Unit 0. Overview

Learning Outcomes

This unit provides an overview of the aims of CS2030/S and how the key concepts covered in CS2030/S are related to each other.

What is This Module About?

CS2030/S is designed for students who have gone through a typical basic programming module and have learned about problem-solving with simple programming constructs such as loops, conditions, and functions. In a typical introductory programming module, such as CS1010 and its variants at NUS, students tend to write small programs (in the order of tens or hundreds of lines of code) to solve a programming homework problem, work alone on their code, and move on to solve the next problem once the homework is done.

The first aim of CS2030/S is to change the students' mindset and to make them learn to write software that will continue to evolve as software requirements change and to write software that will be read and modified by other programmers (including their future selves).

The second aim of CS2030/S is to level up the complexity of programs that the students write, from the order of hundreds of lines to thousands of lines. CS2030/S bridges the students between writing toy programs to solve specific problems in CS1010 and writing larger real-world software in their later modules, such as CS2103 Software Engineering.

A programming language is the medium in which programmers can express their intention and construct software, and thus is critical to supporting the aims above. With the appropriate features and tools, one can tame the complexity of software, make the code written friendlier to other programmers, and easier to evolve. The third aim of CS2030/S is thus to expand the students' minds on different ways one can construct software and the principles behind some of the programming language constructs. In particular, CS2030/S focuses on *objects*, *types*, and *functions*, as three key constructs for building programmer-friendly software. It covers both object-oriented and functional paradigms as two different approaches to constructing software, with a strong emphasis on type safety.

The final aim of CS2030/S is to introduce students to programming language concepts and to bridge them from introductory programming to advanced modules such as programming language design and implementation. Part of CS2030/S introduces students to the design decisions behind some of the constraints and the workings behind the programming language compilation and execution, giving them a glimpse inside the programming system that so far has been mostly treated as a black box in introductory modules.

The Choice of Java

We decided to use one programming language throughout the module. This decision means that we need to pick a language that is strongly typed with static typing and supports both object-oriented and functional programming. Considering multiple factors, we decided to choose Java for CS2030/S, for its popularity, syntax familiarity, and smoother transitions to later modules in the NUS computing curriculum.

While Java is not the most elegant programming language when expressing programs in a functional style, we hope that students can still learn the principles of functional programming and apply them in other programming languages. This choice is a trade-off between having to switch to a different language in the middle of a module.

What This Module is not About

This is not a module on Java programming. We will not comprehensively cover Java syntax and features, except those relevant to the concepts we teach. In fact, we will avoid and even ban students from using certain Java features (such as `var`) for pedagogical purposes.

This is not a module on software engineering either. Software engineering is a broad discipline on its own and deserves another module. Rather, this module is about the programming principles and constructs on top of which programmers can design better software. To motivate the importance of these principles and constructs and see how they can be used, we will inevitably cover some of the software engineering design principles, such as Liskov Substitution Principle (the L in SOLID), Tell-Don't-Ask, Composition over Inheritance, etc. But we will not comprehensively cover object-oriented design or software design in general (e.g., we will not cover S, O, I, and D in SOLID).

Finally, CS2030/S is not a module that focuses on computational efficiency. We have CS2040/S for that. In CS2030/S, although reducing computational cost still plays a role, this is not the only cost that matters. CS2030/S is also concerned with the human cost of

debugging or maintaining software. In striving for simpler software that is easier to maintain and extend, we may have to sacrifice computational efficiency.

Taming Complexity in Software Development

An underlying theme of CS2030/S is taming complexity in software development. There are objective metrics with which one can measure the complexity of software, but here, we will loosely define complexity as anything that increases the likelihood of bugs in a program.

Let's start by considering a simplified view of what a software program is. One can view a software program as a collection of data variables and instructions on how to modify these variables. A program is generally written to meet a given requirement: given one or more input variables, the program should perform the computation to produce the output variables, in a way that meets the requirement. Often, the program stores information in the intermediate variables while performing the computation.

As a student who has gone through an introductory programming module such as CS1010 and its variants, you should be familiar with the view above, and you should have some experience writing a program to solve a given computational problem. The program you have written for these introductory modules are small "toy" programs mostly -- they consist of only a few hundred lines and tens of variables, at most.

Software development in the real world, however, is far more complex than what you have experienced. A software program rarely solves a well-defined computational problem only. It often requires multiple components, such as user interface, data storage, and business rules, intricately interacting with each other to attain a set of functionalities.

As the requirement of the software becomes more complex, the number of variables that need to be kept track of increases; the logic of the computation the programmer needs to maintain the variables becomes more complicated. Further, it is often that the variables are interdependent. For instance, updating a variable might require updating another; how a variable should be updated might depend on another variable. As the number of variables increases, so is the number of relationships between the variables that the programmer has to keep track of. Failure to correctly maintain the variables and the relationship between them most likely will lead to bugs.

Further, real-world software rarely remains static. This property is again different from what you have experienced in your introductory programming module, where once the instructors release a programming assignment, they rarely go back and change the requirement. In the real world, software evolves -- new features are added, business rules change, and better algorithms are deployed. The code needs to be updated accordingly --

adding new variables and new computation; changing how variables are updated or are dependent on each other. Updating the code of an already-complex software program to keep up with the requirement, if not managed properly, can lead to bugs.

Real-world software is often the product of teamwork from multiple programmers, where the software development process is unlike what you have experienced in your introductory programming module, where you solve your homework individually. When multiple programmers work together, the interdependency between the states needs to be communicated and handled properly and consistently across the programmers. One programmer's modification to the code should not introduce bugs into another programmer's code.

Since software evolves, the notion of "multiple programmers" actually applies even to software developed by a single lone programmer across time. Changing one's code should not introduce new bugs to other parts of the code that is written some time ago.

Strategies to Tame Complexity

Good Software Development Practices

If you are taught properly in your introductory programming modules, you should already be familiar with good programming practices that help to tame the complexity and reduce the chances of bugs. These practices include

- Comment your code: Commenting your code provides *in situ* communication between you and other programmers on the team, as well as between you and your future self, on the non-obvious purpose of the states and the relationships between the states. Such comments help to enhance the understanding of what the code is doing and to remind whoever is updating to code to modify appropriately when the requirement changes.
- Use a coding convention: Adhering to a coding convention helps improve code readability, reducing the cognitive barrier when one programmer reads another programmer's code and allowing the reader to understand the code more easily and thoroughly.

CS2030/S will continue to enforce these good programming practices.

Functions

You should also be taught to always break your code down into functions, each one performing a simple, specific, task. The functions can then be composed to solve larger

and more complex tasks. Functions are an important programming structure in taming code complexity, it allows programmers to (i) compartmentalize computation and its effects, reducing the number of interactions to a few well-defined ones (through arguments and return values); (ii) hide the implementation details so that they can be changed later without affecting other parts of the code; (iii) reuse computations and thus write code that is more succinct and easier to understand/change.

