Java Programming for the *FIRST* Robotics Competition

Ziv Scully Antonio Rufo

Team 125: The NUTRONS

2012 *FIRST* Robotics Competition Kickoff

Outline

Outline

Tradition

Code

```
public class Main {
    public static void main(String[] args) {
    System.out.println( "Hello, world!" );
    }
}
```

Tradition

Code

```
public class Main {
    public static void main(String[] args) {
    System.out.println( "Hello, world!" );
    }
}
```

Output

```
Hello, world!
```

Declaring Variables

Code

```
public class Main {
    public static void main(String[] args) {
        String message = "Hello, world!";
        System.out.println(message);
    }
}
```

Declaring Variables

Code

```
public class Main {
    public static void main(String[] args) {
        String message = "Hello, world!";
        System.out.println(message);
    }
}
```

Output

```
Hello, world!
```

Assigning Variables

Code

```
public class Main {
    public static void main(String[] args) {
        String message = "Hello, world!";
        message = "Salutations, denizens of Earth.";
        message = "Yo world, 'sup?";
        System.out.println(message);
    }
}
```

Assigning Variables

Code

```
public class Main {
    public static void main(String[] args) {
        String message = "Hello, world!";
        message = "Salutations, denizens of Earth.";
        message = "Yo world, 'sup?";
        System.out.println(message);
    }
}
```

Output

```
Yo world, 'sup?
```

Order Matters

Code

```
public class Main {
    public static void main(String[] args) {
        String message = "Hello, world!";
        message = "Salutations, denizens of Earth.";
        System.out.println(message);
        message = "Yo world, 'sup?";
    }
}
```

Order Matters

Code

```
public class Main {
    public static void main(String[] args) {
        String message = "Hello, world!";
        message = "Salutations, denizens of Earth.";
        System.out.println(message);
        message = "Yo world, 'sup?";
    }
}
```

Output

```
Salutations, denizens of Earth.
```

Integers

Code

```
public class Main {
    public static void main(String[] args) {
        int x = 2;
        System.out.println(x);
        x = (2 * x) + 1;
        System.out.println(x);
    }
}
```

Integers

Code

```
public class Main {
    public static void main(String[] args) {
        int x = 2;
        System.out.println(x);
        x = (2 * x) + 1;
        System.out.println(x);
    }
}
```

Output

```
2
5
```

Real Numbers

Code

```
public class Main {
    public static void main(String[] args) {
        double x = 2.0;
        x *= 2;
        x += 1;
        System.out.println(x);
        x = Math.exp(x);
        System.out.println(x);
    }
}
```

Real Numbers

Code

```
public class Main {
    public static void main(String[] args) {
        double x = 2.0;
        x *= 2;
        x += 1;
        System.out.println(x);
        x = Math.exp(x);
        System.out.println(x);
    }
}
```

Output

```
5.0
148.4131591025766
```

If You Must Choose

Code

```
public class Main {
    public static void main(String[] args) {
        int x = 42;
        String message = "This is the default message.";
        if(x % 2 == 1) {
            message = "Wow, an odd number!";
        }
        System.out.println(message);
    }
}
```

If You Must Choose

Code

```
public class Main {
    public static void main(String[] args) {
        int x = 42;
        String message = "This is the default message.";
        if(x % 2 == 1) {
            message = "Wow, an odd number!";
        }
        System.out.println(message);
    }
}
```

Output

```
This is the default message.
```

When All Else Fails

Code

```
public class Main {
    public static void main(String[] args) {
        int x = 42;
        String message = "This is the default message.";
        if(x % 2 == 1) {
            message = "Wow, an odd number!";
        }
        else {
            message = "What a nice, even number!";
        }
        System.out.println(message);
    }
}
```

When All Else Fails

Code

```
public class Main {
    public static void main(String[] args) {
        int x = 42;
        String message = "This is the default message.";
        if(x % 2 == 1) {
            message = "Wow, an odd number!";
        }
        else {
            message = "What a nice, even number!";
        }
        System.out.println(message);
    }
}
```

