Java Fundamentals

WELCOME!

Develop Intelligence

Simon Roberts

WELCOME

Develop Intelligence

Mark Your Attendance:

- Go to the Course Calendar Invite
- Click on the Course Event Link
- Press Check-In

Join Us in Making Learning Technology Easier

Develop Intelligence

Our mission...

Over 16 years ago, we embarked on a journey to improve the world by making learning technology easy and accessible to everyone.

...impacts everyone daily.

And it's working. Today, we're known for delivering customized tech learning programs that drive innovation and transform organizations.

In fact, when you talk on the phone, watch a movie, connect with friends on social media, drive a car, fly on a plane, shop online, and order a latte with your mobile app, you are experiencing the impact of our solutions.

Upskilling and Reskilling Offerings

Develop Intelligence

Intimately customized learning experiences just for your teams.

- **Workshop** — 2-3 day upskilling experiences
- **Fast Track** — 5-day reskilling experiences
- **Learning Spike** — 1-day technology overviews
- **Target Topics** — 90-minute instructor-led micro-learnings
- **Hack-a-thon** — Learn and build an MVP in 2-3 days

BACK END DEVELOPMENT

BIG DATA

CLOUD COMPUTING

DEVOPS

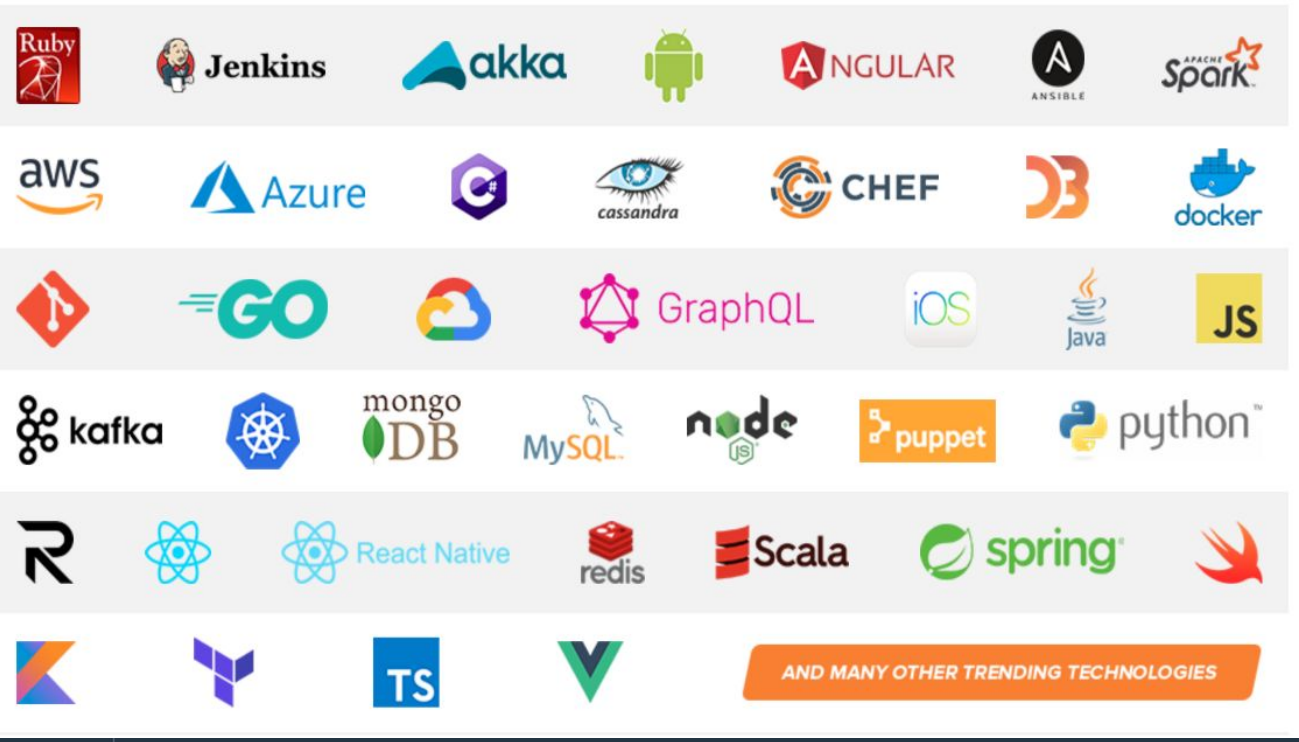
FRONT END DEVELOPMENT

MACHINE LEARNING

MOBILE APP DEVELOPMENT

SOFTWARE ENGINEERING

SYSTEM ADMINISTRATION



World Class Practitioners

Note About Virtual Trainings

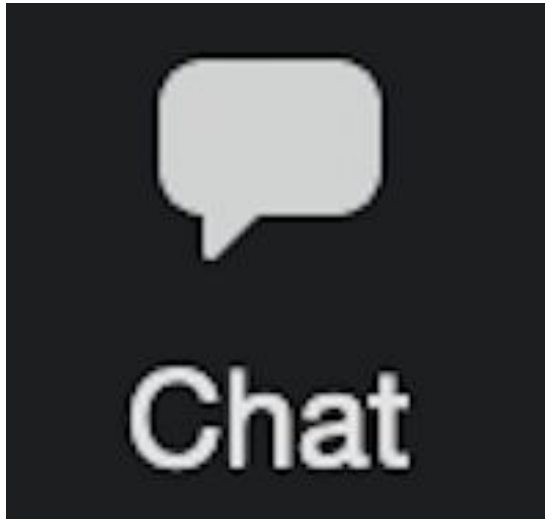
What we want

...what we've got

Virtual Training Expectations for You

Arrive on time / return on time

Mute unless speaking



Use chat or ask questions verbally

Virtual Training Expectations for Me

I pledge to:

- Make this as interesting and interactive as possible
- Ask questions in order to stimulate discussion
- Use whatever resources I have at hand to explain the material
- Try my best to manage verbal responses so that everyone who wants to speak can do so
- Use an on-screen timer for breaks so you know when to be back

Prerequisites

- You should have written code in some other programming language.

Objectives

At the end of this course you will be able to:

- Explain what Java is and why you would use it
- Express computations in a variety of data types
- Write conditional and iterative code
- Factor repeated code into a method or function
- Create simple classes and objects including enumerations
- Use, and explain key limitations of, inheritance
- Throw and catch exceptions
- Use a simple lambda expression