In CS2030/S, you will not only continue to break your computation into functions, but we will kick it up several notches. A major part of CS2030/S is to introduce you to more programming paradigms and language tools that allow you to compartmentalize computations, hide details, and reduce repetition.

The Abstraction Principle

The last point above about why it is important to code in small, reusable functions, follows what is called the *Abstraction Principle*[1]. The principle states that:

> "Each significant piece of functionality in a program should be implemented in just one place in the source code. Where similar functions are carried out by distinct pieces of code, it is generally beneficial to combine them into one by abstracting out the varying parts."

This principle is something that we will visit over and over again in CS2030/S, applying it to different varying parts of a program. In the case of functions, the "varying parts" are the values on which we wish to perform the computation on. We will also apply this principle to (i) types, abstracting them out as parameterized types or subtypes, and (ii) to sub-computation, abstracting them out as first-class functions. These concepts: generics, subtypes, and first-class functions, underlies most of the content of CS2030/S.

Erecting an Abstraction Barrier

Another important strategy for taming complexity is the *abstraction barrier*. Let's separate the role of a programmer into two, in the context of writing functions: the *implementer*, who implements the function, and the *client*, who calls the function. The implementer should compartmentalize the internal variables and the implementation of the function, hiding them behind the abstraction barrier. The parameters and the return values are the only communication gateways across the barrier.

The abstraction barrier is something that we will refer to repeatedly in CS2030/S as well. We will see how we maintain this barrier not only in the context of functions, but also variables and computations on these variables together, by encapsulating them as *objects*,

and hiding details from the client through *access modifiers*. These ideas form two of the core principles of *object-oriented programming*: *encapsulation* and *abstraction*.

Code for Change

The abstraction barrier, if erected and maintained properly, reduces code complexity. It, however, also reduces flexibility as the software evolves. If the client wishes to modify the computation protected by the abstraction barrier, it will need the help of the implementer. In CS2030/S, we will see two ways we can modify the computation behind the abstraction barrier, *without changing the code behind the barrier*.

First, we will introduce the concept of *inheritance* and *polymorphism*, the other two core principles of object-oriented programming. These object-oriented mechanisms allow programmers to easily extend or modify the behavior of existing code.

Second, we will introduce *closure*, an abstraction to computation and its environment, that we can pass into the functions behind the abstraction barrier to perform a computation. The second idea, if carried to the extreme in terms of flexibility, leads to the concept of *monad* in the functional programming paradigm. A monad is a computational structure that allows objects to be composed and manipulated in a succinct and powerful way.

Types

Allowing a programmer to change the behavior of the existing code without changing the code could lead to more bugs, if not managed properly. To prevent this, both the programming language system and the programmers, have to adhere to certain rules when extending or modifying the behavior of the existing code. Java and many other typed languages have *type systems* -- a set of rules that governs how variables, expressions, and functions interact with each other. You will learn about subtyping and the Liskov Substitution Principle, two notions that are important to constraining how inheritance and polymorphism should be used to avoid bugs.

A type system is also an important tool to reduce the complexity of software development. Constraining the interactions among the variables, expressions, and functions, it reduces the possible interdependence between these programming constructs. Furthermore, any attempt by programmers to break the constraint can be caught automatically by the compiler. By utilizing the type system properly, we can detect potential bugs before they manifest themselves.

A reason CS2030/S chooses to use Java is due to its type system. CS2030/S will introduce the concept of types, subtypes, compile-time vs. run-time types, variants of types, parameterized types, and type inferences, in the context of Java. We will see how we can

define our own types (using *classes* and *interfaces*) and define relationships between them. We will see how we can define parameterized types and generic functions that take in types as parameters. These concepts apply to many other programming languages.

Eliminating Side Effects

We have discussed how functions can compartmentalize computations and limit their complexity within their body. For this approach to be effective, the function must not have any side effects -- such as updating a variable that is not within the function. Such functions, called *pure _functions*, are one of the key principles of the functional programming paradigm and is something that we will explore to kick off the section on functional paradigm in CS2030/S.

A related idea in object-oriented programming we will cover in CS2030/S is *immutability* -- once we create an object, the object cannot be changed. In order to update an object, we need to create a new one. With immutability and pure functions, we can guarantee that the same function invoked on the same objects will always return the same value. This certainty can help in understanding and reasoning about the code behavior.

---

1. This principle is formulated by Benjamin C. Pierce in his book "Types and Programming Languages." ↵

Unit 1: Program and Compiler

After reading this unit, students should:

- recap some fundamental programming concepts, including the concept of a program, a programming language, a compiler, an interpreter
- be aware of two modes of running a Java program (compiled vs. interpreted)
- be aware that compile-time errors are better than run-time errors, but the compiler cannot always detect errors during compile time

Software Program

A software program is a collection of data variables and instructions on how to modify these variables. To dictate these instructions to the computer, programmers usually write down the instructions using a programming language, expressing their instructions in code that are made up of keywords, symbols, and names.

A programming language is a formal language that helps programmers specify *precisely* what the instructions are at a higher level of *abstraction* (i.e., at a higher conceptual level) so that a programmer only needs to write a few lines of code to give complex instructions to the computer.

Compiled vs. Interpreted Programs

The processing unit of a computer can only accept and understand instructions written in machine code. A program, written in a higher-level programming language, therefore needs to be translated into machine code before execution. There are different approaches to how such translations can be done. The first approach uses a *compiler* -- a software tool that reads in the entire program written in a higher-level programming language and translates it into machine code. The machine code is then saved into an executable file, which can be executed later. `clang`, a C/C++ compiler, is an example. The second approach uses an *interpreter* -- software that reads in the program one statement at a time interprets what the statement means, and executes its directly. This is how Python and Javascript programs are executed.

Modern programming systems for executing programs are, however, more sophisticated. V8, for instance, is an open-source engine that executes Javascript, and it contains both an

interpreter that first interprets a Javascript into *bytecode* (an intermediate, low-level representation). A just-in-time compiler then reads in the bytecode and generates machine code dynamically at runtime with optimized performance.

Java programs, on the other hand, can be executed in two ways.

- The Java program can first be compiled into bytecode. During execution, the bytecode is interpreted and compiled on-the-fly by the *Java Virtual Machine* (JVM) into machine code.
- The Java program can also be interpreted by the Java interpreter.

