

A painting of a rural landscape. In the foreground, there are several large haystacks made of golden-brown straw. A person wearing a white shirt and dark trousers is standing in the middle ground, looking towards the right. In the background, there are several small buildings with orange roofs, some trees, and a blue sky with a few birds. The overall style is impressionistic with visible brushstrokes.

J01 Introductory Java 1

Imperative programming languages

Java standard library

Types

Hello world

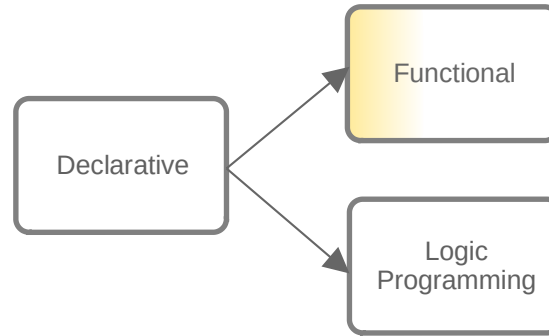


Why Java?

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

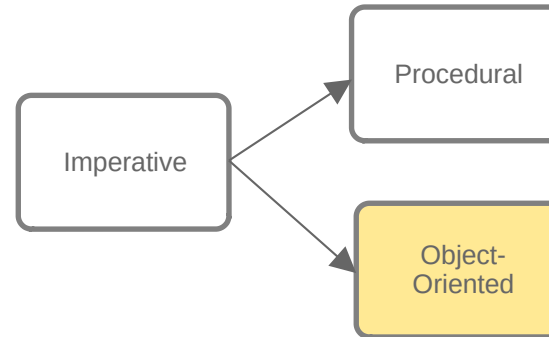
Programming Paradigms

Declarative programming describes the desired result without explicitly listing steps required to achieve that goal.



Pure functional programming only transforms state using functions without side effects.

Imperative programming describes computation in terms of a series of commands that transform state.



Procedural programming uses procedures to transform data structures

Object-oriented programming tightly groups the procedures and data structures together into “objects”

Structured Programming

Another paradigm that imposes a **logical structure to code** making it easier to understand and less error prone.

- Structured control flow (e.g., no GOTOs)
- Callable units (functions / methods / procedures)
- Block structure and scoping

“**Structured program theorem**” gives three building blocks:

- Sequence
- Selection
- Iteration

Type Systems

00110001 ?

$2^5 + 2^4 + 2^0 = 49$?

R of RGB value ?

ASCII Char '1' ?

X86 Opcode XOR ?

The type of a unit of data determines the possible values that data may take on, and the ways it may be operated on.

Ensuring the constraints on types are obeyed is *type checking*:

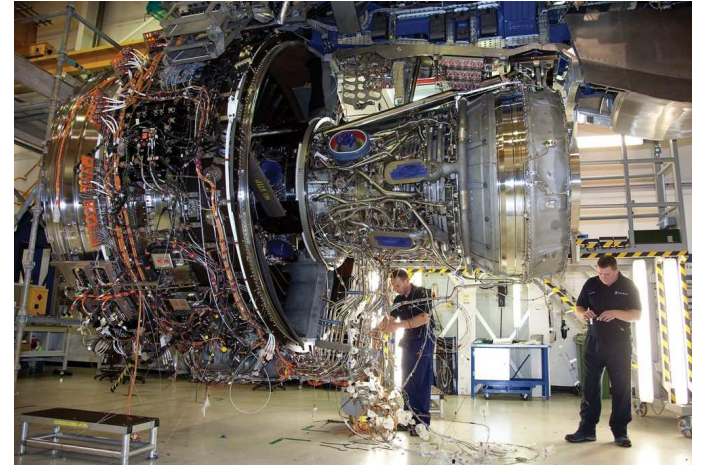
- **Static** type checking: done at **compile time**
 - Java / Haskell / C
- **Dynamic** type checking: done at **runtime**
 - Python / LISP / Javascript

