed	Allowed
Protected	Allowed	Allowed	Allowed	Denied
Default	Allowed	Allowed	Denied	Denied
Private	Allowed	Denied	Denied	Denied

Static Keyword

- Means the method or attribute is bound to the entire Class
  - No object needs creation to call static methods
  - Static attributes are reflected for all objects of a class
    - E.g. - a count int, that counts all objects of a class is static
- To call a static method
  - Use: *ClassName.methodName();*

Static Keyword

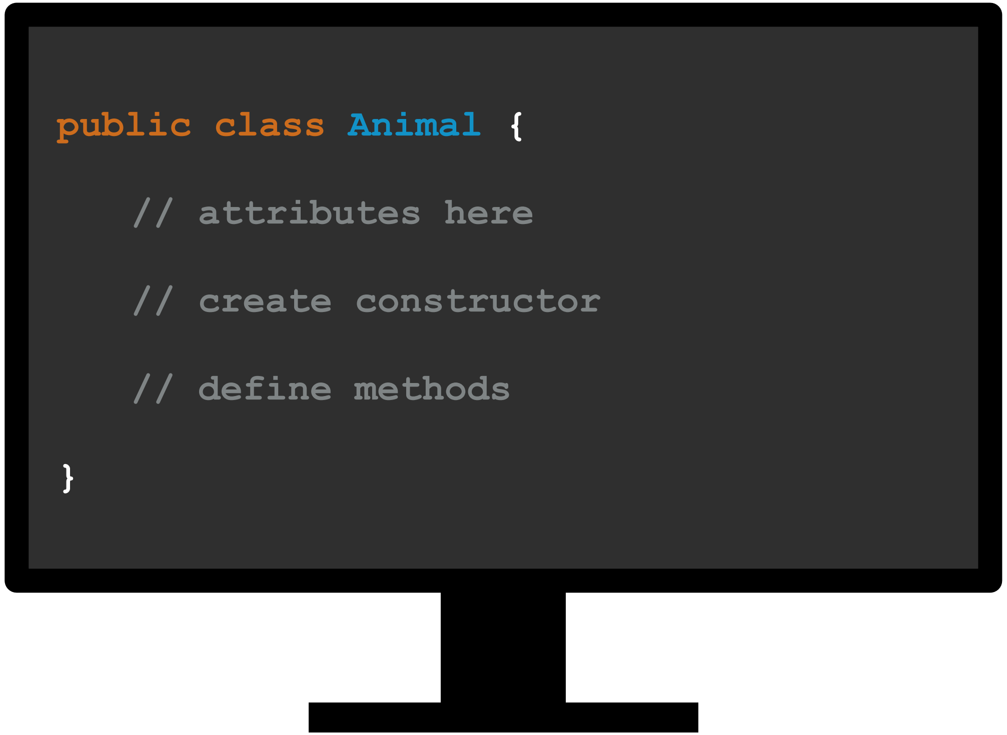
- Static Block
  - Block of code that executes once, when a class is loaded into the program heap.
  - Will execute *BEFORE* a constructor, but only once.



```
// Static Block - executed once, first time the class is loaded
static {
    System.out.println("This is our static block");
}
```

White Board Exercise

Class Example



```
public class Animal {

    // attributes here

    // create constructor

    // define methods

}
```

White board exercise

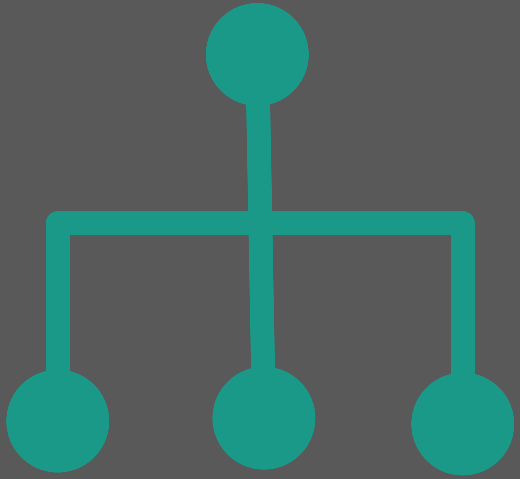
Creating a Class Diagram

Create Class	Class Properties	Child Class	Polymorphism	Explain
Choose a topic and create a class for this, draw it up on the board	Create attributes and methods for this class.	Create a child class that can inherit from this original class. Come up with attributes and methods for this child class.	Create a method that will override one of the methods from the parent.	What is happening in this diagram? Is there encapsulation?

Naming Conventions

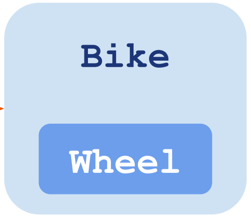
- ➔ *Classes* - should be nouns, in mixed case with the first letter of each internal word capitalized
- ➔ *Interfaces* - should be adjectives, in mixed case with the first letter of each internal word capitalized
- ➔ *Methods* - should be verbs, in mixed case with the first letter of each internal word capitalized
- ➔ *Variables* - should be short yet meaningful. The choice of a variable name should be mnemonic- that is, designed to indicate to the casual observer the intent of its use. Lowercase first letter and camelcased
- ➔ *Final Variables and Enums* - should be all uppercase with words separated by underscores ("_")

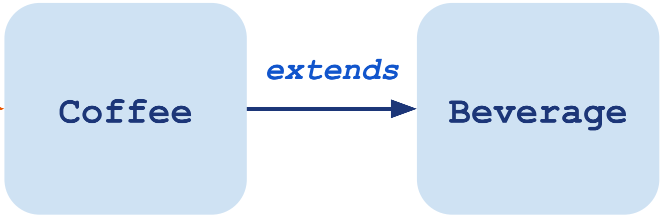
Composition & Inheritance



Has-A vs Is-A Relationship

- **Composition** - a class can contain an instance of another class
    - *Has-A* relationship
        - Ex: a bike ***has a*** wheel
- **Inheritance** - a child class can obtain the properties of its parent class
    - *Is-A* relationship
        - Ex: a coffee ***is a*** beverage
    - Access modifiers determine what attributes/methods are accessible to the child class



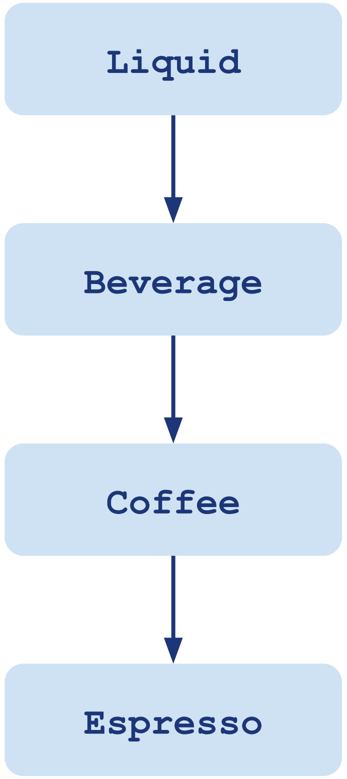


Inheritance Hierarchy

A child class will inherit from its parent class and all the classes its parent inherits from. Because an Espresso is a Liquid, a variable of type Liquid can be assigned an Espresso object. However, a Liquid cannot be an Espresso because a Liquid does not inherit from Espresso.

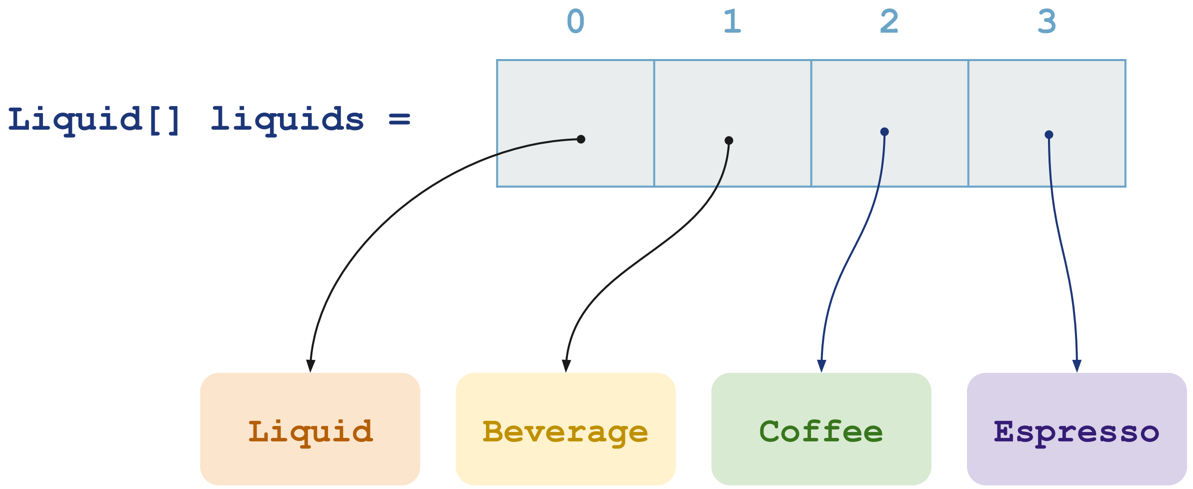
✓
```
Liquid liquid = new Espresso();
```

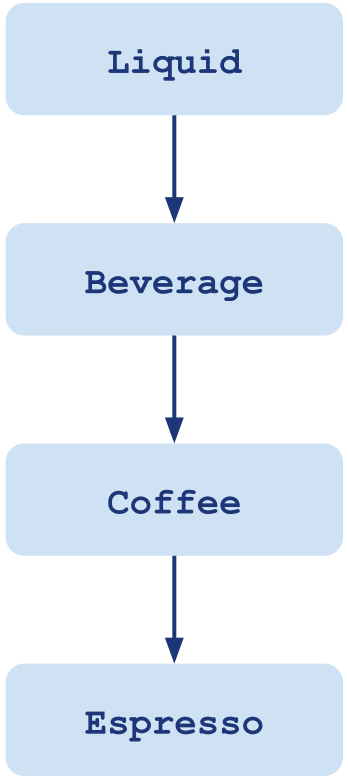
✗
```
Espresso espresso = new Liquid();
```



Inheritance Hierarchy

An array of Liquids can take in any objects that inherit from it. So Beverage, Coffee, and Espresso objects can be placed in this array.





Final Classes and Inheritance

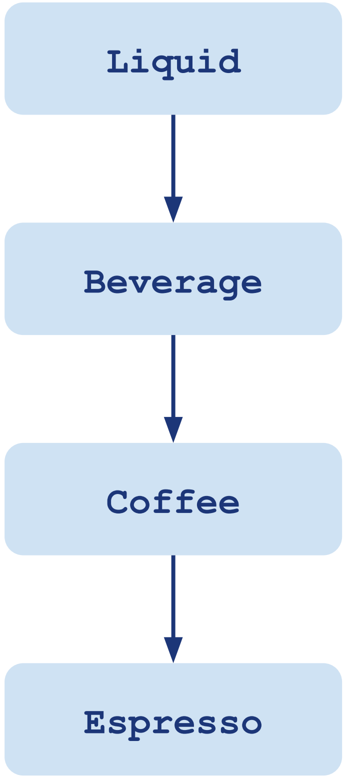
A class declared final cannot be extended by another class. Classes like String are final and use this implementation so no other classes can inherit and use their functionality.

```java
public final class Espresso {
    ...
}
```

✗

```java
public class Latte extends Espresso {
    ...
}
```

Will cause error

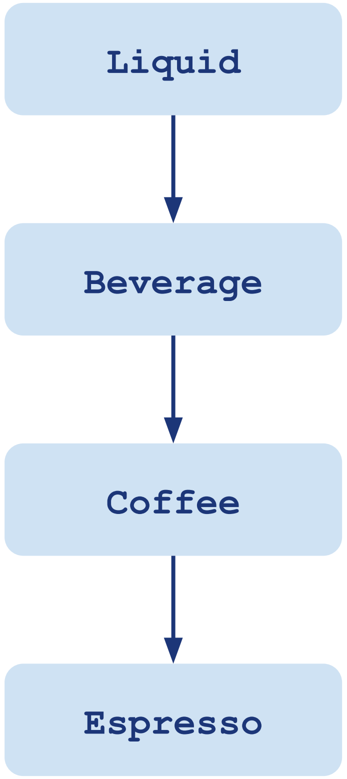


Polymorphism

Polymorphism is the ability for an object to take on different forms. Like how an Espresso is an Espresso, but also a Coffee.

```java
Espresso espresso = new Espresso();
```

```java
Coffee espresso = new Espresso();
```



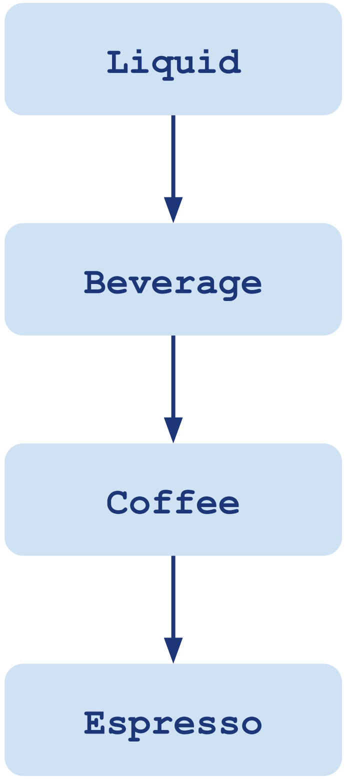
Polymorphism: Methods

RunTime Polymorphism is when **method overriding** is used to redefine a method with the same method signature from a parent class in your child class. Also known as ***dynamic polymorphism***.

```
Espresso espresso = new Espresso();
espresso.whatAmI(); // prints: I am an
                    // espresso
```

```
Coffee espresso = new Espresso();
espresso.whatAmI(); // prints: I am an
                    // espresso
```

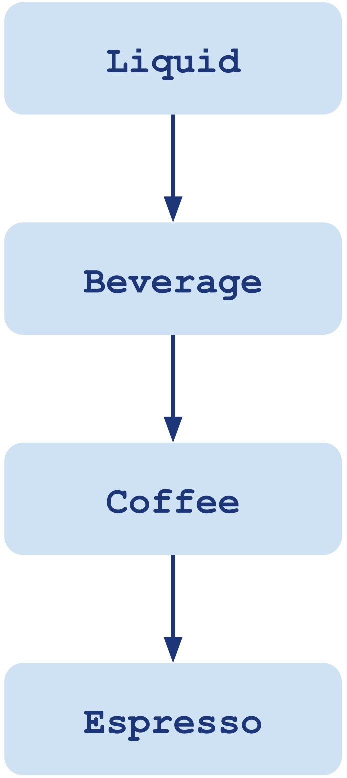
Note: Can't override static methods.



Polymorphism: Methods

Only time you won't be able to override a method is if it is **final** or you try to override a **static** method.

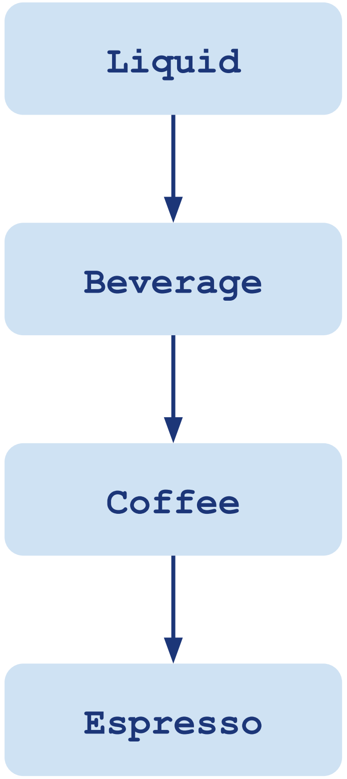
```
public class Espresso {
    // methods cannot be overridden
    public final void whatAmI() {
        ...
    }
    public static void hello() {
        ...
    }
}
```



Polymorphism: Methods

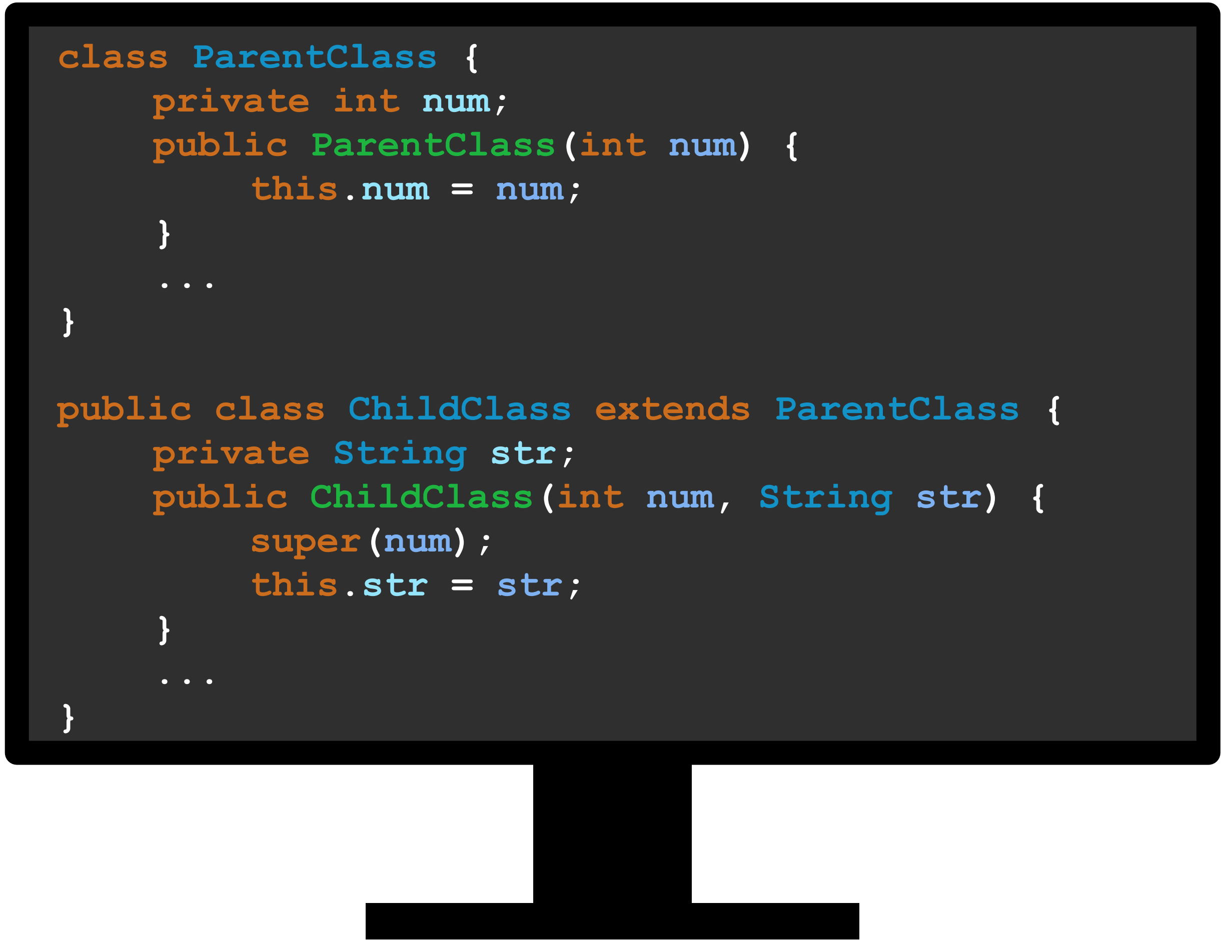
Compile Time Polymorphism is when method overloading is used to define multiple methods with the same name, but different parameters. Also known as ***static polymorphism***.

```
Espresso espresso = new Espresso();

espresso.whatAmI();       // prints: I am an
                          // espresso

espresso.whatAmI("Sam"); // prints: I am an
                         // espresso prepared
                         // by Sam

espresso.whatAmI(3);     // prints: I am 3
                         // espressos
```



