2.1

Classes and Objects

Classes and Objects

CLASS	OBJECT (aka INSTANCE)
A class is a blueprint from which you can create the instance, i.e., objects.	An object is the instance of the class, which helps programmers to use variables and methods from inside the class.
Classes have logical existence.	Objects have a physical existence.
A class doesn't take any memory spaces when a programmer creates one. (The “idea” of a cat)	An object takes memory when a programmer creates one. (A real, live cat)
The class has to be declared only once. (i.e. “Cat”)	Objects can be declared several times depending on the requirement. (i.e. “Buttons”, “Mr. Bigglesworth”, “Garfield”)

Class

`String`

Object

```
String greeting = “Hello world!”;

String favoriteClass = “AP Computer Science”;

String bestTeacher = “Ms. Molina”;
```

Attributes (instance variable) and Behaviors (methods)

An **attribute** or **instance variable** is data the object knows about itself. For example a turtle object knows the direction it is facing or its color.

A **behavior** or **method** is something that an object can do. For example a turtle object can go forward 100 pixels.

2.2

Constructors

Dog

```
public class Dog {
    private String breed;
    private int age;
    private String color;

    public Dog() {
        breed = “pug”;
        age = 3;
        color = “brown”;
    }

    public Dog(String a, int b, String c) {
        breed = a;
        age = b;
        color = c;
    }
}
```

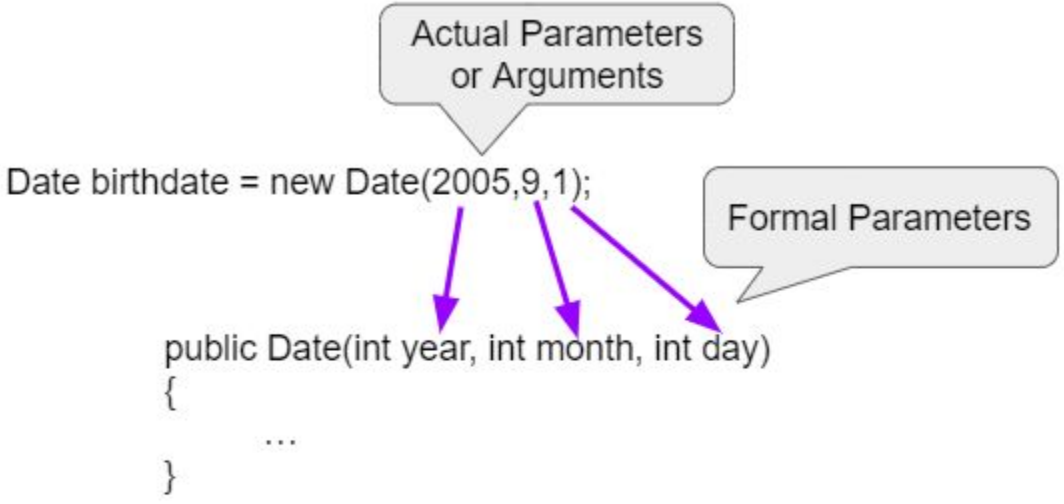
Default constructor

An overloaded constructor that takes parameters

```
World world1 = new World(); // creates a 640x480 world

World world2 = new World(300,400); // creates a 300x400 world


Turtle t1 = new Turtle(world1);

Turtle t2 = new Turtle(50, 100, world1);
```

Notice here that the order of parameters matters

Formal and Actual Parameters



```
Date birthdate = new Date(2005,9,1);

public Date(int year, int month, int day)
{
        ...
}
```

Date(2005,9,1) - This is **call by value** which means that copies of the actual parameter values are passed to the constructor. These values are used to initialize the object’s attributes.

2.3

Methods

What is a method?

A method is an **action** defined for a class that all instances of that class (objects) will support.

Methods can:

1. Provide access to an attribute of an instance
2. Update an attribute of an instance
3. Do something new and interesting with the information stored in an instance

Methods are called using the "." operator, which allows access to the public methods of a class.

An example… what number is printed?

```java
// Dog.java

public class Dog {
      private int age; // an attribute

      public Dog(int dogAge) { // constructor
            age = dogAge;
      }

      // a method that updates an attribute.
      // returns nothing
      public void makeOlder(int years) {
            age += years;
      }

      private int dogYears() { // an internal method
            return 7*age;
      }

      // a method that retrieves an attribute
      public int getAge() {
            return dogYears();
      }
}
```

```java
// TestDog.java

public class TestDog {

      public static void main(String[] args) {

            Dog goodBoy = new Dog(5);
            goodBoy.makeOlder(2);
            int age = goodBoy.getAge();
            System.out.println(age);
      }
}
```

Method declarations

Method declarations, such as `public void makeOlder(int years) { … }`

1. **Define whether the method is accessible to the outside world** (public / private)
    a. Public methods are available externally (e.g. `goodBoy.getAge()`) while private methods are not (calling `goodBoy.dogYears()` in `main` will cause an error)
2. **Determine what the method returns**
    a. Void methods return nothing
    b. String methods promise to return Strings, int methods to return ints
3. **Defines the variables (parameters) passed to the method**
    a. To be described in the next section
4. **Define the body of the method**
    a. The body is the statements of code that will execute when the method is called.

Abstraction – keeping things simple

One of the core concepts in computer science is **abstraction.** Abstraction means that you only need to understand how to interact with an object–you **don't need to understand how the code is actually implemented behinds the scenes.**

E.g. as a user, I should be indifferent between the following implementations:

Option 1

```
private int dogYears() {
    return 7*age;
}

public int getAge() {
    return dogYears();
}
```

Option 2

```
public int getAge() {
    return (age + age +
            age + age +
            age + age +
            age);
}
```

Option 3

```
public int getAge() {
    return 7*age;
}
```

The power of abstraction

Abstraction accomplishes two things:

1. It keeps things simple, minimizing what you need to know to write a program
2. It makes it possible for the class owner to change the technical implementation of the method without impacting its use
   a. e.g., option 3 may be faster for a computer to calculate than option 2… the programmer may want to switch their implementation from 2 to 3. Abstraction means that the user won't notice a difference (besides faster code)

NullPointerException

A variable `Dog dog;` **points to** an instance of `class Dog`.

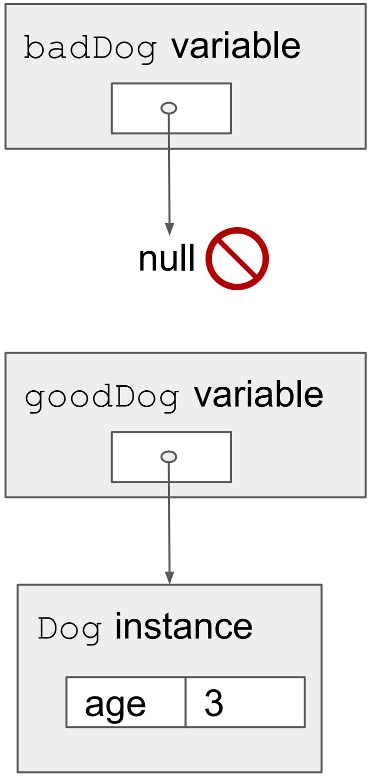
It starts out not pointing to any `Dog`, with the special value `null`.

You have to use `new Dog` to construct a Dog instance that the variable can point to.

If you don't **initialize** a variable to point to a `Dog` instance, and you try to call a method, `NullPointerException` will be thrown.

```
Dog badDog; // badDog == null
badDog.getAge(); // throws NullPointerException
```

```
Dog goodDog = new Dog(3); // goodDog points to instance
goodDog.getAge(); // no problem
```



Instance and static methods

Instance methods act upon instances of a class. We first create an instance and then call on one of its instance methods.

```
String t = "blue”;

t.substring(0,2); // "bl”
```

Static methods aren’t bound to a particular instance of a class. They are called by naming a class following by the dot operator:

```
String.valueOf(1234); // "1234”
```

So what does `public static void` mean?