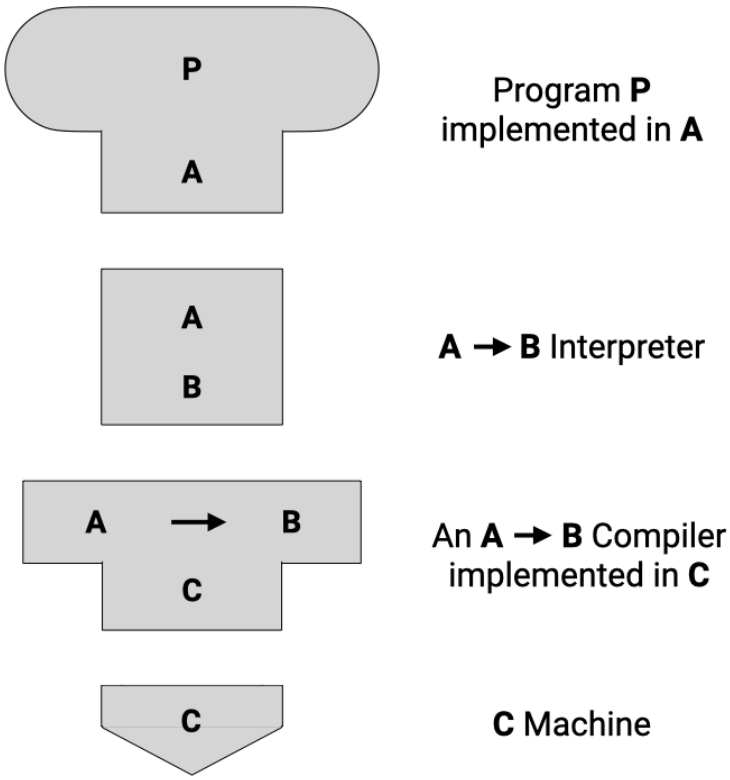
To better understand this distinction, we will introduce a visual aid to describe the relationships between programs, compilers, interpreters, and machines.

Tombstone Diagrams (T-Diagrams)

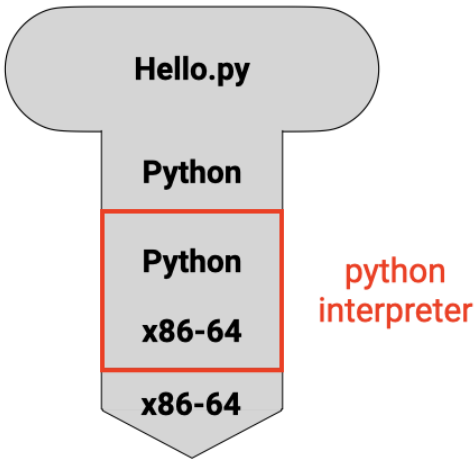
Tombstone Diagrams or T-diagrams consist of combinations of the following four components:

- *Programs* which are implemented in a particular language (i.e. `Java`, `python`, `c/c++`)
- Language **A** to language **B** *Interpreters*
- Language **A** to language **B** *Compilers* which are implemented in a language **C**
- Physical *Machines* implementing a particular language (i.e. x86-64, ARM-64)

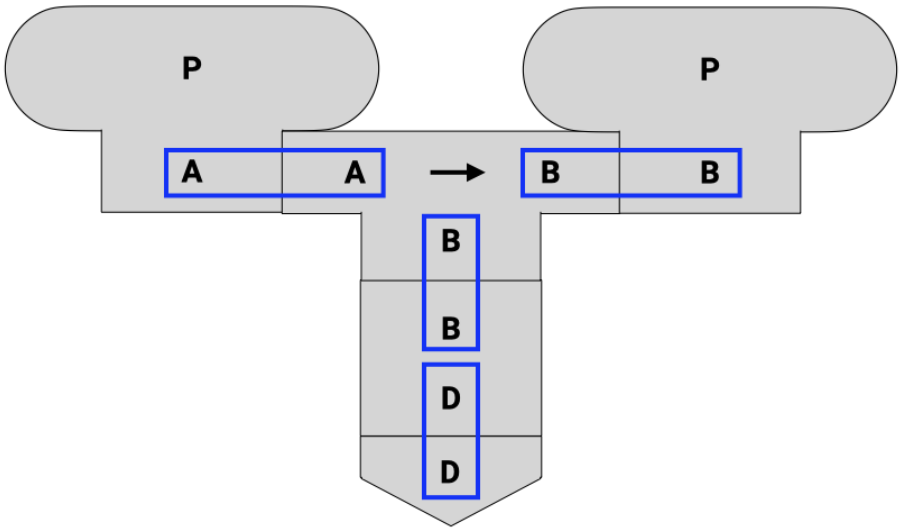
These components are represented in T-diagrams as shown in the figure below:



We can treat these components like "puzzle pieces" and build diagrams to describe various execution, compilation, or interpreted processes. For example, in the diagram below, a python script `Hello.py` is being interpreted by the python interpreter running on the x86-64 architecture.



Note: In order for the diagram to be valid, adjacent connected diagram components need to match. This can be seen in the diagram below (highlighted with blue boxes).



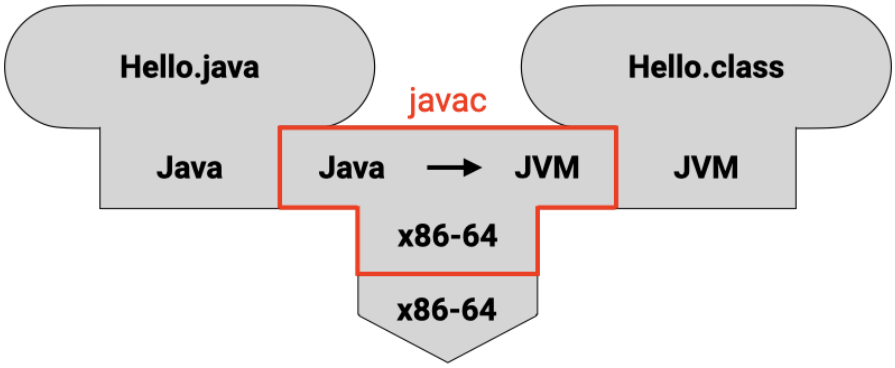
Since CS2030/S uses Java, we will now look at the two approaches to execute a Java program in more detail (without worrying about how to write a single line of Java first).