Syntax and Semantics

- **Syntax**: the ways characters can be structured to create a valid program in the given language
 - $3 + 5$: a valid expression involving a number, a binary operator, and then another number
 - *Fair warning*: You will see some syntax that has not yet been explained, especially in the first weeks.
- **Semantics** (meaning / behaviour): what that syntax represents / how the program will behave
 - $3 + 5$: evaluates to a new integer (8) that is the sum of the two integer operands (informally)

Abstraction

- Controlling **complexity**.
- Forming **modules** / components that hide unimportant details and provide an intuitive **interface** to other components.
- Enabling more of the system to fit in our limited fleshy brains at once, without losing the key interactions.
- **Generalising** capability.
- Critical in all languages / paradigms.



Rolls Royce Trent XWB for the A350. Photo: AINonline

The Java Standard Library

- The Java language is augmented with a large standard library (like libstdc++ for C++, .NET for C#, and many others)
 - I/O (accessing files, network, etc)
 - Standard data structures
 - Graphics
 - And much more

<https://docs.oracle.com/en/java/javase/17/docs/api/index.html>

- Learning to use the standard library is part of learning a major language, such as Java.
- Rich and well-designed standard libraries are a key software engineering tool.

The Oracle Java Tutorials

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

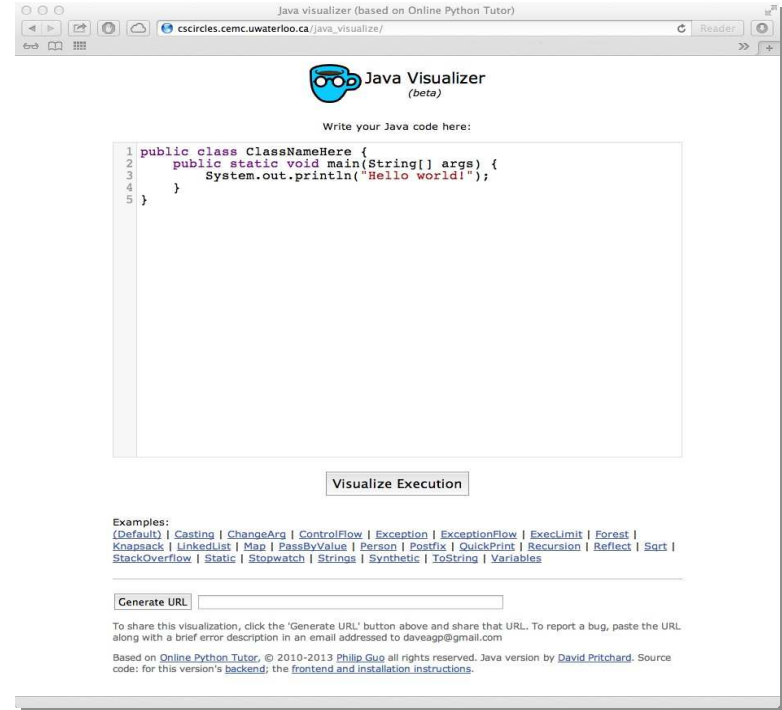
<https://docs.oracle.com/javase/tutorial/>

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

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

The Waterloo Java Visualiser

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



https://cscircles.cemc.uwaterloo.ca/java_visualize/



J02 Introductory Java 2

Objects
Classes
Inheritance

Objects

Java is an *object-oriented* language.

Objects combine state and behaviour

- **State:** fields (data)
 - data can be constant (unchanging through the existence of the object), or changing/variable
- **Behaviour:** methods (code)