Super Keyword

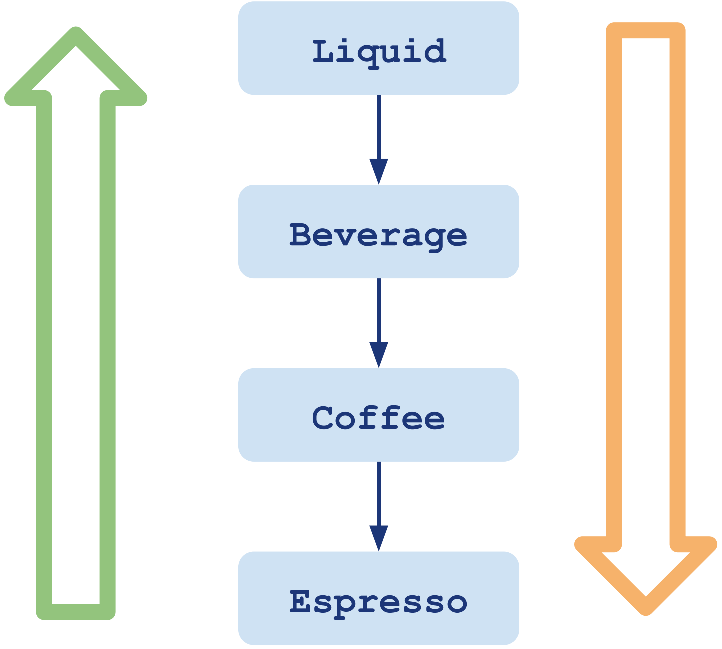
- ➔ The super keyword references the superclass of a class
- ➔ Super can be used to
    - ◆ call constructor of the parent
    - ◆ access data members of parent class



```java
class ParentClass {
     private int num;
     public ParentClass(int num) {
          this.num = num;
     }
     ...
}

public class ChildClass extends ParentClass {
     private String str;
     public ChildClass(int num, String str) {
          super(num);
          this.str = str;
     }
     ...
}
```

Casting Between Types

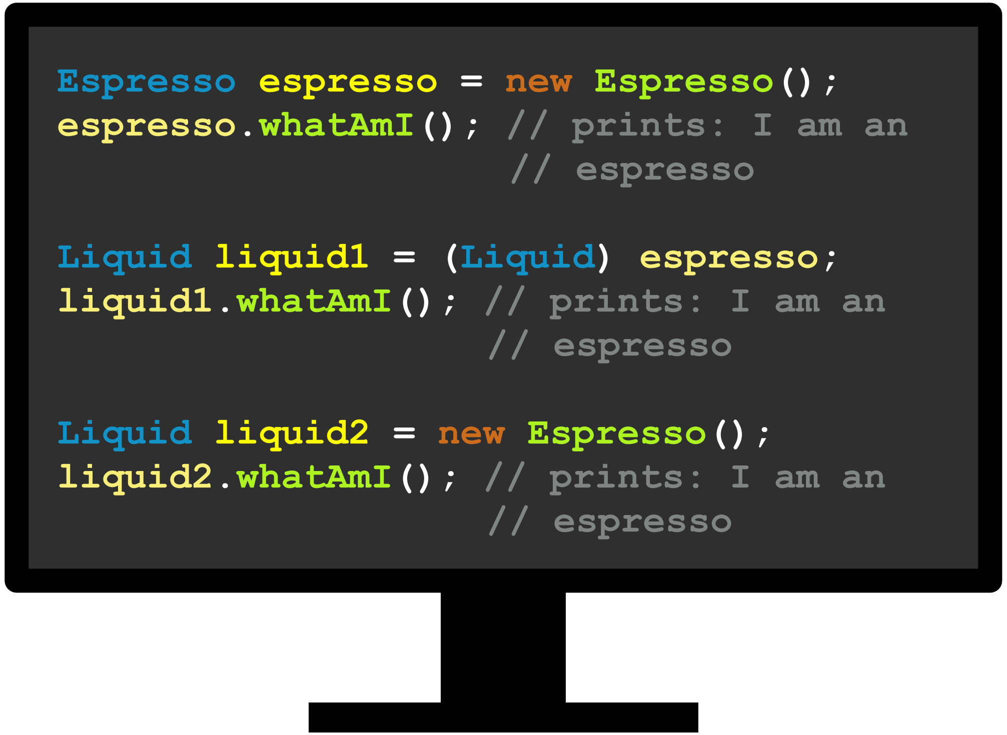
We **Upcast** to convert the type from that of a child class to the type of its parent or any of the classes it inherits from along the chain of inheritance.



We **Downcast** by converting the type from a parent class to a child class or any class that inherits from the original parent.

UpCasting

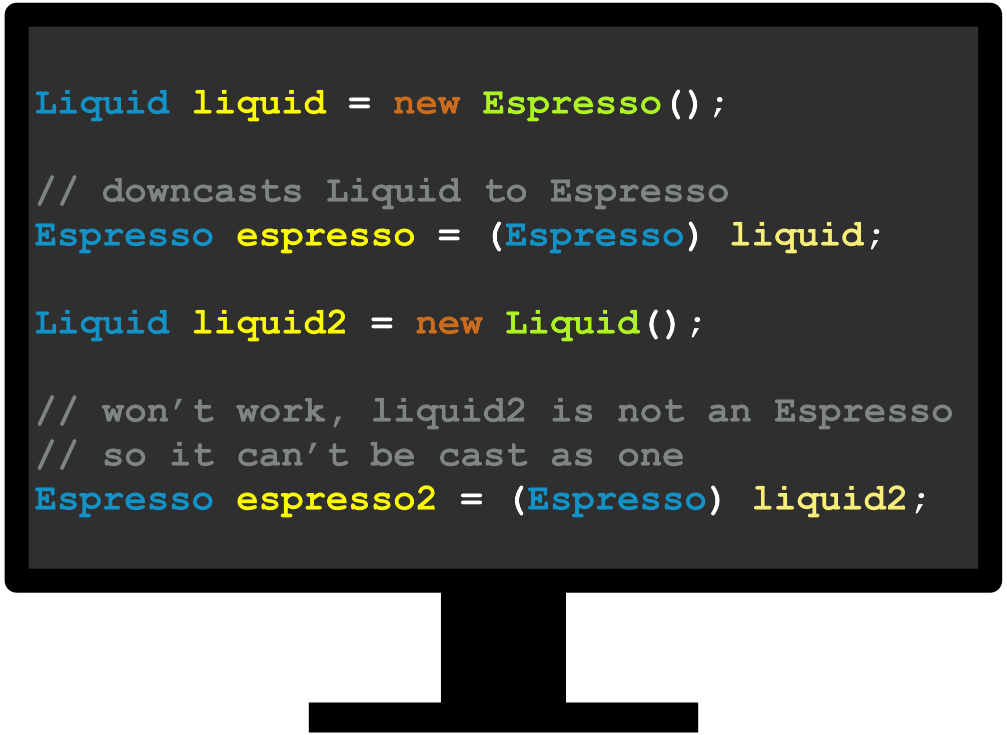
- Can cast by...
  - Explicitly casting with parenthesis and type specified
  - Initializing the object with new keyword
- Why upcast?
  - Want to write code that deals with only supertype



```
Espresso espresso = new Espresso();
espresso.whatAmI(); // prints: I am an
                    // espresso

Liquid liquid1 = (Liquid) espresso;
liquid1.whatAmI(); // prints: I am an
                   // espresso

Liquid liquid2 = new Espresso();
liquid2.whatAmI(); // prints: I am an
                   // espresso
```

DownCasting

- ➔ Can cast by...
  - ◆ Explicitly casting with parenthesis and type specified
- ➔ Why downcast?
  - ◆ Want to access specific behaviors of a subtype

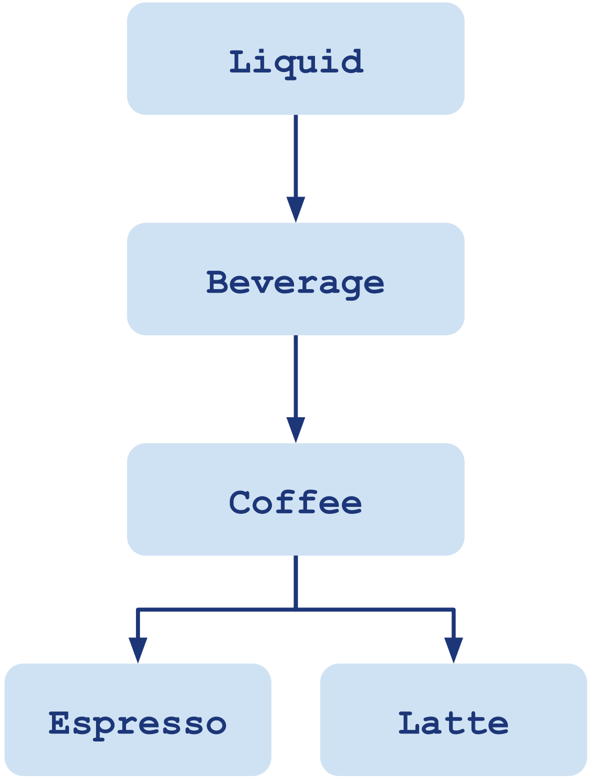


```
Liquid liquid = new Espresso();

// downcasts Liquid to Espresso
Espresso espresso = (Espresso) liquid;

Liquid liquid2 = new Liquid();

// won't work, liquid2 is not an Espresso
// so it can't be cast as one
Espresso espresso2 = (Espresso) liquid2;
```

Instance of an Object

The **instanceof** keyword checks if an object is of a given type and returns back true or false. Checks ***is-a relationship***.

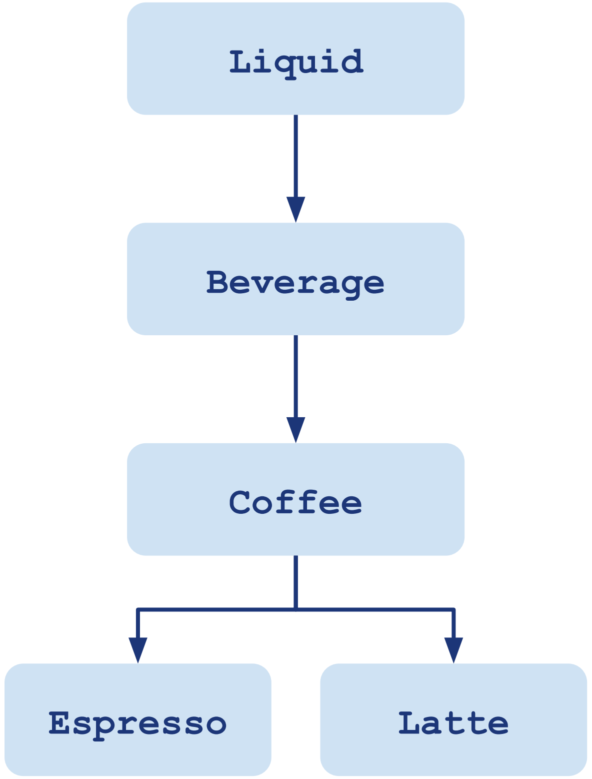
```
Espresso expr = new Espresso();

if ( expr instanceof Espresso ){ // will print
     System.out.println("expr is an Espresso");
}
if ( expr instanceof Liquid ){ // will print
     System.out.println("expr is a Liquid");
}
if ( expr instanceof Latte ){ // compile error
     System.out.println("expr is a Latte");
}
```



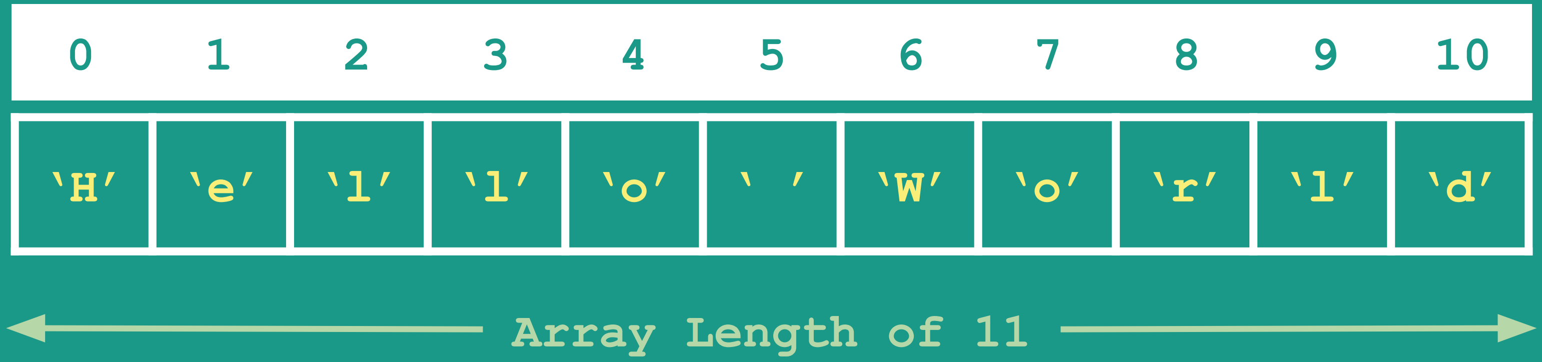
Instance of an Object

The instanceof keyword will return false if object not of the type or if the object is null.

```
Coffee cof1 = new Coffee();
Coffee cof2 = null;

if ( cof1 instanceof Espresso ){ // won't print
     System.out.println("expr is an Espresso");
}
if ( cof2 instanceof Coffee ){ // won't print
     System.out.println("expr is a Liquid");
}
```



Arrays

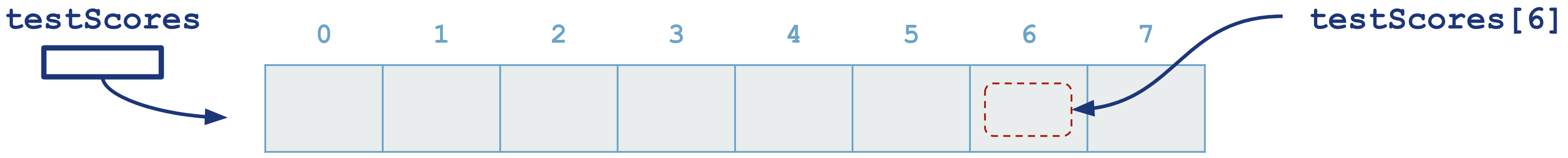


Arrays

```
<data type>[] <variable>;
<variable> = new <data type> [size];
```

```
double[] testScores;
testScores = new double[8];
```

- ➔ An **array** is a collection/group of the same variable types
- ➔ Like an object, must be declared, then have space allocate for it
- ➔ Once the size of an array is set, cannot be changed
- ➔ Each element in array found by its index



Arrays Cont.

- ➔ Arrays can also be declared and initialized at the same time
- ➔ Newly declared arrays with no values initialized will be set to default values
  - ◆ **number types** ➝ zero
  - ◆ **char** ➝ single space
  - ◆ **boolean** ➝ false
- ➔ The **length** from an array is a final variable set when array initialized

`temperatures.length` ➝ `4`

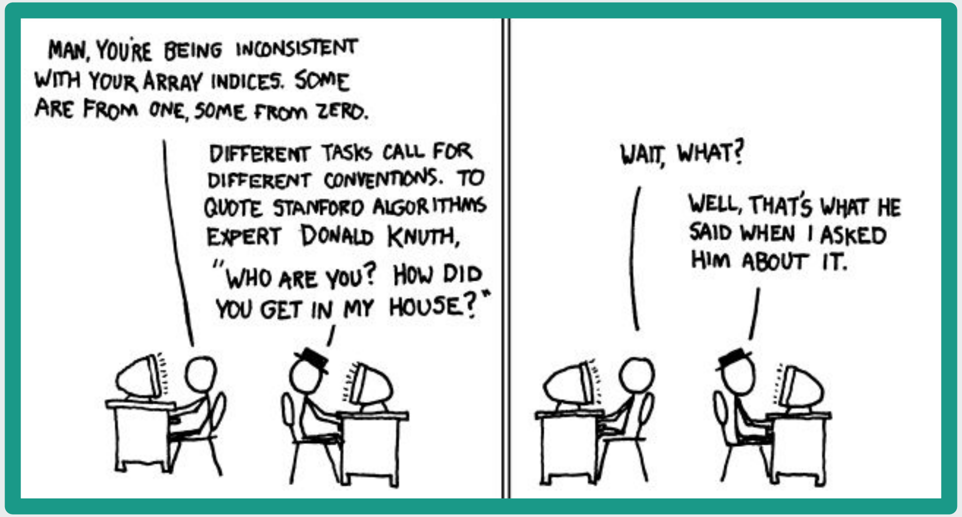
```java
int[] temperatures = {65, 70, 66, 63};
```

0	1	2	3
65	70	66	63

```java
int[] temperatures = new int[4];
temperatures[0] = 65;
```

0	1	2	3
65	0	0	0

White Board Exercise


MAN, YOU'RE BEING INCONSISTENT WITH YOUR ARRAY INDICES. SOME ARE FROM ONE, SOME FROM ZERO.
DIFFERENT TASKS CALL FOR DIFFERENT CONVENTIONS. TO QUOTE STANFORD ALGORITHMS EXPERT DONALD KNUTH, "WHO ARE YOU? HOW DID YOU GET IN MY HOUSE?"
WAIT, WHAT?
WELL, THAT'S WHAT HE SAID WHEN I ASKED HIM ABOUT IT.

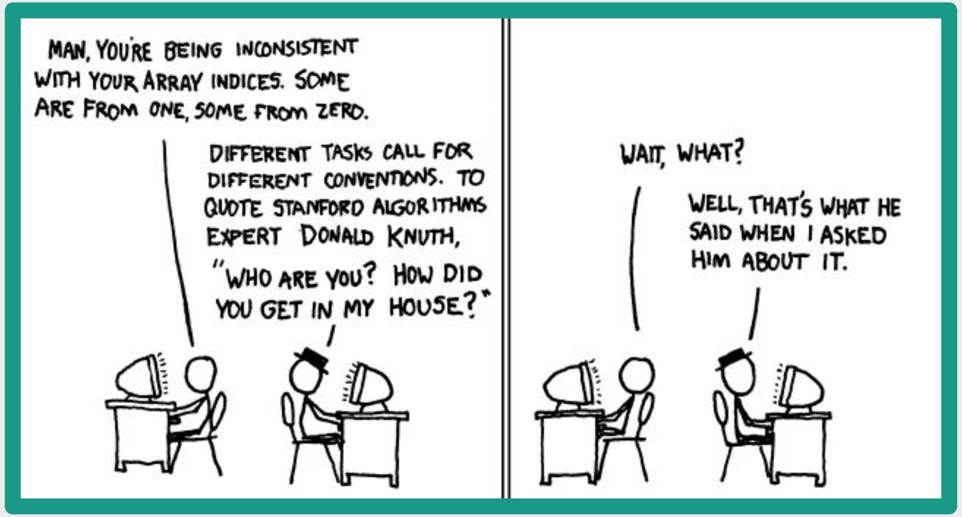
Exercise: Assume you have two arrays of the same data type. You need to check if the two arrays match. The values do not have to be in the same order to match.

✓

```
{1, 3, 5, 0} = {0, 5, 1, 3}
```

✗

```
{ 3, 5, 4, 0} = {0, 5, 1, 3}
```


MAN, YOU'RE BEING INCONSISTENT WITH YOUR ARRAY INDICES. SOME ARE FROM ONE, SOME FROM ZERO.
DIFFERENT TASKS CALL FOR DIFFERENT CONVENTIONS. TO QUOTE STANFORD ALGORITHMS EXPERT DONALD KNUTH, "WHO ARE YOU? HOW DID YOU GET IN MY HOUSE?"
WAIT, WHAT?
WELL, THAT'S WHAT HE SAID WHEN I ASKED HIM ABOUT IT.