Compiling and Running Java Programs

Suppose we have a Java program called `Hello.java`. To compile the program, we type[1]

```
1  $ javac Hello.java
```

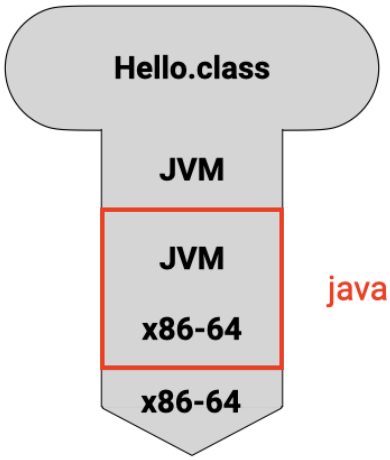
into the command line. `javac` is the Java compiler. This step will either lead to the bytecode called `Hello.class` being created or spew out some errors. This process can be seen in the figure below, where the `Hello.java` program is compiled from Java to the JVM language (bytecode). The Java compiler `javac` in this diagram is implemented in the x86-64 machine language.



Assuming that there is no error in compilation, we can now run

```
$ java Hello
```

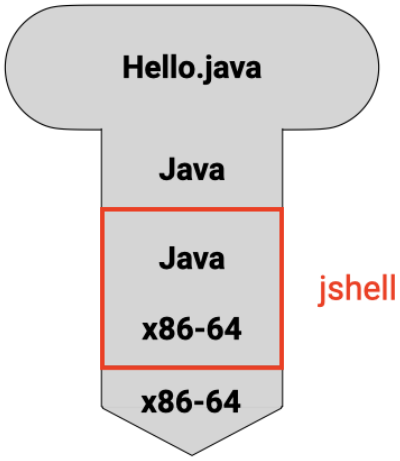
to invoke the JVM `java` and execute the bytecode contains in `Hello.class`. This can be seen in the figure below, where the `Hello.class` program is interpreted from JVM language (bytecode) to the x86-64 machine language.



Beginners tend to confuse between `javac` and `java`, and whether to add the extension `.java` or `.class` when compiling and executing a Java program. Do take note and refer back here if needed[2].

Interpreting a Java program

Java (version 8 or later) comes with an interpreter called `jshell` that can read in Java statements, evaluate them, and print the results[3]. `jshell` is useful for learning and experimenting about Java. This can be seen in the figure below, where the `Hello.java` program is interpreted from Java directly to the x86-64 machine language.



To run `jshell` in interactive mode, we type

```
$ jshell
```

on the command line, which causes an interactive prompt to appear:

```
$ jshell
|  Welcome to JShell -- Version 11.0.2
|  For an introduction type: /help intro

jshell>
```

We can now type in Java code on `jshell>`.

Alternatively, we can also include all the Java statements that we want `jshell` to run in a file and pass it into jshell

```
$ jshell Hello.jsh
```

While `jshell` is a convenient way to test things out and play with new Java concepts as we learn, do keep in mind that `jshell` combined both compilation and run-time into a single step. The error spewed out by `jshell` could be either compile-time error or run-time error, and this could be confusing to beginners who try to distinguish between the two phases of program execution.

Compiler

The compiler does more than just translating source code into machine code or bytecode. The compiler also needs to parse the source code written and check if it follows the precise specification of the programming language (called *grammar*) used, and produces a *syntax error* if the grammar is violated. It therefore can detect any syntax error before the program is run.

It is much better for a programmer to detect any error in its code during compilation -- since this is the phase when the program is still being developed and under the control of the programmer. Runtime error, on the other hand, might occur when the customers are running the program, and so are much less desirable. As such, we try to detect errors as much as possible during compilation. The compiler is a powerful friend for any programmer if used properly.

The power of the compiler, however, is limited. A compiler can only read and analyze the source code without actually running it. Without running the program, the compiler cannot always tell if a particular statement in the source code will ever be executed; it cannot always tell what values a variable will take.

To deal with this, the compiler can either be *conservative*, and report an error as long as *there is a possibility* that a particular statement is incorrect; or, the compiler can be more *permissive*, reporting an error only if *there is no possiblity* that a particular statement is correct. If there is a possibility that a particular statement is correct, it does not throw an error, but rely on the programmer to do the right thing. We will further contrast these two behaviors later in this module.

---

1. The `$` represents the command prompt in a shell and you do not need to type this. ↵
2. To add to the confusion, Java 11 introduces a shortcut where we can run `java Hello.java` directly. This command causes `Hello.java` to be compiled and executed in a single step. We won't do this in CS2030/S (i) to avoid confusion and (ii) to show you the steps explicitly. ↵
3. Such a program is called REPL (Read-Evaluate-Print in a Loop) for short. ↵

Unit 2: Variable and Type

After this unit, students should be able to:

- appreciate the concept of variables as an abstraction
- understand the concept of types and subtypes
- contrast between statically typed language vs. dynamically typed language
- contrast between strongly typed language vs. weakly typed language
- be familiar with Java variables and primitive types
- understand widening type conversion in the context of variable assignments and how subtyping dictates whether the type conversion is allowed.

Data Abstraction: Variable

One of the important abstractions that are provided by a programming language is the *variable*. Data are stored in some location in computer memory. But we should not be referring to the memory location all the time. First, referring to something like `0xFA49130E` is not user-friendly; Second, the location may change. A *variable* is an abstraction that allows us to give a user-friendly name to a piece of data in memory. We use the *variable name* whenever we want to access the *value* in that location, and *pointer to the variable* or *reference to the variable* whenever we want to refer to the address of the location.

Type

As programs get more complex, the number of variables that the programmer needs to keep track of increases. These variables might be an abstraction over different types of data: some variables might refer to a number, some to a string, some to a list of numbers, etc. Not all operations are meaningful over all types of data.

To help mitigate the complexity, we can assign a *type* to a variable. The type communicates to the readers what data type the variable is an abstraction over, and to the compiler/interpreter what operations are valid on this variable and how the operation behaves. In lower-level programming languages like C, the type also informs the compiler how the bit representing the variable should be interpreted.

As an example of how types can affect how an operation behaves, let's consider Python. Suppose we have two variables `x` and `y`, storing the values `4` and `5` respectively and we run `print x + y`. If `x` and `y` are both strings, you would get `45`; if `x` and `y` are integers, you would get `9`; if `4` is an integer and `5` is a string, you would get an error.

In the last instance above, you see that assigning a type to each variable helps to keep the program meaningful, as the operation `+` is not defined over an integer and a string in Python.

Java and Javascript, however, would happily convert `4` into a string for you, and return `45`.

Dynamic vs. Static Type

Python and Javascript are examples of *dynamically typed* programming languages. The same variable can hold values of different types, and checking if the right type is used is done during the execution of the program. Note that, the type is associated with the *values*, and the type of the variable changes depending on the value it holds. For example, we can do the following:

Javascript

```
let i = 4;   // i is an integer
i = "5"; // ok, i is now a string
```

Python

```
i = 4   // i is an integer
i = "5" // ok, i is now a string
```

Java, on the other hand, is a *statically typed* language. We need to *declare* every variable we use in the program and specify its type. A variable can only hold values of the same type as the type of the variable, so we can't assign, for instance, a string to a variable of type `int`. Once a variable is assigned a type, its type cannot be changed.

```
int i;   // declare a variable
i = 4;   // ok
i = "5"; // error, cannot assign a string to an `int`
```

The type that a variable is assigned with when we declare the variable is also known as the *compile-time type*. During the compilation, this is the only type that the compiler is aware of. The compiler will check if the compile-time type matches when it parses the variables, expressions, values, and function calls, and throw an error if there is a type mismatch. This type-checking step helps to catch errors in the code early.