Classes

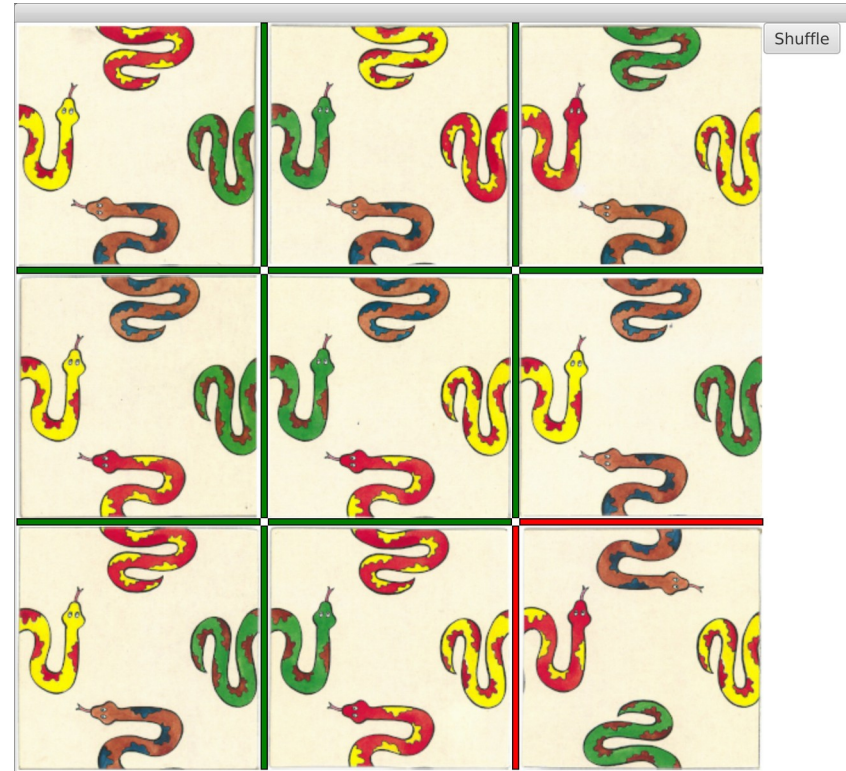
A class is a blueprint or 'type' for an object


- **Class:** definition used for multiple instances (objects)
 - The class defines what fields and methods each instance of it has; methods are shared by all instances.
 - A class can also have fields, that are shared by all instances.
- **Instance (object):** one instantiation of a class
 - Each instance has its own concrete values for (non-class) fields.

Methods

- A subroutine/function/procedure
 - Reusable code to perform a specific task
 - Abstraction: modularity, encapsulation
- In Java, almost all code is a method (`main`, if not another).
- Methods may take arguments (parameters).
- Methods may return a value.

The puzzle consists of nine **tiles**, arranged in a 3x3 **grid**. Each tile has one **pattern**, a snake head or tail of a given colour, on each side. The goal of the game is to place and rotate the tiles so that the patterns match at every edge.



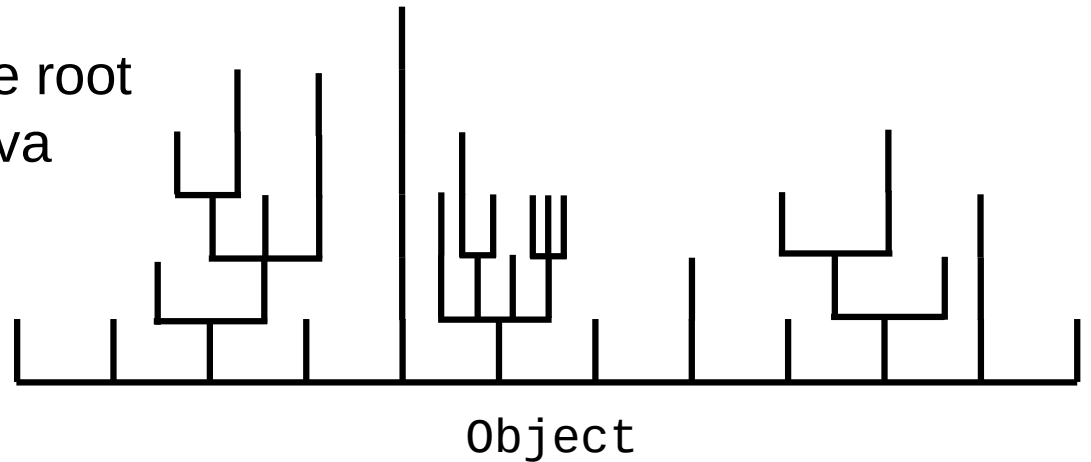
- class Tile:
 - Fields: one Pattern for each side, current rotation
 - Methods: rotate
 - Instances:  ...
- class Grid:
 - Fields: tile in each grid position
 - Methods: move (swap) tiles, check if edges match