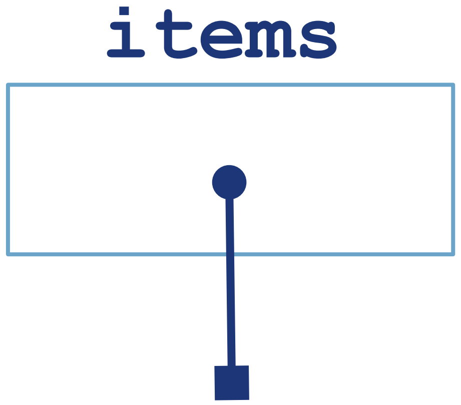
Exercise: There is an integer array with values from 1 to 100, but there is one number missing. Find that missing value.

```
97 is
missing
```

```
{1, 2, 3, 4...96, 98, 99, 100}
```

Arrays of objects

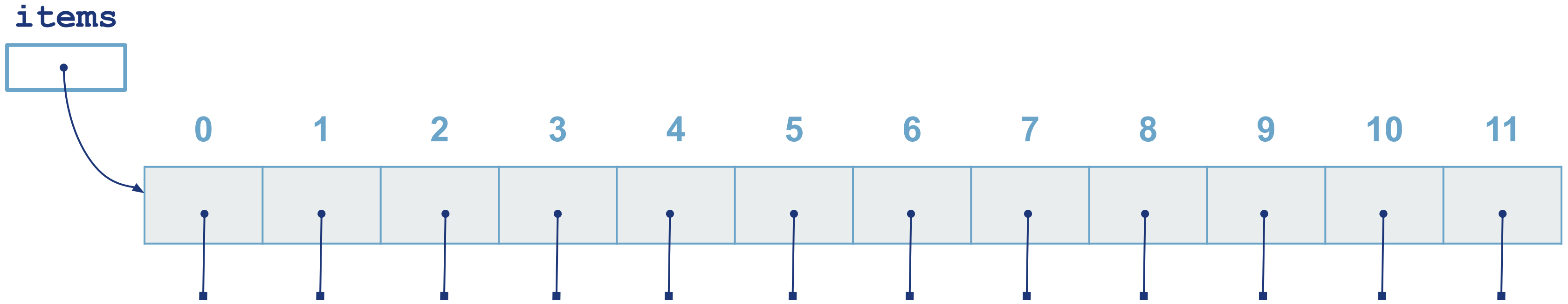
```
Item[] items;

items = new Item[12];

items[0] = new Item();
```



The name items is declared but no allocation has been made for an array.

Arrays of objects

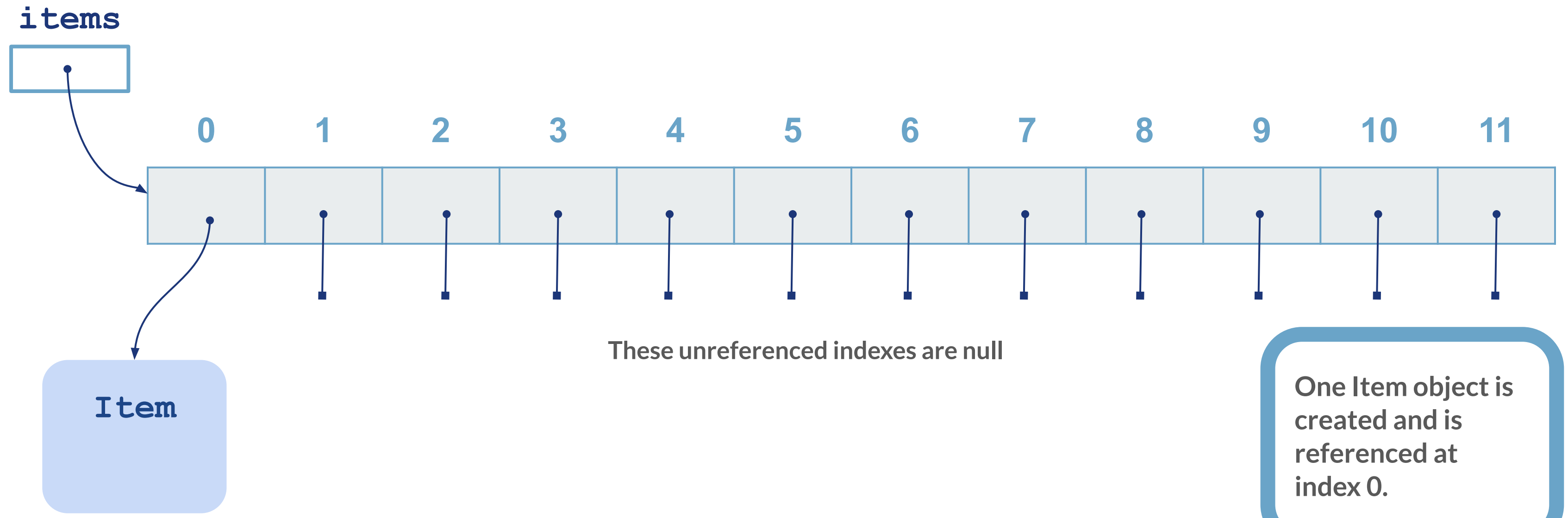
```
Item[] items;

items = new Item[12];

items[0] = new Item();
```



Array allocated for 12 Item objects. No Item objects created yet.

Arrays of objects

```
Item[] items;

items = new Item[12];

items[0] = new Item();
```



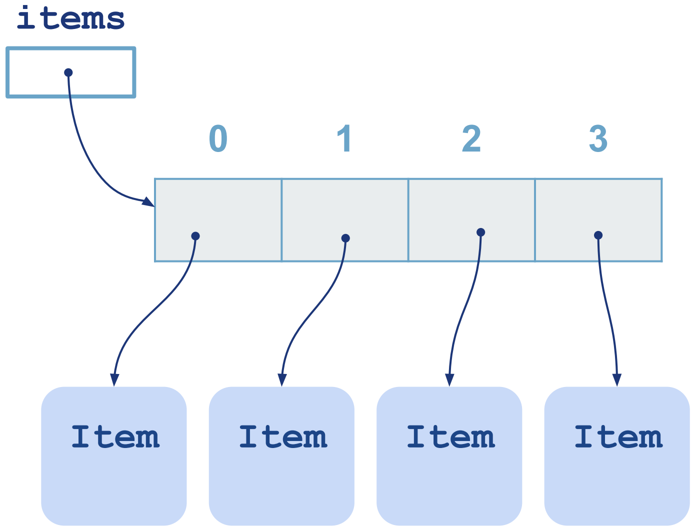
These unreferenced indexes are null

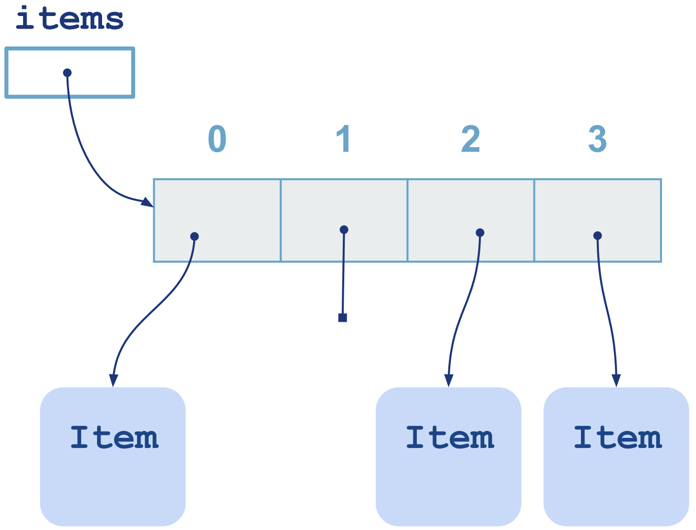
One Item object is created and is referenced at index 0.

Object Deletion

```
Item[] items = new Item[4];
...
items[1] = null;
```

Delete an object by setting it to null.





For Each Loops

```
for(int i = 0; i < items.length; i++) {
      System.out.println(items[i].getName());
}
```

VS

```
for( Item item : items ) {
      System.out.println(item.getName());
}
```

- ➔ Introduced in Java 5
- ➔ Simplifies processing of elements in a collection
- ➔ Constraints:
  - ◆ Read access only (elements can’t be changed)
  - ◆ Can only access a single array at a time
  - ◆ Can’t skip elements
  - ◆ Can’t iterate backwards

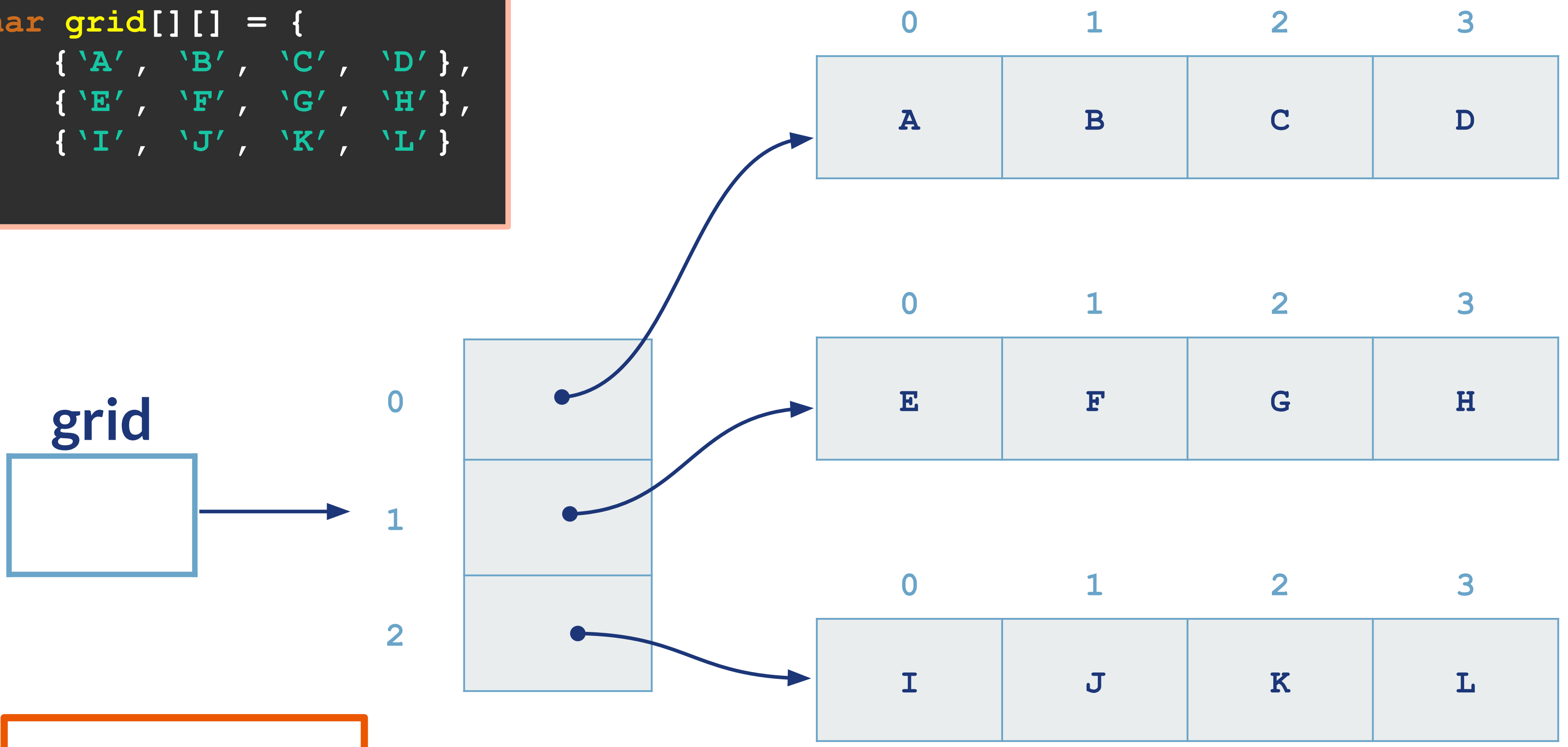
Two-Dimensional Arrays

A **two-dimensional array** is an array that holds a reference to another array in each of its elements.

```
char grid[][] = {
     {‘A’, ‘B’, ‘C’, ‘D’},
     {‘E’, ‘F’, ‘G’, ‘H’},
     {‘I’, ‘J’, ‘K’, ‘L’}
};
```

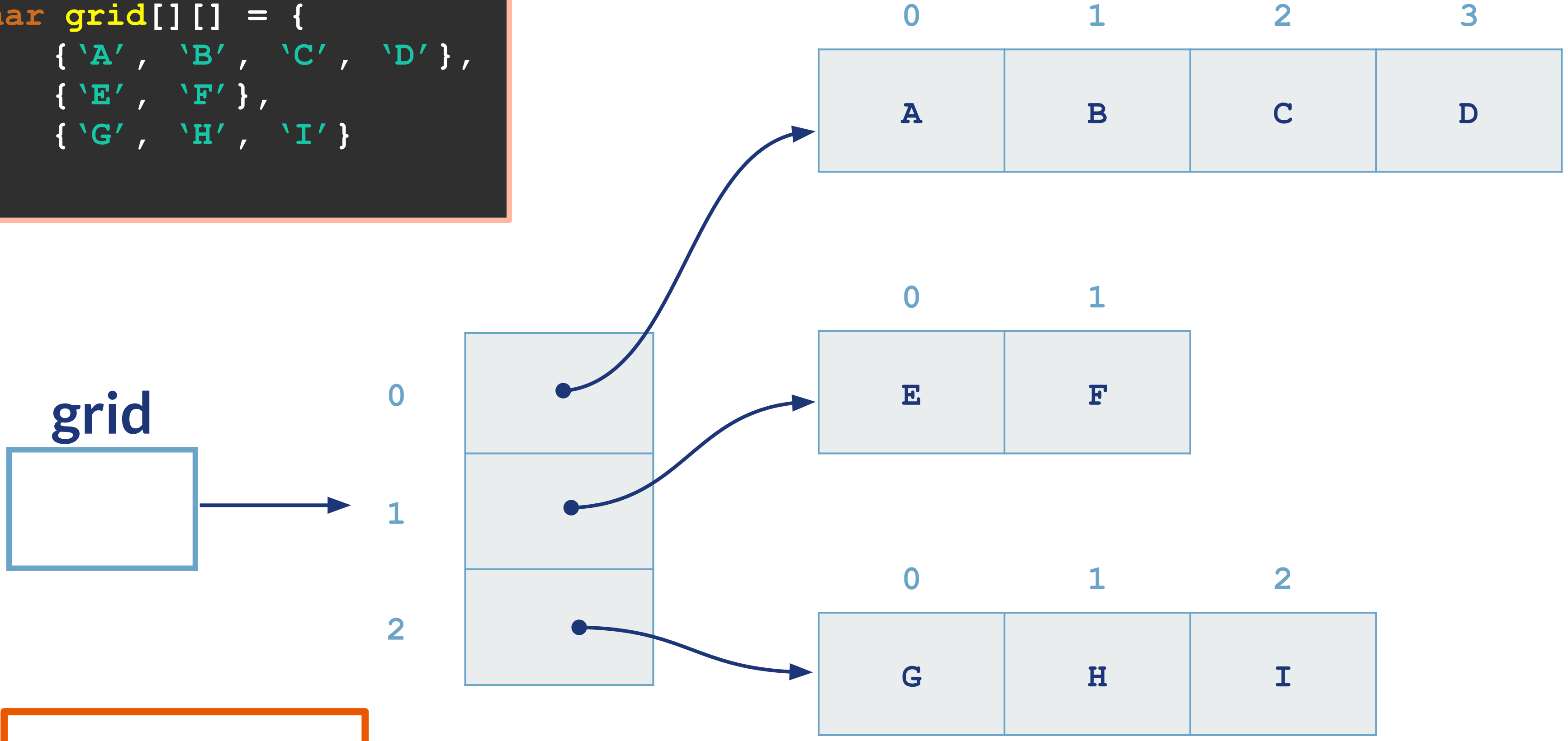
	0	1	2	3
0	A	B	C	D
1	E	F	G	H
2	I	J	K	L

```
char grid[][] = {
     {'A', 'B', 'C', 'D'},
     {'E', 'F', 'G', 'H'},
     {'I', 'J', 'K', 'L'}
};
```



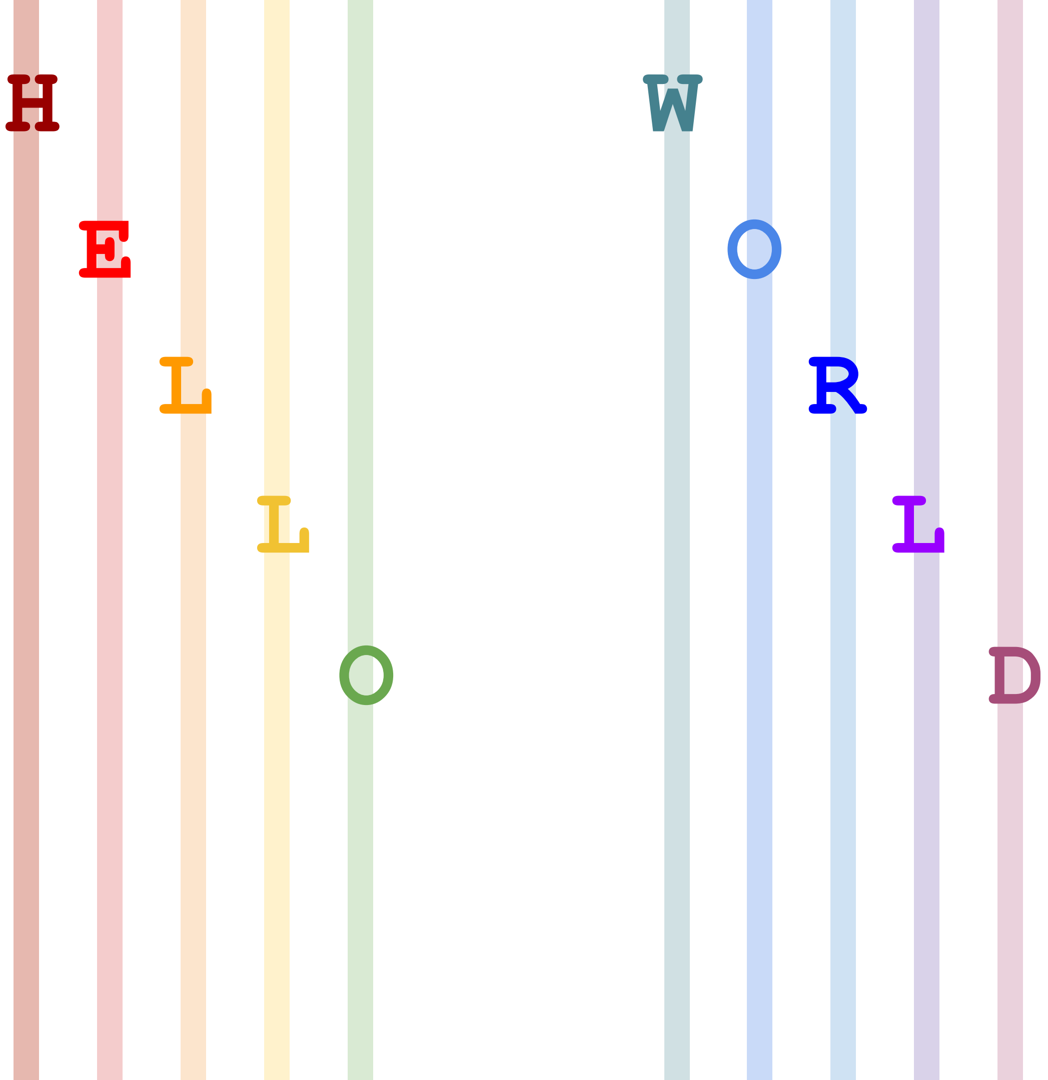
```
arr[1][2] ➞ G
```

```
char grid[][] = {
     {'A', 'B', 'C', 'D'},
     {'E', 'F'},
     {'G', 'H', 'I'}
};
```



```
arr[2][0] → G
```

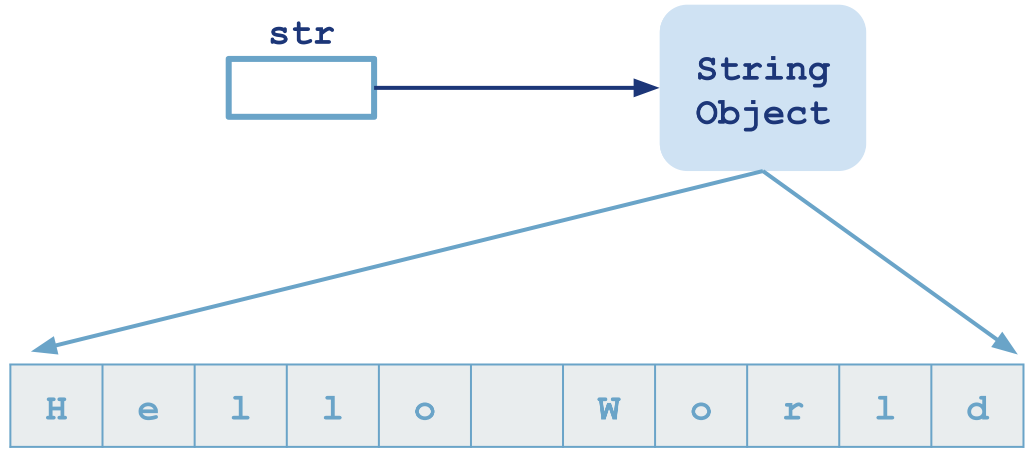
Strings



Strings

```
String str =  "Hello World";
```

- ➔ Strings are not primitive data types, they are objects
- ➔ New object created with name assigned to its memory location
- ➔ Often considered alongside primitives even though they are objects