2.4

Method Parameters

Use parameters!

```
// Person.java
public class Person {
     private String name;

     public Person(String personName) {
          name = personName;
     }

     // Greeting
     public void greet() {
          System.out.println(name + " says: Hello, world!");
    }

     // Greet a particular person
     public void greet(String otherName) {
          System.out.println(name + " says: Hello, " + otherName + "!”);
    }
}
```

Calling methods with parameters

```
// TestPerson.java

public class TestPerson {

    public static void main(String[] args) {

        Person amy = new Person("Amy");

        amy.greet("Ted"); // Prints "Amy says: Hello, Ted!"

        amy.greet("Thursday"); // Prints "Amy says: Hello, Thursday!"


        Person bob = new Person("Bob");

        bob.greet("Amy"); // Prints "Bob says: Hello, Amy!"

    }

}
```

Definitions

Formal Parameter (parameter) -The variable declared in the method header

```
public void greet(String name)
```

Actual Parameter (argument) - The value passed in a method call

```
amy.greet("Ted");
```

Method Overloading

Overloaded methods are two or more methods in the same class that have the same name but different parameters.

```
// Person.java
public class Person {
    private String name;

    public Person(String personName) {
        name = personName;
    }

    // Greeting
    public void greet() {
        System.out.println(name + " says: Hello, world!");
    }

    // Greet a particular person
    public void greet(String otherName) {
        System.out.println(name + " says: Hello, " + otherName + "!”);
    }

    // Greet a number
    public void greet(int aNumber) {
        System.out.println(name + " says: How are you, " + aNumber + "?");
    }
}
```

Calling Overloaded Methods

```
// TestPerson.java
public class TestPerson {
    public static void main(String[] args) {
        Person amy = new Person("Amy”);
        amy.greet(); // Prints "Amy says: Hello, world!”
        amy.greet("Ted”); // Prints "Amy says: Hello, Ted!”
        amy.greet(12); // Prints "Amy says: How are you, 12?”
    }
}
```

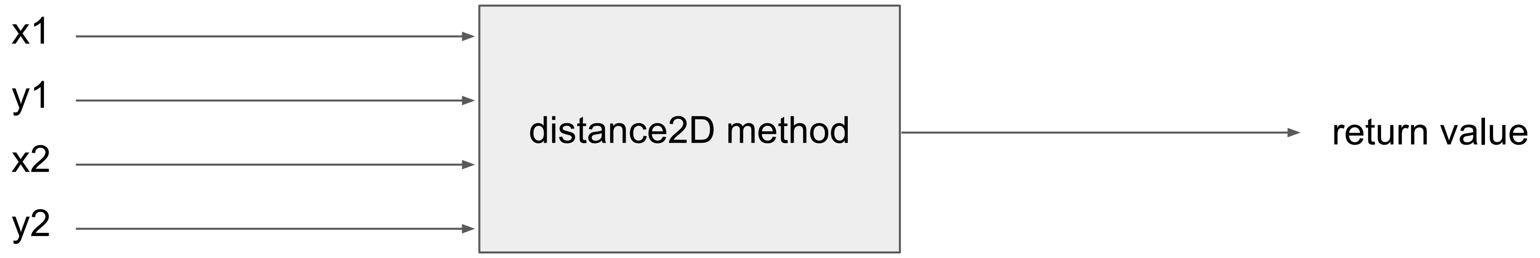
2.5

Return Values

Return Value

Methods can take **inputs** (“arguments” or “parameters”), and they can also spit out a single **output** (“return value”)

```
public double distance2D(double x1, double y1, double x2, double y2)
```



Methods are like functions… what is the difference between a function and a method?

Return Value

Methods that don’t **return** anything have a **void** return type

```
public void printGreeting(String name) {
    System.out.println("Hello " + name + "!");
}
```

Return Value

- The **type** of the return value must match what is declared in the method declaration
- Right:

```
public int getNumberTimesThree(int value) {
    return 3 * value;
}
```

- Wrong (why?):

```
public int getNumberTimesThree(int value) {
    return 3.0 * value;
}
```

- Q: Java only lets you return one value from a method. How might you return multiple pieces of data at once?

Getter and Setter Methods

In Java, you'll commonly find that classes declare `getXYZ` and `setXYZ` methods for their properties (instance variables).

```
public class TurtleTestGetSet
{
  public static void main(String[] args)
  {
      World world = new World(300,300);
      Turtle yertle = new Turtle(world);
      System.out.println("Yertle's width is: " + yertle.getWidth()); // Yertle's width is: 15
(this is the default width)
      yertle.setWidth(200);
      yertle.setHeight(200);
      System.out.println("Yertle's width is: " + yertle.getWidth()); // Yertle's width is: 200
(this is the width after we've set it to 200 2 lines above
      yertle.turnRight();
      world.show(true);
  }
}
```

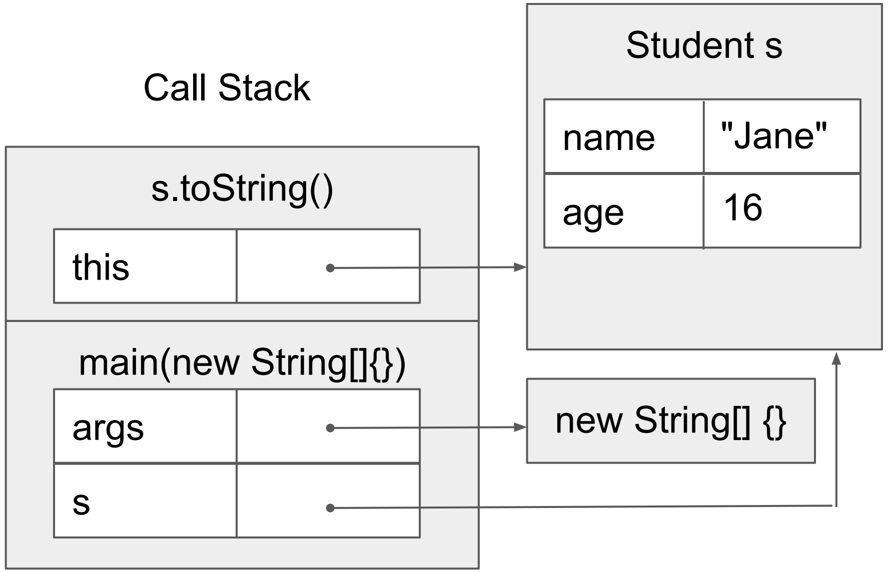
This is considered a **best practice**. Q: Why do you think that is?

toString Methods

- In Java, all objects can be represented in String form by defining a **toString** method.
- This can be useful for programmers to get a visual or textual representation of an otherwise abstract object.
- How can we make the `toString` method to the right more descriptive?
- What gets printed if you don't define a `toString` method?

```java
class Student {
  private String name;
  private int age;

  Student(String name, int age) {
    this.name = name;
    this.age = age;
  }

  public String toString(){
    return name;
  }
}

class HelloWorld {
    public static void main( String args[] ) {
      Student s = new Student("Jane",16);
      System.out.println(s.toString()); //”Jane”
      System.out.println(s); //Also ”Jane”
    }
}
```

`this` is a reference to the current object instance



```
class Student {
  private String name;
  private int age;

  Student(String name, int age) {
    this.name = name;
    this.age = age;
  }

  public String toString(){
    return name;
  }
}

class HelloWorld {
    public static void main( String args[] ) {
      Student s = new Student("Jane",16);
      System.out.println(s.toString()); //"Jane"
      System.out.println(s); //Also "Jane"
    }
}
```

VERY common pattern.

Q: What's the difference here between name and this.name?

Q: What do you think the next thing on the call stack will be on top of main?

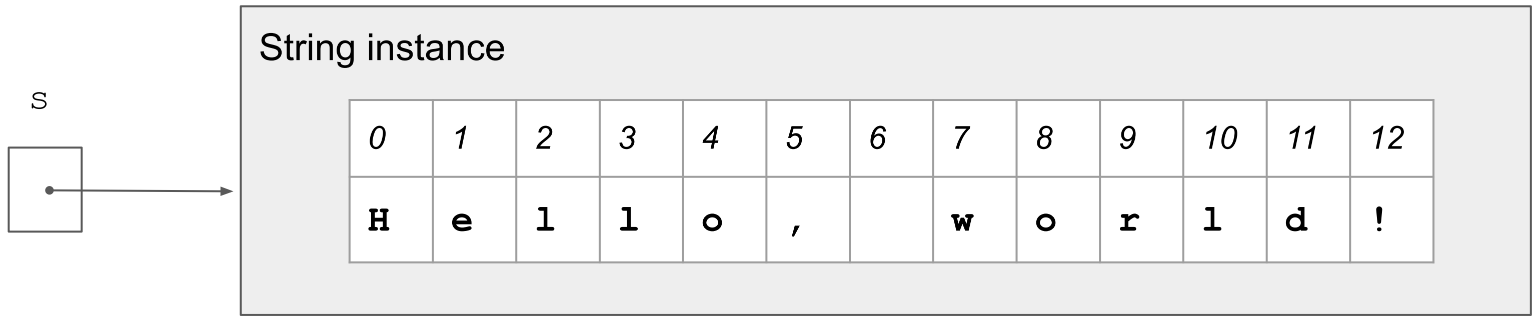
2.6

Strings

Strings

Strings in Java are instances of the `java.lang.String` class that hold sequences of characters (`a`, `b`, `c`, `$`, etc.)

```
String s = "Hello, world!";
```



Creating Strings

`String` is a class, so you can construct them with the `new` operator.

```
String s = new String(“Hello, world!”);
```

Strings can also be constructed using **string literals**:

```
String s = “Hello, world!”;
```

String comparison in Java

In many other languages, like JavaScript and Python, you can use the `==` operator to compare strings for equality.

