

Introductory Java 1



Imperative programming languages
Java Standard Library
Types
Hello World



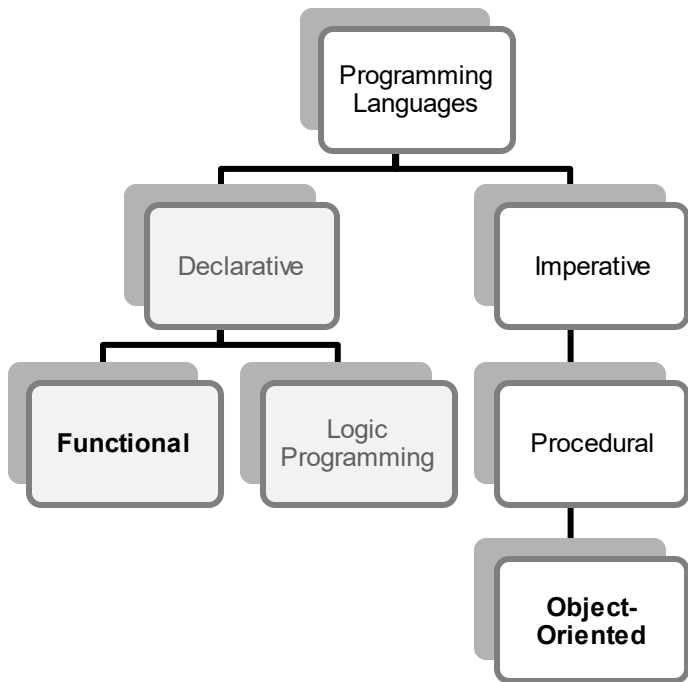
Why Java?

- Learn multiple programming paradigms
- Important example of:
 - Object-oriented programming
 - Large scale programming
 - Programming with a rich standard library

Imperative Programming Languages

Declarative languages describe the desired result without explicitly listing steps required to achieve that goal.

Pure functional languages, like Haskell, only transform state using functions without side effects.



Imperative languages describe computation in terms of a series of statements that transform state.

Object-oriented languages use structured (procedural) code, tightly coupling data with the code that transforms it.

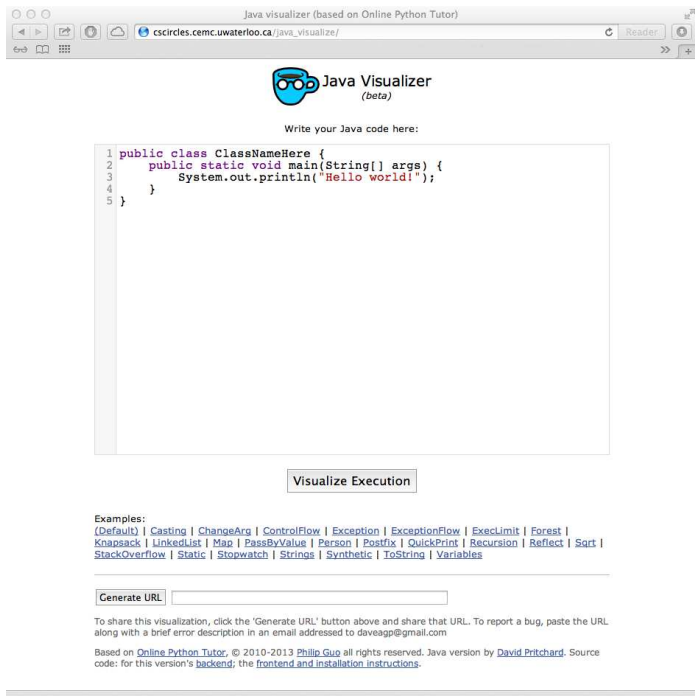
Imperative Programming Languages

- Sequence
- Selection
- Iteration

Object Oriented Programming Languages

- Structured code
- Code (*behavior*) tightly coupled with data (*state*) that it manipulates

The Waterloo Java Visualizer



Java visualizer (based on Online Python Tutor)

cscircles.cmc.uwaterloo.ca/java_visualize/

Java Visualizer
(beta)

Write your Java code here:

```
1 public class ClassNameHere {
2     public static void main(String[] args) {
3         System.out.println("Hello world!");
4     }
5 }
```