Output

```
What a nice, even number!
```

While Loops

Code

```
public class Main {
    public static void main(String[] args) {
        int x = 1;
        while(x < 10) {
            x += 1;
        }
        System.out.println(x);
    }
}
```

While Loops

Code

```
public class Main {
    public static void main(String[] args) {
        int x = 1;
        while(x < 10) {
            x += 1;
        }
        System.out.println(x);
    }
}
```

Output

```
10
```

More Interesting While Loops

Code

```
public class Main {
    public static void main(String[] args) {
        int x = 0;
        int y = 1;
        while(y < 100) {
            int newY = x + y;
            x = y;
            y = newY;
        }
        System.out.println(x);
    }
}
```

More Interesting While Loops

Code

```
public class Main {
    public static void main(String[] args) {
        int x = 0;
        int y = 1;
        while(y < 100) {
            int newY = x + y;
            x = y;
            y = newY;
        }
        System.out.println(x);
    }
}
```

Output

89

Dangerously Interesting While Loops

Code

```
public class Main {
    public static void main(String[] args) {
        int x = 0;
        int y = 1;
        while(y < 100) {
            int newY = x + y;
            x = y;
            y = newY;
        }
        System.out.println(newY);
    }
}
```

Dangerously Interesting While Loops

Code

```
public class Main {
    public static void main(String[] args) {
        int x = 0;
        int y = 1;
        while(y < 100) {
            int newY = x + y;
            x = y;
            y = newY;
        }
        System.out.println(newY);
    }
}
```

Output

```
This code will not compile!
```

For Loops

Code

```
public class Main {
    public static void main(String[] args) {
        int x = 0;
        for(int i = 0; i <= 100; i++) {
            x += i;
        }
        System.out.println(x);
    }
}
```

For Loops

Code

```
public class Main {
    public static void main(String[] args) {
        int x = 0;
        for(int i = 0; i <= 100; i++) {
            x += i;
        }
        System.out.println(x);
    }
}
```

Output

```
5050
```

Defining Functions

Code

```
public class Main {
    public static double half(double x) {
        System.out.println( "I'm in a function!" );
        return x / 2.0;
    }
    public static void main(String[] args) {
        int x = 15;
        System.out.println(half(x));
    }
}
```

Defining Functions

Code

```
public class Main {
    public static double half(double x) {
        System.out.println( "I'm in a function!" );
        return x / 2.0;
    }
    public static void main(String[] args) {
        int x = 15;
        System.out.println(half(x));
    }
}
```

Output

```
I'm in a function!
7.5
```

How Functions are Executed

Code

```
public class Main {
    public static double half(double x) {
        System.out.println( "I'm in a function!" );
        return x / 2.0;
    }
    public static void main(String[] args) {
        int x = 15;
        System.out.println(half(half(x)));
    }
}
```

How Functions are Executed

Code

```
public class Main {
    public static double half(double x) {
        System.out.println( "I'm in a function!" );
        return x / 2.0;
    }
    public static void main(String[] args) {
        int x = 15;
        System.out.println(half(half(x)));
    }
}
```

Output

```
I'm in a function!
I'm in a function!
3.75
```

Classes

- Classes are data structures with benefits.
- A class does two main things:
  - Stores data in variables.
  - Provides functions—called "methods"—that allow for convenient and safe access to and manipulation of data.
- Why use a class?
  - Eliminate repetetive code.
  - Organize data with intuitive objects.
  - Split complicated tasks into simple chunks.
    - This is especially important for FRC.

Classes

- Classes are data structures with benefits.
- A class does two main things:
  - Stores data in variables.
  - Provides functions—called "methods"—that allow for convenient and safe access to and manipulation of data.
- Why use a class?
  - Eliminate repetetive code.
  - Organize data with intuitive objects.
  - Split complicated tasks into simple chunks.
    - This is especially important for FRC.

