# Part 1

- Interfaces
- File I/O

# Inheritance Revisited

Classes can inherit (and thereby **extend**) both concrete and abstract classes.

A Child Class ***IS-A*** (subtype of) Parent Class.

- E.g., Dog is a subtype of (or **IS-A**) Animal.

**But a class can only extend at most one class.**

*When/Why might this be a problem?*

# Java Interfaces

While a class can **extend** only **one parent class**, a class can can **implement multiple interfaces.**

# What is an Interface?

- An **interface** is basically a group of public methods that are **declared** but **not implemented**.
- As such, it is **more abstract** than an **abstract class**.
- The **interface** defines how an object interacts with the outside world.
- Objects that implement the same interface can be grouped much like objects of the same class.
- But unlike classes, interfaces **don't** contain attributes (save for public **static final** attributes, which are effectively constants). *We'll talk more about the **final** keyword shortly, too.*

# Interfaces and ADTS

For example, **collections** are ADTs that support similar behaviours, like iterations.

One kind of collection is an **array**, which is a container object that holds a fixed number of values of a single type:

*int[] anArray = new int[10];*

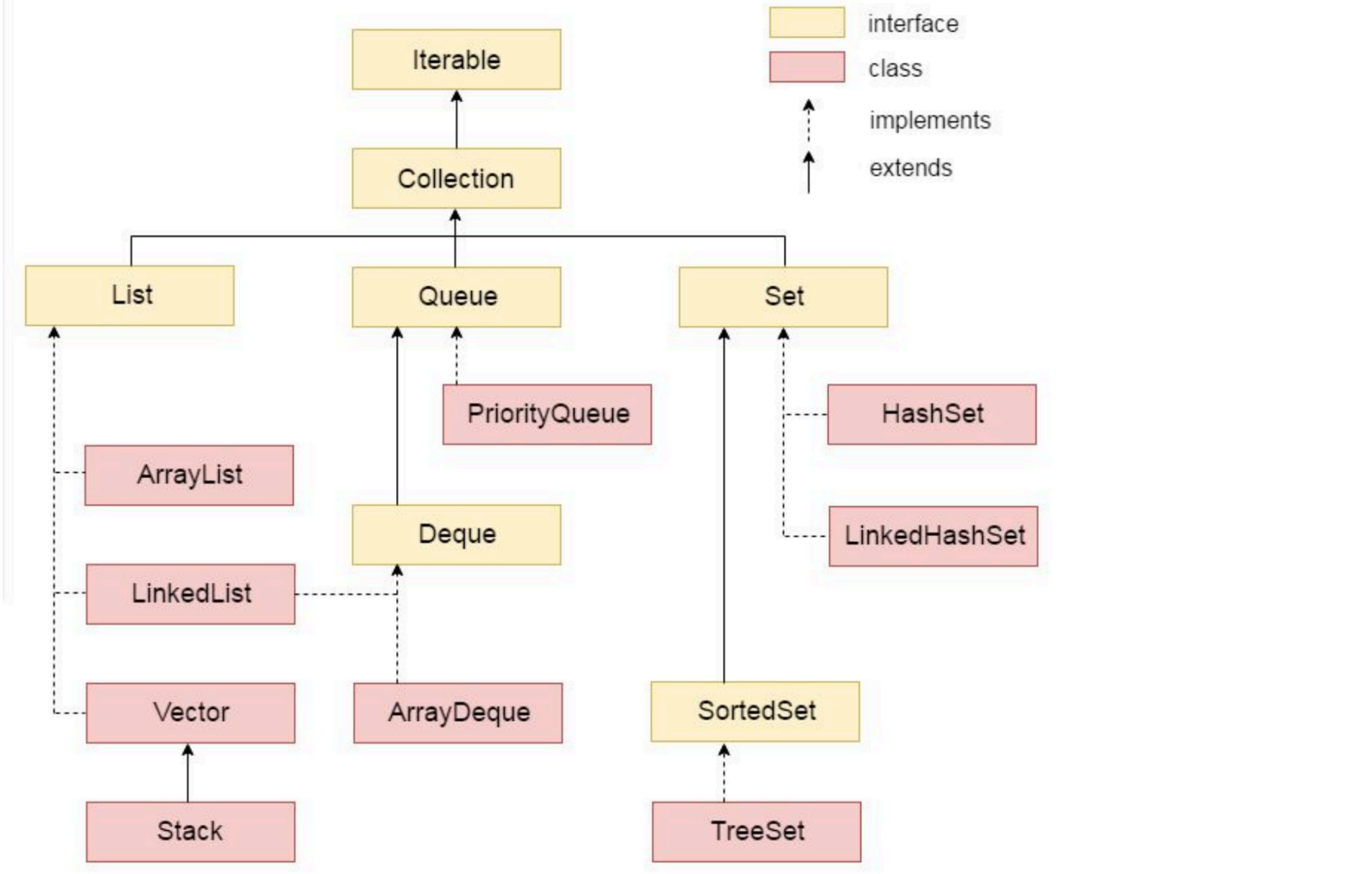
We might want **collections of objects**, however, and to support ways of interacting with these (i.e. to iterate over them).