In Java, the `==` operator compares object references, **not** what's in the referenced objects!

`s1.equals(s2)` is almost always what you want, not `s1 == s2`

String concatenation

Strings can be appended to each other to create a new string using the `+` or `+=` operator. This is also called **concatenation**.

In the expression `x + y`, `x` and `y` are the operands and `+` is the operator.

If x or y is a String, the other operand will be converted to String. That’s why

```
    System.out.println("Temp: " + 43 + " Frozen: " + false);
```

works. What does this print out?

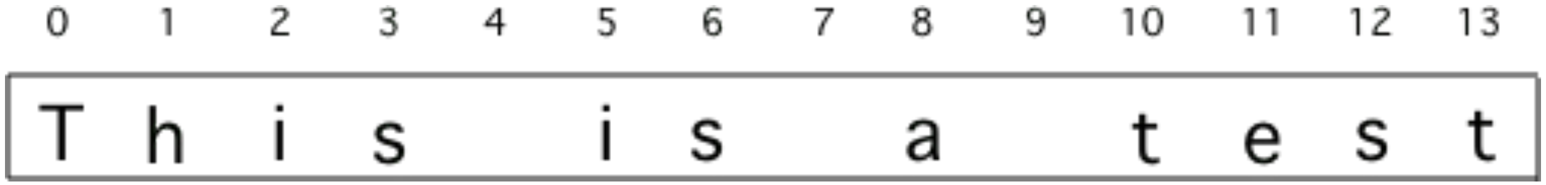
```
    System.out.println("Age: " + 1 + 2);
```

2.7

String Methods

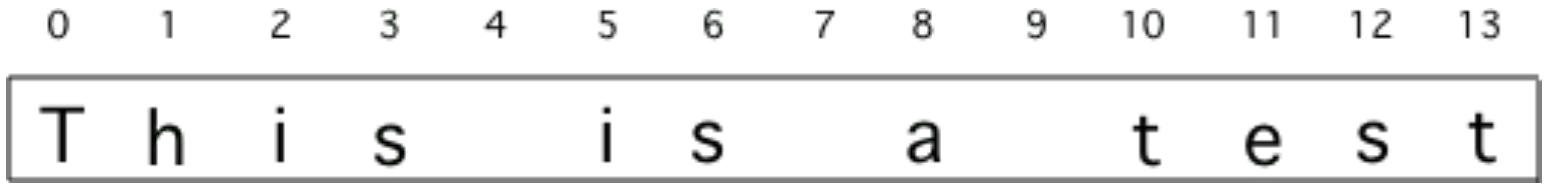
String Methods

- **`int length()`** method returns the number of characters in the string, including spaces and special characters like punctuation.
- **`String substring(int from, int to)`** method returns a new string with the characters in the current string starting with the character at the from index and ending at the character before the to index (if the `to` index is specified, and if not specified it will contain the rest of the string).
- **Remember: In Java, we always start counting from 0**



String Methods

- **`int indexOf(String str)`** method searches for the string str in the current string and returns the index of the beginning of str in the current string or -1 if it isn't found.
- **`int compareTo(String other)`** returns a negative value if the current string is less than the other string alphabetically, 0 if they have the same characters in the same order, and a positive value if the current string is greater than the other string alphabetically.
- **`boolean equals(String other)`** returns true when the characters in the current string are the same as the ones in the other string. This method is inherited from the Object class, but is **overridden** which means that the String class has its own version of that method.



null

If you declare a String variable without initializing it, its value is `null`.

```
    String s;   ← Will contain null
```

Just like any other class, if you invoke a method of `null`, like `s.length()` here, Java will throw a `NullPointerException`.

`null` has its uses. It can represent the absence of a thing. For example, some people don’t have a middle name:

```
    class Person(String firstName, String middleName, String lastName) {...}
    Person person = new Person(“John”, null, “Middlenameless”);
```

Mutable vs Immutable

- **Mutable:** CAN CHANGE, **Immutable:** CANNOT CHANGE
- Strings are immutable. Any methods that seems to change a string actually just creates a **copy** of it, and returns the new version as its return value.

```
String str1 = "Hello!";
// Print str1 in lower case? Will str1 change?
str1.toLowerCase();
System.out.println("In lowercase: " + str1);
```

Why are strings immutable?

Immutability is a powerful concept in Computer Science. It can make it easier to reason about what a program does.

When you pass a `String` to a method, since `Strings` are immutable, you know that the method cannot change your `String` behind your back!

`s += ", world!";` is really the same as `s = s + ", world";`

(but is stylistically better)

Many modern programming languages have immutable strings, such as Python and JavaScript. Some chose to have mutable strings, like Ruby.

2.8

Wrapper Classes

Wrapper classes

Wrapper classes are used to store primitive types inside of ordinary Java classes. `Integer` is a Java class, while `int` is not.

`Integer` has a single attribute storing the value of the `int` used to create the instance.

Constructors:

Integer i = new Integer(4);

Double d = new Double(2.718);

Getters:

int j = i.intValue();

double e = d.doubleValue();

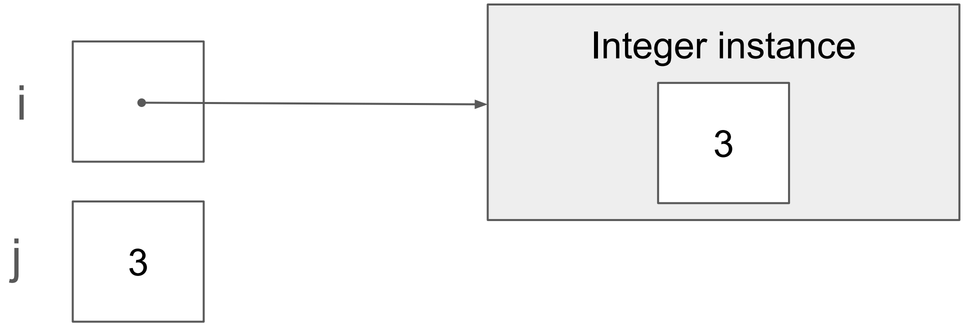
Wrapper classes

A variable whose type is a wrapper class like `Integer`, like any other class, is a reference to an object instance, not an object instance itself.

Using a wrapper class instead of a primitive type means more memory accesses, which is slower, and more memory used.

Wrapper object instances are also called "boxed primitives" – see the box?

```
Integer i = new Integer(4);

int j = 3;
```



Wrapper classes exist for every primitive type

Every primitive type in Java has a corresponding wrapper class.

`boolean`	`java.lang.Boolean`
`byte`	`java.lang.Byte`
`char`	`java.lang.Character`
`double`	`java.lang.Double`
`float`	`java.lang.Float`
`int`	`java.lang.Integer`
`long`	`java.lang.Long`
`short`	`java.lang.Short`

Autoboxing and unboxing

Originally, Java programmers had to explicitly convert between primitive types and the equivalent wrapper objects.

In Java 1.5 (released 9/30/2004), **autoboxing** and **unboxing** were added.

Autoboxing:

Integer i = 4; is the same as Integer i = new Integer(4);

Unboxing:

int j = i; is the same as int j = i.intValue();

Wrapper classes are immutable

The wrapper classes are immutable, like `String`. Once you create an instance of `Integer`, you can't change the `int` inside it.

The wrapper classes have getter methods like `intValue()`, `doubleValue()`, `booleanValue()`, `floatValue()`, but no setter methods.

```
Integer i = 3;
```

`i = 5;` ← This is really creating a whole new `Integer` object instance, and reassigning the value of `i`.

Why even use wrapper classes?

1) They contain useful methods and attributes:
    a) Integer.parseInt(string) converts a string representation of an integer into an int
    b) Integer.MIN_VALUE and Integer.MAX_VALUE store the largest and smallest possible 32-bit integers your computer can store. These lower and upper bounds on all computable integers are very useful in algorithm development
2) Storing primitive types within classes enables us to use Java language constructs that can only be applied to classes…
    a) More on this later with Arrays and Maps

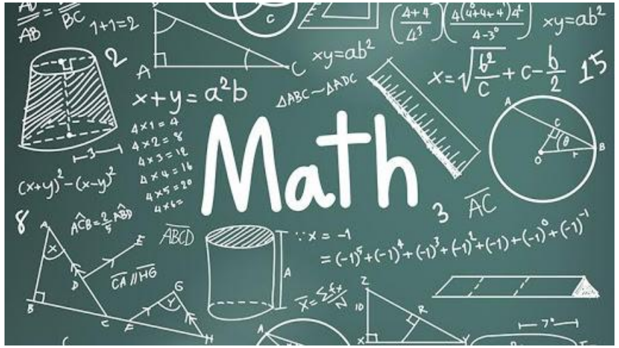
2.9

Math Class

The `Math` Class

This class implements standard mathematical functions and constants.