- Use the List interface and ArrayList class
- Implement a simple unit test

About you

In 90 seconds!

- What you hope to learn
- What your background level is

Agenda

- Introduction to Java and OO
- Organizing source code
- Expressions and computations
- Iteration and conditions
- Creating static methods
- Using enumerations
- Working with exceptions
- Creating classes
- Working with Collections
- Inheritance in Java
- Using Lambdas
- Unit Testing

Introduction to Java and OO

Develop Intelligence

Java provides:

- Performance of compiled languages
- Platform independence
- Dynamic features usually reserved for interpreted languages
- Security (though it's not used as much as it should be)
- Familiar syntax and programming model
  - Much like C, C++, Objective C, JavaScript, C#
  - Object oriented with functional features too
- Carefully improving, backward compatible, development
- Huge library base
- Huge programmer skill base

Organizing source code

Develop Intelligence

- Java code is organized hierarchically
  - Packages contain types (e.g. classes)
    - Types contain code and data models
- Source code layout maps to the package and class structure
  - Directory/folder ⇔ package
    - Source (dot-java) file ⇔ type (class)

- Finding where to look is an important problem when maintaining code in a large project

- Binary (classfile) layout parallels the source layout
- Binaries are searched for on "classpath" or "modulepath" depending on deployment model

Organizing source code

- If using a class from another package it can be referred to "longhand"
  `java.util.List`
- Or the class can be imported
  Either: `import java.util.List;`
  Or: `import java.util.*;`

- Features of the package java.lang do not need to be imported, they are always in scope

- Importing is a "namespace" issue only
  - Importing does not make any changes in the generated binary
  - Importing does not change the size or speed of the running executable

Expressions and computations

Develop Intelligence

- Java has 8 "primitive" types:
  - boolean
  - char
  - byte, short, int, long
  - float, double
- Their behavior is predictable across platforms
- Variables must be declared, and have a single type throughout their lifetime

```
<type> <varname> [ = <expression> ] ;
int x = 99;
```

- Variables must be definitely assigned before reading / use
- Key binary arithmetic operators are:
  - +, -, *, /, %
- Binary operators "promote" operands to at least int, and to at least the larger of the two operant types.