String Concatenation

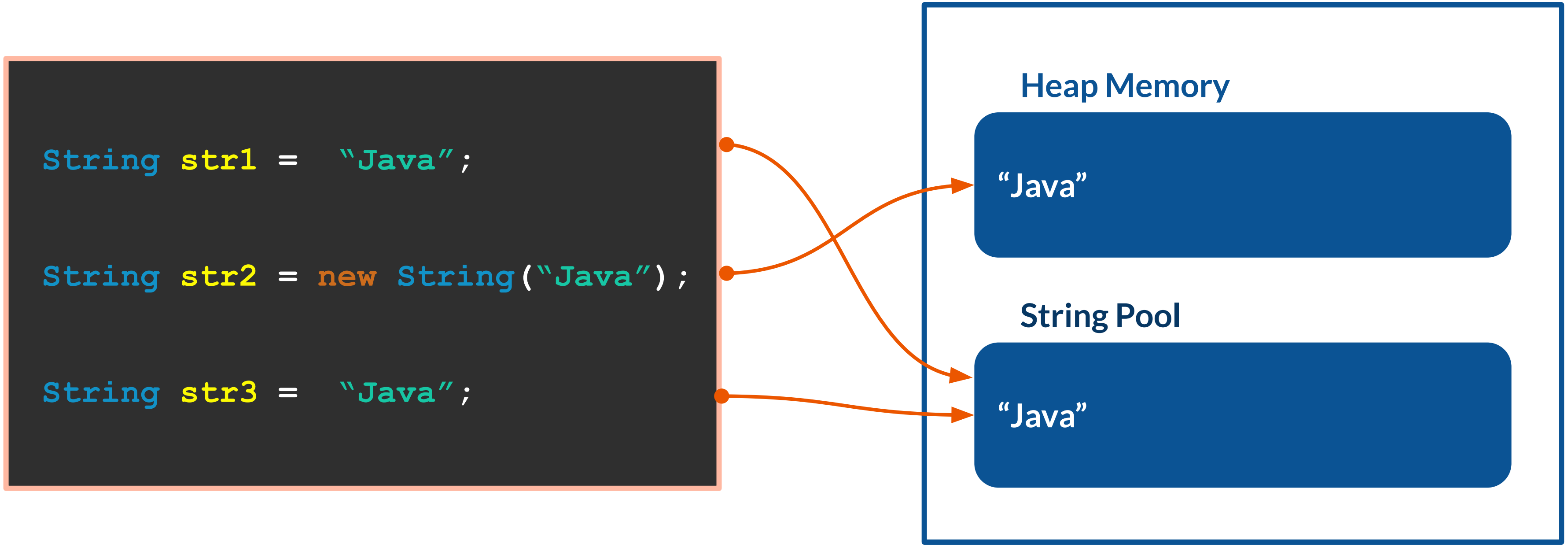
- ➔ The **+ operator** can concat a string with other data types
  - ◆ Will convert other data types to strings
- ➔ Using **concat()** method
  - ◆ Doesn’t edit string, returns new string with concatenation
  - ◆ Only accepts Strings
  - ◆ Will throw exception if String passed is null

```
String str1 =  “Hello ”;

// str2 = “Hello World”
String str2 = str1.concat(“World”);

boolean bool = true;

// str3 = “Hello There2.3true”
String str3 = str1 + “There” + 2.3 +
bool;
```

String Immutability and String Pool

- ➔ Strings are ***final and immutable*** — can't be altered (adding/removing a character)
- ➔ New object created each time value reassigned
- ➔ **String Pool** — place in memory where the JVM stores each unique character sequence that is created
- ➔ When a new string is created,
    - ◆ JVM searches for that char sequence in String Pool
    - ◆ Sequence found, assigns new object that value in the pool
    - ◆ Not found, creates char sequence in String Pool and assigns the variable to that memory location
- ➔ Using new keyword → String stored in heap memory

String Pool Diagram



```
String str1 =  "Java";

String str2 = new String("Java");

String str3 =  "Java";
```

Comparing Strings

- ➔ Consider the following scenario:

```
String str1 =  “Java”;
String str2 = new String( “Java”);
```

- ➔ Two strings have the same char sequence. However, if we try and compare them:

```
str1 == str1; // true
str2 == str1; // false
```

- ➔ Strings are objects, so the variables we assign them are references to memory locations, not the char sequences themselves.
- ➔ To check the equivalence of strings, we must use the .equals( ) method:

```
str1.equals(str1); // true
str2.equals(str1); // true
```

String Methods

There are almost 50 methods defined in the String class. Important methods include:

- **Length**:
  - Returns the number of chars in the string as an int
- **CharAt**:
  - Returns the char at the given index
- **IndexOf**:
  - Returns the first index of a given char in the string
- **Substring**:
  - Extracts a section of the string at the given indexes
- String **Concatenation**, while not a defined method, is an important operation when working with strings

Length of a String

Index:	0	1	2	3	4	5	6	7	8	9	10
Character:	H	e	l	l	o		W	o	r	l	d

- ➔ Strings are arrays of characters
- ➔ *length( )* returns number of characters in the array

```
System.out.println("string length:  " + str.length());
// string length: 11
```

- ➔ <u>Important</u>: Array indexes start at 0, so the length of the string is one more than the last index

Single Character Access

Index:	0	1	2	3	4	5	6	7	8	9	10
Character:	H	e	l	l	o		W	o	r	l	d

➔ Individual characters can be accessed with the ***charAt()***:

```
System.out.println("first letter:  " + str.charAt(0));
// first letter: H
```

➔ Trying to access a char outside the length of an array results in an error

Finding an Index

Index:	0	1	2	3	4	5	6	7	8	9	10
Character:	H	e	l	l	o		W	o	r	l	d

- ➔ ***indexOf( )*** method returns index where a given substring appears in the string
- ➔ Is overloaded operator
  - ◆ can accept a string or a char
  - ◆ takes second argument of a starting index

```
String s1 = "first letter ‘H’: " + str.indexOf(‘H’); // first letter ‘H’: 0
String s2 = "first ‘W’: " + str.indexOf(‘W’); // first letter ‘W’: 6
String s3 = "‘o’ after index 5: " + str.indexOf(‘o’, 5); // ‘o’ after index 5: 7
```

Substrings

Index:	0	1	2	3	4	5	6	7	8	9	10
Character:	H	e	l	l	o		W	o	r	l	d

- ➔ ***substring( )*** retrieves specific part of a string
- ➔ Returns new string, original string not modified
- ➔ Overloaded, accepts one or two arguments
  - ◆ *.substring(i, j)* return new string starting with character at ***i*** and ending before index ***j***
  - ◆ *.substring(i)* returns new string starting at index ***i*** till the end of string

```
String s1 = str.substring(3, 7); // “lo W”
String s2 = str.substring(4); // “o World”
```

Other String Methods

- **compareTo( )**
  - Compares two strings lexicographically
- **trim( )**
  - Removes leading and trailing whitespace characters
- **valueOf( )**
  - Converts a primitive to a string
- **startsWith( )**
  - Returns true if the string starts with the given char
- **endsWith( )**
  - Returns true if the string ends with the given char
- Most IDEs will give a user access to all available methods for a given class.
  - String methods detailed in Java Documentation: https://docs.oracle.com/en/java/javase/13/docs/api/java.base/java/lang/String.html

StringBuffer and StringBuilder

- ➔ **StringBuffer** or the newer **StringBuilder** (Java 5) used to work with mutable character sequences
  - ◆ StringBuffer is thread safe, but Stringbuilder is faster.
  - ◆ Both have the same methods, interchangeable
- ➔ Contain methods for:
  - ◆ Replacing chars
  - ◆ Appending or prepending any primitive data type
  - ◆ Inserting new char sequences at a given position

```
StringBuffer sb = new StringBuffer(“Hello”);
sb.append(“ World”); // Hello World
sb.setCharAt(5,‘@’); // Hello@World
sb.append(1);        // Hello@World1
sb.append(true);     // Hello@World1true
```

WHITE BOARD EXERCISE

Coding challenge:

Create a method that takes a string, reverses it, and returns the reversed string. You can use a built in method to do this

Part 2:

Create a second method that does not use any built in methods to reverse the string (i.e. will reverse the string for you).

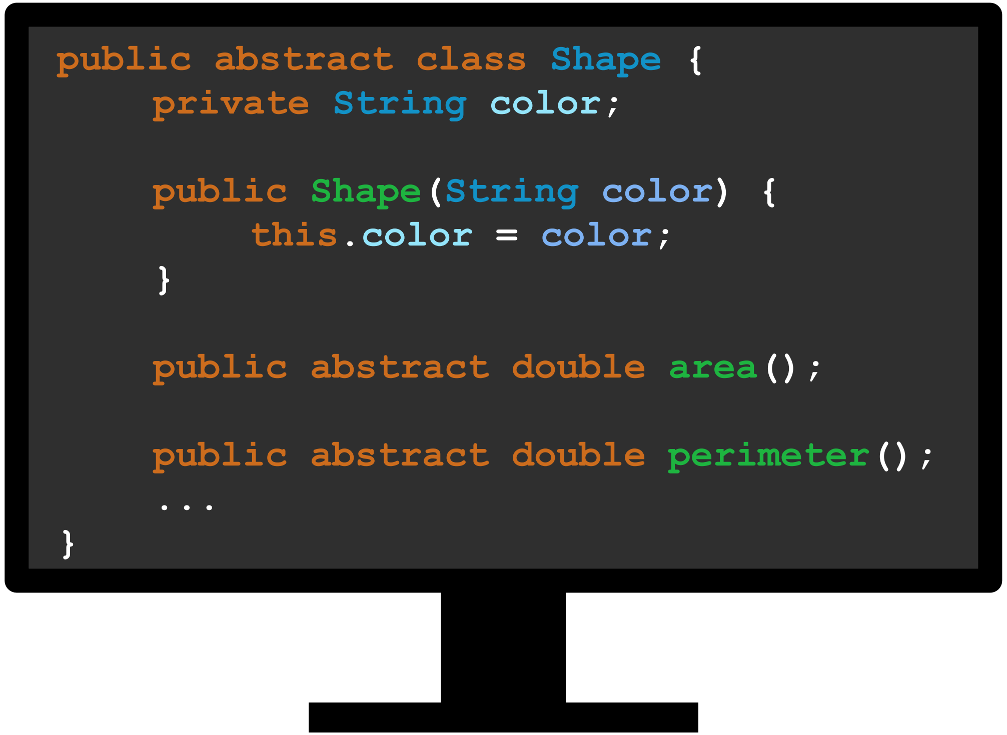
Bonus:

For a given string with multiple words, reverse each word individually

Interfaces

Abstract Class

- ➔ An **abstract class**
  - ◆ defined with the modifier ***abstract***
  - ◆ can contain ***abstract methods***
  - ◆ doesn't implement inherited abstract methods
  - ◆ Can't create instance of abstract class
- ➔ **Abstract method** - a method with keyword ***abstract***, ends with a semicolon instead o f method body
  - ◆ Private and static methods can't be ***abstract***.



```
public abstract class Shape {
    private String color;

    public Shape(String color) {
        this.color = color;
    }

    public abstract double area();

    public abstract double perimeter();
    ...
}
```

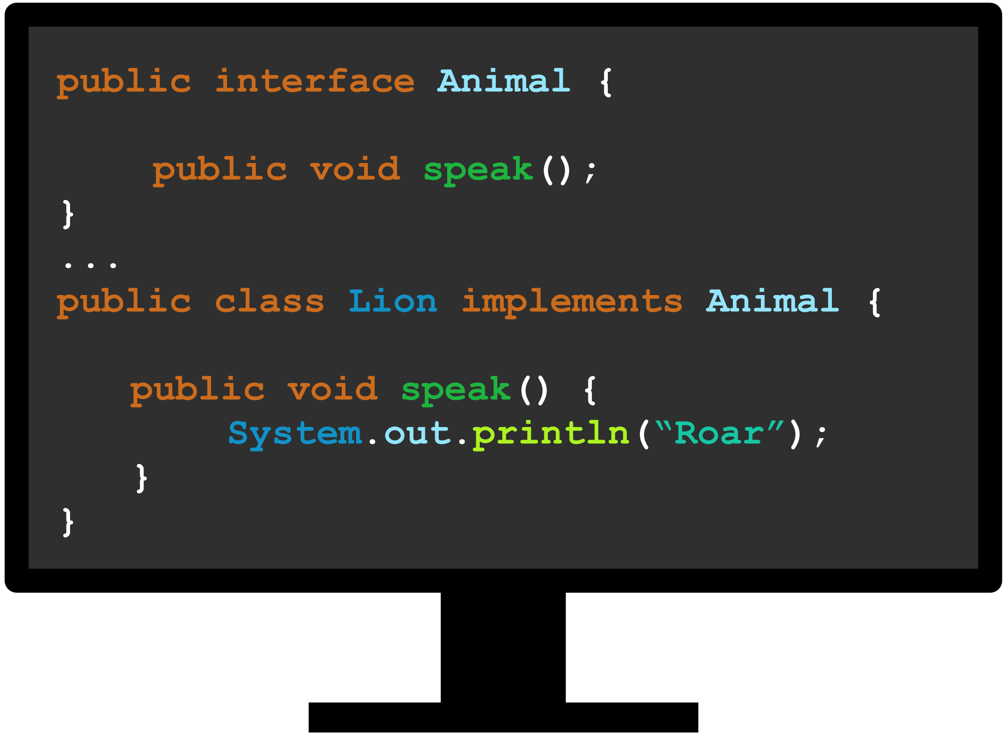
Interface

➔ **Interface** is a contract

➔ Contains **method signatures** (methods without implementation) and **static constants** (final static)

➔ Can only be implemented by classes and extended by other interfaces

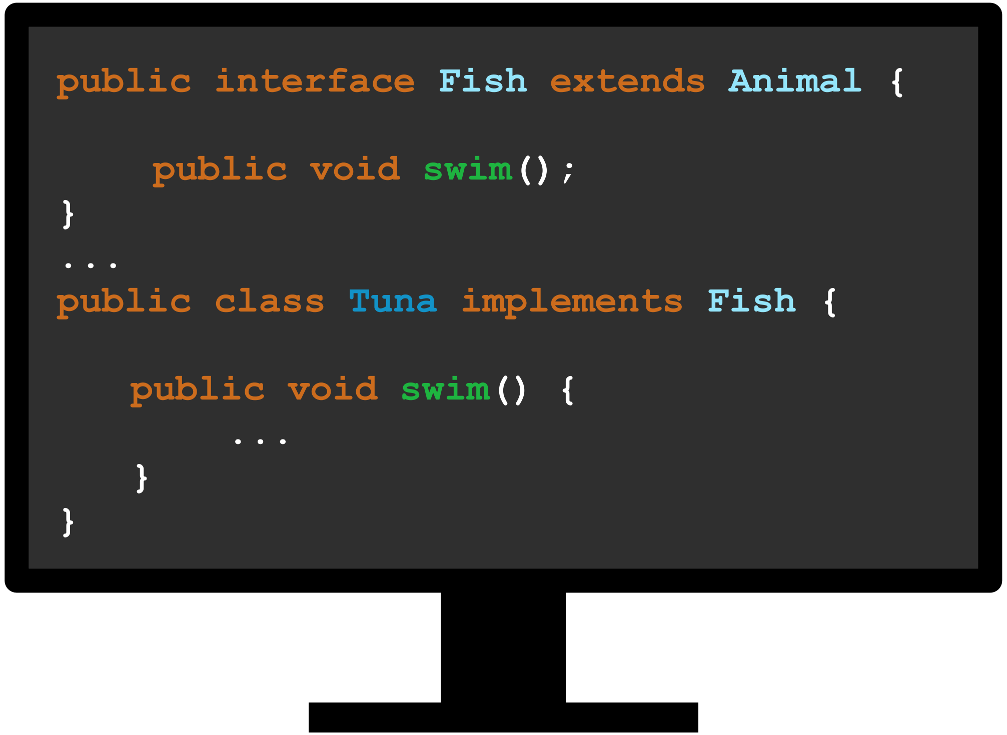
➔ Used as a “template” for how a class should be structured



```
public interface Animal {

    public void speak();
}
...
public class Lion implements Animal {

   public void speak() {
        System.out.println(“Roar”);
   }
}
```

Interface Inheritance

- ➔ A class inheriting from an interface uses the keyword **implements**
- ➔ An interface inheriting from another interface uses the keyword **extends**



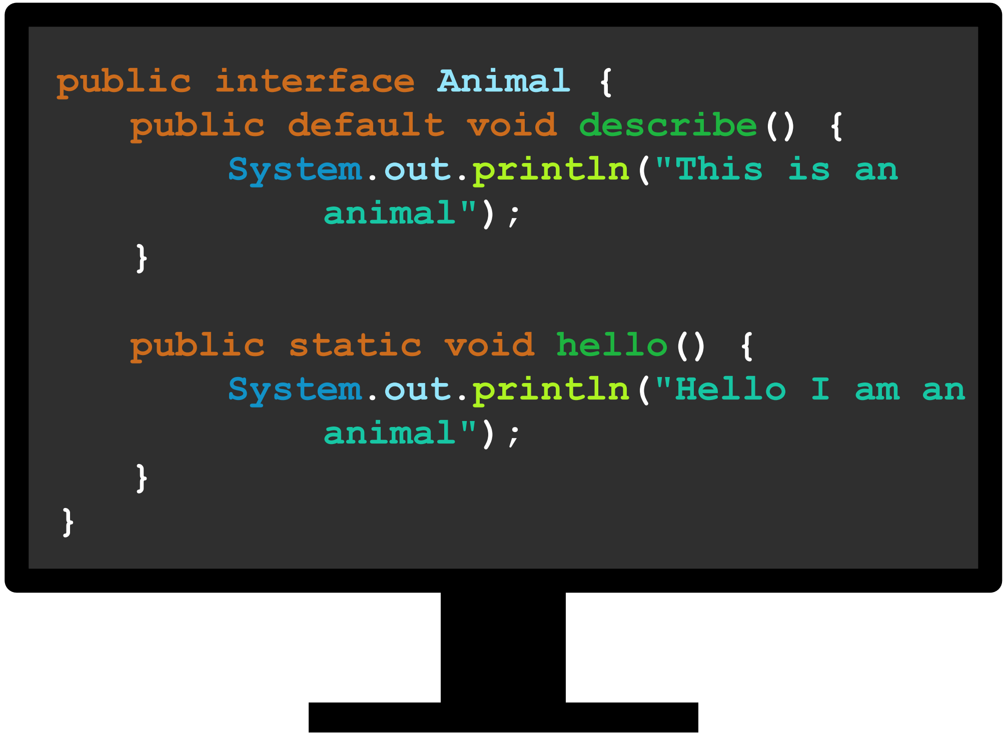
```
public interface Fish extends Animal {

    public void swim();
}
...
public class Tuna implements Fish {

   public void swim() {
        ...
   }
}
```

Java 8 Interfaces

➔ With Java 8, interfaces can have default and static methods

➔ **Default methods** allow for a default method for an interface, can be overwritten

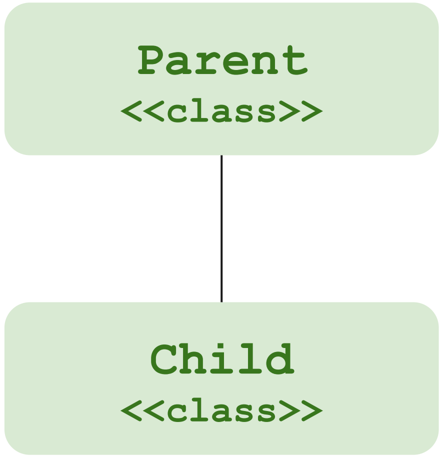
➔ **Static methods** are methods that belong to the interface