Classes

- Classes are data structures with benefits.
- A class does two main things:
  - Stores data in variables.
  - Provides functions—called "methods"—that allow for convenient and safe access to and manipulation of data.
- Why use a class?
  - Eliminate repetetive code.
  - Organize data with intuitive objects.
  - Split complicated tasks into simple chunks.
    - This is especially important for FRC.

Classes

- Classes are data structures with benefits.
- A class does two main things:
  - Stores data in variables.
  - Provides functions—called "methods"—that allow for convenient and safe access to and manipulation of data.
- Why use a class?
  - Eliminate repetetive code.
  - Organize data with intuitive objects.
  - Split complicated tasks into simple chunks.
    - This is especially important for FRC.

Classes

- Classes are data structures with benefits.
- A class does two main things:
  - Stores data in variables.
  - Provides functions—called "methods"—that allow for convenient and safe access to and manipulation of data.
- Why use a class?
  - Eliminate repetetive code.
  - Organize data with intuitive objects.
  - Split complicated tasks into simple chunks.
    - This is especially important for FRC.

Classes

- Classes are data structures with benefits.
- A class does two main things:
  - Stores data in variables.
  - Provides functions—called "methods"—that allow for convenient and safe access to and manipulation of data.
- Why use a class?
  - Eliminate repetetive code.
  - Organize data with intuitive objects.
  - Split complicated tasks into simple chunks.
    - This is especially important for FRC.

Classes

- Classes are data structures with benefits.
- A class does two main things:
  - Stores data in variables.
  - Provides functions—called "methods"—that allow for convenient and safe access to and manipulation of data.
- Why use a class?
  - Eliminate repetetive code.
  - Organize data with intuitive objects.
  - Split complicated tasks into simple chunks.
    - This is especially important for FRC.

Classes

- Classes are data structures with benefits.
- A class does two main things:
  - Stores data in variables.
  - Provides functions—called "methods"—that allow for convenient and safe access to and manipulation of data.
- Why use a class?
  - Eliminate repetetive code.
  - Organize data with intuitive objects.
  - Split complicated tasks into simple chunks.
    - This is especially important for FRC.

Classes

- Classes are data structures with benefits.
- A class does two main things:
  - Stores data in variables.
  - Provides functions—called "methods"—that allow for convenient and safe access to and manipulation of data.
- Why use a class?
  - Eliminate repetetive code.
  - Organize data with intuitive objects.
  - Split complicated tasks into simple chunks.
    - This is especially important for FRC.

Objects

- Classes are descriptions of how to make a certain data type.
  - A point has an x-coordinate and a y-coordinate.
- Objects, also called instances, are examples of that data type with the blanks filled in.
  - A particular point might have coordinates $(3, 4)$.

Objects

- Classes are descriptions of how to make a certain data type.
  - A point has an x-coordinate and a y-coordinate.
- Objects, also called instances, are examples of that data type with the blanks filled in.
  - A particular point might have coordinates $(3, 4)$.

Objects

- Classes are descriptions of how to make a certain data type.
  - A point has an x-coordinate and a y-coordinate.
- Objects, also called instances, are examples of that data type with the blanks filled in.
  - A particular point might have coordinates $(3, 4)$.

Objects

- Classes are descriptions of how to make a certain data type.
  - A point has an x-coordinate and a y-coordinate.
- Objects, also called instances, are examples of that data type with the blanks filled in.
  - A particular point might have coordinates $(3, 4)$.