Visualize Execution

Examples:
[\(Default\)](#) | [Casting](#) | [ChangeArg](#) | [ControlFlow](#) | [Exception](#) | [ExceptionFlow](#) | [ExecLimit](#) | [Forest](#) | [Knapsack](#) | [LinkedList](#) | [Map](#) | [PassByValue](#) | [Person](#) | [Postfix](#) | [QuickPrint](#) | [Recursion](#) | [Reflect](#) | [Sort](#) | [StackOverflow](#) | [Static](#) | [Stopwatch](#) | [Strings](#) | [Synthetic](#) | [ToString](#) | [Variables](#)

Generate URL

To share this visualization, click the 'Generate URL' button above and share that URL. To report a bug, paste the URL along with a brief error description in an email addressed to daveagp@gmail.com

Based on [Online Python Tutor](#), © 2010-2013 Philip Guo all rights reserved. Java version by [David Pritchard](#). Source code: for this version's [backend](#); the [frontend](#) and [installation instructions](#).

A great resource. Type in simple Java programs and watch step-by-step execution. A great way to enhance your understanding of a new language.

The Oracle Java Tutorials

This course follows the structure of the *Oracle Java Tutorials* for the basic introduction to Java.

The tutorials are subject to Oracle's 'Java Tutorial Copyright and License' (Berkeley license).

We will move very fast for the first few weeks. You should use the tutorials to **ensure that you rapidly become proficient in the basics of Java.**

The Java Standard Library

- The Java language is augmented with a large standard library
(.NET does the same for C#)
 - IO (accessing files, network, etc.)
 - Graphics
 - Standard data structures
 - Much more
- Using and understanding the standard library is part of learning a major language like Java or C#.
- Rich standard libraries are a key software engineering tool.

Data types

The *type* of a unit of data determines the possible values that data may take on.

- Weak v strong
 - Must all data be typed? Can types be coerced or converted?
- Static v dynamic
 - Is checking done at compile-time or run-time?

Haskell: strong, static

Java: strong, static and dynamic

Introductory Java 2

J2

Types

Objects

Classes

Inheritance

Interfaces

Objects

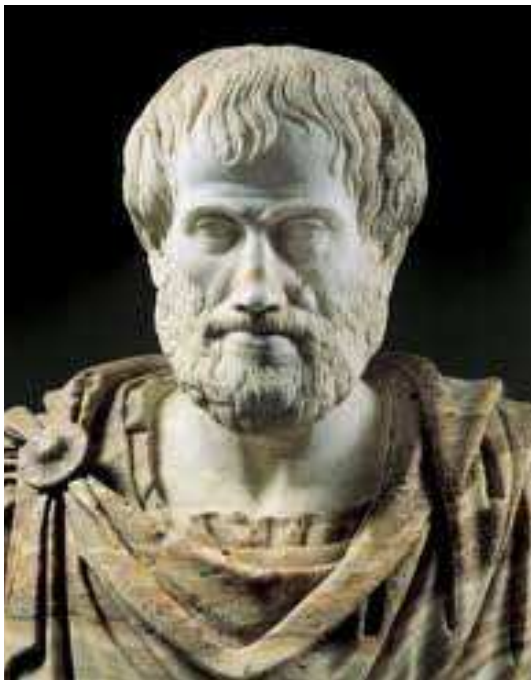
Objects combine state and behavior

- State: *fields* (data types)
- Behavior: *methods* (imperative code)

Example: bicycle

- State: current speed, direction, cadence, & gear
- Behavior: change cadence, change gear, brake

Classes



Aristotle 384-322BC

‘Blood-bearing animals’:

1. Four-footed animals with live young,
2. Birds,
3. Egg-laying four-footed animals,
4. Whales,
5. Fishes

Classes

A class is a blueprint or 'type' for an object

- Instance: one instantiation of a class (aka object)
- Class: blueprint / definition for many instances

Example: bicycle

- Instance: your bike
- Class: Kona Jake The Snake 2012



Inheritance

Classes may form a hierarchy

- sub-class: **extends** a super-class

Example: bicycle

- class: KonaJakeTheSnake2012
- super-class: CyclocrossBike
- super-class: UprightBike
- super-class: Bike
- super-class: Object



Java Interfaces

Methods define behavior

- An **interface** is a group of methods *without* implementations

Example: an interface `MovableThing` might include:

- `brake()`
- `speedup()`

Any class that **implements** `MovableThing` must include definitions of these methods.