Math has only static methods and attributes. It cannot be instantiated with the new operator… it has no public constructor!



The `Math` Class

You can prefix with `Math.` to access Math methods and attributes, or use **static imports** to bring some or all of Math into your code's default scope.

These two code samples are equivalent:

```
public class Main {
  public static void main(String args[]) {
    System.out.println(Math.PI);
    System.out.println(Math.sqrt(9));
  }
}
```

```
import static java.lang.Math.*;

public class Main {
  public static void main(String args[]) {
    System.out.println(PI);
    System.out.println(sqrt(9));
  }
}
```

Q: Why was no import statement required on the left?

Absolute Value

`static int abs(int x)`	Returns the absolute value of an `int` value
`static double abs(double x)`	Returns the absolute value of a `double` value

Remember from lesson 2.4:

Overloaded methods are two or more methods in the same class that have the same name but different parameters.

Power

`static double pow(double base, double exp)`

Returns `base`$^{\texttt{exp}}$.

Assumes `base` > 0, or `base` = 0 and `exp` > 0, or `base` < 0 and `exp` is an integer

(What happens if base < 0 and exp is not an integer? It returns NaN, which means "Not A Number." double can't represent imaginary numbers. Math.pow(-1, 0.5) is equivalent to Math.sqrt(-1), which also returns NaN since the answer is an imaginary number.)

P.S. Imaginary numbers can be represented in Java, but it's not built-in to the primitive data types or the standard library. You can use a third party library or write your own.

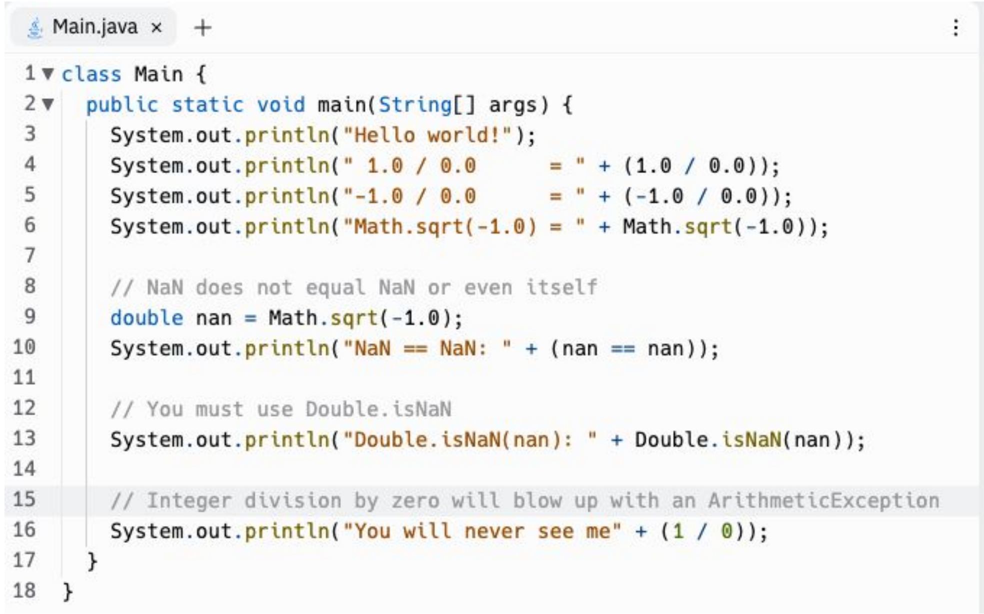
Square root

```
static double sqrt(double x)
```

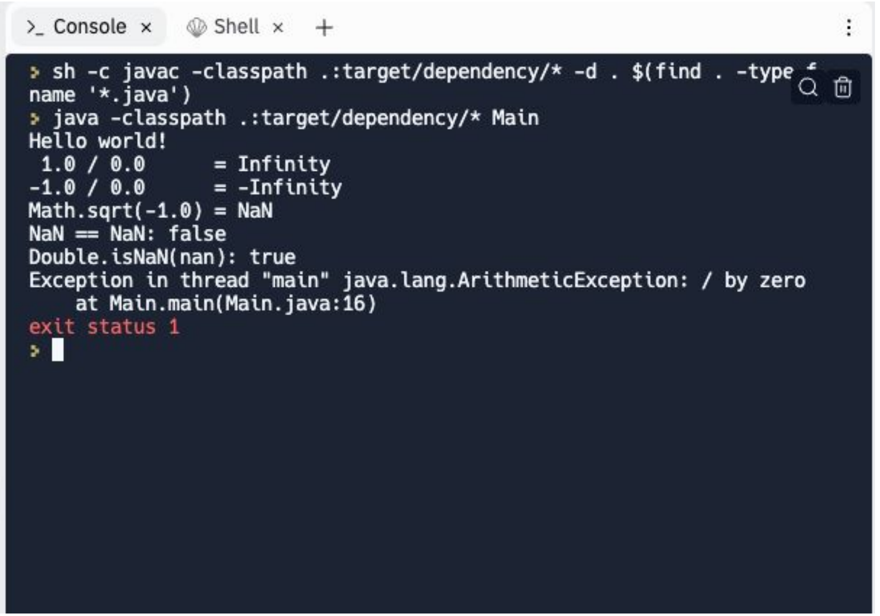
Returns the positive square root of a double value.

(If `x` is negative, this will return `NaN` - Not A Number.)

NaN: Not A Number (floats and doubles only)



```java
class Main {
  public static void main(String[] args) {
    System.out.println("Hello world!");
    System.out.println(" 1.0 / 0.0       = " + (1.0 / 0.0));
    System.out.println("-1.0 / 0.0       = " + (-1.0 / 0.0));
    System.out.println("Math.sqrt(-1.0) = " + Math.sqrt(-1.0));

    // NaN does not equal NaN or even itself
    double nan = Math.sqrt(-1.0);
    System.out.println("NaN == NaN: " + (nan == nan));

    // You must use Double.isNaN
    System.out.println("Double.isNaN(nan): " + Double.isNaN(nan));

    // Integer division by zero will blow up with an ArithmeticException
    System.out.println("You will never see me" + (1 / 0));
  }
}
```



```
> sh -c javac -classpath .:target/dependency/* -d . $(find . -type f
name '*.java')
> java -classpath .:target/dependency/* Main
Hello world!
 1.0 / 0.0      = Infinity
-1.0 / 0.0      = -Infinity
Math.sqrt(-1.0) = NaN
NaN == NaN: false
Double.isNaN(nan): true
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at Main.main(Main.java:16)
exit status 1
> 
```

Random number

```
static double random()
```

Returns a `double` value greater than or equal to `0.0` and less than `1.0`

Example:

```
double randomValue = Math.random();
// Example output: 0.6573016382857277
```