Inheritance

Classes form a hierarchy

- a sub-class **extends** a super-class
- a child-class **extends** a parent-class

Class `java.lang.Object` is the root (ultimate ancestor) class of all Java Classes



A painting of a landscape. In the foreground, there is a field of yellow flowers, possibly rapeseed, with some blue flowers in the lower left. In the middle ground, there are several large, leafy trees. In the background, a village with several buildings, including a church with a tower, is visible on a hillside. The sky is a pale, overcast blue.

J03 Introductory Java 3

Naming
Types
Literals

Java Naming

- Java names:
 - Alphanumeric characters and underscores, must not start with a digit.
 - Keywords and reserved words cannot be used.
 - Case-sensitive.
- Capitalisation conventions
 - Class names start with capital letters (`Bi`ke)
 - Variable/field and method names start with lower case, and use upper case for subsequent words (`current`Gear)
 - Constant names use all caps and underscores (`MAX_GEAR_RATIO`)

Java Packages

Every name used in a program must be unambiguous.

A namespace is a part of the code within which names are unique

In Java, a **package** (among other things) defines a namespace.

For example,

```
comp1110.snake.Tile
```

```
comp1110.ass1.Tile
```

Mary Who?



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**
 - Variables local to a method, given values by method's caller

Java's Primitive Data Types

In addition to objects, Java has 8 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 or false | true, false | false |
| char | 16-bit Unicode character | 0 - 65535 | 0 |

Java Literals

- Integer literals (e.g., 1) default to type `int`.
 - An integer value is a `long` if it ends with 'l' or 'L'
 - The prefix 0x indicates hexadecimal, 0b is binary, 0 octal:
 - `0x30` // 48 expressed in hex
 - `0b110000` // 48 expressed in binary
 - `060` // 48 expressed in octal
 - Underscores can be used to break up numbers:
 - `long creditCardNumber = 1234_5678_9012_3456L;`
- A decimal value ending in 'f' is a `float`, while 'd' indicates `double` (default).
- Text in "double quotes" is a `String`.
- A single character in single quotes (e.g., 'A') is a `char`.

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, all zeros
```

```
values = new int[]{1, 2, 3, 4}; // with specific values
```

- Accessing:

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

- Copying:

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




J04 Introductory Java 4

Expressions
Operators
Statements
Methods

Expressions

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

Java Operators

- Assignment (=)
- Arithmetic (+ - * / % += -= *= /= %=)
- Unary (+ - ++ -- !)
- Equality (== !=), relational (> >= < <=), logical (&& ||) and `instanceof`
- Bitwise (~ & ^ | << >> >>>)
- Object creation (`new`)

Statements

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

Block

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

Methods

- A function/procedure/subroutine
 - Reusable code to perform a specific task
 - Abstraction: modularity, encapsulation
- In Java, almost all code is in a method (`main`, if not another).
- Methods may take arguments (parameters).
- Methods may return a value.

Method Declaration

A method declaration consists of 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), a block

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

Parameters (method arguments)

Parameters are the mechanism for passing information from one method to another method (or constructor).

When a method is called, it must be given a list of argument expressions that match the number and types of the method's parameters.