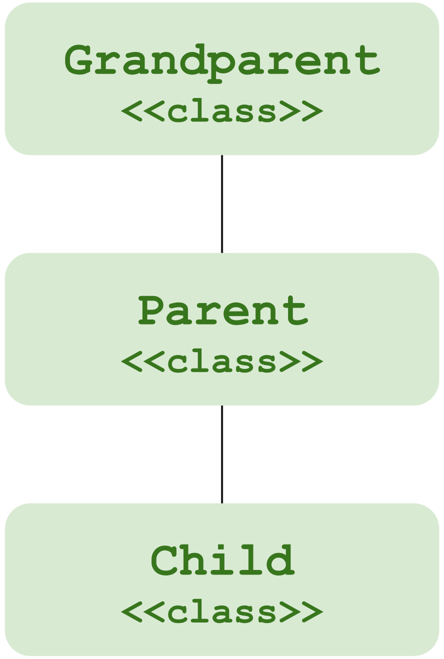


```
public interface Animal {
    public default void describe() {
        System.out.println("This is an
            animal");
    }

    public static void hello() {
        System.out.println("Hello I am an
            animal");
    }
}
```

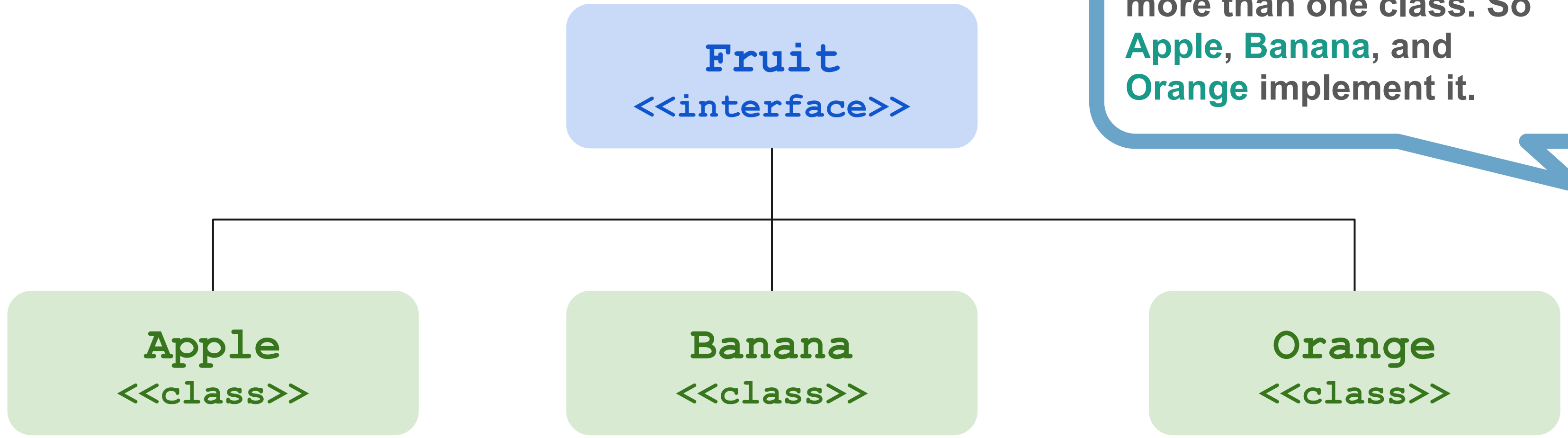
Single and Multilevel Inheritance

Child can inherit from Parent on left, but the multilevel on right allows Child to inherit from both Parent and Grandparent.



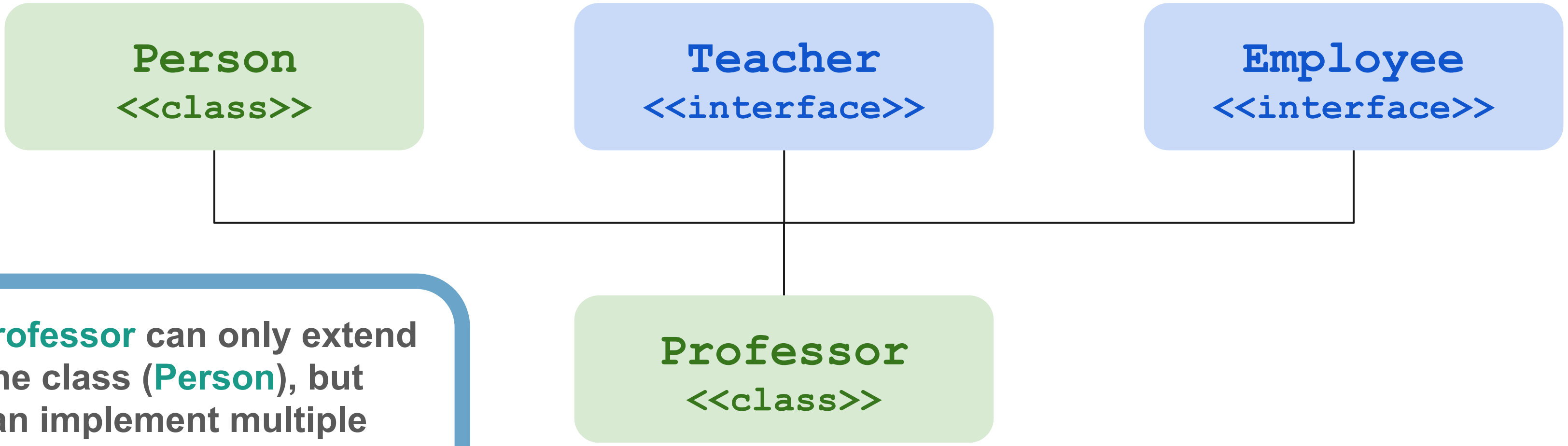


Hierarchical Inheritance



Fruit can be inherited by more than one class. So Apple, Banana, and Orange implement it.

Multiple Inheritance

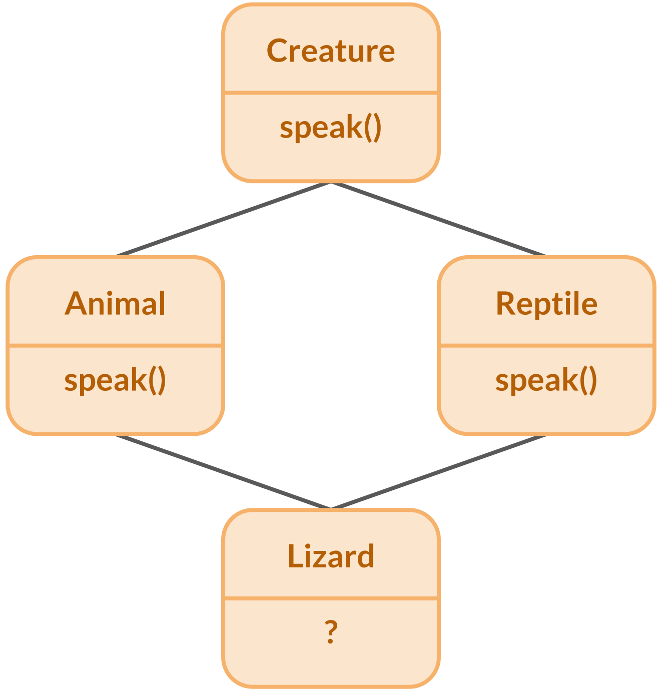


Professor can only extend one class (Person), but can implement multiple interfaces (Teacher, Employee).

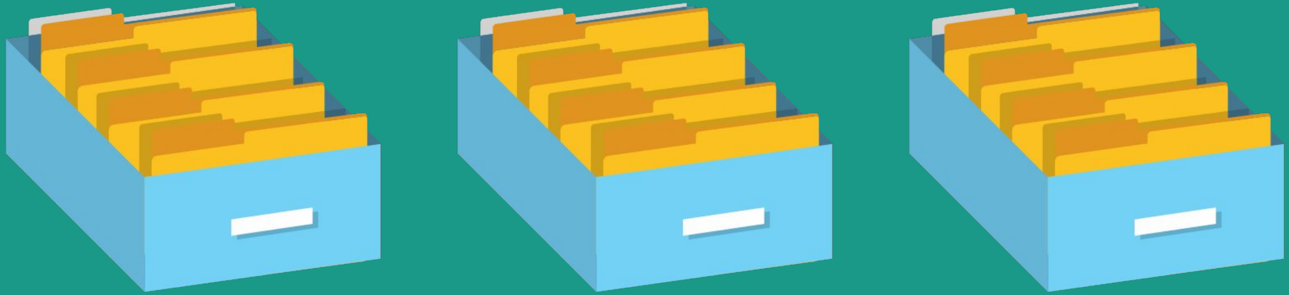
Hybrid Inheritance

Diamond Problem

Will Lizard inherit the method speak from Animal or Reptile? Or could it possibly inherit it from Creature?



Packages and Imports



Packages and Imports

➔ **Packages** - Java mechanism to organize classes

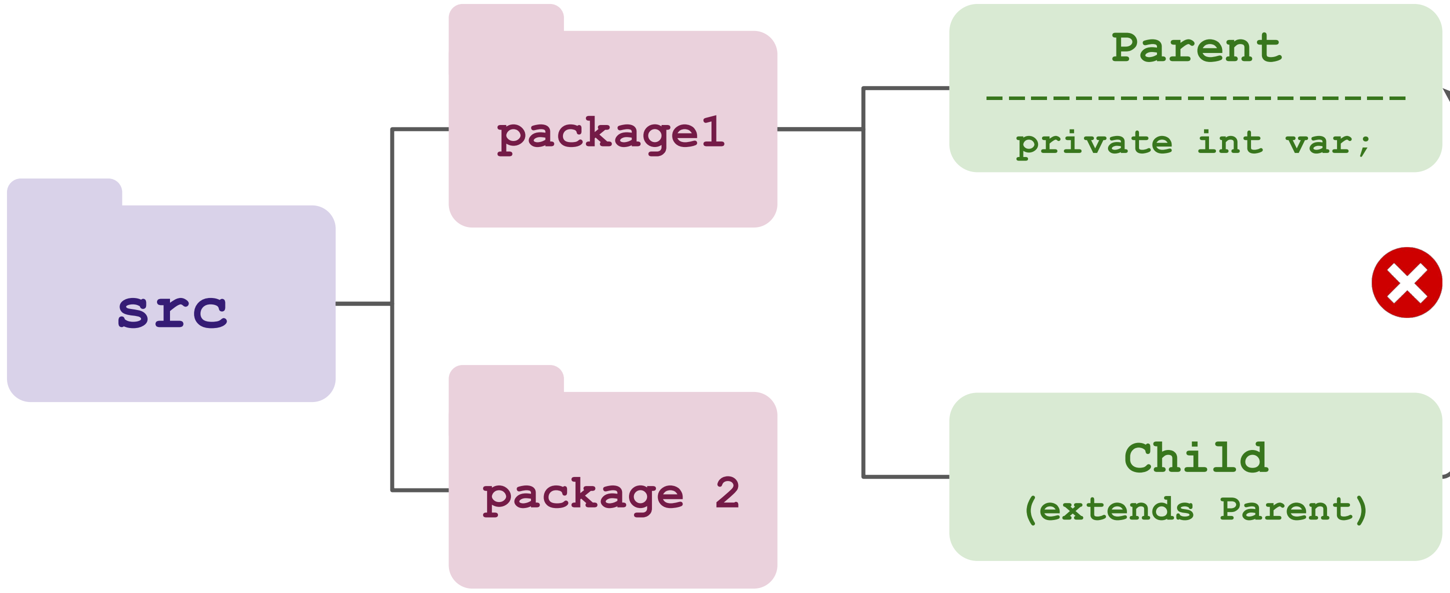
- Naming Conventions:
  - i.e. - com.cognixia.jump.corejava
  - OrganizationType.CompanyName.OrganizationTopic

➔ **Imports** - Java needs to know what libraries to reference

to use certain Classes

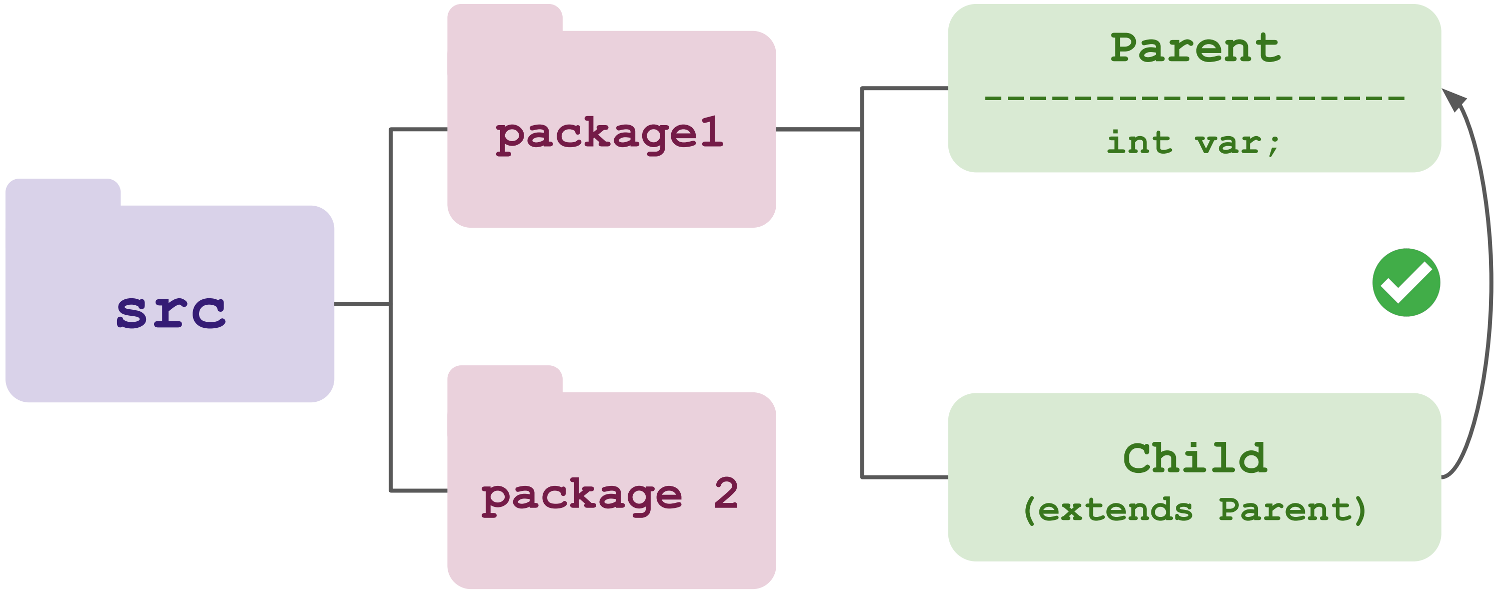
- I.e. - java.lang, java.util

Access Modifiers: Default



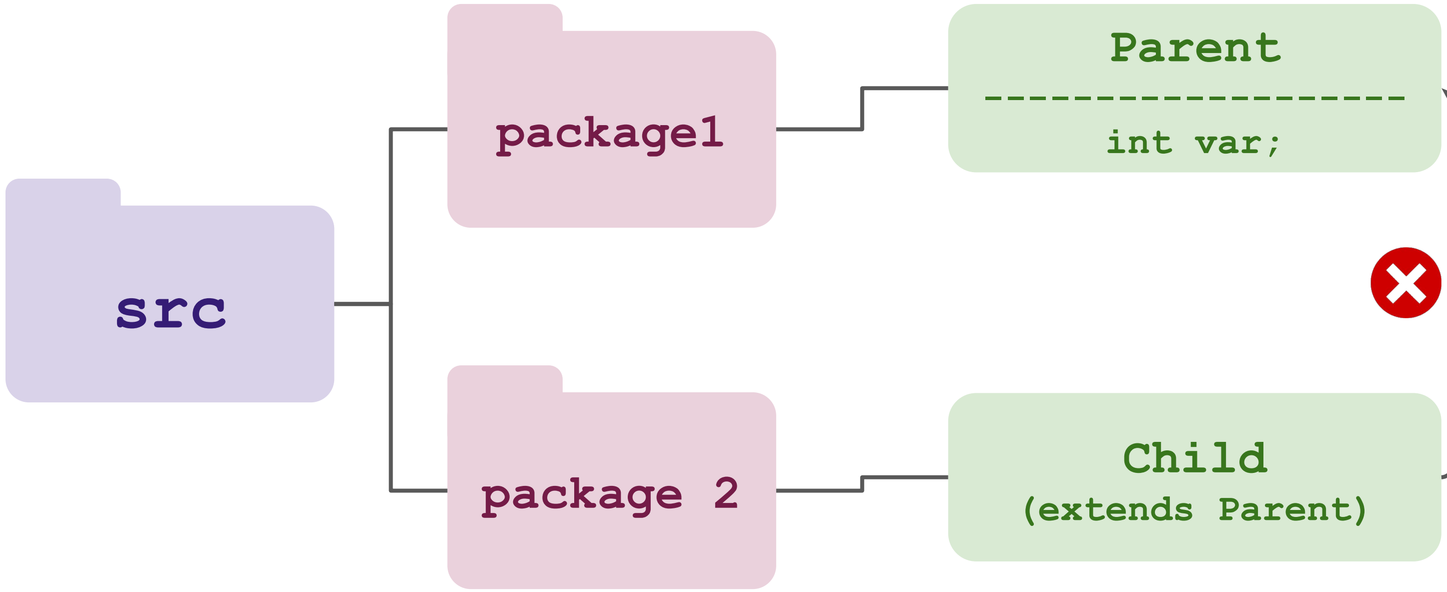
How to access *var* directory?

Access Modifiers: Default



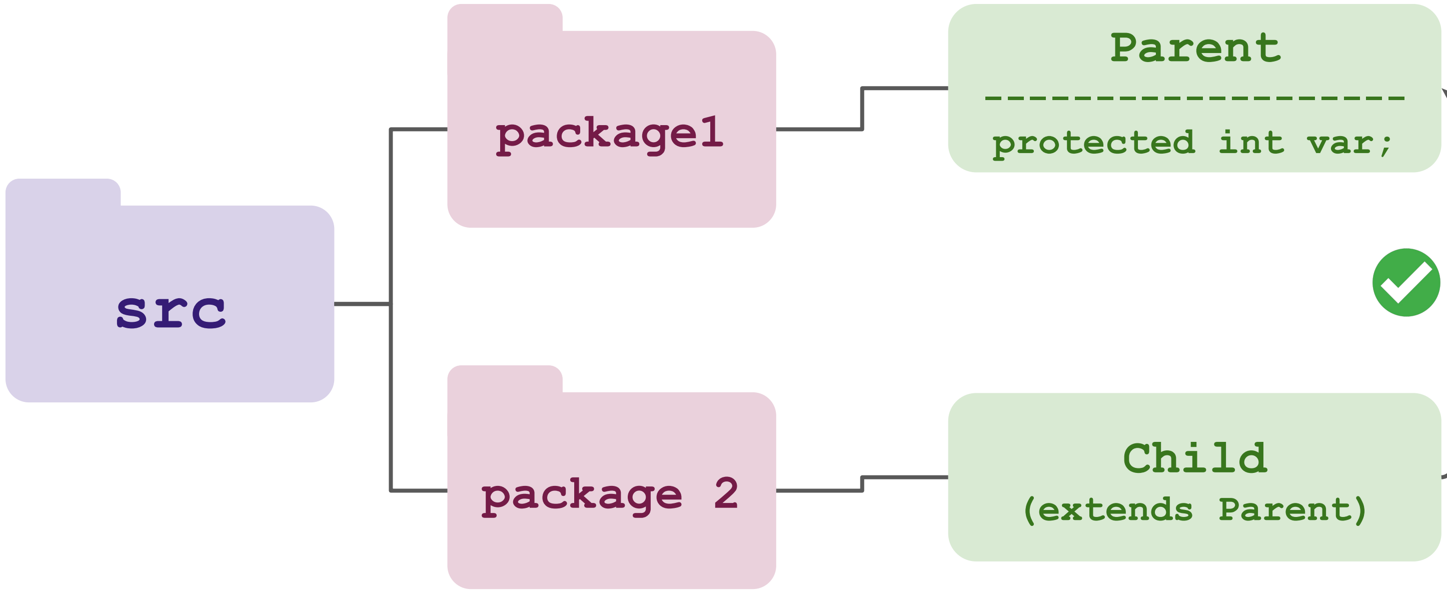
Set modifier to default (no modifier at all).