Write code that generates a random `int` between 0 to 9

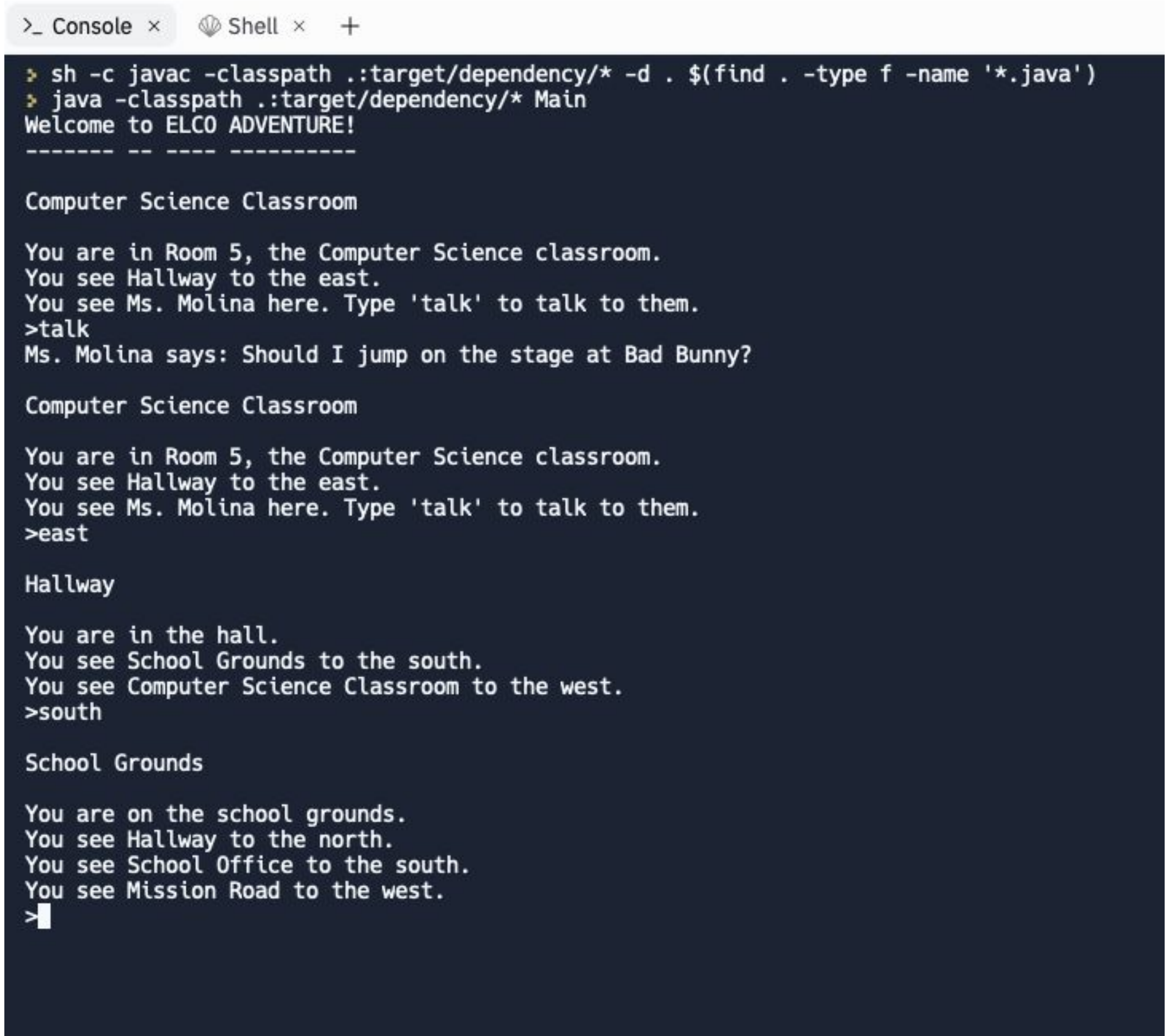
```
int random = (int) (Math.random() * 10);
```

Write code that generates a random `int` between 1 and 10

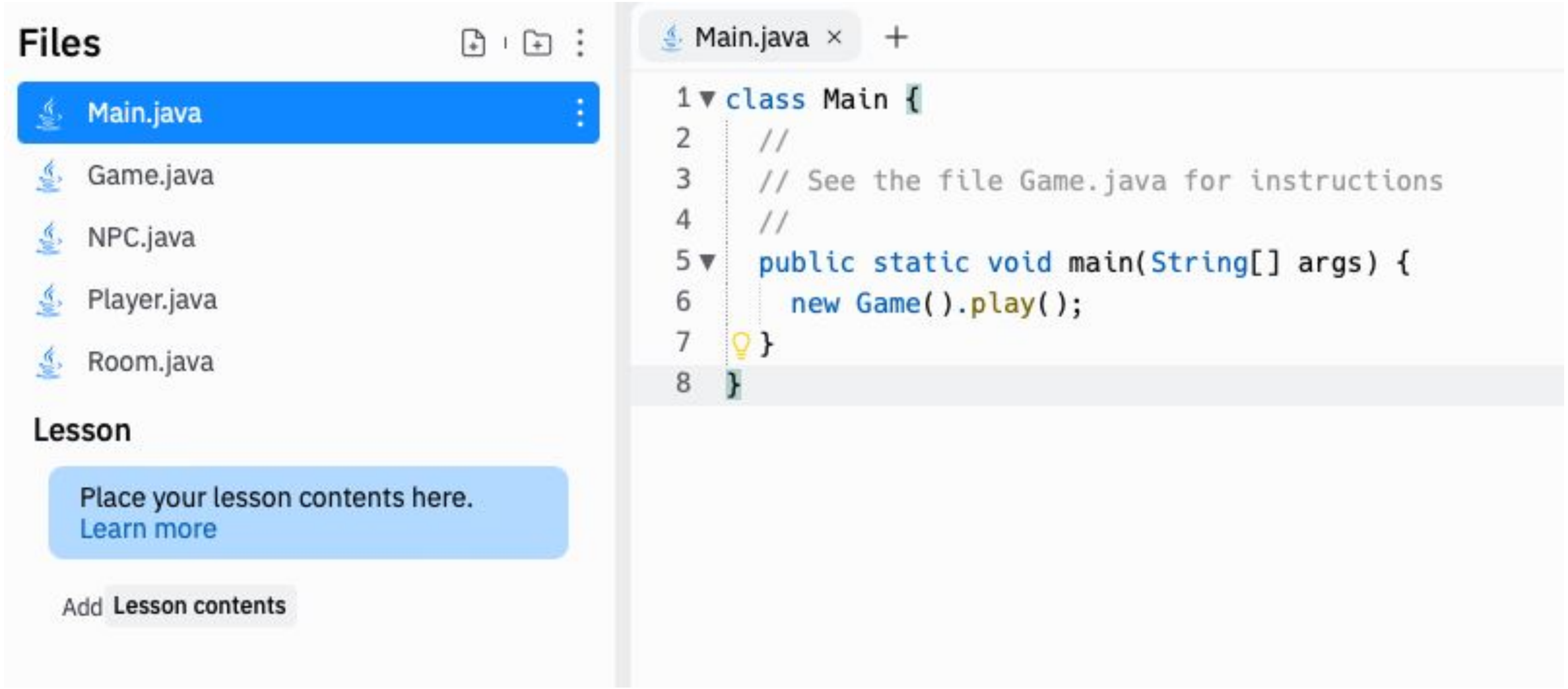
```
int random = (int) (Math.random() * 10) + 1;
```

Practice

Replit: "Adventure"