```
byte[] bytes = myString.getBytes("UTF-8");
```

(The semantics of parameter passing are not so simple, and we will come back to them in a later lecture.)

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 (more in a later lecture)

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).

Class and Instance methods

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

- Class methods
 - Can be called without reference to an object
 - But may use **class fields** only.
- Instance methods
 - Must be called on an object (`anObject.aMethod(...)`)
 - May use both class and **instance fields**.



J05 Control Flow

Control flow
if-then-else
switch

Control Flow

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

Structured 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.



J06 Control Flow 2

while and do-while
for
break, continue, return

The `while` and `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.

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

The for statement

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

Branching statements within a loop

- The **break** statement terminates a loop construct
 - *Unlabelled* terminates the inner-most loop
 - *Labelled* terminates the loop named by the label
- The **continue** statement skips the current iteration of a loop
 - *Unlabelled* skips the current iteration of the inner-most loop
 - *Labelled* skips the current iteration of the loop named by the label
- The **return** statement exits the current method



J07 Methods

Methods

Stack, Parameters and References

Exceptions

Methods

- A function/procedure/subroutine
 - Reusable code to perform a specific task
 - Abstraction: modularity, encapsulation
- In Java, almost all code is in a method (`main`, if not another).
- Methods may take arguments (parameters).
- Methods may return a value.

Method Declaration

Method declarations will have the following, in order:

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

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


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).

Parameters and arguments

When a method is called, it must be given a list of argument expressions that match the number and types of the method's parameters.

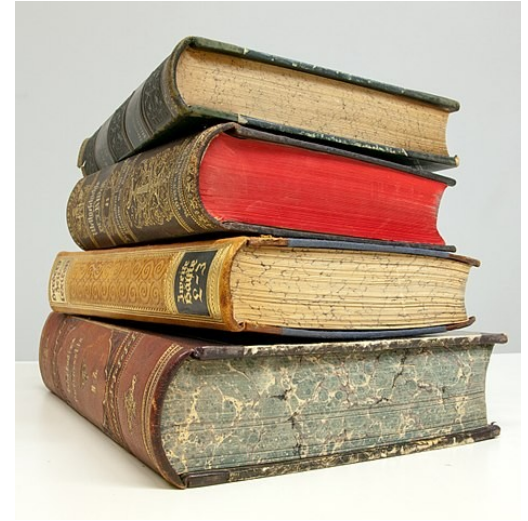
```
byte[] bytes = myString.getBytes("UTF-8");
```

Argument expressions are evaluated *before* the method is called (left-to-right) and their *values* passed as arguments.

The Call Stack: Method after Method after...

- Call Stack: a data structure that tracks method calls
 - Not directly interacted with in high-level languages like Java
 - Each call to a method pushes a *stack frame* to the stack with*:
 - **Return address**: where to continue in the **calling** method after **called** method finishes
 - The **parameters** to pass the **called** method
 - Space to store **local variables** for **called** method

* Not specific to Java, the details depend on the language, compiler, instruction set, operating system etc...



By Ch. Maderthoner - Own work, CC0,<https://commons.wikimedia.org/w/index.php?curid=37987842>

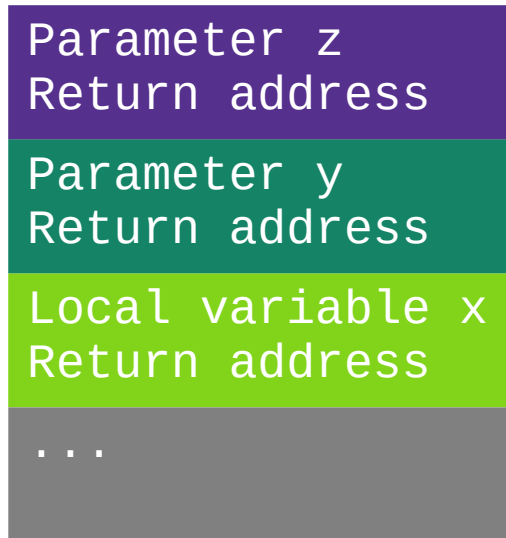


The Call Stack

```
static int twice(int z) {  
    return 2 * z;  
}
```

```
static int process(int y) {  
    y = twice(y);  
    y = y + 1;  
    return y;  
}
```

```
static int number() {  
    int x = 11;  
    return process(x);  
}
```



Parameters

- **Primitive types** passed by value (copied into stack frame)
 - Changes to parameter are **not seen** by caller
- **References** passed by value (copied into stack frame)
 - Changes to the *reference* are **not seen** by caller
 - Changes* to *object referred to* **are seen** by caller

* Some types (e.g., String) are designed to be *immutable* – no public methods modify any class or instance fields.

Parameter Passing