Introductory Java 3

J3

Naming

Literals

Primitives

Java Modules

- A module is a named group of packages and related resources
- Strong encapsulation
- Explicit dependencies

```
module java.sql {  
    requires transitive java.logging;  
    requires transitive java.transaction.xa;  
    requires transitive java.xml;  
  
    exports java.sql;  
    exports javax.sql;  
  
    uses java.sql.Driver;  
}
```

Java Packages



Which Mary?

Mary Queen of Scots

‘Queen of Scots’ provides a namespace within which ‘Mary’ is well defined. In Java a **package** provides a namespace.

Java Variables

- Instance (non-static fields, object-local)
 - Each object has its own version (instance) of the field
- Class (static fields, global)
 - Exactly one version of the field exists
- Local
 - Temporary state, limited to execution scope of code
- Parameters
 - Temporary state, limited to execution scope, passed from one method to another

Java Naming

- Java names are case-sensitive
 - Whitespace not permitted
 - \$, _ to be avoided
 - Java keywords and reserved words cannot be used
- Capitalization conventions
 - Class names start with capital letters (`Bike`)
 - Variable names start with lower case, and use upper case for subsequent words (`currentGear`)
 - Constant names use all caps and underscores (`MAX_GEAR_RATIO`)

Java's Primitive Data Types

In addition to *objects*, Java has 8 special, built-in '*primitive*' data types.

| type | Description | Range | Default |
|----------------|--|----------------------|--------------|
| byte | 8-bit signed 2's complement integer | -128 - 127 | 0 |
| short | 16-bit signed 2's complement integer | -32768 - 32767 | 0 |
| int | 32-bit signed 2's complement integer | $-2^{31} - 2^{31}-1$ | 0 |
| long | 64-bit signed 2's complement integer | $-2^{63} - 2^{63}-1$ | 0L |
| float | single precision 32-bit IEEE 754 floating point number | | 0.0f |
| double | double precision 64-bit IEEE 754 floating point number | | 0.0d |
| boolean | logically just a single bit: true <i>or</i> false | true, false | false |
| char | 16-bit Unicode character | 0 - 65535 | 0 |

Java Literals

- When a numerical value (e.g. '1') appears, the compiler normally knows exactly what it means.
- Special cases:
 - An integer value is a `long` if it ends with 'l' or 'L'
 - The prefix `0x` indicates hexadecimal, `0b` is binary:
 - `0x30` // 48 expressed in hex
 - `0b110000` // 48 expressed in binary
 - An 'f' indicates a float, while 'd' indicates double.
 - Underscores may be used to break up numbers:
 - `long creditCardNumber = 1234_5678_9012_3456L;`

Introductory Java 4

J4

Arrays, Operators, Expressions
Statements, Blocks, Random

Java Arrays

- Arrays hold a fixed number of values of a given type (or sub-type) that can be accessed by an index

- Declaring:

```
int[] values;
```

- Initializing:

```
values = new int[8]; // 8 element array
```

- Accessing:

```
int x = values[3]; // the 4th element
```

- Copying:

```
System.arraycopy(x, 0, y, 0, 8);
```


Java Operators

- Assignment

=

- Arithmetic

+ - * / %

- Unary

+ - ++ -- !

- Equality, relational, conditional and `instanceof`

== != > >= < <= && || `instanceof`

- Bitwise

~ & ^ | << >> >>>

Expressions

- A construct that evaluates to a **single value**.
- Made up of
 - variables
 - operators
 - method invocations
- Compound expressions follow precedence rules
 - Use parentheses (clarity, disambiguation)

Statements

- A complete unit of execution.
- **Expression** statements (expressions made into statements by terminating with ‘;’):
 - Assignment expressions
 - Use of ++ or --
 - Method invocations
 - Object creation expressions
- **Declaration** statements
- **Control flow** statements

Blocks

- Zero or more statements between balanced braces ('{' and '}')
- Can be used anywhere a single statement can