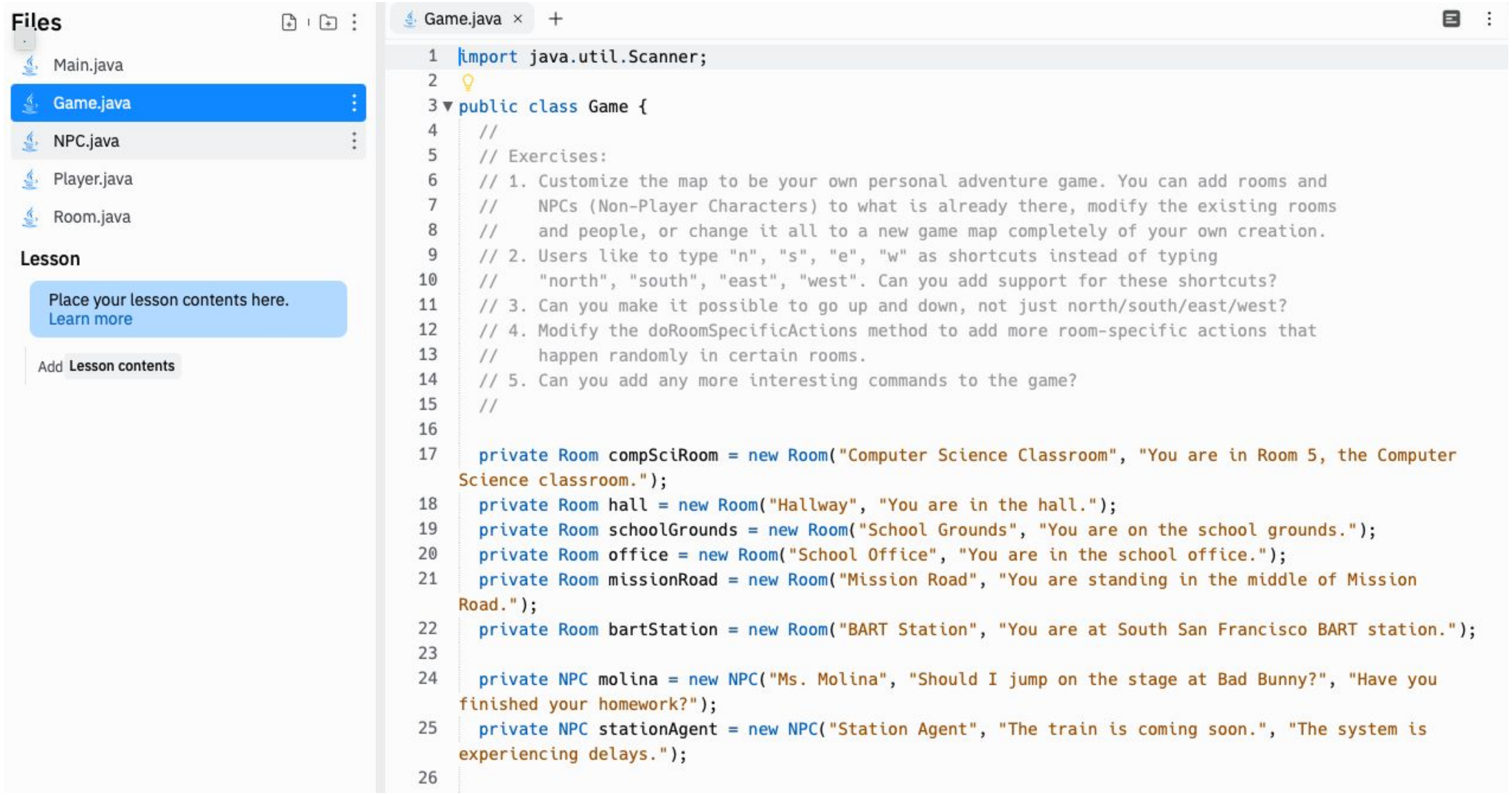

Console Shell

```
> sh -c javac -classpath .:target/dependency/* -d . $(find . -type f -name '*.java')
> java -classpath .:target/dependency/* Main
Welcome to ELCO ADVENTURE!
------- -- ---- ----------

Computer Science Classroom

You are in Room 5, the Computer Science classroom.
You see Hallway to the east.
You see Ms. Molina here. Type 'talk' to talk to them.
>talk
Ms. Molina says: Should I jump on the stage at Bad Bunny?

Computer Science Classroom

You are in Room 5, the Computer Science classroom.
You see Hallway to the east.
You see Ms. Molina here. Type 'talk' to talk to them.
>east

Hallway

You are in the hall.
You see School Grounds to the south.
You see Computer Science Classroom to the west.
>south

School Grounds

You are on the school grounds.
You see Hallway to the north.
You see School Office to the south.
You see Mission Road to the west.
>
```



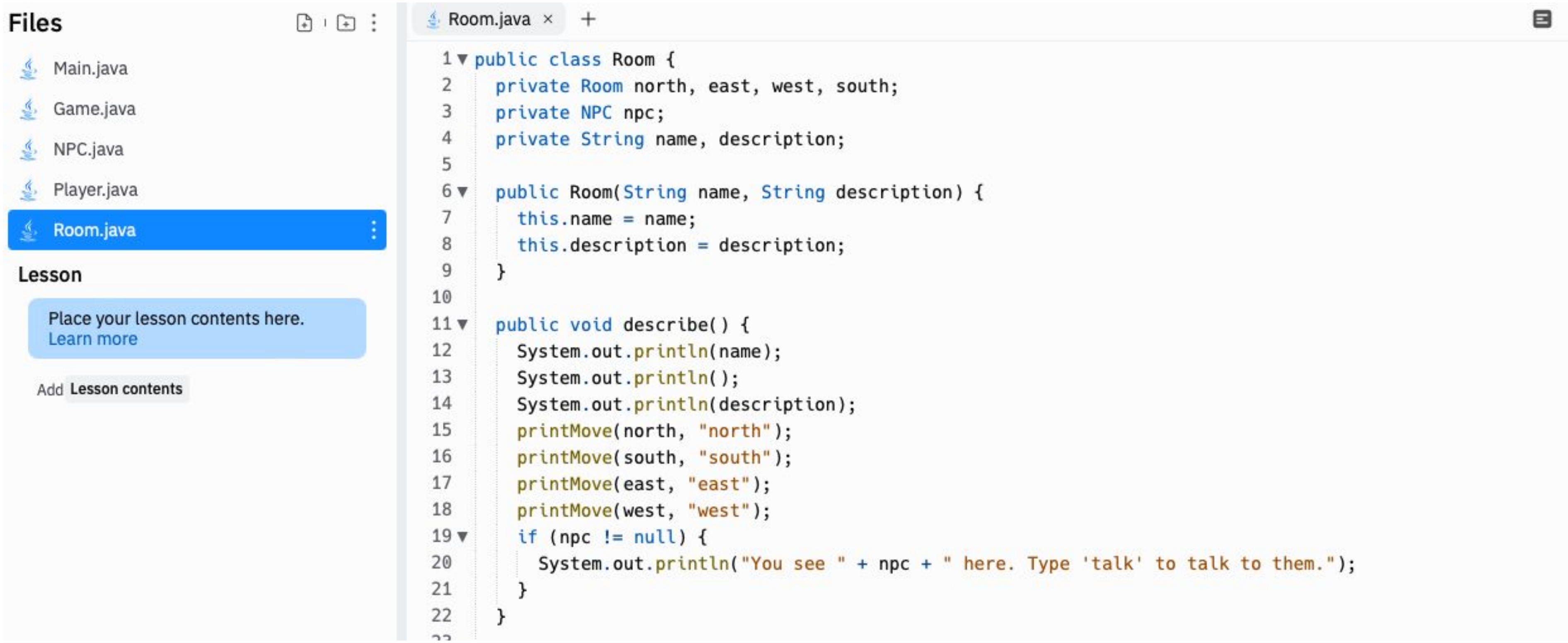
Place your lesson contents here.
Learn more

```
class Main {
  //
  // See the file Game.java for instructions
  //
  public static void main(String[] args) {
    new Game().play();
  }
}
```