Expressions and computations

Develop Intelligence

- Conditional expressions produce boolean results:
  - <, <=, >=, >
  - ==, !=
- Equals / not-equals works as expected for primitive types
  - but provides "object identity" test on non-primitive types

- Text may be represented using String:

```
String s = "Hello";
String t = s + " world!";
```

- String values are immutable, any operation that appears to change the string actually creates a new one
- Strings cannot safely be compared with ==
  - use dot-equals instead (helpful for many, but ***not all*** object types)

```
boolean same = s.equals(t); // value is false
```

Expressions and computations

- Bitwise and logical operations:

	Bitwise	Logical
and	&	&&
or	|	||
not	~	!
exclusive-or	^ `	^

- && and || are "short-circuiting"; right hand operand is not evaluated if the left hand side defines the result

Expressions and computations

Develop Intelligence

- Arrays create "many" variables accessed with a subscript

```
// ten numbers numbers[0] through numbers[9]
int [] numbers = new int[10];
System.out.println("count of numbers is "
  + numbers.length);

int [] num2 = {1, 3, 5, 7, 9};
```

- Arrays are of fixed length once created
  - Lists, a library feature, are typically preferred

Iteration and conditions

Develop Intelligence

- Java has four loop types:
- while, do while, C-style for, enhanced for

- While loop

```
int count = 0;
while (count < 3) {
  System.out.println("count is " + count);
  count++;
}
```

- Java tests must be `boolean` expressions
  - Java avoids the notion of "truthiness"

Iteration and conditions

Develop Intelligence

- do while loop

```
int count = 0;
do {
  System.out.println("count is " + count);
  count++;
} while (count < 3);
```

Iteration and conditions

Develop Intelligence

- C-style for loop

```
for (int count = 0; while (count < 3); count++) {
  System.out.println("count is " + count);
}
```

Iteration and conditions

Develop Intelligence

- Enhanced for loop works over arrays and collections

```
int [] numbers = {1, 3, 5, 7};
for (int n : numbers) {
  System.out.println("number is " + n);
}
```

- The enhanced for loop does not maintain a counter

Iteration and conditions

Develop Intelligence

- Java has three conditional structures
  - if/else, switch/case, and the conditional expression

- The if / else structure:

```
int x = 99;
if (x > 50) {
  System.out.println("x is large");
} else {
  System.out.println("x is not so big");
}
```

- Java does not need "elif", simply use `else if (...)`

Iteration and conditions

Develop Intelligence

- The switch / case structure:

```
int x = 99;
String message;
switch (x) {
  case 99: case 100: case 101: message = "x is 100 ish";
  break;
  case 0: case 1: case 2: message = "x is small";
  break;
  default: message = "x is boring";
  break;
}
```

Creating classes

Develop Intelligence

- A static method is code that can be accessed in the context of the class as a whole
  - These are the methods we have already defined
  - It does not require an instance of a date to operate
  - E.g. determine if a year is a leap year
  - Static methods have the prefix static

```
public class MyDate {
  static boolean isLeapYear(int year) {
    return year % 4 == 0 && year % 100 != 0
      || year % 400 == 0;
  }
}
```

Iteration and conditions

Develop Intelligence

- Switch on `int` (or smaller), `String`, and enumeration types
- `case` values must be constants
- `case` acts as a target for a "go-to" effect
  - Be careful to use `break` or execution flow will "fall through"
  - `return` from a method also avoids fall-through
- Falling through is helpful sometimes to allow the same path of execution for several conditions

Iteration and conditions

Develop Intelligence

- The conditional (aka "ternary") expression creates an if-else effect in a single expression:

```
boolean sometest = ???
String message =
  sometest ?
    "test value is true"
    : "test value is false";
```