Don't Panic

Code

```
public class Point {
    private double x = 0;
    private double y = 0;
    public Point(double x, double y) {
        this.x = x;
        this.y = y;
    }
    public void setX(double newX) {
        x = newX;
    }
    public void setY(double newY) {
        y = newY;
    }
    public double getX() {
        return x;
    }
    public double getY() {
        return y;
    }
    public double getR() {
        return Math.sqrt(x*x + y*y);
    }
    public double getTheta() {
        return Math.atan2(x, y);
    }
}
```

Outline

Victors and Jaguars

- Speed controllers regulate voltage to motor based on PWM signal.
- WPILib defines `Victor` and `Jaguar` classes.
  - Constructor takes PWM port as an argument.
- Speed controllers have a `set(double speed)` method, where $-1 \leq \texttt{speed} \leq 1$.
  - Victors have nonlinear voltage output relative to `speed`.

Victors and Jaguars

- Speed controllers regulate voltage to motor based on PWM signal.
- WPILib defines `Victor` and `Jaguar` classes.
  - Constructor takes PWM port as an argument.
- Speed controllers have a `set(double speed)` method, where $-1 \leq \texttt{speed} \leq 1$.
  - Victors have nonlinear voltage output relative to `speed`.

Victors and Jaguars

- Speed controllers regulate voltage to motor based on PWM signal.
- WPILib defines `Victor` and `Jaguar` classes.
  - Constructor takes PWM port as an argument.
- Speed controllers have a `set(double speed)` method, where $-1 \leq \texttt{speed} \leq 1$.
  - Victors have nonlinear voltage output relative to `speed`.

Victors and Jaguars

- Speed controllers regulate voltage to motor based on PWM signal.
- WPILib defines `Victor` and `Jaguar` classes.
  - Constructor takes PWM port as an argument.
- Speed controllers have a `set(double speed)` method, where $-1 \leq \texttt{speed} \leq 1$.
  - Victors have nonlinear voltage output relative to speed.

Victors and Jaguars

- Speed controllers regulate voltage to motor based on PWM signal.
- WPILib defines `Victor` and `Jaguar` classes.
  - Constructor takes PWM port as an argument.
- Speed controllers have a `set(double speed)` method, where $-1 \leq \texttt{speed} \leq 1$.
  - Victors have nonlinear voltage output relative to `speed`.

Other Hardware

- Robot Sensors:
  - `Gyro` and `Encoder` for those sensors.
  - `DigitalIO` for limit switches, KOP light sensors, etc.
  - `AnalogIO` for ultrasonic sensors, potentiometers, etc.
- Driver station input:
  - `Joystick` class for USB controllers.
  - Analog and digital IO from Cypress module or driver station interface.
- Additional actuators such as `Servo` and `Solenoid`.

Other Hardware

- Robot Sensors:
  - `Gyro` and `Encoder` for those sensors.
  - `DigitalIO` for limit switches, KOP light sensors, etc.
  - `AnalogIO` for ultrasonic sensors, potentiometers, etc.
- Driver station input:
  - `Joystick` class for USB controllers.
  - Analog and digital IO from Cypress module or driver station interface.
- Additional actuators such as `Servo` and `Solenoid`.

Other Hardware

- Robot Sensors:
  - `Gyro` and `Encoder` for those sensors.
  - `DigitalIO` for limit switches, KOP light sensors, etc.
  - `AnalogIO` for ultrasonic sensors, potentiometers, etc.
- Driver station input:
  - `Joystick` class for USB controllers.
  - Analog and digital IO from Cypress module or driver station interface.
- Additional actuators such as `Servo` and `Solenoid`.

Other Hardware

- Robot Sensors:
  - `Gyro` and `Encoder` for those sensors.
  - `DigitalIO` for limit switches, KOP light sensors, etc.
  - `AnalogIO` for ultrasonic sensors, potentiometers, etc.
- Driver station input:
  - `Joystick` class for USB controllers.
  - Analog and digital IO from Cypress module or driver station interface.
- Additional actuators such as `Servo` and `Solenoid`.