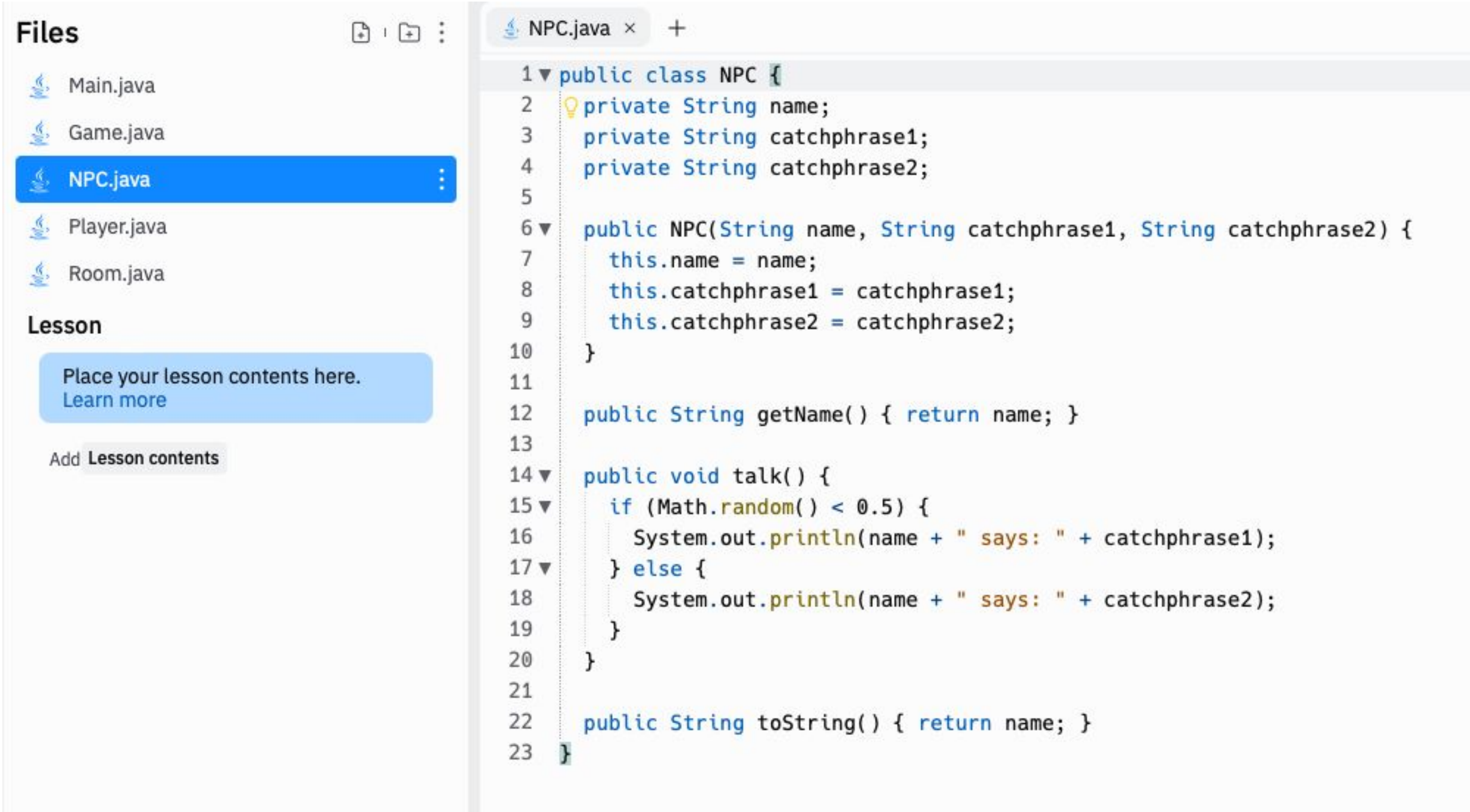


Place your lesson contents here.
Learn more

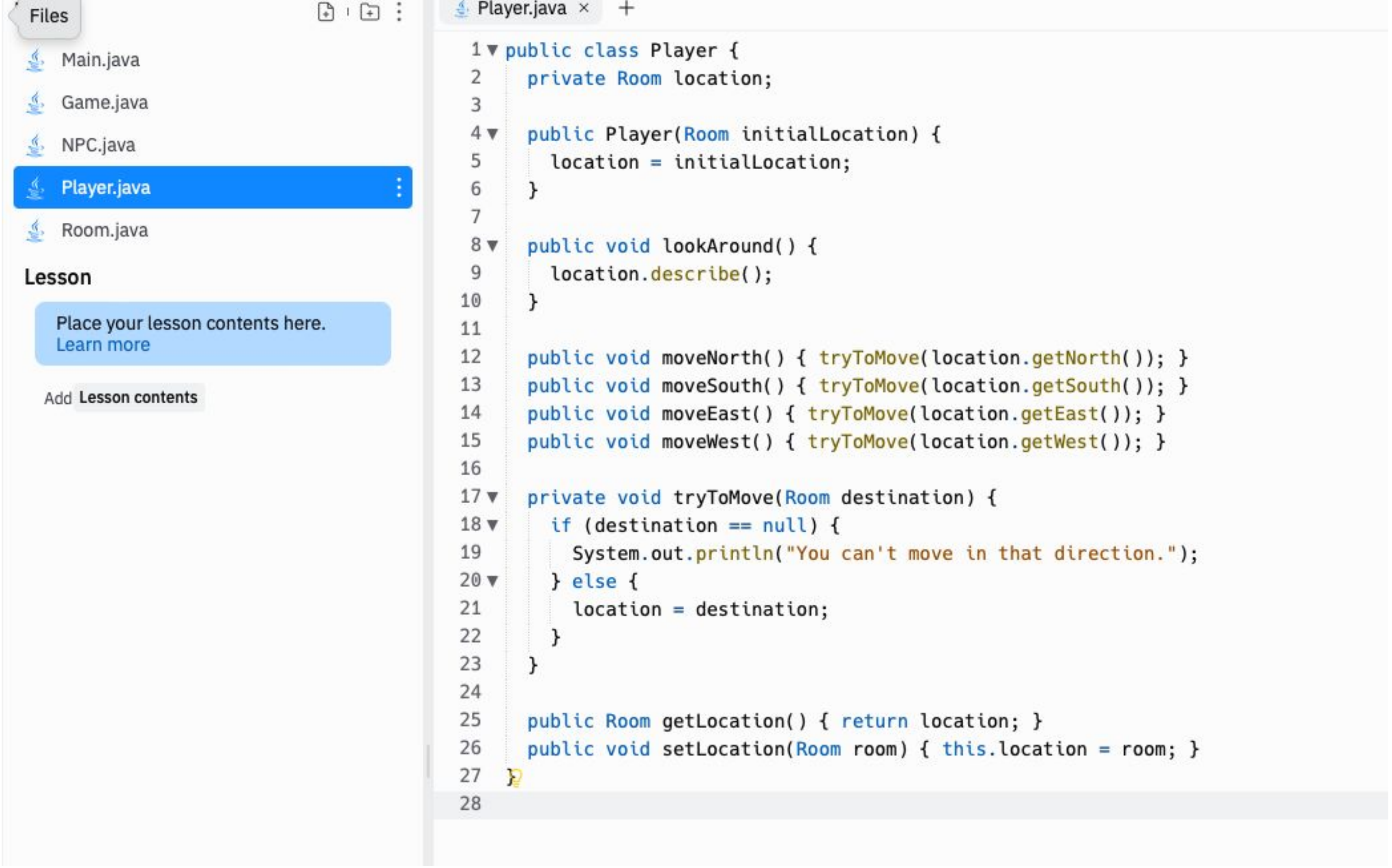
```java
import java.util.Scanner;

public class Game {
  //
  // Exercises:
  // 1. Customize the map to be your own personal adventure game. You can add rooms and
  //    NPCs (Non-Player Characters) to what is already there, modify the existing rooms
  //    and people, or change it all to a new game map completely of your own creation.
  // 2. Users like to type "n", "s", "e", "w" as shortcuts instead of typing
  //    "north", "south", "east", "west". Can you add support for these shortcuts?
  // 3. Can you make it possible to go up and down, not just north/south/east/west?
  // 4. Modify the doRoomSpecificActions method to add more room-specific actions that
  //    happen randomly in certain rooms.
  // 5. Can you add any more interesting commands to the game?
  //

  private Room compSciRoom = new Room("Computer Science Classroom", "You are in Room 5, the Computer Science classroom.");
  private Room hall = new Room("Hallway", "You are in the hall.");
  private Room schoolGrounds = new Room("School Grounds", "You are on the school grounds.");
  private Room office = new Room("School Office", "You are in the school office.");
  private Room missionRoad = new Room("Mission Road", "You are standing in the middle of Mission Road.");
  private Room bartStation = new Room("BART Station", "You are at South San Francisco BART station.");

  private NPC molina = new NPC("Ms. Molina", "Should I jump on the stage at Bad Bunny?", "Have you finished your homework?");
  private NPC stationAgent = new NPC("Station Agent", "The train is coming soon.", "The system is experiencing delays.");

```



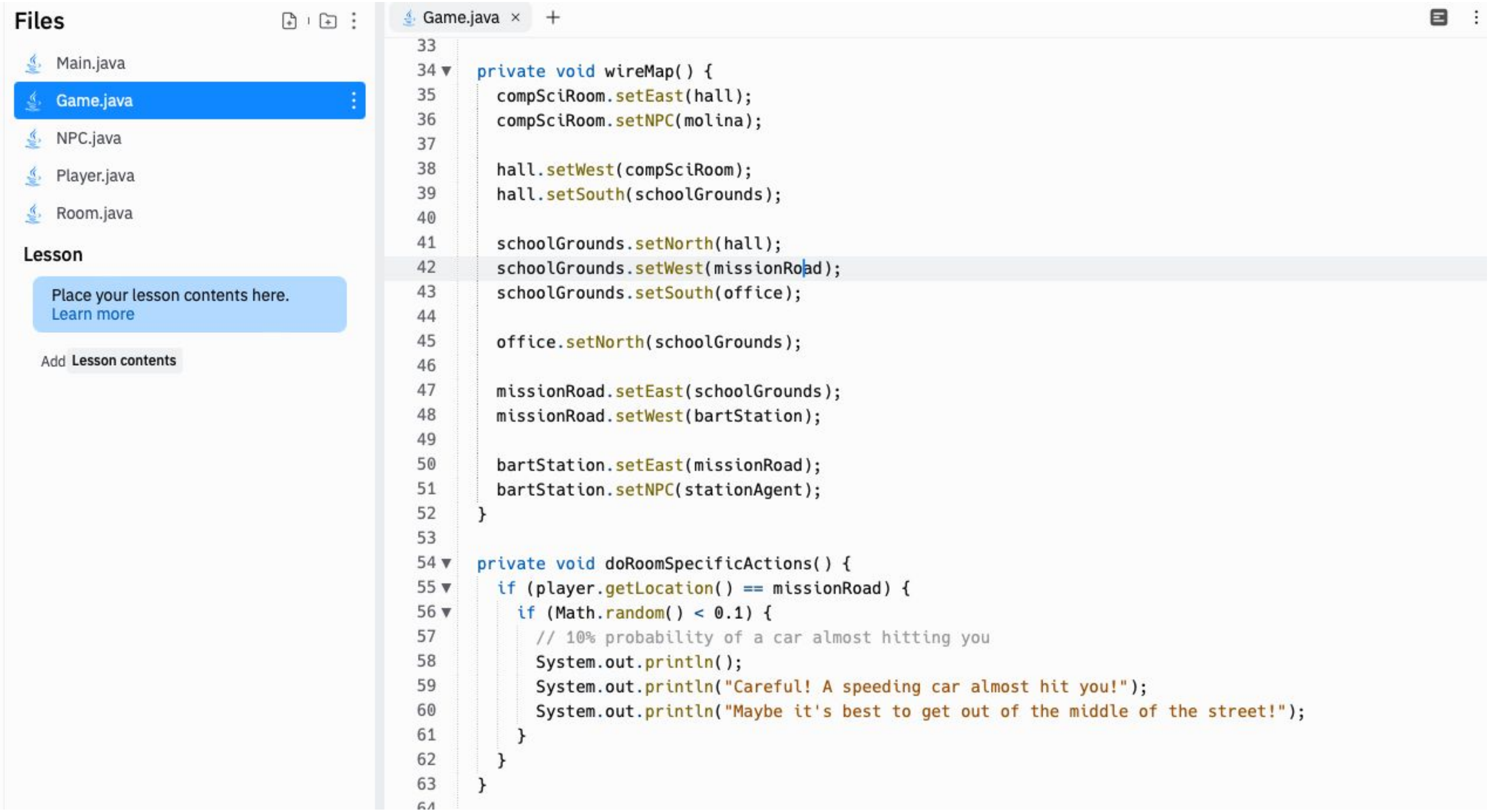
```java
public class Room {
  private Room north, east, west, south;
  private NPC npc;
  private String name, description;

  public Room(String name, String description) {
    this.name = name;
    this.description = description;
  }

  public void describe() {
    System.out.println(name);
    System.out.println();
    System.out.println(description);
    printMove(north, "north");
    printMove(south, "south");
    printMove(east, "east");
    printMove(west, "west");
    if (npc != null) {
      System.out.println("You see " + npc + " here. Type 'talk' to talk to them.");
    }
  }
```