The Random Class

The `Random` class provides a pseudo-random number generator:

```
Random rand = new Random();
```

You can optionally provide a seed (for determinism):

```
Random rand = new Random(12345);
```

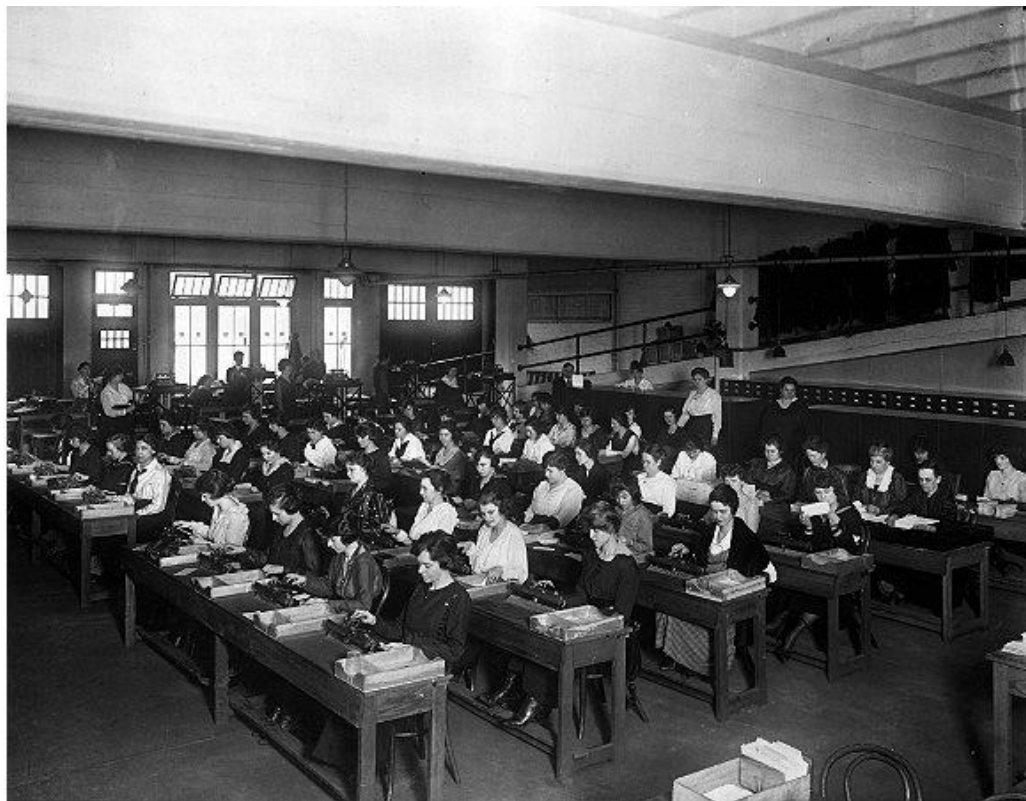
You can then generate random numbers of different types:

```
int i = rand.nextInt(10); // number in 0-9
```