Other Hardware

- Robot Sensors:
  - `Gyro` and `Encoder` for those sensors.
  - `DigitalIO` for limit switches, KOP light sensors, etc.
  - `AnalogIO` for ultrasonic sensors, potentiometers, etc.
- Driver station input:
  - `Joystick` class for USB controllers.
  - Analog and digital IO from Cypress module or driver station interface.
- Additional actuators such as `Servo` and `Solenoid`.

Other Hardware

- Robot Sensors:
  - `Gyro` and `Encoder` for those sensors.
  - `DigitalIO` for limit switches, KOP light sensors, etc.
  - `AnalogIO` for ultrasonic sensors, potentiometers, etc.
- Driver station input:
  - `Joystick` class for USB controllers.
  - Analog and digital IO from Cypress module or driver station interface.
- Additional actuators such as `Servo` and `Solenoid`.

Other Hardware

- Robot Sensors:
  - `Gyro` and `Encoder` for those sensors.
  - `DigitalIO` for limit switches, KOP light sensors, etc.
  - `AnalogIO` for ultrasonic sensors, potentiometers, etc.
- Driver station input:
  - `Joystick` class for USB controllers.
  - Analog and digital IO from Cypress module or driver station interface.
- Additional actuators such as `Servo` and `Solenoid`.

Other Hardware

- Robot Sensors:
  - `Gyro` and `Encoder` for those sensors.
  - `DigitalIO` for limit switches, KOP light sensors, etc.
  - `AnalogIO` for ultrasonic sensors, potentiometers, etc.
- Driver station input:
  - `Joystick` class for USB controllers.
  - Analog and digital IO from Cypress module or driver station interface.
- Additional actuators such as `Servo` and `Solenoid`.

Encapsulation with Subsystem Classes

- Subsystem objects correspond to physical robot subsystems.
- Extend WPILib's `Subsystem` class.
  - `PIDSubsystem` class also available.
- Minimal number of public methods.
- Subsystems don't think for themselves—they follow orders.
  - Good: Arm stops when it reaches maximum or minimum height.
  - Good: Elevator moves towards setpoint of a PID controller.
  - Bad: Drive train has automated routine to turn 90 degrees.

Encapsulation with Subsystem Classes

- Subsystem objects correspond to physical robot subsystems.
- Extend WPILib's `Subsystem` class.
  - `PIDSubsystem` class also available.
- Minimal number of public methods.
- Subsystems don't think for themselves—they follow orders.
  - Good: Arm stops when it reaches maximum or minimum height.
  - Good: Elevator moves towards setpoint of a PID controller.
  - Bad: Drive train has automated routine to turn 90 degrees.

Encapsulation with Subsystem Classes

- Subsystem objects correspond to physical robot subsystems.
- Extend WPILib's `Subsystem` class.
  - `PIDSubsystem` class also available.
- Minimal number of public methods.
- Subsystems don't think for themselves—they follow orders.
  - Good: Arm stops when it reaches maximum or minimum height.
  - Good: Elevator moves towards setpoint of a PID controller.
  - Bad: Drive train has automated routine to turn 90 degrees.

Encapsulation with Subsystem Classes

- Subsystem objects correspond to physical robot subsystems.
- Extend WPILib's `Subsystem` class.
  - `PIDSubsystem` class also available.
- Minimal number of public methods.
- Subsystems don't think for themselves—they follow orders.
  - Good: Arm stops when it reaches maximum or minimum height.
  - Good: Elevator moves towards setpoint of a PID controller.
  - Bad: Drive train has automated routine to turn 90 degrees.

Encapsulation with Subsystem Classes

- Subsystem objects correspond to physical robot subsystems.
- Extend WPILib's `Subsystem` class.
  - `PIDSubsystem` class also available.
- Minimal number of public methods.
- Subsystems don't think for themselves—they follow orders.
  - Good: Arm stops when it reaches maximum or minimum height.
  - Good: Elevator moves towards setpoint of a PID controller.
  - Bad: Drive train has automated routine to turn 90 degrees.