- This code is typically laid out on a single line if space permits easy reading

Creating static methods

Develop Intelligence

- Where code is repeated, or creates a meaningful "utility", it should be extracted into a method
  - Sometimes called function, procedure, subroutine in other languages

<modifiers> <return type> <name> (<formal parameter list>) <body>

- For a static method `static` is one of the modifiers
- Other modifiers exist, `public` and `private` are common
- Name must be a legal Java "identifier"
  - Starts with letter, underscore (Unicode definitions)
  - Continues with letters, underscores, and digits
  - Avoid $, and single _
- Method body is enclosed in curly-braces

Creating static methods

Develop Intelligence

<modifiers> <return type> <name> (<formal parameter list>) <body>

- Formal parameters are local variable declarations in a comma separated list
  - <type> <name>, <type> <name> …
  - e.g. `String s, int c`
  - Their values are initialized for each method call by copying from the caller's values
  - The enclosing parentheses are always required, even if no formal parameters exist for this method
- Return value, matching the declared type must be supplied by a `return` statement in the body
- A method with return type `void` must not return a value, but may run off the end of the method, or use `return;` (without an expression)

Creating static methods

Develop Intelligence

```
public static int addUp(int a, int b) {
   System.out.println("Adding numbers");
   return a + b;
}

public static void show(String s, long l) {
   System.out.println("String is " + s);
   System.out.println("long is " + l);
   return; // optional, can run off end
}

public static void hello() {
   System.out.println("Hello!");
}
```

Using enumerations

Develop Intelligence

- For situations where a specific, known, and never-changing set of values exist, an enumeration is appropriate:

```
public enum DayName {
  MONDAY, TUESDAY, WEDNESDAY, THURSDAY,
  FRIDAY, SATURDAY, SUNDAY;
}

DayName dn = DayName.WEDNESDAY;
```

- Enumeration types can reduce the potential errors of using simple `int` values (how will code respond to day number -17, for example?)

Working with exceptions

Develop Intelligence

- If something goes wrong and it's necessary to abandon some operation as a result, an exception might be appropriate.
- When identifying the bad situation, throw an exception:

```
if (dayNumber < 1 || dayNumber > 7) {
  throw new IllegalArgumentException(
    "Bad day number " + dayNumber);
}
```

Working with exceptions

Develop Intelligence

- If an exception is thrown by code, it can be "caught"
  - This gives a chance to recover
  - Catching it, even if nothing is done, is considered to have handled the problem and execution flow returns to normal
- Use a try/catch or try/catch/finally construction
- "Happy path" goes in the try block
- Multiple catch blocks can be provided to handle different potential problems
- If no catch block matches the exception, then the exception has not been recovered and propagates out of this method
- A finally block can be used for code that must be run in success, recovered-failure, and unrecovered failure
  - This is less common in modern Java

Working with exceptions

Develop Intelligence

```
try {
  // something that might throw and exception
} catch (SomeException e1) {
  // recover from Some problem
} catch (OtherException e2) {
  // recover from Other problem
} finally {
  // do something in all situations
}
```

- In this structure a try cannot exist unless at least one catch or a finally exists
  - It would be pointless

Working with exceptions

Develop Intelligence

- Exceptions are very common with IO operations
  - Closing files is important whether the operation succeeds or fails
  - Try-with-resources is the right way to handle this (since Java 7)

```
try (
     FileReader fr = new FileReader("data.txt");
     BufferedReader br = new BufferedReader(fr);
) {
   System.out.println(br.readLine());
} catch (IOException ioe) {
   System.out.println("That broke: " + ioe);
}
```

- "Resources" (`AutoCloseable`) opened in `try (...)` are closed

Working with exceptions

Develop Intelligence

- Some exceptions are "checked"
- These are normal problems in the environment
  - A quality program should include code for recovery, rather than simply crashing
  - Java identifies these exceptions (not always perfectly) and requires some code to address the problem
- Either surround the source of the problem with a try/catch
- Or declare that the method containing the problem "throws" the exception(s) with a comma separated list of their types:

```
public static void mightBreak()
    throws IOException, SQLException {
  // method code that might break
}
```