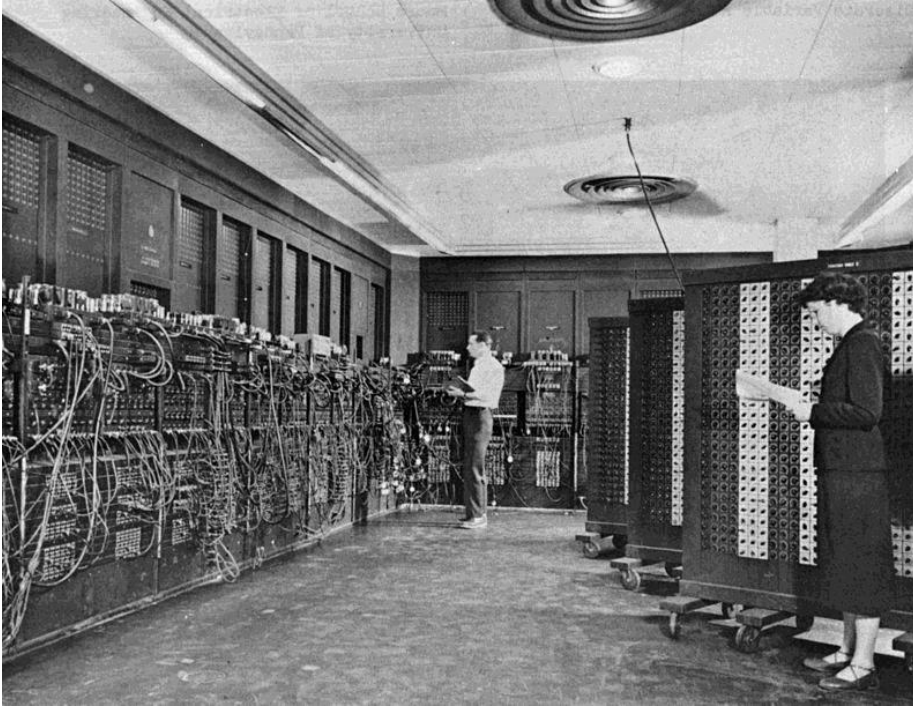
Control Flow 1

J5

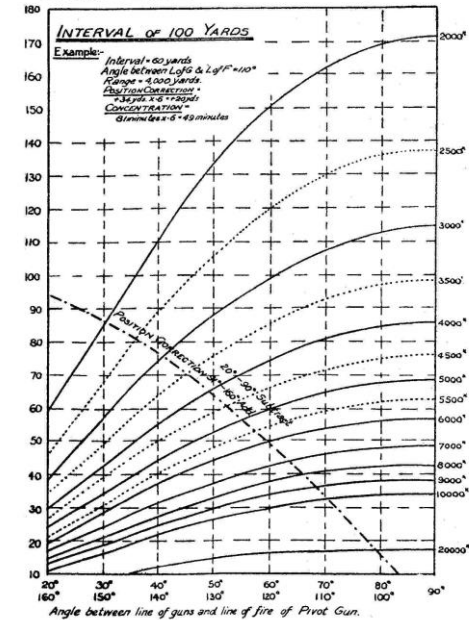
Control flow
if-then-else
switch



Women workers ('computers') in a calculation "factory," 1930s. Courtesy of the Library of Congress.



GRAPH SHOWING:-
 1. **CONCENTRATIONS**, at varying angles between line of guns and line of fire for 100 yards interval. (Continuous and dotted curves).
 2. **POSITION CORRECTIONS**, as above. (Chain dotted curve).



Calculating a trajectory could take up to 40 hours using a desk-top calculator. The same problem took 30 minutes or so on the Moore School's differential analyzer. But the School had only one such machine, and since each firing table involved hundreds of trajectories it might still take the better part of a month to complete just one table. [Winograd & Akera 1996]



Source: Ad Meskens, Wikimedia Commons



Control Flow

Control flow statements allow the execution of the program to deviate from a strictly sequential execution of statements (*selection*).

Imperative programming: sequence, ***selection***, iteration.

if-then & *if-then-else* statements

- The *if-then* construct *conditionally* executes a block of code.
- The *if-then-else* construct *conditionally* executes one of two blocks of code

The **switch** statement

- The **switch** statement selects one path among *many*.
- Execution *jumps* to the first matching **case**.
- Execution *continues* to the end of the **switch** unless a **break** statement is issued.

The `switch` expression

- The `switch` expression selects one *value* among many.
- Execution jumps to the first matching `case`.
- The value of the expression is given by the `yield` operator in the matching case.

Control Flow 2

J6

Control flow
while & do-while
for
break, continue, return

The **while** & **do-while** statements

- The **while** statement continuously executes a block while a condition is **true**.
- The **do-while** construct evaluates the condition at the *end* of the block rather than at the start.

Imperative programming: sequence, selection, ***iteration***.