Access Modifiers: Protected



If child is in another package?

Access Modifiers: Protected



Add a protected modifier.

Access Modifiers

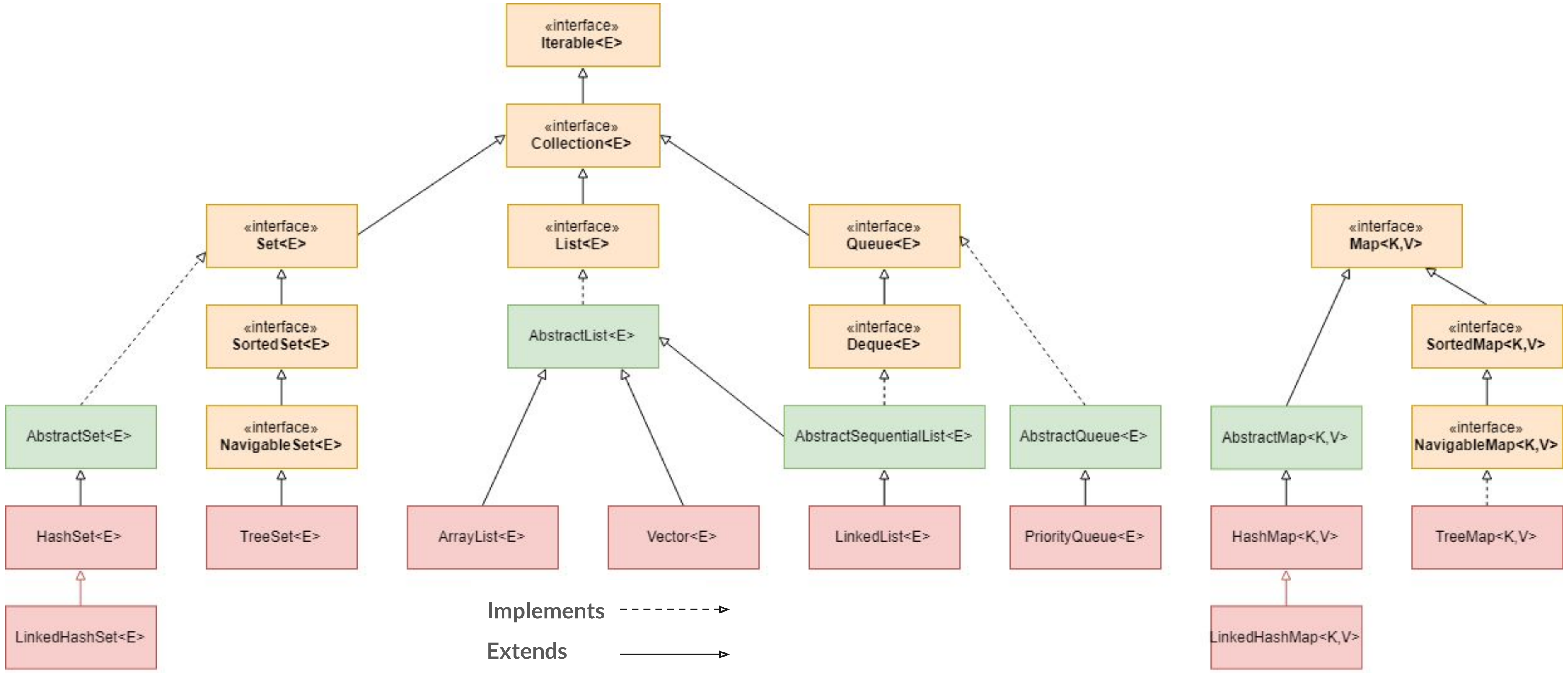
Modifier	Class	Package	Subclass	Global
Public	Allowed	Allowed	Allowed	Allowed
Protected	Allowed	Allowed	Allowed	Denied
Default	Allowed	Allowed	Denied	Denied
Private	Allowed	Denied	Denied	Denied

Collections and Generics

Primitive vs Autoboxed

- ➔ Java is NOT a pure OOP language
  - ◆ has primitives
- ➔ Objects are required in many applications for Java
- ➔ Wrapper Classes - objects that wrap around the primitive type
- ➔ Can use Collections - Collections cannot store primitives on their own

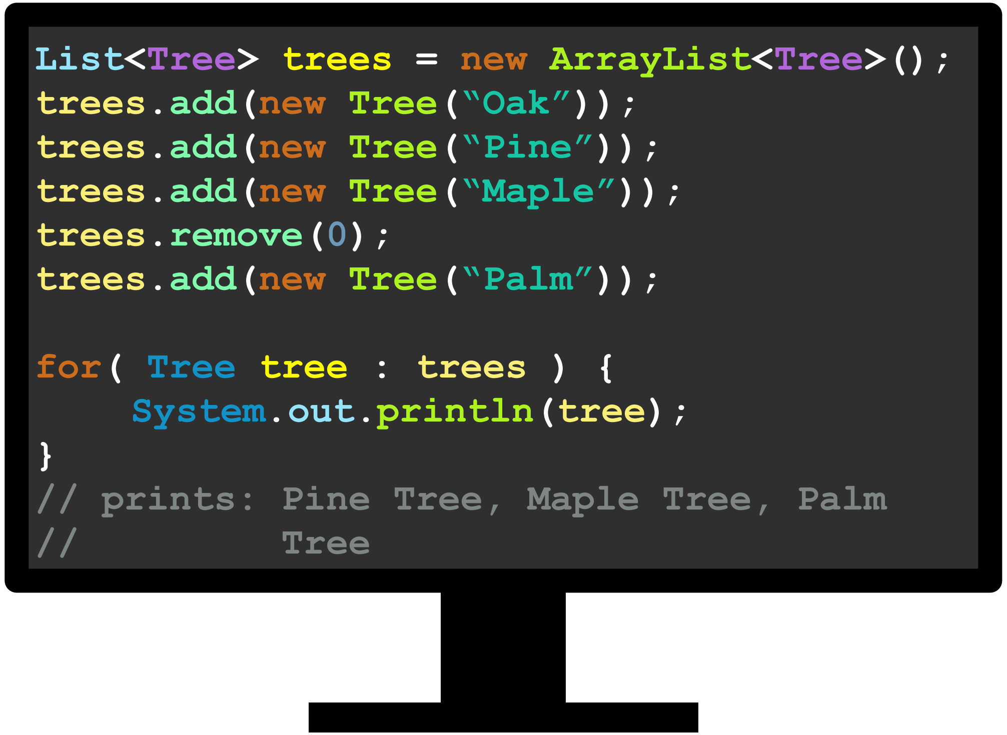
```java
double dubs = 5.0;
int num = (int) dubs;

// passes int to boxed type
Integer boxed = num;
```



Collections Framework - provides a set of classes and interfaces that can store and manipulate groups of objects

List

- ➔ List supports methods to maintain a collection of objects as a linear list
  - ◆ $L = (l_0, l_1, l_2, \ldots, l_N)$
- ➔ No limit to objects that can be added
- ➔ Size is dynamic
- ➔ Classes that implement List:
  - ◆ ArrayList - array used to manage data
  - ◆ LinkedList - stores objects using linked-node representation
  - ◆ Vector - like ArrayList, but synchronized



```
List<Tree> trees = new ArrayList<Tree>();
trees.add(new Tree("Oak"));
trees.add(new Tree("Pine"));
trees.add(new Tree("Maple"));
trees.remove(0);
trees.add(new Tree("Palm"));

for( Tree tree : trees ) {
     System.out.println(tree);
}
// prints: Pine Tree, Maple Tree, Palm
//         Tree
```

Frequently Used List Methods

`boolean add( E e )`	Add an element to the list (must be an object)
`int size()`	Returns the number of elements in the list
`E get( int index )`	Returns the element at the index given
`E remove( int index )`	Removes element at index given, returns the element removed
`boolean remove( E e )`	Removes the element passed, will return true or false if element given was found and removed
`void clear()`	Clears list (removes all elements)
`boolean isEmpty()`	Returns true or false if list is empty

Homogeneous vs Heterogeneous Collections

- ➔ Homogeneous collections include objects of a single type

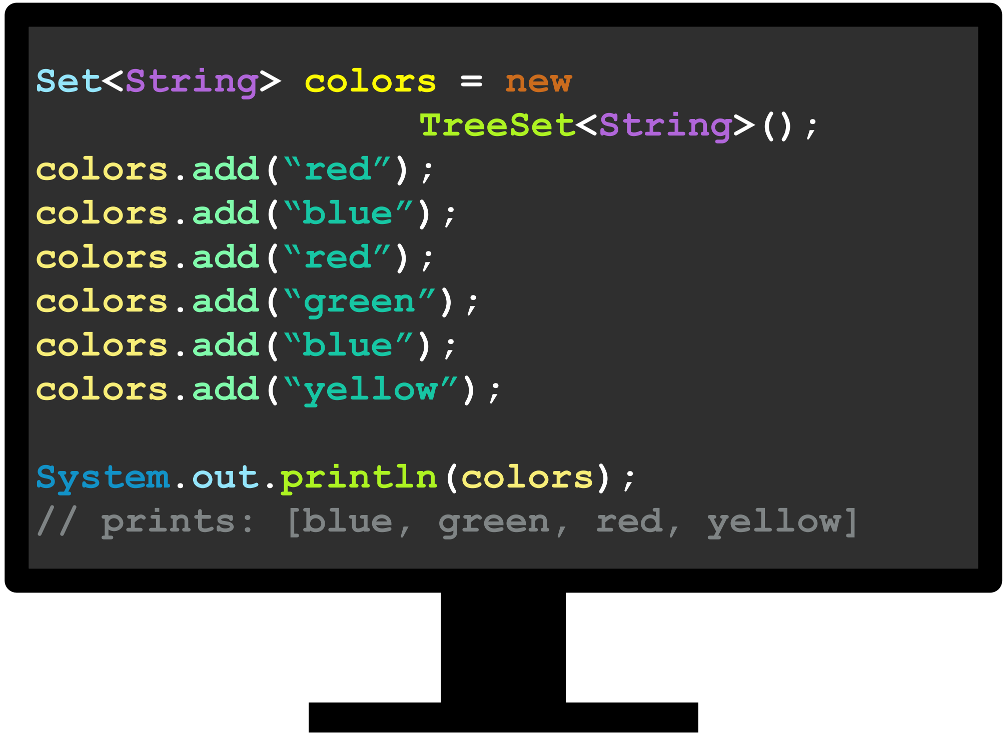
```
List<Monster> monsters = new ArrayList<Monster>();
monsters.add(new Monster());
monsters.add(new Monster());
...
```

- ➔ Heterogeneous collections include objects of a variety of types (derived from the same base class)

```
List<Monster> monsters = new ArrayList<Monster>();
monsters.add(new Vampire()); // Vampire and Mummy inherit
monsters.add(new Mummy());   // from Monster
...
```

Set

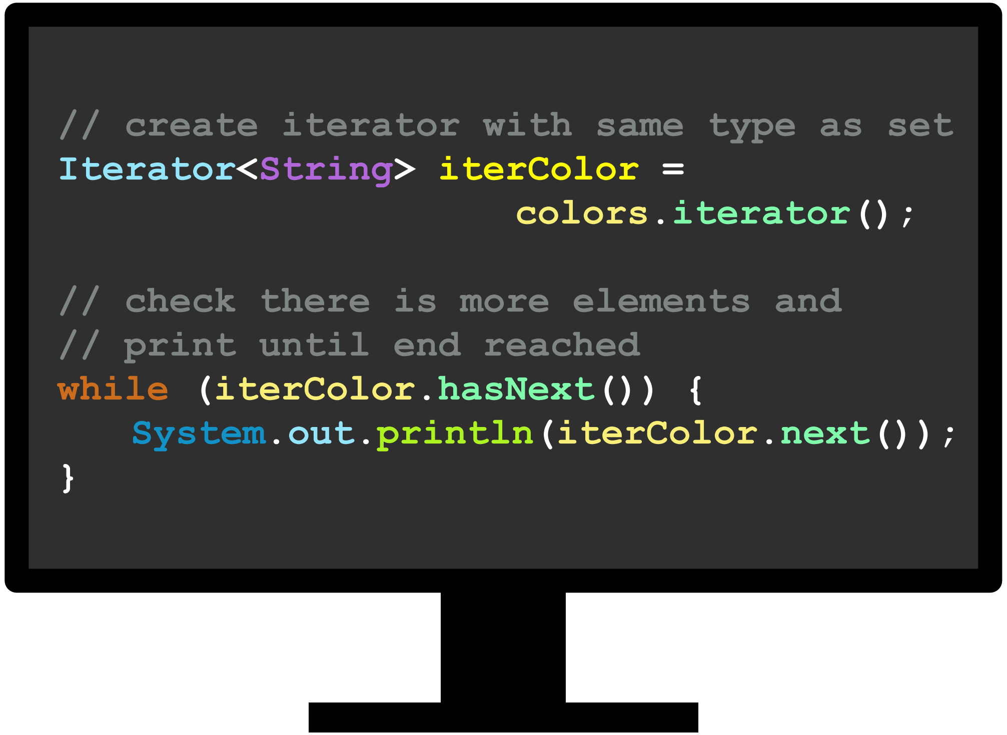
- ➔ Sets are an unordered collection of objects with no duplicates
- ➔ Classes that implement Set:
  - ◆ TreeSet - sorts any element added to it
  - ◆ HashSet - implemented using hash table and is faster than a TreeSet at access data



```
Set<String> colors = new
                  TreeSet<String>();
colors.add(“red”);
colors.add(“blue”);
colors.add(“red”);
colors.add(“green”);
colors.add(“blue”);
colors.add(“yellow”);

System.out.println(colors);
// prints: [blue, green, red, yellow]
```

Iterator

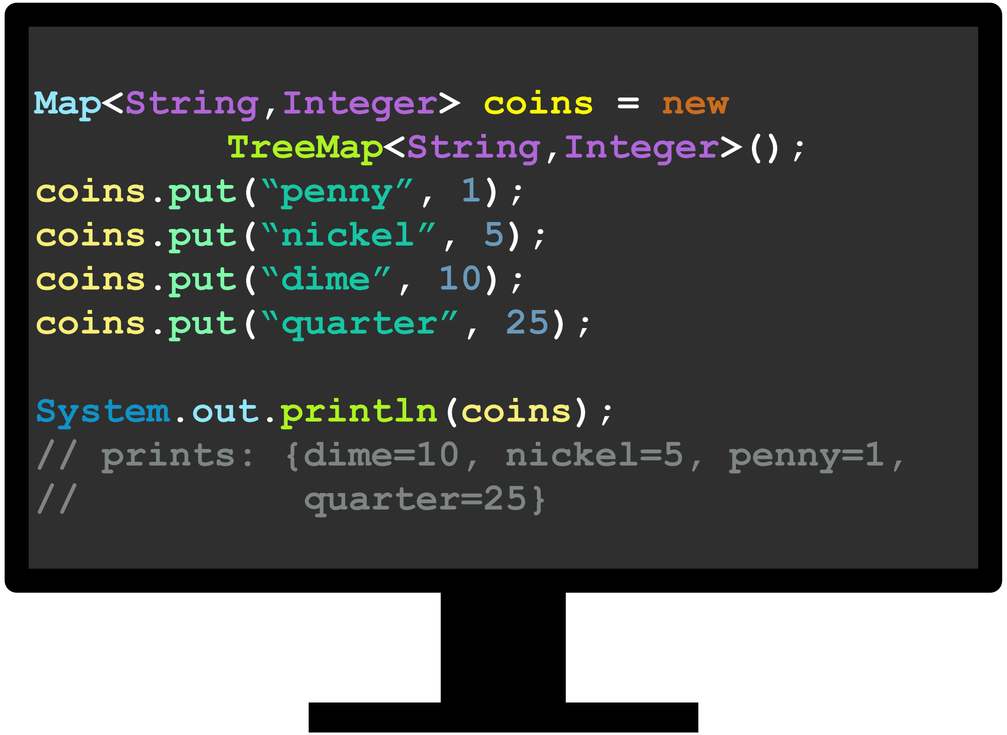
- ➔ An **Iterator** is an object used with collections to provide sequential access to that collection's elements
- ➔ Can impose ordering on elements if none
- ➔ Can alternately use ***for-each loop*** for certain situations



```
// create iterator with same type as set
Iterator<String> iterColor =
                    colors.iterator();

// check there is more elements and
// print until end reached
while (iterColor.hasNext()) {
    System.out.println(iterColor.next());
}
```

Map

- ➔ **Maps** contain methods that maintain a collection of objects with ***key-value pairs*** called map entries
- ➔ Classes that implement Map:
  - ◆ **TreeMap** - sorts pairs within it
  - ◆ **HashMap** - not sorted, uses a hash table to access pairs by their key



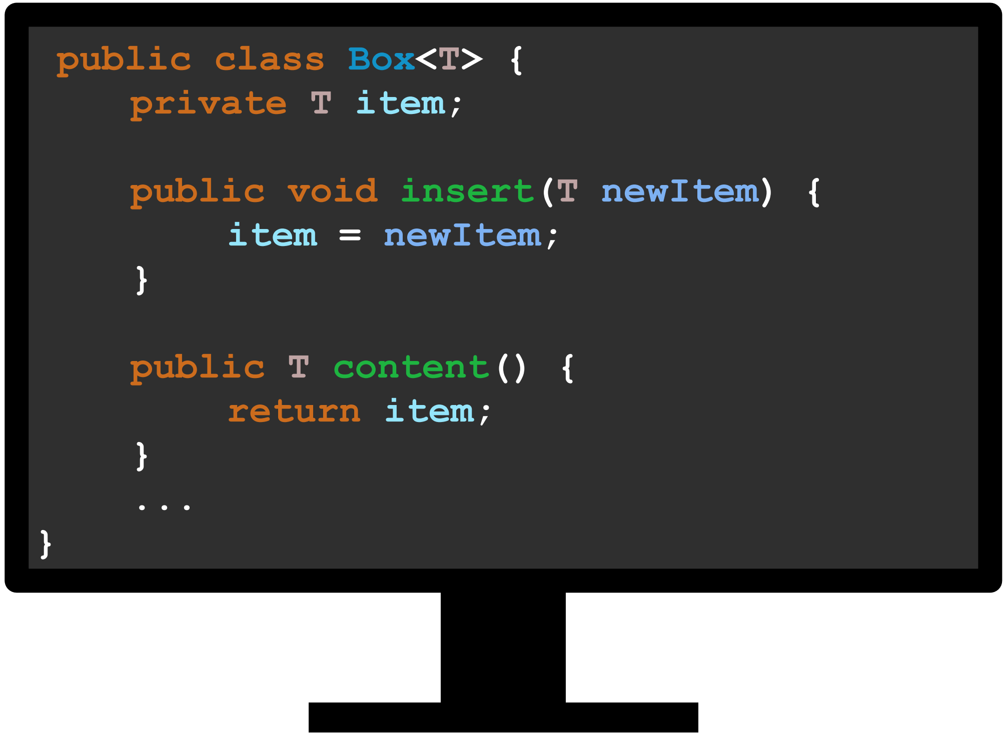
```
Map<String,Integer> coins = new
            TreeMap<String,Integer>();
coins.put(“penny”, 1);
coins.put(“nickel”, 5);
coins.put(“dime”, 10);
coins.put(“quarter”, 25);

System.out.println(coins);
// prints: {dime=10, nickel=5, penny=1,
//               quarter=25}
```

Hashtable vs HashMap vs ConcurrentHashMap

Hashtable	HashMap	ConcurrentHashMap
➔ ***Synchronized*** ➔ ***No null keys or values*** can be passed to it ➔ Puts ***lock on whole map*** so all methods and access to data synchronized	➔ ***Not synchronized***, not thread safe ➔ Allows for ***one null key*** and ***multiple null values***	➔ ***Synchronized*** ➔ ***No null keys or values*** can be passed to it ➔ Doesn't lock whole map, ***locks it in segments***, data that needs to be updated, will lock only segment its in