Creating classes

Develop Intelligence

- Classes are the "home" of code that provides the behavior (code) that implements a single concept that is being modeled in a software system
- Classes also act as a template for how to build the data model for that same concept, e.g. for a calendar date:

  int day, month, year;

- The code in the class might act on the data representation as a whole, or on other data that relate, e.g. in modeling a date:
  - find the name of day number 3
  - find the number of days in March 2020
  - find the day of the week of July 4th, 1776
  - find the date four years, 2 months, and 13 days after this date

Creating classes

- Define a class with this syntax:

<modifiers> class <name> [<parent information>] <class body>

- Modifiers often include public, and might be abstract or final in special cases
- The name is any valid identifier, but stylistically should start uppercase
- Class body is enclosed in curly-braces
- Optional parent information describes superclasses or interfaces E.g.:

```
public class MyDate
    extends CalendarItem implements Printable {
  // class body code
}
```

Creating classes

- The class body can declare fields, methods, and other classes
- An instance "field" becomes a variable that is a part of an object when a class is instantiated.
  - E.g. day would be a field in a date.
  - Create ten date objects, get ten day fields, one in each
- Instance field accessibility can be `private`, <default> `protected`, or `public`
  - Omitting the keyword gives <default> access
  - `private` is "normal"
- Define instance fields in classes as for simple variables:

```
public class MyDate {
  private int day;
  private int month;
  private int year;
}
```

Creating classes

- Accessibility governs what other code can see and use an element
- This concept relates to all field types, and method types
- Some accessiblities may be used on classes too, depending on circumstances
- `private` means "visible withing the enclosing top-level curly braces surrounding this declaration
- <default> means visible anywhere within the same package
- `protected` means visible anywhere <default> is visible plus in sub-classes (with some constraints)
- `public` means visible anywhere in this module, or anywhere within the JVM if non-modular
  - `public` elements of exported packages may be visible in other modules if the containing package is exported by the owning module, and read by the other

Creating classes

Develop Intelligence

- A "static field" is a variable that is a part of the class.
  - One variable exists in the class, and may be accessed from any instance
  - The value is effectively "shared'
  - E.g. daysInJanuary could be a static field in the MyDate class.
  - Create zero or ten date objects, always have one daysInJanuary value
- Define static fields in classes as for simple variables with the prefix `static`:

```
public class MyDate {
  static int daysInJanuary = 31;
}
```

Creating classes

Develop Intelligence

- Static methods define behavior that relates to the concept modeled by the class, but that do not relate specifically to an individual instance
  - find the name of day number 3
  - find the number of days in March 2020

```
public class MyDate {
  static String dayName(int dayNum) {
    // body of method
  }
  static int numberOfDays(int monthNum) {
    // body of method
  }
}
```

Creating classes

Develop Intelligence

- Instance methods define behavior that relates to the concept modeled by the class, and specifically act on an individual instance, e.g.:
  - find the day of the week of July 4th, 1776
  - find the date four years, 2 months, and 13 days after this date
- Instance methods:
  - ***Omit*** the word static
- The first formal parameter of an instance method:
  - Is of exactly the enclosing class type
  - Has/must have the special/reserved name `this`
  - My optionally be omitted from the parameter list, in which case it is implicit
- Instance methods must be invoked using a subject-verb grammar rather than a verb-object grammar

Creating classes

- Instance method declaration example using explicit `this`:

```
public class MyDate {
  int day, month, year;

  void setDayOfMonth(MyDate this, int day) {
    this.day = day;
  }
}
```

Creating classes

- Instance method declaration example using implicit `this`:

```
public class MyDate {
  int day, month, year;

  void setDayOfMonth(int day) {
    this.day = day;
  }
}
```

- Note that this example is identical in effect to the preceding one.