```
public static void main(...) {  
    int xCaller = 5;  
    String nameCaller = "Barbara"  
    int[] arrayCaller = new int[] {1, 2, 3};  
    method(xCaller, nameCaller, arrayCaller);  
}
```

```
static void method(int x,  
                  String name,  
                  int[] array) {  
    x = 100;  
    name = "Greg";  
    array[2] = 1000;  
    array = new int[]{10, 11};  
}
```

Stack

x := 5 := 100

name := #ref1 := #ref3

array := #ref2 := #ref4

xCaller := 5

nameCaller := #ref1

arrayCaller := #ref2

Heap

#ref1 "Barbara"

#ref2 {1, 2, 3} ({1, 2, 1000})

#ref3 "Greg"

#ref4 {10, 11}

Exceptions Basics

- A method can either execute normally and **return** a value (passing execution back to caller), or **throw** an exception to signal something went wrong.
- When an exception is thrown, exception control flow kicks in: *unwinds* the call stack until either a method further down the stack “handles” the exception, or the process exits.
- We will revisit the types of exception and how to catch them later on. For now you will just likely want exceptions to crash your program so it is obvious something went wrong.

Class and Instance methods

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

- Class methods
 - The method called is determined **statically** from the class of the referring variable/expression.
- Instance methods
 - The method called is determined **dynamically** from the class of the value (object) that the method is called on.
- Same with static fields.

The background of the slide is a vibrant, impressionistic painting of a rural landscape. It features a village with several houses, some with red roofs, nestled among lush green trees. In the foreground, there are large, sunlit fields of yellow and green, possibly a field of flowers or a field being prepared for planting. A small stream or path winds through the fields. The overall style is reminiscent of the Impressionist movement, with visible brushstrokes and a rich, warm color palette.

J08 Nested Classes

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
- **Inner classes** (non-static) has direct access to the instance fields and members of its enclosing class.

Anonymous Inner Classes

An instance of an inner class implementing a specified interface can be created without defining the inner class:

```
comp = new Comparable<String>() {  
    Public int compareTo(String other) { ... };  
};
```



J09 Lambda Expressions

Lambda expressions

Functional Interfaces

A *functional interface* is 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();  
}
```


Lambda Expressions

Lambda expressions in Java are a shorthand for creating anonymous inner classes that implement functional interfaces.

- 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; }
```



J10 Number and Autoboxing

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 are subclasses of `Object`.
- 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` (primitives *wrapped* in an object). Java offers automatic support (syntactic sugar) for boxing and unboxing (wrapping / unwrapping).

- Boxing an `int` literal: `Integer i = 5;`
- Unboxing to an `int` variable: `int j = i;`

The Math class

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

- Constants: `Math.PI`, `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()`



J11 Character and String

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

- Strings are *immutable*: no operations modify original `String`
- 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.

A painting of a rural landscape. In the foreground, there is a field of tall, green and yellow grasses. In the middle ground, there are several trees, including a tall, dark green cypress tree and a large, rounded green tree. In the background, there is a small, white house with a yellow roof and a chimney. The sky is blue with white clouds. The overall style is impressionistic with visible brushstrokes.

J12 Generics

Generics

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 of types 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 Parameters

- By default, the only thing that is assumed about a type parameter `T` is that it is an object: i.e. it extends `Object`.
 - No primitives can be used as a generic type (big part of the reason for boxing primitives)
 - When working with a variable that has a generic type, all we can do is pass it around and call methods that are defined for `Object`.
- *Bounds* can be put on type parameters to make them “less generic”.
 - E.g., `public <T extends Number & MyInterface> void method(T t) {...}`
 - This **restricts** the types that can be used with the generic.
 - This **increases** the assumptions that can be made about a variable of this generic type.
- Limits on generic method overloading (type erasure).

A painting of a rural landscape. In the foreground, a person is working in a field of tall green crops. A large wooden wheel is visible in the middle ground. In the background, there are several buildings, including a barn and a windmill, and a range of mountains under a blue sky.

J13 Type Inference

Generic type inference
Lambda expressions
Local variables

Type Inference

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

Instantiating generic classes:

```
LinkedList<String> list = 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 - too vague
```

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 as a local variable:

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

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

```
IntFunction<Integer> lambda = x -> x + 1; // OK
```

Passing a lambda expression directly to a method normally works, as the method parameter provides the type information.

The background of the slide is a reproduction of a painting in the style of J.M.W. Turner or Vincent van Gogh. It depicts a vast landscape with a bright yellow field in the foreground and middle ground, and a deep blue sky filled with numerous dark birds in flight. The brushwork is visible and expressive, with thick, textured strokes. The overall mood is dynamic and somewhat somber due to the dark birds against the bright sky.

J14 Collections

The collections framework
Common collection types
Iterator and Stream interfaces
Ordering collections

The Collections Framework

(in `java.util`)

- 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
 - Iterators (abbreviated for loop syntax), `forEach()`.
- 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** (ordered sequence with restrictions)
 - **Map** (<key, value> pairs)
- Each collection type is defined as an interface
 - You need to choose a concrete class to instantiate
 - 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** or **list** using a linked list
 - First-in-first-out (FIFO) queue ordering

Iterable<T> interface

Collections implement the `Iterable<T>` interface, which enables use of the “For-Each loop”:

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

and also a `forEach` method to apply lambda expression:

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

Stream<T> Interface

Collections can be accessed as a stream via the `stream()` method, enabling a more **functional programming** style:

```
List<Integer> list = List.of(1, 2, 3, 4, 5); // immutable list!
var count = list.stream()
    .filter(x -> x > 2)
    .count();
var nList = list.stream()
    .filter(x -> x > 2)
    .map(x -> Integer.toString(x + 2))
    .toList(); // immutable, otherwise collect(...)
```

Ordering Collections

The Comparable interface defines a 'natural' ordering for all instances of a given type, T:

```
public interface Comparable<T> {  
    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 *functional* interface allows a type T to be ordered in ad-hoc ways:

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

java.util.Collections

Some useful static methods for collections:

- `sort`, `min`, `max`, `reverse`, `frequency`, `addAll`

`List` also has a `sort` instance method:

- When provided with `null` it uses the natural order of elements (given by `Comparable`)
- Can use bespoke ordering when provided a lambda expression (`Comparator` functional interface):

```
(T a, T b) -> { 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[] array = new Long[1];
```

```
objectArray[0] = "I don't fit in"; // Throws ArrayStoreException
```

```
// Won't compile!
```

```
List<Object> list = new ArrayList<Long>(); // Incompatible types
```

```
list.add("I don't fit in");
```

A painting of a Dutch landscape, likely a windmill scene. The scene is dominated by a large, green, grassy hill in the foreground. In the background, several windmills are visible, some with their sails partially open. The sky is a mix of light and dark tones, suggesting a cloudy day. The overall style is characteristic of 17th-century Dutch landscape painting.

J15 Exceptions

Java Exceptions
Catch or Specify
Java syntax

Exceptions

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

Some similarity to event handling (lecture topic **X02**)

- 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 (call stack is *unwound*) until a matching handler is found.