The **for** statement

- A compact way to *iterate* over a set of values.
- The statement has three logical parts:
 - Initialization
 - Termination condition
 - Increment statement
- The ‘enhanced’ **for** statement *infers* the initialization, termination and increment statements, given an array or collection

Branching statements

- The **break** statement terminates a loop construct
 - *Unlabeled* terminates the loop in which it is called
 - *Labeled* terminates the loop named by the label
- The **continue** statement skips the current iteration of a loop
 - *Unlabeled* skips the current iteration of the loop in which it is called
 - *Labeled* skips the current iteration of the loop named by the label
- The **return** statement exits the current method

Methods

J7

Methods

Parameters

Return values

Methods

- A subroutine
 - Reusable code to perform a specific task
 - Modularity, encapsulation
- May take arguments (parameters)
- May return a value

Method Declaration

Method declarations will have the following, in order:

- Any **modifiers** (`public`, `private`, etc.)
- **return type**
- **method name**
- **parameters**, in parentheses
- Any **exceptions** the method may throw
- The method **body** (code)

```
class String {  
    public byte[] getBytes(String charsetName)  
    throws UnsupportedOperationException {...}  
    ... }
```

Class and Instance methods

A method declared with the **static** modifier is a **class** method (otherwise it is an **instance** method)

- Class methods
 - May operate on class fields only
- Instance methods
 - May operate on class *and* instance fields

Parameters (method arguments)

Parameters are the mechanism for passing information to a method or constructor.

- Primitive types passed by *value*
 - Changes to parameter **are not seen** by caller
- Reference types passed by *value*
 - Changes to the *reference* **are not seen** by caller
 - Changes to *object referred to* **are seen** by caller
- Your last parameter may in fact be more than one parameter (*varargs*), and treated as an array

Returning a Value from a Method

The **return** statement exits the current method

Methods return to caller when:

- all statements in method executed, or
- a **return** statement is reached, or
- the method throws an exception (later)

Methods declared **void** do not return a value.

All other methods must return a value of the declared type
(or a *subclass* of the declared type, described later).

Nested Classes

J8

Nested classes

Nested Classes

A class may be defined within another class. Such a class is called a *nested class*. The main motivation for nested classes is to improve encapsulation and clarity.

- **Static nested classes** (use `static` keyword) behave as if declared elsewhere, but happen to be packaged together in a single file, cannot refer directly to instance fields of parent
- an **inner class** (non-static) has direct access to the instance fields and members of its enclosing class.

Lambda Expressions

J9

Lambda Expressions

From Java version 8, lambda expressions allow code to be passed as a parameter, just like data.

- Particularly useful for event handling; can pass *behavior* as an argument ('do *this* when x happens').
- Syntax
 - Comma-separated formal parameters (`x`)
 - Arrow (`->`)
 - Body (either single expression or statement block, which may contain return)

```
x -> x > 100    or    x -> { ... return true; }
```

Functional Interfaces

A lambda expression implements a *functional interface*: an interface which only defines a single method.

Commonly-used functional interfaces are defined in package `java.util.function`, e.g.

```
public interface IntPredicate {  
    boolean test(int value);  
}
```

```
public interface DoubleSupplier {  
    double getAsDouble();  
}
```

Number, Autoboxing

J10

Number, Integer, Short, Float, etc
Autoboxing

Math

The Number Classes

Normally you will represent numbers with the **primitive** types `int`, `short`, `float`, etc. Java includes ‘boxed’ object analogues to each of these: `Integer`, `Short`, `Float`, etc.

- Number classes have methods (primitives don’t)
 - `toString()`, `parseInt()`, etc.
- Number classes have constants
 - `Integer.MIN_VALUE`, `Short.MAX_VALUE`, etc
- Number classes have a space overhead
 - They are instantiated as true objects

Autoboxing

Classes such as `Integer` and `Character` are ‘boxed’ versions of the primitive types `int` and `char` (i.e. object versions of the primitives). Java offers automatic support for boxing and unboxing.

- Boxing: `Integer i = 5;`
- Unboxing: `int j = i;`

The `Math` class

The `Math` class contains methods and constants useful for basic mathematics:

- Constants: `Math.PI` and `Math.E`
- Trigonometry: `sin()`, `cos()`, **etc.**
- Rounding: `abs()`, `ceil()`, `floor()`, **etc.**
- Comparison functions: `max()`, `min()`
- Exponentials and logs: `exp()`, `log()`, `pow()`, **etc.**
- Random number generation: `random()`

Character and String

J11

Character and String

The Character Class

The `Character` class boxes `char`, just as `Integer` boxes `int`. It contains methods and constants useful for manipulating characters:

- **Property methods:** `isLetter()`, `isDigit()`, **etc.**
- **Conversion:** `toString()` (a single character string!)