Encapsulation with Subsystem Classes

- Subsystem objects correspond to physical robot subsystems.
- Extend WPILib's `Subsystem` class.
  - `PIDSubsystem` class also available.
- Minimal number of public methods.
- Subsystems don't think for themselves—they follow orders.
  - Good: Arm stops when it reaches maximum or minimum height.
  - Good: Elevator moves towards setpoint of a PID controller.
  - Bad: Drive train has automated routine to turn 90 degrees.

Encapsulation with Subsystem Classes

- Subsystem objects correspond to physical robot subsystems.
- Extend WPILib's `Subsystem` class.
  - `PIDSubsystem` class also available.
- Minimal number of public methods.
- Subsystems don't think for themselves—they follow orders.
  - Good: Arm stops when it reaches maximum or minimum height.
  - Good: Elevator moves towards setpoint of a PID controller.
  - Bad: Drive train has automated routine to turn 90 degrees.

Encapsulation with Subsystem Classes

- Subsystem objects correspond to physical robot subsystems.
- Extend WPILib's `Subsystem` class.
  - `PIDSubsystem` class also available.
- Minimal number of public methods.
- Subsystems don't think for themselves—they follow orders.
  - Good: Arm stops when it reaches maximum or minimum height.
  - Good: Elevator moves towards setpoint of a PID controller.
  - Bad: Drive train has automated routine to turn 90 degrees.

Program Logic with Commands

- Commands describe the actions a robot can do.
  - A subsystem's methods are basic motions that a command strings together into a meaningful game action.
- Extend project's `CommandBase` class, which has a static instance of each subsystem.
- As well as combining subsystem methods, commands can combine other commands.
  - WPILib offers `CommandGroup` class.
- Specify which subsystems a command needs using `requires(Subsystem subsys)` method.
- Can be used for both autonomous and teleoperated behavior.

Program Logic with Commands

- Commands describe the actions a robot can do.
  - A subsystem's methods are basic motions that a command strings together into a meaningful game action.
- Extend project's `CommandBase` class, which has a static instance of each subsystem.
- As well as combining subsystem methods, commands can combine other commands.
  - WPILib offers `CommandGroup` class.
- Specify which subsystems a command needs using `requires(Subsystem subsys)` method.
- Can be used for both autonomous and teleoperated behavior.

Program Logic with Commands

- Commands describe the actions a robot can do.
  - A subsystem's methods are basic motions that a command strings together into a meaningful game action.
- Extend project's `CommandBase` class, which has a static instance of each subsystem.
- As well as combining subsystem methods, commands can combine other commands.
  - WPILib offers `CommandGroup` class.
- Specify which subsystems a command needs using `requires(Subsystem subsys)` method.
- Can be used for both autonomous and teleoperated behavior.

Program Logic with Commands

- Commands describe the actions a robot can do.
  - A subsystem's methods are basic motions that a command strings together into a meaningful game action.
- Extend project's `CommandBase` class, which has a static instance of each subsystem.
- As well as combining subsystem methods, commands can combine other commands.
  - WPILib offers `CommandGroup` class.
- Specify which subsystems a command needs using `requires(Subsystem subsys)` method.
- Can be used for both autonomous and teleoperated behavior.

Program Logic with Commands

- Commands describe the actions a robot can do.
  - A subsystem's methods are basic motions that a command strings together into a meaningful game action.
- Extend project's `CommandBase` class, which has a static instance of each subsystem.
- As well as combining subsystem methods, commands can combine other commands.
  - WPILib offers `CommandGroup` class.
- Specify which subsystems a command needs using `requires(Subsystem subsys)` method.
- Can be used for both autonomous and teleoperated behavior.

Program Logic with Commands

- Commands describe the actions a robot can do.
  - A subsystem's methods are basic motions that a command strings together into a meaningful game action.
- Extend project's `CommandBase` class, which has a static instance of each subsystem.
- As well as combining subsystem methods, commands can combine other commands.
  - WPILib offers `CommandGroup` class.
- Specify which subsystems a command needs using `requires(Subsystem subsys)` method.
- Can be used for both autonomous and teleoperated behavior.