Strong Typing vs. Weak Typing

A *type system* of a programming language is a set of rules that govern how the types can interact with each other.

A programming language can be strongly typed or weakly typed. There are no formal definitions of "strong" vs. "weak" typing of a programming language, and there is a spectrum of "strength" between the typing discipline of a language.

Generally, a *strongly typed* programming language enforces strict rules in its type system, to ensure *type safety*, i.e., to catch type errors during compile time rather than leaving it to runtime.

On the other hand, a *weakly typed* (or *loosely typed*) programming language is more permissive in terms of typing checking. C is an example of a static, weakly typed language. In C, the following is possible:

```
int i;   // declare a variable
i = 4;   // ok
i = (int)"5"; // you want to treat a string as an int? ok, as you wish!
```

The last line forces the C compiler to treat the string (to be more precise, the address of the string) as an integer, through typecasting.

In contrast, if we try the following in Java:

```
int i;   // declare a variable
i = 4;   // ok
i = (int)"5"; // error
```

we will get the following compile-time error message:

```
|  incompatible types: java.lang.String cannot be converted to int
```

because the compiler enforces a stricter rule and allows typecasting only if it makes sense.

Type Checking with A Compiler

In addition to checking for syntax errors, the compiler can check for type compilability according to the compile-time type, to catch possible errors as early as possible. Such type checking is made possible with static typing. Consider the following Python program:

```
i = 0
while (i < 10):
    # do something that takes time
```

```
4       i = i + 1
5   print("i is " + i)
```

The type mismatch error on Line 5 is only caught when Line 5 is executed. Since the type of the variable `i` can change during run time, Python (and generally, dynamically typed languages) can not tell if Line 5 will lead to an error until it is evaluated at run-time.

In contrast, statically typed language like Java can detect type mismatch during compile time since the compile-time type of a variable is fixed.

Primitive Types in Java

We now switch our focus to Java, particularly to the types supported. There are two categories of types in Java, the *primitive types* and the *reference types*. We will first look at primitive types in this unit.

Primitive types are types that holds numeric values (integers, floating-point numbers) as well as boolean values (`true` and `false`).

For storing integral values, Java provides four types, `byte`, `short`, `int`, and `long`, for storing 8-bit, 16-bit, 32-bit, 64-bit signed integers respectively. The type `char` stores 16-bit unsigned integers representing UTF-16 Unicode characters.

For storing floating-point values, Java provides two types, `float` and `double`, for 32-bit and 64-bit floating-point numbers.

Unlike reference types, which we will see later, primitive type variables never share their value with each other, i.e., if we have:

```
1   int i = 1000;
2   int j = i;
3   i = i + 1;
```

`i` and `j` each store a copy of the value `1000` after Line 2. Changing `i` on Line 3 does not change the content of `j`.

Subtypes

An important concept that we will visit repeatedly in CS2030/S is the concept of subtypes.

Let S and T be two types. We say that T is a *subtype* of S if *a piece of code written for variables of type S can also safely be used on variables of type T.*

We use the notation $T <: S$ or $S :> T$ to denote that T is subtype of S.

The subtype relationship is transitive, i.e., if $S <: T$ and $T <: U$, then $S <: U$. It is also reflexive, for any type S, $S <: S$.

We also use the term *supertype* to denote the reversed relationship: if T is a subtype of S, then S is a supertype of T.

Subtyping Between Java Primitive Types

Consider the range of values that the primitive types can take, Java defines the following subtyping relationship:

`byte` <: `short` <: `int` <: `long` <: `float` <: `double`

`char` <: `int`

Valid subtype relationship is part of what the Java compiler checks for when it compiles. Consider the following example:

```
1  double d = 5.0;
2  int i = 5;
3  d = i;
4  i = d; // error
```

Line 4 above would lead to an error:

```
1  |  incompatible types: possible lossy conversion from double to int
```

but Line 3 is OK.

This example shows how subtyping applies to type checking. *Java allows a variable of type T to hold a value from a variable of type S only if $S <: T$*. This step is called *widening type conversion*. Such conversion can happen during assignment or parameter passing.

Additional Readings

- Java Tutorial: Primitive Data Types and other Language Basics

Unit 3: Functions

After reading this unit, students should

- understand the importance of function as a programming constructor and how it helps to reduce complexity and mitigate bugs.
- be aware of two different roles a programmer can play: the implementer and the client
- understand the concept of abstraction barrier as a wall between the client and the implementer, including in the context of a function.

Function as an Abstraction over Computation

Another important abstraction provided by a programming language is the *function* (or *procedure*). This abstraction allows programmers to group a set of instructions and give it a name. The named set of instructions may take one or more variables as input parameters, and return one or more values.

Like all other abstractions, defining functions allow us to think at a higher conceptual level. By composing functions at increasingly higher level of abstractions, we can build programs with increasing level of complexity.

Functions help us deal with complexity in a few ways:

- Functions allow programmers to compartmentalize computation and its effects. We can isolate the complexity to within its body: the intermediate variables exists only as local variables that has no effect outside of the function. A function only interacts with the rest of the code through its parameters and return value, and so, reduces the dependencies between variables to these well-defined interactions. Such compartmentalization reduces the complexity of code.
- Functions allow programmers to hide *how* a task is performed. The caller of the function only needs to worry about *what* the function does. By hiding the details of *how*, we gain two weapons against code complexity. First, we reduce the amount of information that we need to communicate among programmers. A fellow programmer only needs to read the documentation to understand what the parameters are for, what the return values are. There is no need for a fellow programmer to know about the intermediate variables or the internal computation used to implement the functions. Second, as the design and requirement evolve, the implementation of a

function may change. But, as long as the parameters and the return value of a function remains the same, the caller of the function does not have to update the code accordingly. Reducing the need to change as the software evolves reduces the chances of introducing bugs accordingly.

- Functions allows us to reduce repetition in our code through *code reuse*. If we have the same computation that we need to perform repeatedly on different *values*, we can construct these computations as functions, replacing the values with parameters, and pass in the values as arguments to the function. This approach reduces the amount of boiler-plate code and has two major benefits in reducing code complexity and bugs. First, it makes the code more succinct, and therefore easier to read and understand. Second, it reduces the number of places in our code that we need to modify as the software evolves, and therefore, decreases the chance of introducing new bugs.