Shared behaviours among collections are realized by **implementing interfaces** called **Iterable** and **Iterator**.

Other ADTs share (e.g. Maps) share common methods; **interfaces** are used to establish them.

# Java Collections

Let us see the hierarchy of collection framework.The **java.util** package contains all the classes and interfaces for Collection framework.




Iterable Interface: https://docs.oracle.com/en/java/javase/18/docs/api/java.base/java/lang/Iterable.html

Iterator interface: https://docs.oracle.com/en/java/javase/18/docs/api/java.base/java/util/Iterator.html

# Methods in the Iterator Interface

**remove():** **r**emoves from the underlying collection the last element returned by this iterator.

**hasNext():** this returns if the iterator is not at the end of the collection.

**next():** this returns the current item and moves the iterator one step forward.

***Methods in the Iterable interface:***

**iterator():** returns an object that can iterate over the collection

Iterable Interface: https://docs.oracle.com/en/java/javase/18/docs/api/java.base/java/lang/Iterable.html

Iterator interface: https://docs.oracle.com/en/java/javase/18/docs/api/java.base/java/util/Iterator.html

# Example: ArrayList implements Iterable

```java
List<String> collection = new ArrayList<String>();

collection.add("zero");
collection.add("one");
collection.add("two");

Iterator<String> iterator = collection.iterator();

// while loop
while (iterator.hasNext()) {
    System.out.println("value= " + iterator.next());
}

// for loop
for (iterator = collection.iterator(); iterator.hasNext();) {
    System.out.println("value= " + iterator.next());
}

// for-each loop
for (String s : collection) {
    System.out.println("value= " + s); //
}
```

# A quick aside on Generics ….

When we syntax like this:

```
Iterator<String> iterator = collection.iterator();
```

It means that “**iterator**” is an **Iterator object** that can iterate over collections of **Strings**.
But we could make iterators that iterate over many kind of collections using syntax like:

*Iterator<Integer> iterator = ….*

*Iterator<Animal> iterator = …*

and so on.

This is because **Iterator** is a **Generic class,** meaning it can be related to collections of many types.