Program Logic with Commands

- Commands describe the actions a robot can do.
  - A subsystem's methods are basic motions that a command strings together into a meaningful game action.
- Extend project's `CommandBase` class, which has a static instance of each subsystem.
- As well as combining subsystem methods, commands can combine other commands.
  - WPILib offers `CommandGroup` class.
- Specify which subsystems a command needs using `requires(Subsystem subsys)` method.
- Can be used for both autonomous and teleoperated behavior.

Anatomy of a Command

- `initialize()`: called once when command starts.
- `execute()`: called periodically; subsystem movement goes here.
- `isFinished()`: called periodically and returns a boolean; when true, triggers command end.
- `end()`: called once when command ends.
- `interrupted()`: called if command is terminated before it can finish.

Anatomy of a Command

- `initialize()`: called once when command starts.
- `execute()`: called periodically; subsystem movement goes here.
- `isFinished()`: called periodically and returns a boolean; when true, triggers command end.
- `end()`: called once when command ends.
- `interrupted()`: called if command is terminated before it can finish.

Anatomy of a Command

- `initialize()`: called once when command starts.
- `execute()`: called periodically; subsystem movement goes here.
- `isFinished()`: called periodically and returns a boolean; when true, triggers command end.
- `end()`: called once when command ends.
- `interrupted()`: called if command is terminated before it can finish.

Anatomy of a Command

- `initialize()`: called once when command starts.
- `execute()`: called periodically; subsystem movement goes here.
- `isFinished()`: called periodically and returns a boolean; when true, triggers command end.
- `end()`: called once when command ends.
- `interrupted()`: called if command is terminated before it can finish.

Anatomy of a Command

- `initialize()`: called once when command starts.
- `execute()`: called periodically; subsystem movement goes here.
- `isFinished()`: called periodically and returns a boolean; when true, triggers command end.
- `end()`: called once when command ends.
- `interrupted()`: called if command is terminated before it can finish.

Starting Commands with Buttons

- `Button` is an interface (template) for classes that have a method that returns a boolean value.
- Using `whenPressed(Command cmd)` or `whileHeld(Command cmd)` methods, can link a button action to starting or stopping `cmd`.

Starting Commands with Buttons

- `Button` is an interface (template) for classes that have a method that returns a boolean value.
- Using `whenPressed(Command cmd)` or `whileHeld(Command cmd)` methods, can link a button action to starting or stopping `cmd`.

Types of Buttons

- `JoystickButton` uses buttons from USB Joystick.
- `AnalogIOButton` and `DigitalIOButton` use input from Cypress module or driver station interface.
- `InternalButton` can be triggered by an arbitrary condition originating from robot sensors, driver input or a combination.

Types of Buttons

- `JoystickButton` uses buttons from USB Joystick.
- `AnalogIOButton` and `DigitalIOButton` use input from Cypress module or driver station interface.
- `InternalButton` can be triggered by an arbitrary condition originating from robot sensors, driver input or a combination.

Types of Buttons

- `JoystickButton` uses buttons from USB Joystick.
- `AnalogIOButton` and `DigitalIOButton` use input from Cypress module or driver station interface.
- `InternalButton` can be triggered by an arbitrary condition originating from robot sensors, driver input or a combination.

Outline

Use the Internet, Luke

Additional Information

- `docs.oracle.com/javase/tutorial/getStarted`: Oracle's Java tutorial.
- `firstforge.wpi.edu/sf/projects/wpilib`: WPILib project page with file releases and documentation.

Additional Information

- `docs.oracle.com/javase/tutorial/getStarted`: Oracle's Java tutorial.
- `firstforge.wpi.edu/sf/projects/wpilib`: WPILib project page with file releases and documentation.