Abstraction Barrier

We can imagine an *abstraction barrier* between the code that calls a function and the code that defines the function body. Above the barrier, the concern is about using the function to perform a task, while below the barrier, the concern is about *how* to perform the task.

While many of you are used to writing a program solo, in practice, you rarely write a program with contributions from only a single person. The abstraction barrier separates the role of the programmer into two: (i) an *implementer*, who provides the implementation of the function, and (ii) a *client*, which uses the function to perform the task. Part of the aim in CS2030/S is to switch your mindset into thinking in terms of these two roles.

The abstraction barrier thus enforces a *separation of concerns* between the two roles.

The concept of abstraction barrier applies not only to a function, but it can be applied to different levels of abstraction as well. We will see how it is used for a higher-level of abstraction in the next unit.

Unit 4: Encapsulation

After reading this unit, students should

- understand composite data type as a even-higher level abstraction over variables
- understand encapsulation as an object-oriented (OO) principle
- understand the meaning of class, object, fields, methods, in the context of OO programming
- be able to define a class and instantiate one as an object in Java
- appreciate OO as a natural way to model the real world in programs
- understand reference types in Java and its difference from the primitive types

Abstraction: Composite Data Type

Just like functions allow programmers to group instructions, give it a name, and refer to it later, a *composite data type* allows programmers to group *primitive types* together, give it a name (a new type), and refer to it later. This is another powerful abstraction in programming languages that help us to think at a higher conceptual level without worrying about the details. Commonly used examples are mathematical objects such as complex numbers, 2D data points, multi-dimensional vectors, circles, etc, or everyday objects such as a person, a product, etc.

Defining composite data type allows programmers to abstract away (and be separated from the concern of) how a complex data type is represented.

For instance, a circle on a 2D plane can be represented by the center (`x`, `y`) and its radius `r`, or it can be represented by the top left corner (`x`, `y`) and the width `w` of the bounding square.

In C, we build a composite data type with `struct`. For example,

```
typedef struct {
  double x, y; // (x,y) coordinate of the center.
  double r; // radius
} circle;
```

Once we have the `struct` defined, we are not completely shielded from its representation, until we write a set of functions that operates on the `circle` composite

type. For instance,

```
double circle_area(circle c) { ... };
bool   circle_contains_point(circle c, double x, double y) { ... };
  :
```

Implementing these functions requires knowledge of how a circle is represented. Once the set of functions that operates on and manipulates circles is available, we can use the *circle* type without worrying about the internal representation.

If we decide to change the representation of a circle, then only the set of functions that operates on a circle type need to be changed, but not the code that uses circles to do other things. In other words, the representation of the circle and the set of functions that operation on and manipulate circles, fall on the same side of the abstraction barrier.

Abstraction: Class and Object (or, Encapsulation)

We can further bundle the composite data type *and its associated functions* on the same side of the abstraction barrier together, into another abstraction called a *class*.

A class is a data type with a group of functions associated with it. We call the functions as *methods* and the data in the class as *fields* (or *members*, or *states*, or *attributes*[1]). A well-designed class maintains the abstraction barrier, properly wraps the barrier around the internal representation and implementation, and exposes just the right *method interface* for others to use.

The concept of keeping all the data and functions operating on the data related to a composite data type together within an abstraction barrier is called *encapsulation*.

Let's see how we can encapsulate the fields and methods associated together, using `Circle` as an example, in Java.

```
// Circle v0.1
class Circle {
  double x;
  double y;
  double r;

  double getArea() {
    return 3.141592653589793 * r * r;
  }
}
```

The code above defines a new class using the keyword `class`, give it a name `Circle`[2], followed by a block listing the member variables (with types) and the function definitions.

Just like we can create variables of a given type, we can create *objects* of a given class. Objects are *instances* of a class, each allowing the same methods to be called, and each containing the same set of variables of the same types, but (possibly) storing different values.

In Java, the keyword `new` creates an object of a given class. For instance, to create a `Circle` object, we can use

```
1  Circle c = new Circle();
```

To access the fields and the methods, we use the `.` notation. For example, `object.field` or `object.method(..)`. For instance,

```
1  c.r = 10;    // set the radius to 10
2  c.getArea(); // return 314.1592653589793
```

Object-Oriented Programming

A program written in an *object-oriented language* such as Java consists of classes, with one main class as the entry point. One can view a running object-oriented (or OO) program as something that instantiates objects of different classes and orchestrates their interactions with each other by calling each others' methods.

One could argue that an object-oriented way of writing programs is much more natural, as it mirrors our world more closely. If we look around us, we see objects all around us, and each object has certain properties, exhibits certain behavior, and they allow certain actions. We interact with the objects through their interfaces, and we rarely need to know the internals of the objects we used every day (unless we try to repair them)[3].

To model a problem in an object-oriented manner, we typically model the nouns as classes and objects, the properties or relationships among the classes as fields, and the verbs or actions of the corresponding objects as methods.

Reference Types in Java

We mentioned in Unit 2 that there are two kinds of types in Java. You have been introduced to the primitive types. Everything else in Java is a reference type.

The `Circle` class is an example of a reference type. Unlike primitive variables, which never share the value, a reference variable stores only the reference to the value, and therefore two reference variables can share the same value. For instance,

```
1  Circle c1 = new Circle();
2  Circle c2 = c1;
3  System.out.println(c2.r); // print 0
4  c1.r = 10.0;
5  System.out.println(c2.r); // print 10.0
```

The behavior above is due to the variables `c1` and `c2` referencing to the same `Circle` object in the memory. Therefore, changing the field `r` of `c1` causes the field `r` of `c2` to change as well.

Special Reference Value: `null`

Any reference variable that is not initialized will have the special reference value `null`. A common error for beginners is to declare a reference variable and try to use it without instantiating an object:

```
1  Circle c1;
2  c1.r = 10.0;  // error
```

Line 2 would lead to a run-time error message

```
1  |  Exception java.lang.NullPointerException
```

Remember to *always instantiate a reference variable* before using it.

Further Readings

- Oracle's Java Tutorial on Classes and Objects

---

1. Computer scientists just could not decide what to call this :(↵
2. As a convention, we use PascalCase for class name and camelCase for variable and method names in Java. ↵
3. This is a standard analogy in an OOP textbook. In practice, however, we often have to write programs that include abstract concepts with no tangible real-world analogy as classes. ↵

Unit 5: Information Hiding

After taking this unit, students should:

- understand the drawback of breaking the abstraction barrier
- understand the concept of information hiding to enforce the abstraction barrier
- understand how Java uses access modifiers to enforce information hiding
- understand what is a constructor and how to write one in Java

Breaking the Abstraction Barrier

In the ideal case, the code above the abstraction barrier would just call the provided interface to use the composite data type. There, however, may be cases where a programmer may intentionally or accidentally break the abstraction barrier.