Kinds of Java Exception and Compile-time Check

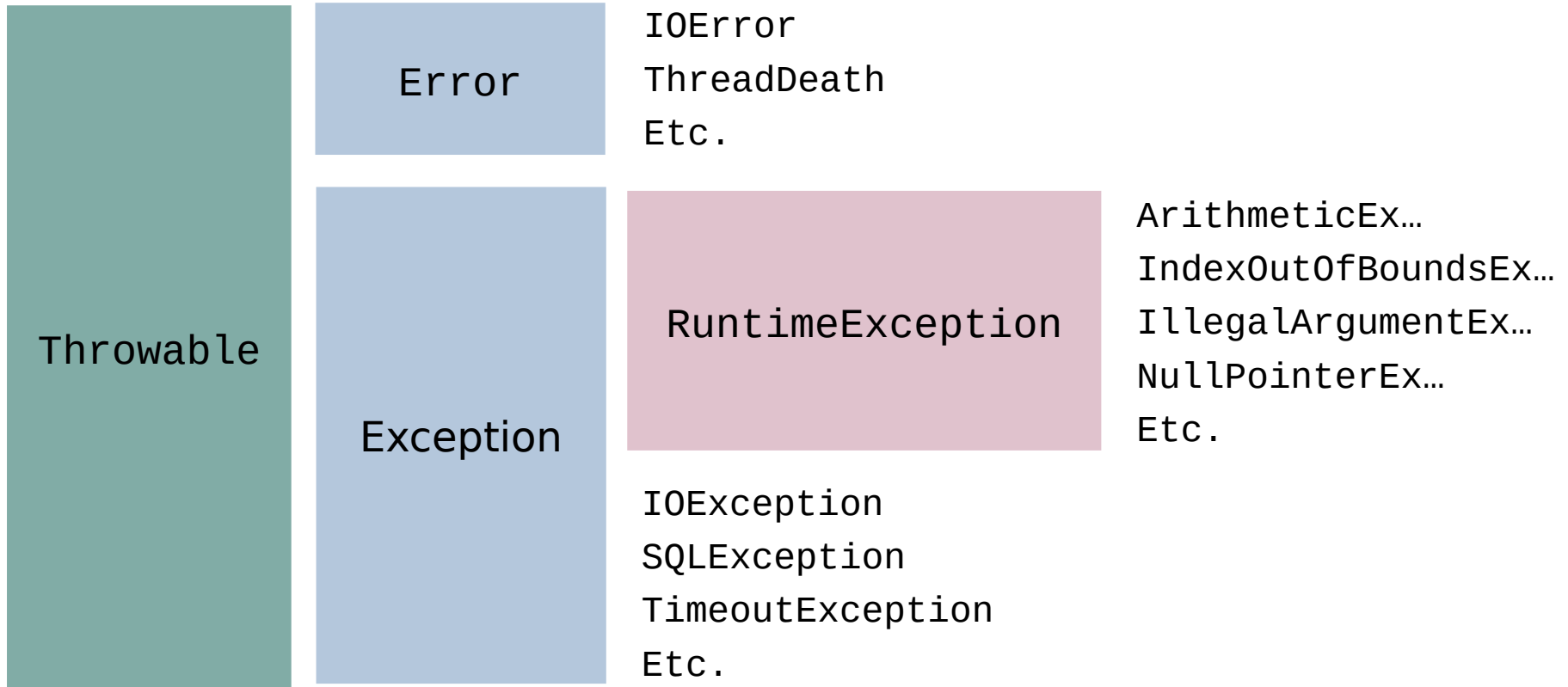
- **error** (Error and its subclasses),
 - serious problems that a reasonable application probably shouldn't attempt to catch
- **runtime exception** (RuntimeException and its subclasses),
 - exceptional situation that often cannot be anticipated or recovered from (e.g., program bugs, logic error, API misuse): probably should fix the bug rather than catch
- **checked exception** (everything else)
 - can be thrown during normal operation and can be reasonably anticipated and handled

unchecked
exceptions

Code that may throw a checked exception must comply with the **catch or specify** requirement, i.e. must be enclosed by either:

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

Java Exception Type Class Hierarchy



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  
}
```

An impressionist landscape painting with a vibrant blue sky, green and yellow fields, and a foreground of white flowers. The brushstrokes are thick and visible, creating a textured, layered effect.

J16 Java Threads

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:
 - Subclass Thread, overriding its run() method.
 - Correspondence between instances of the class and threads.
 - Disadvantages: can't subclass anything else.
 - Use the Runnable interface and implement its run() method.
 - Use Thread.currentThread() to access the thread that is executing the run() method.

start(), join() and sleep()

- Calling `t.start()` will start execution of the `run()` method within the thread `t` (then continue 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