Escape sequences are used to represent characters that have a special meaning in Java syntax:

- `\'`, `\"`, `\\`, `\n`, **etc.**

The `String` Class

The `String` class is provided by Java to store and manipulate strings (by contrast, in C, a string is simply an array of characters).

- Implicit creation from literal:

```
String x = "foo";
```

- Concatenation with "+":

```
String y = x + "bar";
```

- `StringBuilder` class

Operations on Strings

- Get `length` (number of characters):

```
if (x.length() > 3) ...
```

- Get a character with `charAt()`
- Get a substring with `substring()`
- Others: `split()`, `trim()`, `toLowerCase()`, **etc.**
- Finding: `indexOf()`, `contains()`, **etc.**
- Replacing: `replace()`, `replaceAll()`, **etc.**

Generics

J12

Generics

Sometimes it is useful to parameterize a class with a type, `T`.

Rather than `IntContainer`, `LongContainer`, etc. we can just write `Container<T>`, and then create instances such as `Container<Integer>`.

We can also create generic methods that accept type parameters:
`static <T> void acceptSomeValue(T value) { ... }`

Prior to the introduction of Java generics, programmers often used `Object` as a work-around as it can refer to any non-primitive type.

Type Inference

J13

Generic Type Inference

Lambda Expressions

Local Variables

Type Inference

The Java compiler can infer many types from context, cutting down on boilerplate code.

Instantiating generic classes:

```
LinkedList<String> s = new LinkedList<>();
```

Generic methods:

```
public <T> void add(T value) { }
```

```
list.add("A String");
```

Local Variables

With the `var` keyword, Java can infer the type of a local variable from its initialization expression.

The most specific type is inferred.

```
var theAnswer = 42;
```

```
var bike = new Bike();
```

```
var mystery; // invalid - no initializer
```

```
var nothing = null; // invalid - null has no type
```

Lambda Expressions

Types of **parameters** to lambda expressions:

```
Predicate<String> nonEmpty = x -> x.length() > 0;
```

However, can't infer the type of a lambda expression:

```
var lambda = x -> x + 1; // invalid - what type is x?
```

```
var lambda = (int x) -> x + 1; // invalid - what is lambda?
```

```
IntFunction lambda = (int x) -> x + 1; // OK
```

Collections

J14

The Collections Framework
for Each
Ordering Collections

The Collections Framework

- Interfaces
 - Implementation-agnostic interfaces for collections
- Implementations
 - Concrete implementations
- Algorithms
 - Searching, sorting, etc.

Using the framework saves writing your own: better performance, fewer bugs, less work, etc.

The Collection Interface

- Basic operators
 - `size()`, `isEmpty()`, `contains()`, `add()`, `remove()`
- Traversal
 - for-each, and iterators
- Bulk operators
 - `containsAll()`, `addAll()`, `removeAll()`, `retainAll()`, `clear()`
- Array operators
 - convert to and from arrays

Collection Types

- Primary collection types:
 - Set (no duplicates, mathematical set)
 - List (ordered elements)
 - Queue (shared work queues)
 - Map (<key, value> pairs)
- Each collection type is defined as an interface
 - You need to choose a concrete collection
 - Your choice will depend on your needs

Concrete Collection Types

| | <i>Implemented Using</i> | | | | |
|-------------------|--------------------------|-----------------|---------|-------------|--------------------------|
| <i>Interfaces</i> | Hash table | Resizable array | Tree | Linked list | Hash table + linked list |
| Set | HashSet | | TreeSet | | LinkedHashSet |
| List | | ArrayList | | LinkedList | |
| Queue | | ArrayDeque | | LinkedList | |
| Map | HashMap | | TreeMap | | LinkedHashMap |

Based on table from <http://docs.oracle.com/javase/tutorial/collections/implementations/index.html>

Four Commonly Used Collection Types

- `HashSet` implements a **set** as a hash table
 - Makes no ordering guarantees
- `ArrayList` implements a **list** using an array
 - Very fast access
- `HashMap` implements a **map** using a hash table
 - Makes no ordering guarantees
- `LinkedList` implements a **queue** using a linked list
 - First-in-first-out (FIFO) queue ordering

forEach

- Collections implement the `forEach` method, which applies an action to every element in the collection.