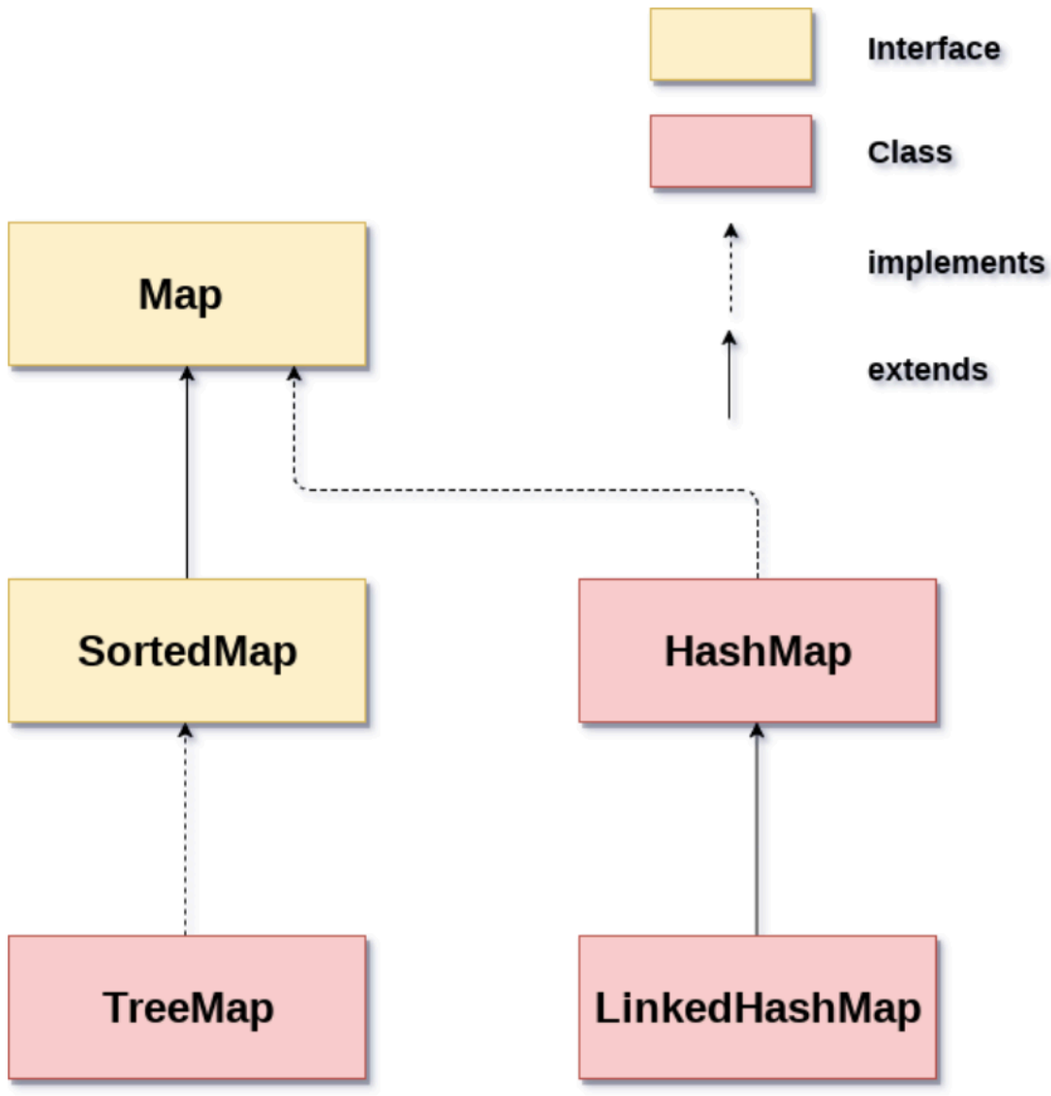
***But don’t worry too much about this now; we will talk more about Generics next time.***

# More Interfaces and ADTS

Other ADTs are maps; common ways of interacting with these are to put key/value pairs into them and get values from them by key.

This interface that realizes these behaviours is called **Map**.

# Java Maps implement Interfaces




https://docs.oracle.com/en/java/javase/18/docs/api/java.base/java/util/Map.html

https://docs.oracle.com/en/java/javase/18/docs/api/java.base/java/util/Set.html

# Methods in the Map Interface

**forEach()** performs an action on each entry in the map until all entries have been processed

**get()** returns a value for a key.

**put()** associates a value with a key.

We will return to Maps later, as maps depend on the ability to assess **equality** of Objects and to map Objects onto unique hash codes.

https://docs.oracle.com/en/java/javase/18/docs/api/java.base/java/util/Map.html

# Custom Interface Example

You can of course make your own interfaces, too.

```
public interface Shape{
   public double calculateArea();
}
public class Rectangle implements Shape{
   double length, width;
    public double calculateArea(){
       return length * width;
    }
}
public class Circle implements Shape{
    public double radius;
    public double calculateArea(){
        return Math.PI*radius*radius;
    }
}
```

# Polymorphism of Objects

Objects that are **descended from the same parent** can be operated upon similarly in Java code.