Generics

- Introduced in Java 5
- Before, could store any type of objects in a collection
- Generics create type safe collections
- Advantages:
  - *Type safe*
  - *Type casting not required*
  - *Compile-time checking*



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

    public void insert(T newItem) {
        item = newItem;
    }

    public T content() {
        return item;
    }
    ...
}
```

Kahoot!

1. Close notes, use phone or laptop
2. Go to kahoot.it
3. Click on Enter game PIN

Exceptions

Exceptions

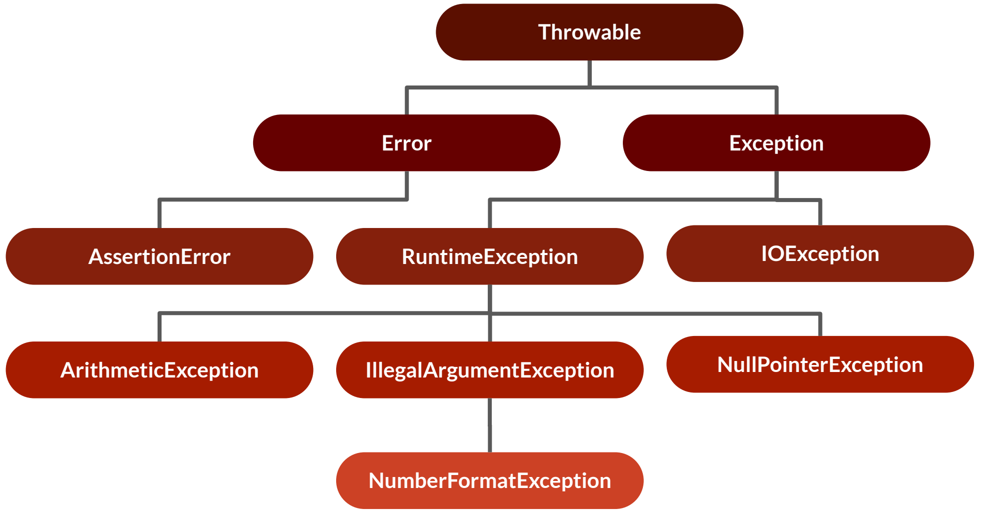
- ➔ An **Exception** is an interruption in the execution of a program
- ➔ Normal flow of a program is terminated
  - ◆ exception is ***thrown***
- ➔ Exception-handling routine is executed
  - ◆ exception is ***caught***

WARNING

PROGRAMMERS MUST HANDLE ALL EXCEPTIONS OR RISK TERMINATING THEIR PROGRAM

Throwable Hierarchy

- ➔ **Exceptions** are objects that represent situations that the developer should handle
- ➔ **Errors** are problems within the JVM itself
  - ◆ More serious than Exceptions
- ➔ Exceptions and Errors extensions of ***Throwable***



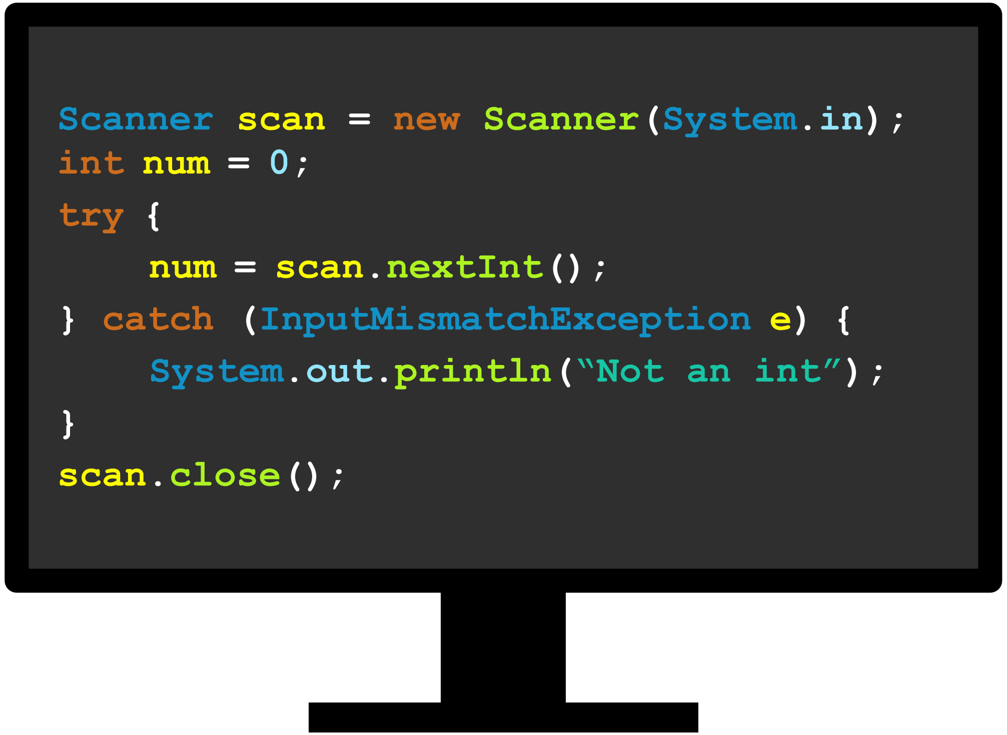
Uncaught Exceptions

- ➔ JVM searches call stack for code containing instructions matching the Exception
  - ◆ Called **Exception Handler**
- ➔ If no matching handler, JVM hands Exception object to ***Default Exception Handler***
  - ◆ Program will terminate abnormally and print **stack trace** to console

```
Exception in thread "main" java.util.InputMismatchException
    at java.base/java.util.Scanner.throwFor(Scanner.java:939)
    at java.base/java.util.Scanner.next(Scanner.java:1594)
    at java.base/java.util.Scanner.nextInt(Scanner.java:2258)
    at java.base/java.util.Scanner.nextInt(Scanner.java:2212)
    at com.cognixia.jump.advancedJava.exceptions.
        .UncaughtExceptions.main(UncaughtExceptions.java:14)
```

Try/Catch Block

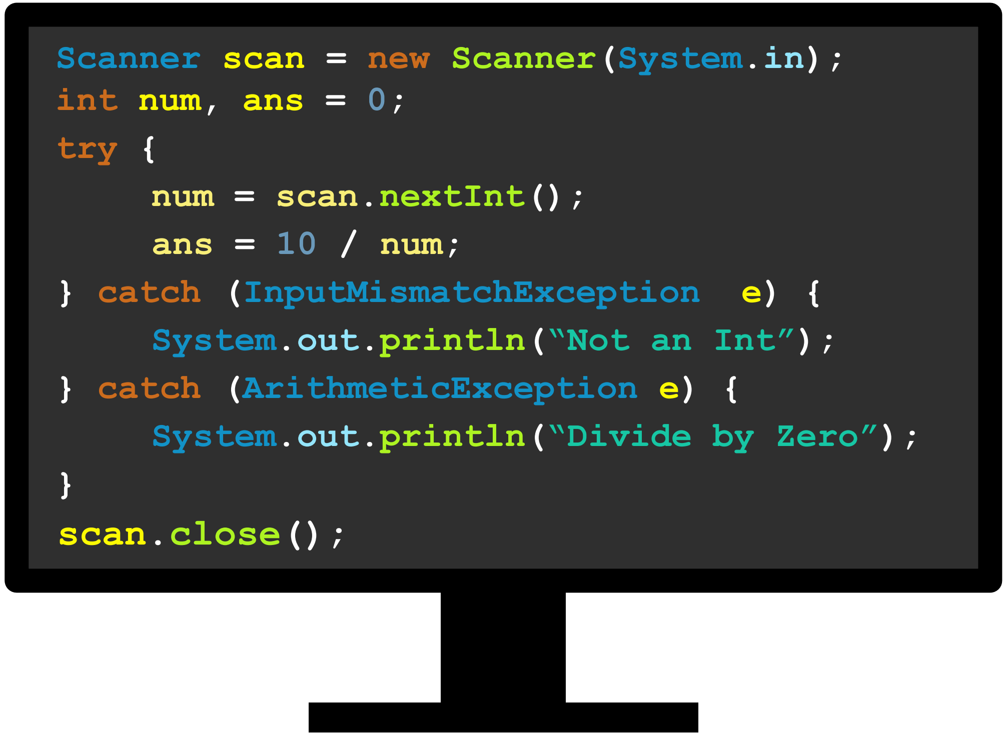
- ➔ A try/catch block must be used for exception handling
- ➔ The code within ***try block*** executes normally until exception occurs
- ➔ Remaining code in try block skipped, code in ***catch block*** immediately executed



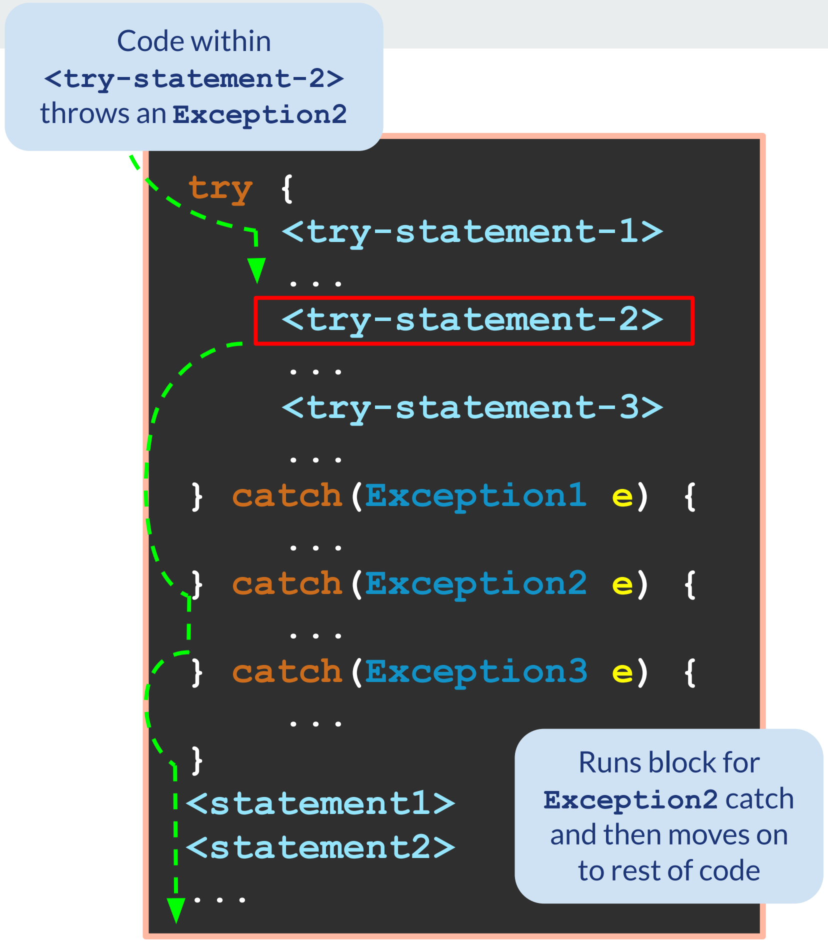
```
Scanner scan = new Scanner(System.in);
int num = 0;
try {
    num = scan.nextInt();
} catch (InputMismatchException e) {
    System.out.println("Not an int");
}
scan.close();
```

Multiple Catch

- ➔ A single **try** block can be paired with multiple **catch** blocks
- ➔ Each **catch** block will handle a different kind of exception
- ➔ A more specific kind of exception cannot follow a more general exception



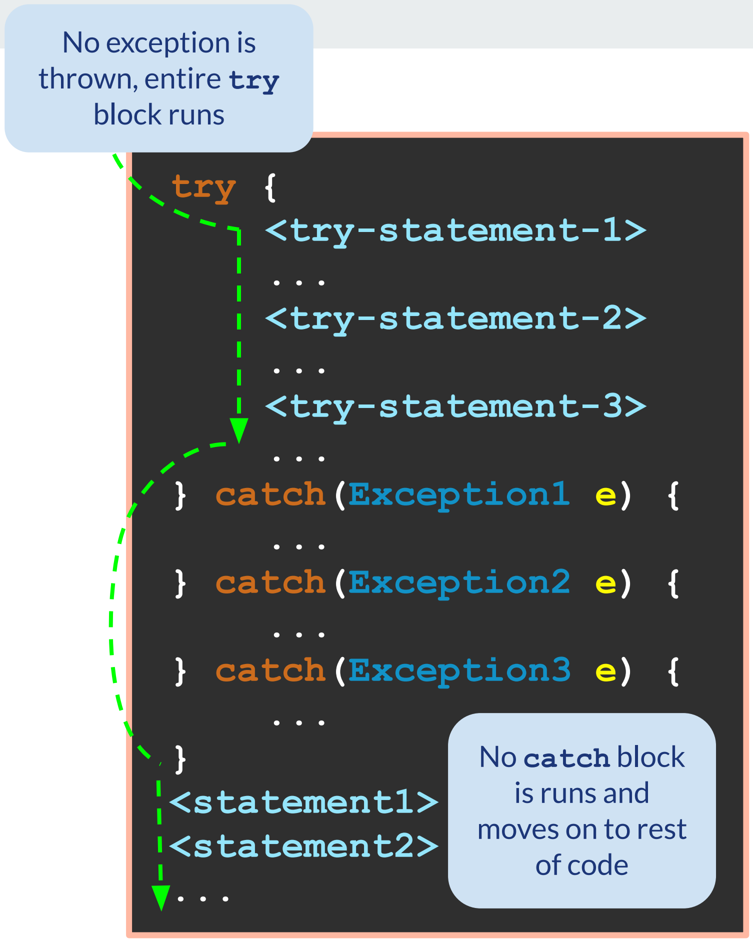
```
Scanner scan = new Scanner(System.in);
int num, ans = 0;
try {
      num = scan.nextInt();
      ans = 10 / num;
} catch (InputMismatchException  e) {
      System.out.println(“Not an Int”);
} catch (ArithmeticException e) {
      System.out.println(“Divide by Zero”);
}
scan.close();
```



Code within <try-statement-2> throws an Exception2

```
try {
    <try-statement-1>
    ...
    <try-statement-2>
    ...
    <try-statement-3>
    ...
} catch(Exception1 e) {
    ...
} catch(Exception2 e) {
    ...
} catch(Exception3 e) {
    ...
}
<statement1>
<statement2>
...
```

Runs block for Exception2 catch and then moves on to rest of code



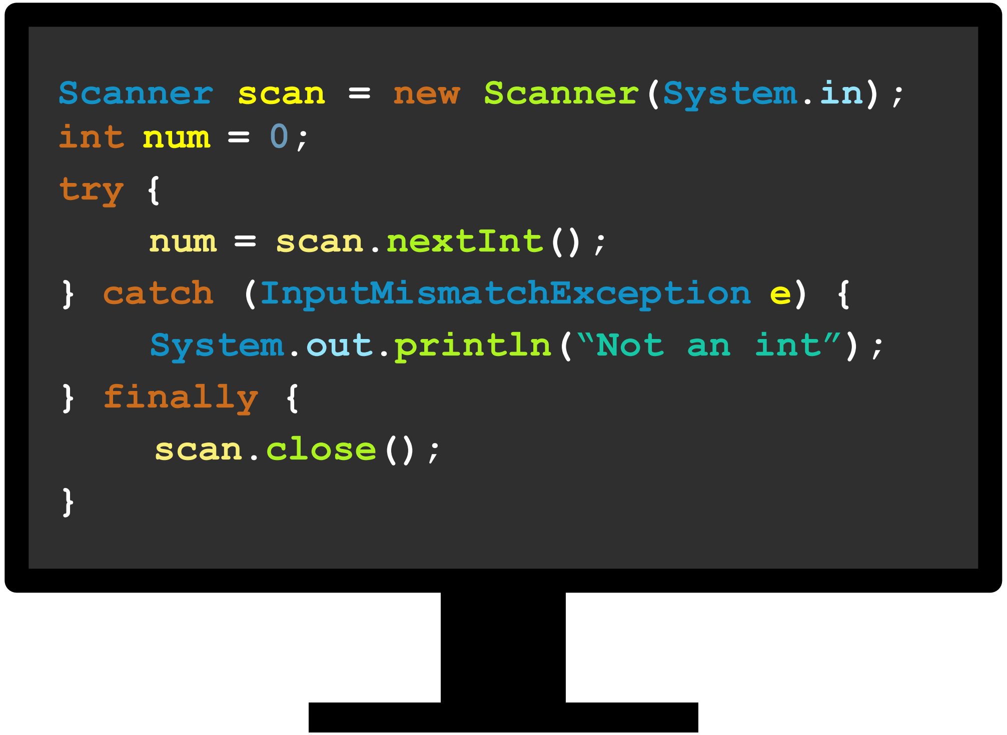
No exception is thrown, entire try block runs

```
try {
    <try-statement-1>
    ...
    <try-statement-2>
    ...
    <try-statement-3>
    ...
} catch(Exception1 e) {
    ...
} catch(Exception2 e) {
    ...
} catch(Exception3 e) {
    ...
}
<statement1>
<statement2>
...
```

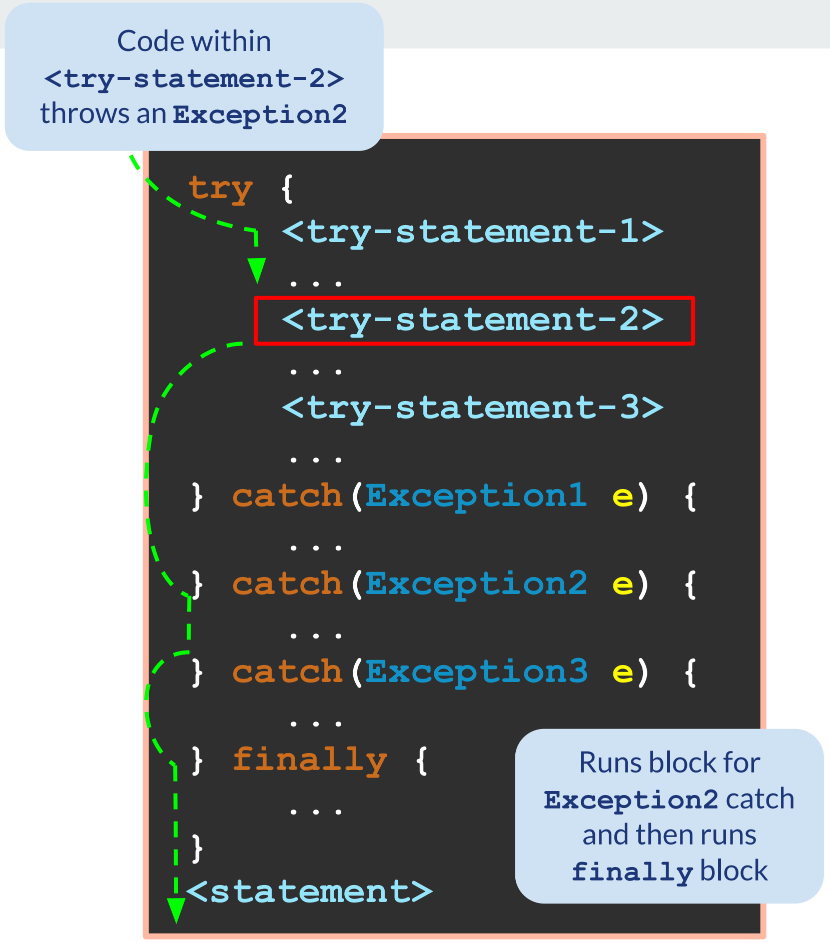
No catch block is runs and moves on to rest of code

Finally Block

- ➔ Cases where code should be executed whether an Exception occurs or not
- ➔ Code within a **finally** block is always executed
    - ◆ <u>Exception</u>: program exits by *System.exit()* or by fatal error aborts process



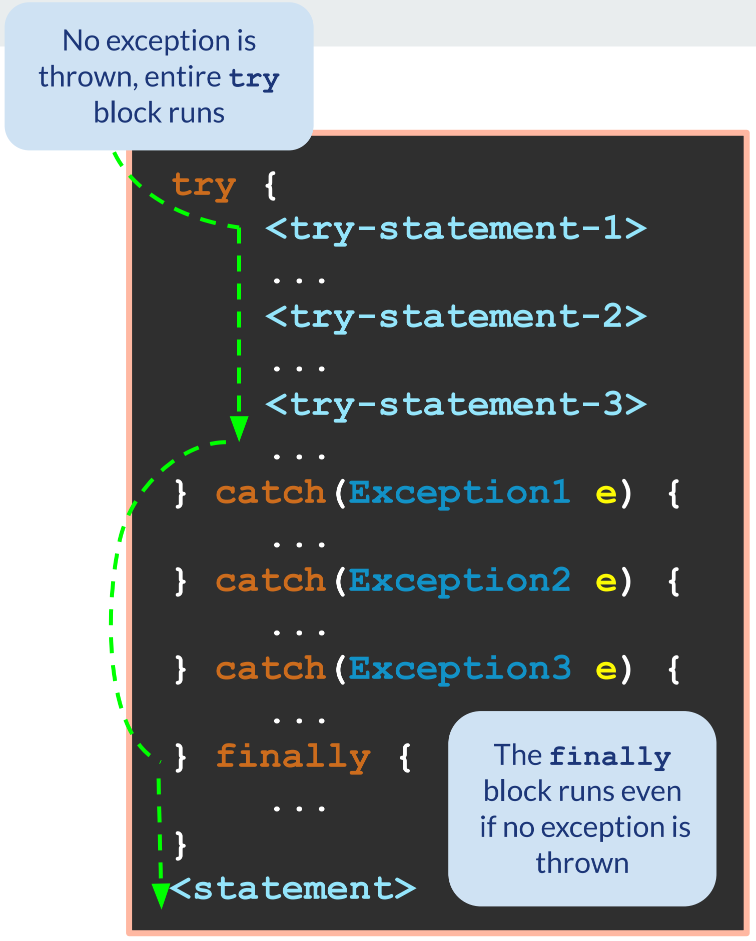
```
Scanner scan = new Scanner(System.in);
int num = 0;
try {
    num = scan.nextInt();
} catch (InputMismatchException e) {
    System.out.println("Not an int");
} finally {
    scan.close();
}
```



Code within **`<try-statement-2>`** throws an **`Exception2`**