Creating classes

- Instance method invocation:

```
MyDate md … // initialized in some way
md.setDayOfMonth(3);
```

- Note that the prefix value `md` is assigned to the value **`this`** inside the invoked method body
- The prefix is mandatory for an instance method and must not be null

Creating classes

Develop Intelligence

- To initialize the fields of a new object, provide initialization behavior in a special behavior called a "constructor"

```
public class MyDate {
  private int day, month, year;
  public MyDate(int day, int month, int year) {
    this.day = day;
    this.month = month;
    this.year = year;
  }
}
```

Inheritance in Java

Develop Intelligence

- A subclass creates a variation on the parent class
  - Additional features can be added
  - Implementation details may be changed
  - Instances of the subclass may be substituted where instances of the parent class are expected (the "is a" relationship)
  - All the features of the parent class will exist in the subclass by default
- To define a subclass, specify the parent in the extends clause of the subclass' declaration:

```
public class MyHoliday extends MyDate {
  private String name;
}
```

Inheritance in Java

Develop Intelligence

- Subclass constructors must make their first action to properly initialize the parent features:

```
public class MyHoliday extends MyDate {
  private String name;
  public MyHoliday(String name,
      int day, int month, int year) {
    super(day, month, year);
    this.name = name;
  }
}
```

Inheritance in Java

Develop Intelligence

- Subclasses can modify the definition of parent instance methods.
  - This is called "overriding"
  - Define an instance method with identical name and argument type sequence as that of the parent method
  - Return type must be "compatible"
- Objects are converted to text using the method `toString`, defined in the base class `java.lang.Object` in this manner:

```
public class Object {
  public String toString(Object this) {
    return <textual representation>;
  }
}
```

Inheritance in Java

Develop Intelligence

- User defined types can redefine the behavior of text conversion by overriding the default/inherited behavior like this:

```
public class MyDate {
  @Override // optional, but advised
  public String toString() { // implicit this
    return "My Date, day is " + this.day +
      " month is " + this.month;
  }
}
```

Inheritance in Java

Develop Intelligence

- An instance of a subclass can be used anywhere an instance of the parent class is expected "Liskov Substitution"
- Therefore a declaration of this kind is legal:

```
MyDate md = new Holiday("New Years", 1, 1, 2022);
```

- And given a method of this form `doStuff(Date d)` and an instance of `Holiday` called `myHoliday`, a call of this form is also legal:

```
doStuff(myHoliday);
```

Inheritance in Java

Develop Intelligence

- Given the example:

```
MyDate md = new Holiday("New Years", 1, 1, 2022);
```

- An invocation like the following will invoke the `toString` behavior of the `Holiday` class (if overridden) rather than that of the `MyDate` class

```
String text = md.toString();
```

- This is known as:
  - Dynamic method invocation
  - Virtual method invocation
  - Late binding

Working with Collections

Develop Intelligence

- The package `java.util` defines several useful "collection" types:
  - `Iterable`
  - `List`
  - `Set`
  - `Map`
- These are *interfaces*, which describe the methods available separately from the implementation by some other class
- Differing implementations are available for several of these
  - The implementations have differing performance characteristics depending on usage patterns
- These types are "generic", which refers to a language mechanism that allows indicating the type of data that should be contained and provides compile-time verification that the usages made are correct

Working with Collections

Develop Intelligence

- The `java.util.List` interface:
  - Provides an array-like storage for multiple items
  - The items have a user-controlled order
  - Items can be added to the list at any given position
  - The length grows and shrinks automatically as needed
  - Items can be changed, or removed
  - Items can be retrieved by index
  - Items can be searched (provided the items implement `equals` properly)

```
List<String> names = new ArrayList<>();
names.add("Fred");
System.out.println(names.get(0));
boolean found = names.contains("Fred");
names.set(0, "Frederick");
```

Working with Collections

Develop Intelligence

- The `java.util.Set` interface:
  - Provides storage for multiple items
  - Duplicates are not added
  - The order of items is not under user control and might change
  - Items can be searched for efficiently
    - Some Set implementations require items implement `equals` properly, others expect the items to have an ordering

```
Set<String> names = new HashList<>();
names.add("Fred"); // succeeds, returns true
names.add("Fred"); // fails, returns false
boolean found = names.contains("Fred"); // true
```