Objects that **share a common interface** can also be similarly grouped in Java code.

The ability for Java to operate on diverse objects that share common features is called **Polymorphism**.

**Polymorphism** is derived from Greek; it means “many forms”.

# Basic File I/O

- Reading and writing from and to files can generate **IOExceptions**, which require some handling. The typical syntax surrounding File I/O operations will be as follows:

```
try
{
   //File I/O operations here
}
catch (IOException e)
{
  //Handle exception here, as you please
  System.out.println("Exception: " + e.getMessage());
}
```

# I/O Streams

- Input and output in Java is accomplished by classes called **streams.**

- **Input streams** move bytes of data into a program. **Output streams** move data in the opposite direction.

- **Streams you may know already:**

- **System.in**
  - An InputStream object, usually to handle input from the keyboard

- **System.out**
  - a buffered PrintStream object, usually tied to the active window

- **System.err**
  - an unbuffered PrintStream object, usually tied to the console window

# Objects to Support Basic File I/O

- Several Java Objects exist to support File I/O Operations. For example:

- **Scanner Objects**
  - These are simple text scanners that can be linked to a **File** or **InputStream**
  - A **Scanner** breaks input into tokens (whitespace is default delimiter).
  - **Tokens** may then be converted into values of different types using various '*next*' methods

- **FileWriter Objects**
  - These objects write text to **output streams** using a buffer

# Basic File I/O Example

```java
private static void readText(String fileName) {
    try {
        File myObj = new File(fileName);
        Scanner myReader = new Scanner(myObj);
        while (myReader.hasNextLine()) {
            String data = myReader.nextLine();
            System.out.println(data);
        }
        myReader.close();
    } catch (IOException e) {
        System.out.println("An error occurred.");
        e.printStackTrace();
    }

}

1 usage
private static void writeText(String fileName, String text,
                              boolean appendMode) {
    try {
        FileWriter myWriter = new FileWriter(fileName, appendMode);
        myWriter.write(text);
        myWriter.close();
        System.out.println("Successfully wrote to the file.");
    } catch (IOException e) {
        System.out.println("An error occurred.");
        e.printStackTrace();
    }
}
```

UNIVERSITY OF TORONTO MISSISSAUGA

# Basic CSV File I/O

```java
/**
 * Populates the records map from the file at path filePath.
 * @param filePath the path of the data file
 * @throws FileNotFoundException if filePath is not a valid path
 */
1 usage
public void readFromCSVFile(String filePath)
        throws FileNotFoundException {

    // FileInputStream can be used for reading raw bytes, like an image.
    Scanner scanner = new Scanner(new FileInputStream(filePath));
    String[] record;
    Student student;

    while(scanner.hasNextLine()) {
        record = scanner.nextLine().split( regex: ",");
        student = new Student(record[0].split( regex: " "),
                record[1], record[2], record[3]);
        students.put(student.getId(), student);
    }
    scanner.close();
}
```

# Part 2

- Shadowing, Overriding, Overloading, Name Resolution and Comparisons

UNIVERSITY OF TORONTO MISSISSAUGA

# Some Notes on Style

You've likely noticed some style conventions for Java.  For example:

- We use **camelCase** to name attributes and methods, generally.

- Class names are typically a **noun phrases** starting with a **capital letter**.

- Method names are typically **verb phrases** starting with **lower case**.

- Most variables are **noun phrases** starting with **lower case.**

- **Constants** may be uppercase and pothole_case (e.g. MAX_ENROLMENT)

# The Keyword Super and Overriding

We have discussed **access modifiers.**

An object of a ***Child*** class can access ***Parent's*** variables and methods if and only if they are **public** or **protected**.

A *Child* ***can't*** access those declared as **private** or **default**.

In a *Child* class (or subclass), **super** refers to a method or variable of the parent class, I.e.:

**super.variable:** accesses a **variable** of the parent class

**super.method():** accesses a **method** of the parent class

**super(arguments):** accesses the parent's **constructor**

The "**super**" keyword will only refer to a class' immediate parent, not to its grandparent, great grandparent, etc.