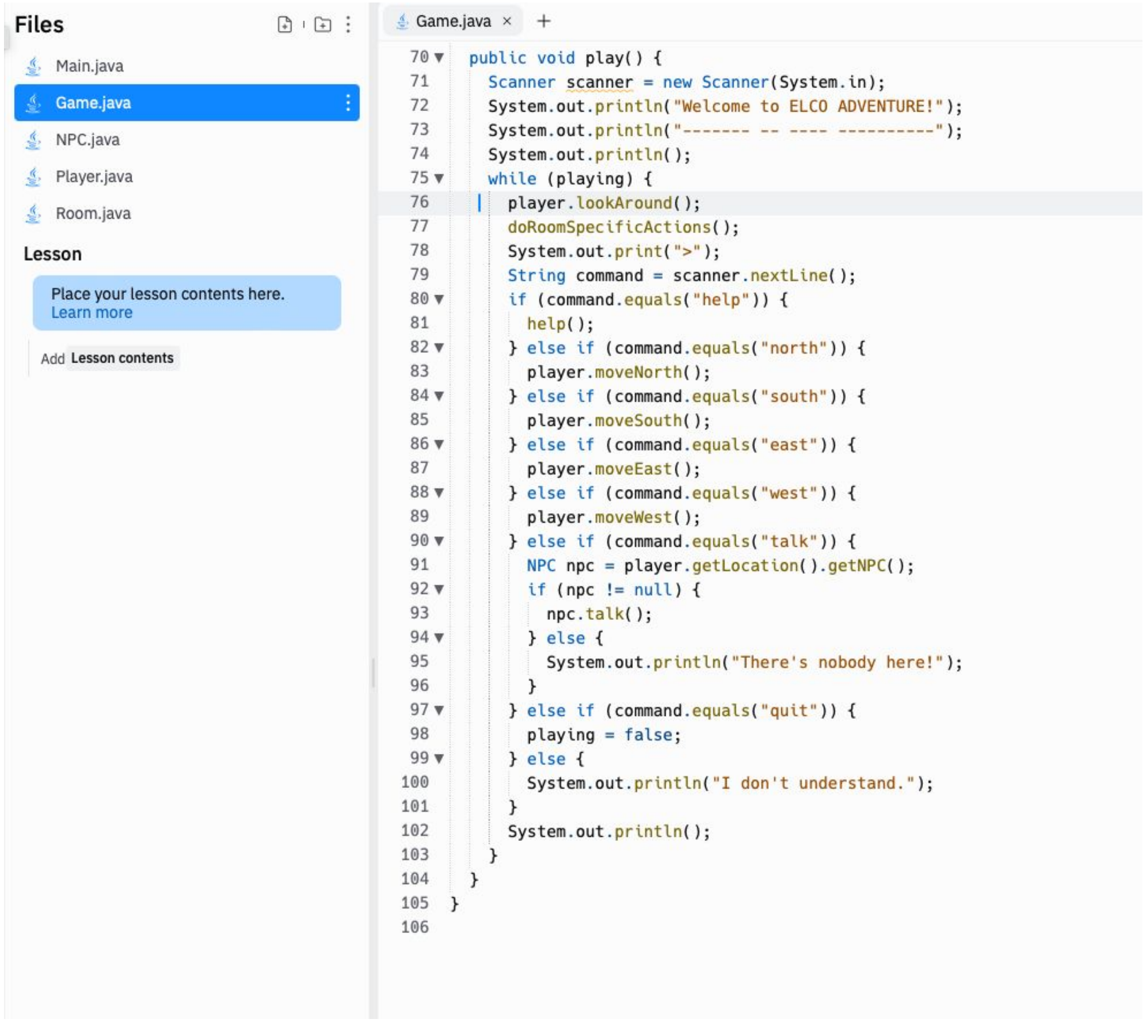
```java
public class NPC {
  private String name;
  private String catchphrase1;
  private String catchphrase2;

  public NPC(String name, String catchphrase1, String catchphrase2) {
    this.name = name;
    this.catchphrase1 = catchphrase1;
    this.catchphrase2 = catchphrase2;
  }

  public String getName() { return name; }

  public void talk() {
    if (Math.random() < 0.5) {
      System.out.println(name + " says: " + catchphrase1);
    } else {
      System.out.println(name + " says: " + catchphrase2);
    }
  }

  public String toString() { return name; }
}
```



```java
public class Player {
  private Room location;

  public Player(Room initialLocation) {
    location = initialLocation;
  }

  public void lookAround() {
    location.describe();
  }

  public void moveNorth() { tryToMove(location.getNorth()); }
  public void moveSouth() { tryToMove(location.getSouth()); }
  public void moveEast() { tryToMove(location.getEast()); }
  public void moveWest() { tryToMove(location.getWest()); }

  private void tryToMove(Room destination) {
    if (destination == null) {
      System.out.println("You can't move in that direction.");
    } else {
      location = destination;
    }
  }

  public Room getLocation() { return location; }
  public void setLocation(Room room) { this.location = room; }
}
```



```java

private void wireMap() {
  compSciRoom.setEast(hall);
  compSciRoom.setNPC(molina);

  hall.setWest(compSciRoom);
  hall.setSouth(schoolGrounds);

  schoolGrounds.setNorth(hall);
  schoolGrounds.setWest(missionRoad);
  schoolGrounds.setSouth(office);

  office.setNorth(schoolGrounds);

  missionRoad.setEast(schoolGrounds);
  missionRoad.setWest(bartStation);

  bartStation.setEast(missionRoad);
  bartStation.setNPC(stationAgent);
}

private void doRoomSpecificActions() {
  if (player.getLocation() == missionRoad) {
    if (Math.random() < 0.1) {
      // 10% probability of a car almost hitting you
      System.out.println();
      System.out.println("Careful! A speeding car almost hit you!");
      System.out.println("Maybe it's best to get out of the middle of the street!");
    }
  }
}
```



```java
    public void play() {
      Scanner scanner = new Scanner(System.in);
      System.out.println("Welcome to ELCO ADVENTURE!");
      System.out.println("------- -- ---- ----------");
      System.out.println();
      while (playing) {
        player.lookAround();
        doRoomSpecificActions();
        System.out.print(">");
        String command = scanner.nextLine();
        if (command.equals("help")) {
          help();
        } else if (command.equals("north")) {
          player.moveNorth();
        } else if (command.equals("south")) {
          player.moveSouth();
        } else if (command.equals("east")) {
          player.moveEast();
        } else if (command.equals("west")) {
          player.moveWest();
        } else if (command.equals("talk")) {
          NPC npc = player.getLocation().getNPC();
          if (npc != null) {
            npc.talk();
          } else {
            System.out.println("There's nobody here!");
          }
        } else if (command.equals("quit")) {
          playing = false;
        } else {
          System.out.println("I don't understand.");
        }
        System.out.println();
      }
    }
  }

```