Consider the case of `Circle` above, where we modify the radius `r` directly with `c.r = 10`. In doing so, we, as the client to `Circle`, make an explicit assumption of how `Circle` implements a circle. The implementation details have been leaked outside the abstraction barrier. Now, if the implementer wishes to change the representation of the `Circle`, to say, store the diameter, instead. This small implementation change would invalidate the code that the client has written! The client will have to carefully change all the code that makes the assumption, and modify accordingly, increasing the chances of introducing a bug.

Data Hiding

Many OO languages allow programmers to explicitly specify if a field or a method can be accessed from outside the abstraction barrier. Java, for instance, supports `private` and `public` access modifiers. A field or a method that is declared as `private` cannot be accessed from outside the class, and can only be accessed within the class. On the other hand, as you can guess, a `public` field or method can be accessed, modified, or invoked from outside the class.

Such a mechanism to protect the abstraction barrier from being broken is called *data hiding* or *information hiding*. This protection is enforced by the *compiler* at compile time.

In our original `Circle` class (v0.1) in Unit 4, we did not specify any access modifier -- this amounts to using the *default* modifier, the meaning of which is not our concern right now[1] For a start, we will explicitly indicate `private` or `public` for all our methods and fields.

```
1  // Circle v0.2
2  class Circle {
3    private double x;
4    private double y;
5    private double r;
6
7    public double getArea() {
8      return 3.141592653589793 * r * r;
9    }
10 }
```

Now the fields `x`, `y`, and `r` are hidden behind the abstraction barrier of the class `Circle`. Note that these fields are not accessible and modifiable outside of the class `Circle`, but they can be accessed and modified within `Circle` (inside the abstraction barrier), such as in the methods `getArea`.

Breaking Python's Abstraction Barrier

Python tries to prevent *accidental* access to internal representation by having a convention of prefixing the internal variables with `_` (one underscore) or `__` (two underscores). This method, however, does not prevent a lazy programmer from directly accessing the variables and possibly planting a bug/error that will surface later.

Constructors

With data hiding, we completely isolate the internal representation of a class using an abstraction barrier. But, with no way for the client of the class to modify the fields directly, how can the client initialize the fields in a class? To get around this, it is common for a class to provide methods to initialize these internal fields.

A method that initializes an object is called a *constructor*.

A constructor method is a special method within the class. It cannot be called directly but is invoked automatically when an object is instantiated. In Java, a constructor method *has the same name as the class* and *has no return type*. A constructor can take in arguments just like other functions. Let's add a constructor to our `Circle` class:

```
1  // Circle v0.3
2  class Circle {
3    private double x;
```

```
4     private double y;
5     private double r;
6
7     public Circle(double x, double y, double r) {
8       this.x = x;
9       this.y = y;
10      this.r = r;
11    }
12
13    public double getArea() {
14      return 3.141592653589793 * this.r * this.r;
15    }
16  }
```

Now, to create a `Circle` object, we need to pass in three arguments:

```
1  Circle c = new Circle(0.0, 0.5, 10.0);
```

Constructor in Python and JavaScript

In Python, the constructor is the `__init__` method. In JavaScript, the constructor is simply called `constructor`.

The `this` Keyword

The code above also introduces the `this` keyword. `this` is a reference variable that refers back to self, and is used to distinguish between two variables of the same name. In the example above, `this.x = x` means we want to set the field `x` of this object to the parameter `x` passed into the constructor.

Now that you have been introduced to `this`, we have also updated the method body of `getArea` and replaced `r` with `this.r`. Although there is nothing syntactically incorrect about using `r`, sticking to the idiom of referring to members through the `this` reference makes the code easier to understand to readers. We are making it explicit that we are referring to a field in the class, rather than a local variable or a parameter.

---

1. The other access modifier is `protected`. Again, we do not want to worry about this modifier for now. ↵

Unit 6: Tell, Don't Ask

After taking this unit, students should:

- understand what accessor and mutator are used for, and why not to use them
- understand the principle of "Tell, Don't Ask"

Accessors and Mutators

Similar to providing constructors, a class should also provide methods to retrieve or modify the properties of the object. These methods are called the *accessor* (or *getter*) or *mutator* (or *setter*).

The use of accessor and mutator methods is a bit controversial. Suppose that we provide an accessor method and a mutator method for every private field, then we are exposing the internal representation, therefore breaking the encapsulation. For instance:

```
1  // Circle v0.4
2  class Circle {
3    private double x;
4    private double y;
5    private double r;
6
7    public Circle(double x, double y, double r) {
8      this.x = x;
9      this.y = y;
10     this.r = r;
11   }
12
13   public double getX() {
14     return this.x;
15   }
16
17   public void setX(double x) {
18     this.x = x;
19   }
20
21   public double getY() {
22     return this.y;
23   }
24
25   public void setY(double y) {
26     this.y = y;
27   }
28
29   public double getR() {
```

```
30      return this.r;
31    }
32
33    public void setR(double r) {
34      this.r = r;
35    }
36
37    public double getArea() {
38      return 3.141592653589793 * this.r * this.r;
39    }
40  }
```

The "Tell Don't Ask" Principle

The mutators and accessors above are pretty pointless. If we need to know the internal and do something with it, then we are breaking the abstraction barrier. The right approach is to implement a method within the class that does whatever we want the class to do. For instance, suppose that we want to check if a given point (x,y) calls within the circle, one approach would be:

```
1  double cX = c.getX();
2  double cY = c.getY();
3  double r = c.getR();
4  boolean isInCircle = ((x - cX) * (x - cX) + (y - cY) * (y - cY)) <= r * r;
```

where `c` is a `Circle` object.

A better approach would be to add a new `boolean` method in the `Circle` class, and call it instead:

```
1  boolean isInCircle = c.contains(x, y);
```

The better approach involves writing a few more lines of code to implement the method, but it keeps the encapsulation intact. If one fine day, the implementer of `Circle` decided to change the representation of the circle and remove the direct accessors to the fields, then only the implementer needs to change the implementation of `contains`. The client does not have to change anything.

The principle around which we can think about this is the "Tell, Don't Ask" principle. The client should tell a `Circle` object what to do (compute the circumference), instead of asking "what is your radius?" to get the value of a field then perform the computation on the object's behalf.

While there are situations where we can't avoid using accessor or modifier in a class, for beginner OO programmers like yourself, it is better to not define classes with any

accessor and modifier to the private fields, and forces yourselves to think in the OO way -- to tell an object what task to perform as a client, and then implement this task within the class as a method as the implementer.