Working with Collections

Develop Intelligence

- The `java.util.Iterable` interface:
  - Provides for taking items one at a time from some source
  - Order might or might not be significant
  - Any Iterable implementation can be used in the enhanced-for loop
  - An Iterable is usually used to obtain an `Iterator`, which performs, and maintains the progress of, the iteration process
  - Iterator provides `hasNext()` and `next()` methods for the iteration
  - Both `List` and `Set` implement `Iterable` (but `Map` does not)

```
List<String> aList = List.of("Fred", "Jim", "Sheila");
Iterator<String> it = aList.iterator();
while (it.hasNext()) {
  System.out.println(it.next());
}
```

Working with Collections

Develop Intelligence

- The `java.util.Map` interface:
  - Provides storage for multiple key-value pairs
  - Storing against a duplicate key overwrites the previous value
  - Keys can be searched for efficiently, yielding the associated value

```
Map<String, String> names = new HashMap<>();
names.put("Fred", "Jones");
String name = names.get("Fred"); // returns "Jones"
name = names.get("Alan"); // returns null
```

Using Lambdas

Develop Intelligence

A lambda expression in Java defines an object in a context.

- The context must require an implementation of an interface
- The interface must declare ***exactly one*** abstract method
- We must ***only*** want to implement that one abstract method
- We provide a modified method argument list and body with an "arrow" between them:

```
(Student s) -> {
  // function body
}
```

- The argument list and return type must conform to the abstract method's signature

Using Lambdas

Develop Intelligence

- Argument types can be omitted if they're unambiguous
  - This is "all or nothing"
- Since Java 10, argument types can be replaced with `var` if they're unambiguous
  - This is also "all or nothing"
- If a single argument carries zero type information the parentheses can be omitted
- If the method body consists of a single `return` statement, the entire body can be replaced with the expression that is to be returned.

Using Lambdas

- If an interface is intended for use with lambdas, it must define exactly one abstract method.
- The `@FunctionalInterface` annotation asks the compiler to verify this and create an error if this is not the case.

Using LambdasPredefined interfaces

Develop Intelligence

Due to Java's strong static typing, and the restrictions preventing generics being used with primitives, different interfaces must be provided for a variety of different situations.

Key interface categories are:

`Function` - takes argument, produces result

`Supplier` - zero argument, produces result

`Consumer` - takes argument, returns void

`Predicate` - takes argument, returns boolean

`Unary/Binary Operator` - `Function` variants that lock arguments and returns to identical type

Using LambdasPredefined interface variations

For two arguments, expect a "`Bi`" prefix

For primitives:

- `Int`, `Long`, `Double` prefix usually means "primitive argument"
  - Note for `Supplier`, this prefix refers to return type (there are zero arguments.)
- `ToInt`, `ToLong`, `ToDouble` prefix means "primitive return"

Unit Testing

Develop Intelligence

- The main method is the traditional entry point to an entire program
- Testing usually involves an entry point per test
  - I.e. a great many entry points
- JUnit and TestNG are common test frameworks that provide for executing many tests as separate operations
  - Each test is annotated with `@Test` and becomes an entry point
  - Tests should be zero argument instance methods in classes dedicated to testing
  - Test classes are placed in a different directory structure from code implementing the main project (this allows separating the tests from the deliverable binaries)

Unit Testing

Develop Intelligence

- Tests should be written in three parts:
  - Configuration of the conditions of the test
  - The action to be tested
  - The expected results described using "assertions"
- The test system is implemented in libraries that must be made available
- For example, if using maven, add this to the project's `pom.xml`:

```
<dependencies>
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>RELEASE</version>
        <scope>test</scope>
    </dependency>
</dependencies>
```

Unit Testing

Develop Intelligence

```
import org.junit.Assert;
import org.junit.Test;

public class SimpleTest {
  @Test
  public void testTwoPlusThree() {
    int two = 2;
    int three = 3;
    int result = two + three;
    Assert.assertEquals("2 + 3 should be 5", 5, result);
  }
}
```

- Many tests, and many test classes, can be added

THANK YOU

Develop Intelligence