Instead of:

```
for (Thing t : things) {  
    System.out.println(t);  
}
```

You can do this:

```
things.forEach(t -> System.out.println(t));
```

Ordering Collections

- The `Comparable` interface defines a ‘natural’ ordering for all instances of a given type, `T`:

```
public interface Comparable<T> {  
    public int compareTo(T o);  
}
```

The return value is either negative, 0, or positive depending if the receiver comes before, equal, or after the argument, `o`.

- The `Comparator` interface allows a type `T` to be ordered in additional ways:

```
public interface Comparator<T> {  
    int compare(T o1, T o2);  
}
```

`Collections.sort()`

- No arguments
 - uses *natural* order for type
- Single Lambda argument:
 - uses order defined by lambda expression
 - `(a T, b T) -> { return <expression>; }`

Josh Bloch Item 25: Prefer lists to arrays

- Why?
 - Arrays are covariant, Generics are invariant
 - if A **extends** B, then A[] is a subclass of B[]
 - but List<A> has no relationship to List

```
// Fails at runtime!
Object[] objectArray = new Long[1];
objectArray[0] = "I don't fit in";           // Throws ArrayStoreException

// Won't compile!
List<Object> ol = new ArrayList<Long>(); // Incompatible types
ol.add("I don't fit in");
```

Exceptions

J15

Java Exceptions
Catch or Specify
Java syntax

Exceptions

Exceptions are a **control flow construct** for **error management**.

- Some similarity to event handling (lecture topic X2)
 - Both disrupt the normal flow of execution, transferring to event handler or exception handler
 - However: exceptions are *exceptional* situations (events are expected)
 - A file is not found or is inaccessible
 - An array is accessed incorrectly (out of bounds)
 - Division by zero
 - A null pointer is dereferenced, etc.

Java Exceptions

Exceptions are *thrown* either:

- Implicitly (via a program error) or
- Explicitly (by executing the `throw` statement).

Exceptions are *caught* with a `catch` block.

Exceptions are propagated from callee to caller until a matching handler is found. Methods throwing uncaught exceptions must have the `throws` clause in their declaration.

Java's **Catch** or **Specify** Requirement

Three kinds of exception:

- **error** (`Error` and its subclasses),
- **runtime exception** (`RuntimeException` and its subclasses),
- **checked** (everything else, must comply with **Catch** or **Specify**)

Java requires that code that may throw a checked exception must be enclosed by either:

- a **try** statement with a suitable handler, or
- a method that declares that it **throws** the exception

Java **try/catch** Block Syntax

```
try {  
    // do something that may generate an exception  
} catch (ArithmeticException e1) { // first catch  
    // this is an arithmetic exception handler  
    // handle the error and/or throw an exception  
} catch (Exception e2) { // may have many catch blocks  
    // this an generic exception handler  
    // handle the error and/or throw an exception  
} finally {  
    // this code is guaranteed to run  
    // if you need to clean up, put the code here  
}
```

Java Threads

J16

Thread and Runnable
start(), join() and sleep()
Races and synchronized

Thread **and** Runnable

- The `Thread` class is used to create threads and interact with them.
- Two ways to create a thread:
 1. Subclass `Thread`, extending its `run()` method.
 - Advantages: class inherits all of `Thread`'s methods
 - Disadvantages: can't subclass anything else
 2. Use the `Runnable` interface and implement its `run()` method.
 - General, but does not inherit `Thread`'s methods

`start()`, `join()` and `sleep()`

- Calling `t.start()` will start execution of the `run()` method within the thread `t` (and continue with execution of the current thread).
- Calling `t.join()` will cause the current thread to wait until thread `t` terminates.
- Calling `Thread.sleep(ms)` will cause the current thread to go to sleep for `ms` milliseconds.

Races and the **synchronized** keyword

- Too many cooks...
 - Coordination is the big challenge of concurrency
 - How do we avoid conflicts?
 - How do we impose some level of coherence and order?
- A 'race condition' is a situation where one or more threads race non-deterministically to be the first to read or write a variable
- The **synchronized** keyword
 - Qualify a method, ensures only one thread executes that method at any time