# The Keyword Super and Overriding

In a *Child* class, if there is no explicit call to the parent's constructor, the parent's default constructor with no arguments will still be called, i.e., **super().**

Two phenomena relate the methods and variables of *Parent* and *Child* classes; these are:

**overriding** and **shadowing**

# Overriding a method

**Overriding** happens when a *Child* class **re-implements a method** that exists in its *Parent*.

The *Child* method will then **take precedence** over the *Parent* method when the compiler resolves calls to **child.method().**

# Overriding Example

```
public class Peanut {
    1 usage   1 override
    public void bitePeanut() {
        System.out.println("What a crunchy peanut.");
    }
}

public class CircusPeanut extends Peanut {
    1 usage
    public void bitePeanut() {
        System.out.println("What a squishy peanut.");
    }
}
```

```
public class Main {
    public static void main(String [] args) {
        CircusPeanut p = new CircusPeanut();
        p.bitePeanut();
    }
}
```

*What will be printed as a result of our call to p.bitePeanut()?*

# The final keyword

If you **don't** want a *Parent* method to be overridden by any *Child* class, you can declare the methods as **final.**

The **final keyword** is like the static keyword in that it can be applied to **class variables** and **methods** of a class.

When related to a **class variables**, it means its value can't be modified; it is essentially, a constant. *Final attributes must be initialized.*

```
final int THRESHOLD = 5;
// Final variable

static final double PI = 3.141592653589793;
// Final static variable PI
```

When related to a class **method**, it means the method cannot be overridden by subclasses.

```
class ChessAlgorithm {
   public final ChessPlayer getFirstPlayer() {
        return ChessPlayer.WHITE;
    }
}
```

# Shadowing a Variable

**Shadowing** happens when a *Child* class **re-declares a variable** that exists in *Parent*.

The *Parent's* variable then gets **shadowed** by the *Child's* variable

This is confusing, and should almost never be used.  If you do it, make sure to ask yourself why.

```java
public class ShadowingExample{

    String name = "Xavier";
    int age = 21;
    public String toString(){
        String name = "Vishnu";
        int age = 22;
        return "Name: "+ name + "; age: "+ age;
    }

    public static void main(String args[]){
        System.out.println(new ShadowingExample());
    }
}
```

*What will be printed when we run the main method?*

# Overloading and constructors

Overloading is like both shadowing and overriding in that it **redefines something that has been declared**, but overloading has differences.

**Overloading** happens when **multiple methods share the same name**, but have **different parameterizations.** We saw this last week with **constructors**:

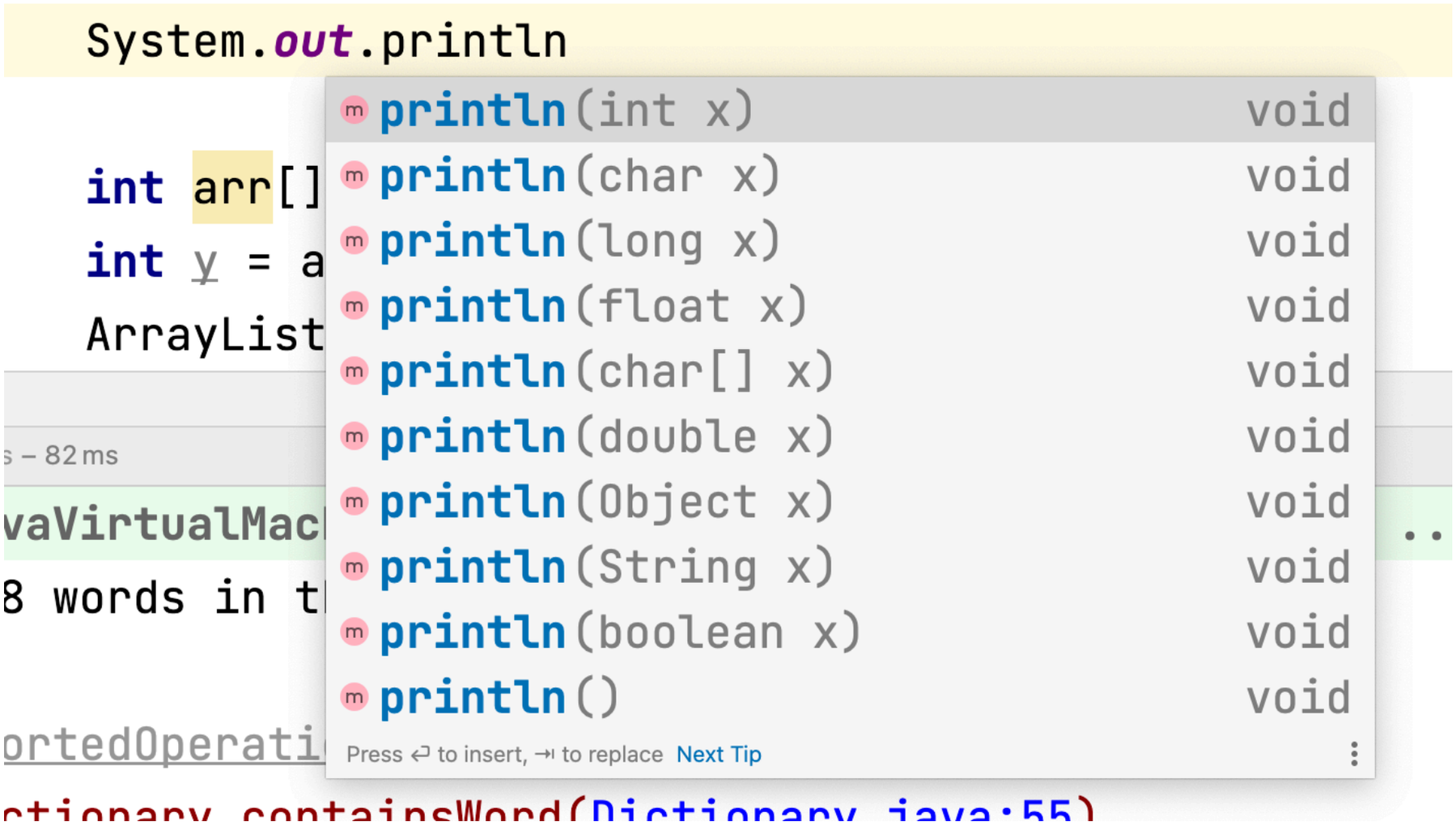
```
public Balloon(String color)
{
   this.amount = 0;
   this.capacity = 10;
   this.popped = false;
   this.color = color;
}

public Balloon(String color, int capacity)
{
   this.amount = 0;
   this.capacity = capacity;
   this.popped = false;
   this.color = color;
}

Balloon left = new Balloon("red");
Balloon right = new Balloon("red", 15);
```

# Overloading more generally

Note that you can overload other methods, too. For example, **System.out.println()** is overloaded and contains many parameterizations, which you can access through your IDE:



# Name Resolution in Java

Calling a method in Java often looks like this:

**objectName.method(arguments)** or **objectName.variable**

The compiler must work to relate these method and variable names to one of many possible candidates, should there be any **over-riding**, **over-loading** or **shadowing** in our code.

To resolve method or attribute names, Java will look for the **most specific** method or attribute.

It will seek to resolve the name in the **child** class, and if it can't it will look up the object hierarchy until the name can be resolved (or throw an error).

# Names can be defined dynamically

When you have multiple methods with the same name, how does Java decide which one to invoke?

```
1 usage
public int hello ( int x){
    return 1;
}

1 usage
public int hello (String s){
    return 2;
}

//the functions called here are decided
//at compile time! And based on type information.
int r1 = hello( x: 3);
int r2 = hello( s: "3");
```

```
Person p = new Person();

int r = (int) (Math.random() * 5 + 1);

if (r > 3) {
    p = new Student();
} else {
    p = new Doctor();
}

//the function called is determined
//dynamically! We don't know until runtime.
p.hello();
```

Names of variables and methods can be defined **statically** or **dynamically**, i.e. at runtime.