```
try {
    <try-statement-1>
    ...
    <try-statement-2>
    ...
    <try-statement-3>
    ...
} catch(Exception1 e) {
    ...
} catch(Exception2 e) {
    ...
} catch(Exception3 e) {
    ...
} finally {
    ...
}
<statement>
```

Runs block for **`Exception2`** catch and then runs **`finally`** block



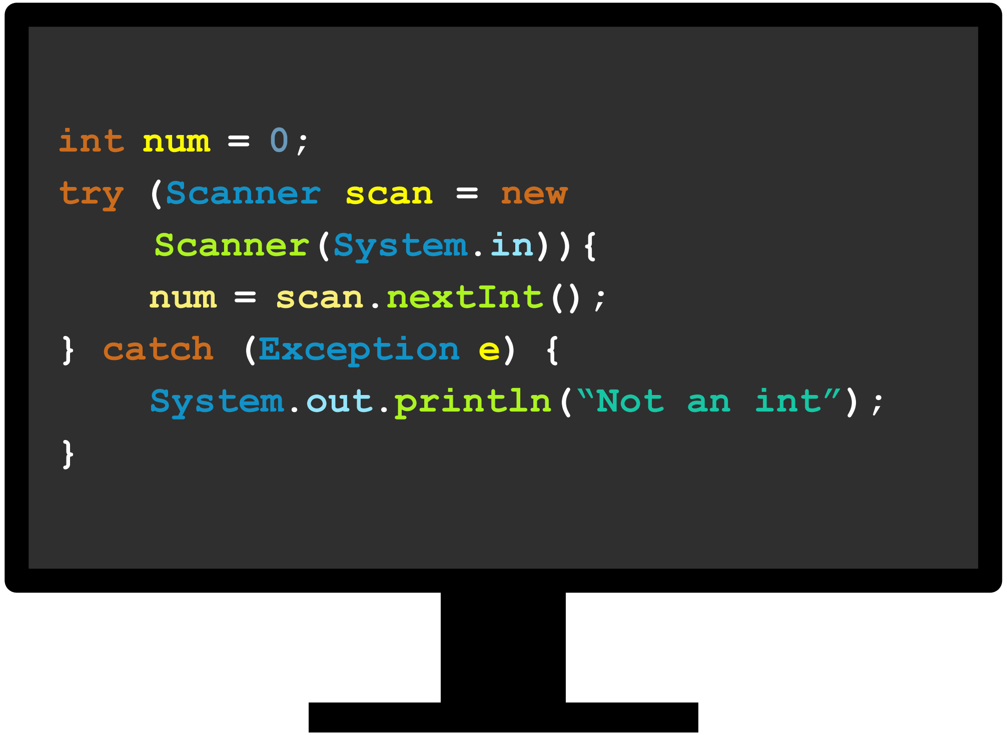
No exception is thrown, entire **`try`** block runs

```
try {
    <try-statement-1>
    ...
    <try-statement-2>
    ...
    <try-statement-3>
    ...
} catch(Exception1 e) {
    ...
} catch(Exception2 e) {
    ...
} catch(Exception3 e) {
    ...
} finally {
    ...
}
<statement>
```

The **`finally`** block runs even if no exception is thrown

Try With Resources

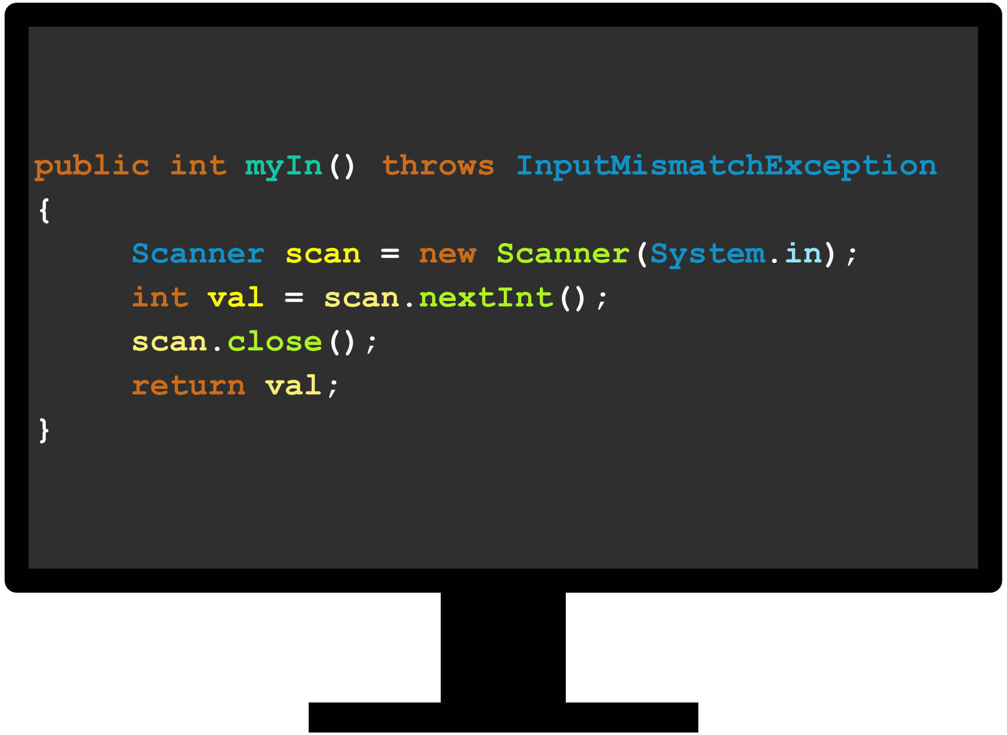
- A **try-with-resources** block automatically closes resources opened within the try
- Required resources must be “passed” into try block like a parameter
- Added in Java 7



```
int num = 0;
try (Scanner scan = new
     Scanner(System.in)){
     num = scan.nextInt();
} catch (Exception e) {
     System.out.println(“Not an int”);
}
```

Propagation

- ➔ Exceptions don't have to be handled in method they occur
- ➔ The **throws** keyword in method header propagates exception up call stack



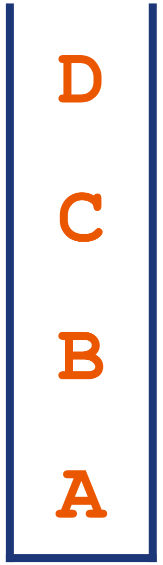
```
public int myIn() throws InputMismatchException
{
      Scanner scan = new Scanner(System.in);
      int val = scan.nextInt();
      scan.close();
      return val;
}
```

Propagates

Propagates

Catches

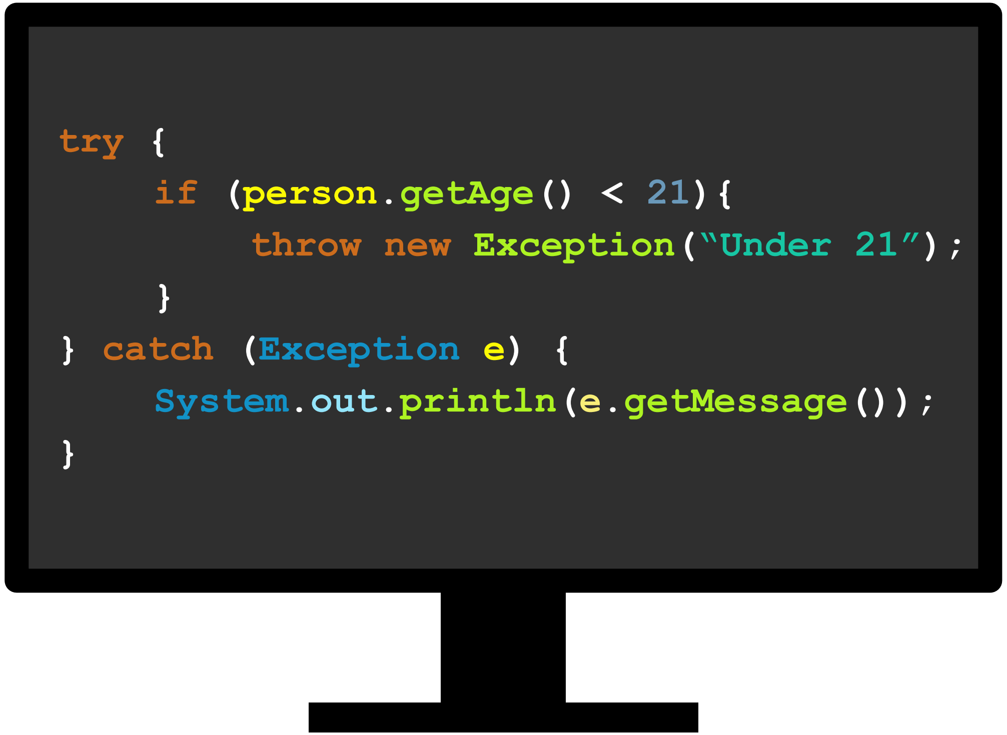
```
public void D() throws Exception {
      if (cond) {
            throw new Exception();
      }
      ...
}

public void C() throws Exception {
      D();
      ...
}

public void B() throws Exception {
      C();
      ...
}

public void A() {
      try {
            B();
            ...
      } catch(Exception e) {
            ...
      }
}
```



Stack Trace

Throwing Exceptions

- ➔ Exceptions can be thrown directly via **throw** keyword
- ➔ Are objects so can be instantiated with the **new** keyword
- ➔ Have a constructor with string parameter
    - ◆ Sets the message that describes the exception thrown



```
try {
     if (person.getAge() < 21){
          throw new Exception("Under 21");
     }
} catch (Exception e) {
     System.out.println(e.getMessage());
}
```

Compile Time vs Runtime Exceptions

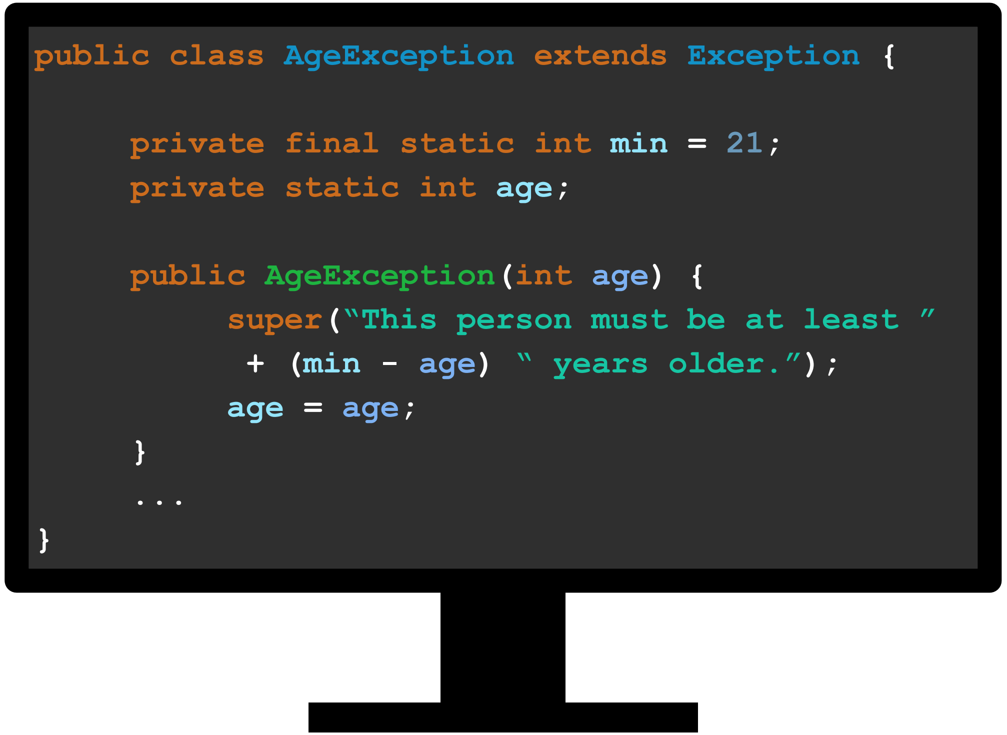
- ➔ **Checked** or **Compile Time** Exceptions are *checked* at compile time
  - ◆ Must be handled with a try/catch block or throws keyword
- ➔ **Unchecked** or **Runtime** Exceptions occur at runtime
  - ◆ Handling with a try/catch or throws keyword is optional
  - ◆ Methods containing them are implicit propagators

```
public void checked throws Exception {
      ...
      throw new Exception();
      ...
}
```

```
public void unchecked {
      ...
      throw new RuntimeException();
      ...
}
```

Custom Exceptions

- **Programmer Defined Exceptions** can contain custom data members to give more details about an exception
- Created by extending the Exception class



```
public class AgeException extends Exception {

     private final static int min = 21;
     private static int age;

     public AgeException(int age) {
          super(“This person must be at least ”
           + (min - age) “ years older.”);
          age = age;
     }
     ...
}
```

Assertions

Assertions

- The **assert** keyword is used to check if a condition is true
  - used on conditions that must be true if code is working correctly
- If assertion false, **AssertionError** thrown and program stops

```
assert condition;
```

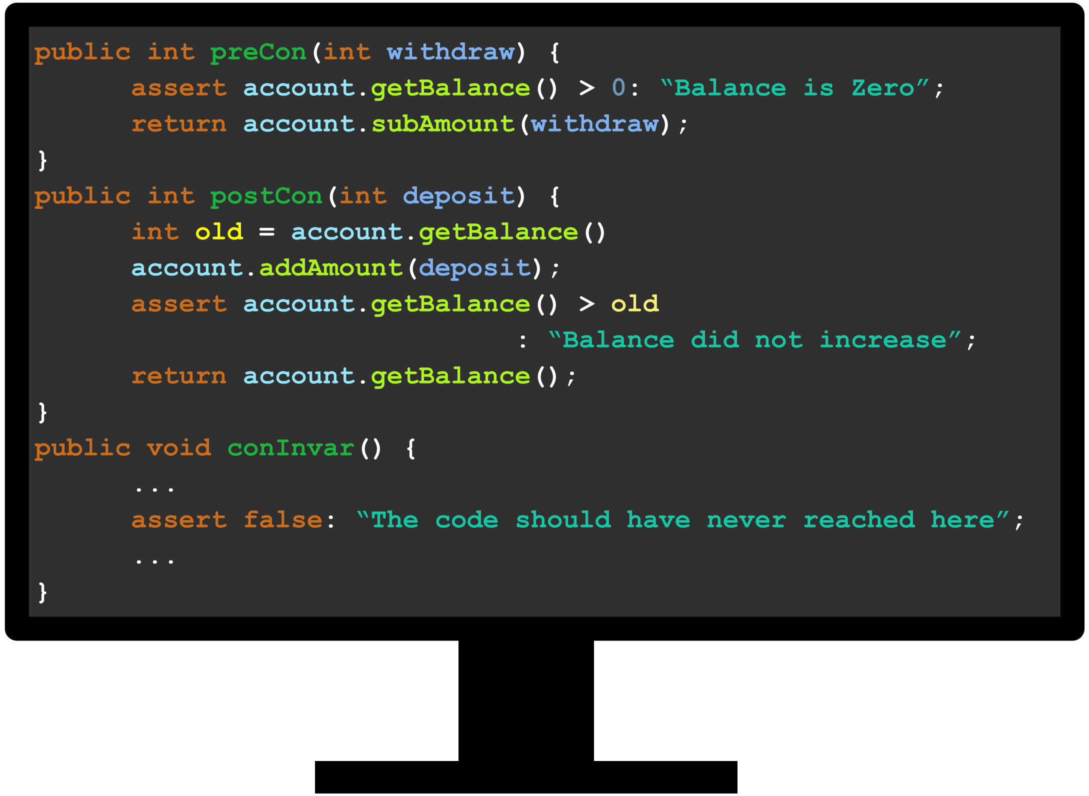
- Message may be displayed in the console after assertion using following syntax:

```
assert condition : “Condition is false”;
```

- By default, Java skips assertions on Runtime
  - Run Java file with **-ea** in the terminal will check assertions

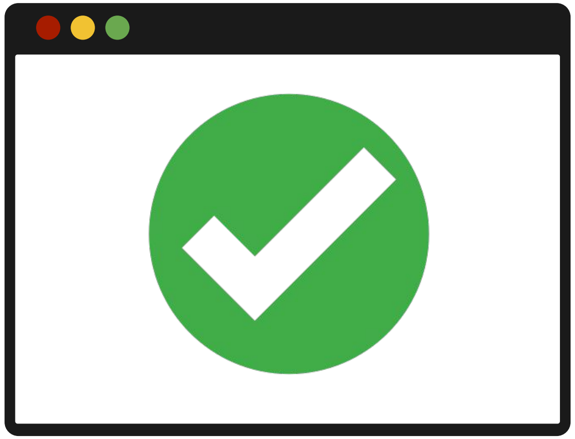
Uses of Assertions

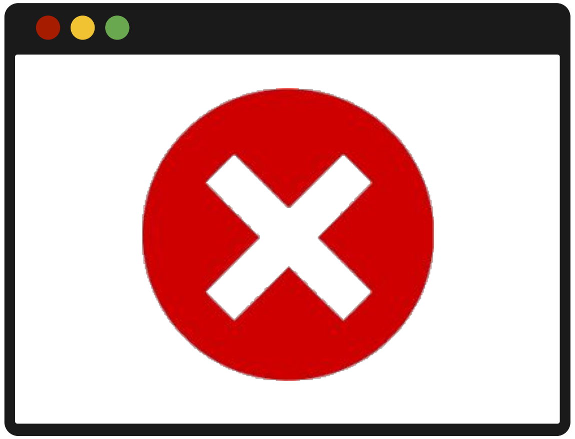
- ➔ **Precondition Assertions** check a condition before executing a method
- ➔ **Postcondition Assertions** check a condition after a method is executed
- ➔ **Control-Flow Invariant Assertions** check that the flow of control is guaranteed to pass a certain way in a program



```
public int preCon(int withdraw) {
      assert account.getBalance() > 0: “Balance is Zero”;
      return account.subAmount(withdraw);
}
public int postCon(int deposit) {
      int old = account.getBalance()
      account.addAmount(deposit);
      assert account.getBalance() > old
                              : “Balance did not increase”;
      return account.getBalance();
}
public void conInvar() {
      ...
      assert false: “The code should have never reached here”;
      ...
}
```

JUnit and Test Driven Development





FIN

Cognixia™