# More Overriding Examples

Remember also that **all** Objects inherit the following methods:

**equals**

**hashCode**

**toString**

When you override **toString**, you determine what will be displayed when an object is printed to the console.

We touched briefly on the relationship between **hashCode** and **equals** when we discussed **Maps**.

You will talk more about between **hashCode** and **equals** during lab.

# Overriding Equals requires Care

Overriding equals requires some care!

**equals()** must be:

- **reflexive**: an object must equal itself
- **symmetric**: x.equals(y) must return the same result as y.equals(x)
- **transitive**: if x.equals(y) and y.equals(z) then also x.equals(z)
- **consistent**: the value of equals() should change only if a property that is contained in equals() changes

# Overriding Equals requires Care

Consider what happens if we make a new class that extends Cash, e.g.

```
class Coin extends Cash
```

If Coin ***also*** overrides equals, what might be the result of the following statements?

```
Cash money = new Cash(25, “CAN");

Coin quarter = new Coin(25, “CAN");

assert money.equals(quarter) == quarter.equals(money);
```

# Overriding hashCode

**hashCode()** returns an integer that represents the current instance of the class.

We must calculate this value consistent with the definition of equality for the class.

Thus if we override the **equals()** method, **we also have to override hashCode().**

# Overriding hashCode

Overriding **hashCode()** demands we consider:

- **internal consistency:** the value of hashCode() may only change if the equals() property changes
- **equals consistency:** objects that are equal to each other must return the same hashCode
- **collisions:** unequal objects may have the same hashCode

# Overriding hashCode

```java
class Team {
    3 usages
    String city, name;

    4 usages
    public Team(String city, String name) {
        this.city = city;
        this.name = name;
    }

    @Override
    public final boolean equals(Object o) {
        if (o == this)
            return true;
        if (!(o instanceof Team))
            return false;
        return this.city.equals(((Team) o).city) && this.name.equals(((Team) o).name);
    }
}
```

```java
public class Main {

    public static void main(String[] args) {
        Map<Team,String> leaders = new HashMap<>();
        leaders.put(new Team( city: "Vancouver",  name: "Moondogs"), "Anne");
        leaders.put(new Team( city: "Toronto",  name: "Leopards"), "Xavier");
        leaders.put(new Team( city: "Calgary",  name: "Koalas"), "Nima");

        Team myTeam = new Team( city: "Vancouver",  name: "Moondogs");
        String myTeamLeader = leaders.get(myTeam);
        System.out.println(myTeamLeader);
    }
}
```

Will running main assign “Anne” to myTeamLeader?

Example from https://www.baeldung.com/java-equals-hashcode-contracts

# Overriding hashCode

```java
@Override
public final int hashCode() {
    int result = 17;
    if (city != null) {
        result = 31 * result + city.hashCode();
    }
    if (name != null) {
        result = 31 * result + name.hashCode();
    }
    return result;
}
```

Overriding **hashCode** within the Team Class can fix this issue.  Using multipliers of prime numbers **reduces the possibility of collisions**, as we are reducing the possibility that multiple objects will map onto the same number.

# Equality of Objects fuels Comparisons

Object **equality** can fuel **comparisons** of objects in our code.

The **Java Comparable Interface,** from the **java.lang** package, helps.

The interface provides a method (**compareTo**) that lets us compare objects to one another. We will implement this when we use the interface. Objects that are comparable can be **sorted** as follows:

*Collections.sort(myComparableCollection)*

Note that **compareTo** compares **objects** while operators '<' and '>' compare **primitive types.**

```
public interface Comparable<T> {

    int compareTo(T other);

}
```

# Comparable Objects

The **compareTo()** method takes a single object as parameter and returns an int value:

- **A positive value** (1 or larger) signals that the object the compareTo() is called on is larger than the parameter object.
- **A value of zero** (0) signals that the two objects are equal.
- **A negative value** (-1 or smaller) signals that the object the compareTo() methods is called on is smaller than the parameter object.