Further Reading

- Tell Don't Ask by Martin Fowler
- Why getters and setters are evil, by Allen Holub, JavaWorld
- Getters and setters are evil. Period, by Yegor Bygayenko.

Unit 7: Class Fields

After this unit, students should:

- understand the difference between instance fields and class fields
- understand the meaning of keywords `final` and `static` in the context of a field
- be able to define and use a class field
- be able to use `import` to access classes from the Java standard libraries

Class Fields

Let's revisit the following implementation of `Circle`.

```
1  // Circle v0.3
2  class Circle {
3    private double x;
4    private double y;
5    private double r;
6
7    public Circle(double x, double y, double r) {
8      this.x = x;
9      this.y = y;
10     this.r = r;
11   }
12
13   public double getArea() {
14     return 3.141592653589793 * this.r * this.r;
15   }
16 }
```

In the code above, we use the constant π but hardcoded it as 3.141592653589793. Hardcoding such a magic number is a *no-no* in terms of coding style. This constant can appear in more than one places. If we hardcode such a number and want to change its precision later, we would need to trace down and change every occurrence. Every time we need to use π, we have to remember or look up what is the precision that we use. Not only does this practice introduce more work, it is also likely to introduce bugs.

In C, we define π as a macro constant `M_PI`. But how should we do this in Java? This is where the ideal that a program consists of only objects with internal states that communicate with each other can feel a bit constraining. The constant π is universal, and does not really belong to any object (the value of π is the same for every circle!).

Another example is the method `sqrt()` that computes the square root of a given number. `sqrt` is a general function that is not associated with any object as well.

A solution to this is to associate these *global* values and functions with a *class* instead of with an *object*. For instance. Java predefines a `java.lang.Math` class[1] that is populated with constants `PI` and `E` (for Euler's number *e*), along with a long list of mathematical functions. To associate a method or a field with a class in Java, we declare them with the `static` keyword. We can additionally add a keyword `final` to indicate that the value of the field will not change and `public` to indicate that the field is accessible from outside the class. In short, the combination of `public static final` modifiers is used for constant values in Java.

```
class Math {
  :
  public static final double PI = 3.141592653589793;
  :
  :
}
```

We call these `static` fields that are associated with a class as *class fields*, and fields that are associated with an object as *instance fields*. Note that, a `static` class field needs not be `final` and it needs not be `public`. Class fields are useful for storing pre-computed values or configuration parameters associated with a class rather than individual objects.

Accessing Class Fields

A class field behaves just like a global variable and can be accessed in the code, anywhere the class can be accessed. Since a class field is associated with a class rather than an object, we access it through its *class name*. To use the static class field `PI`, for instance, we have to say `java.lang.Math.PI`.

```
public double getArea() {
  return java.lang.Math.PI * this.r * this.r;
}
```

A more common way, however, is to use `import` statements at the top of the program. If we have this line:

```
import java.lang.Math;
```

Then, we can save some typing and write:

```
public double getArea() {
  return Math.PI * this.r * this.r;
```

```
}
```

Class Fields and Methods in Python

Note that, in Python, any variable declared within a `class` block is a class field:

```
class Circle:
  x = 0
  y = 0
```

In the above example, `x` and `y` are class fields, not instance fields.

Example: The Circle class

Now, let revise our `Circle` class to improve the code and make it a little more complete. We now add in comments for each method and variable as well, as we always should.

```
// version 0.4
import java.lang.Math;

/**
 * A Circle object encapsulates a circle on a 2D plane.
 */
class Circle {
  private double x;  // x-coordinate of the center
  private double y;  // y-coordinate of the center
  private double r;  // the length of the radius

  /**
   * Create a circle centered on (x, y) with given radius
  */
  public Circle(double x, double y, double r) {
    this.x = x;
    this.y = y;
    this.r = r;
  }

  /**
   * Return the area of the circle.
   */
  public double getArea() {
    return Math.PI * this.r * this.r;
  }

  /**
   * Return true if the given point (x, y) is within the circle.
   */
  public boolean contains(double x, double y) {
    return false;
    // TODO: Left as an exercise
```

```
34      }
35  }
```

---

1. The class `Math` is provided by the package `java.lang` in Java. A package is simply a set of related classes (and interfaces, but I have not told you what is an interface yet). To use this class, we need to add the line `import java.lang.Math` at the beginning of our program. ↵

Unit 8: Class Methods

After this unit, students should:

- understand the differences between instance methods and class methods
- be able to define and use a class method
- know that the `main` method is the entry point to a Java program
- the modifies and parameters required for a `main` method

Let's suppose that, in our program, we wish to assign a unique integer identifier to every `Circle` object ever created. We can do this with the additions below:

```
1  class Circle {
2    private double x;  // x-coordinate of the center
3    private double y;  // y-coordinate of the center
4    private double r;  // the length of the radius
5    private final int id; // identifier
6    private static int lastId = 0; // the id of the latest circle instance
7
8    /**
9     * Create a circle centered on (x, y) with a given radius
10    */
11   public Circle(double x, double y, double r) {
12     this.x = x;
13     this.y = y;
14     this.r = r;
15     this.id = Circle.lastId;
16     Circle.lastId += 1;
17   }
18
19   /**
20    * Return how many circles have ever existed.
21    */
22   public static int getNumOfCircles() {
23     return Circle.lastId;
24   }
25 }
```

- On Line 5, we added a new instance field `id` to store the identifier of the circle. Note that, since the identifier of a circle should not change once it is created, we use the keyword `final` here.
- On Line 6, we added a new class field `lastId` to remember that the `lastId` of the latest circle instance. This field is maintained as part of the class `Circle` and is initialized to 0.

- On Line 15 and 16, as part of the constructor, we initialize `id` to `lastId` and increment `lastId`. We explicitly access `lastId` through `Circle` to make it clear that `lastId` is a class field.

Note that all of the above are done privately beneath the abstraction barrier.

Since `lastId` is incremented by one every time a circle is created, we can also interpret `lastId` as the number of circles created so far. On Line 22-24, we added a method `getNumOfCircles` to return its value.

The interesting thing here is that we declare `getNumOfCircles` with a `static` keyword. Similar to a `static` field, a `static` method is associated with a class, not to an instance of the class. Such method is called a *class method*. A class method is always invoked without being attached to an instance, and so it cannot access its instance fields or call other of its instance methods. The reference `this` has no meaning within a class method. Furthermore, just like a class field, a class method should be accessed through the class. For example, `Circle.getNumOfCircles()`.

Other examples of class methods include the methods provided in `java.lang.Math`: `sqrt`, `min`, etc. These methods can be invoked through the `Math` class: e.g., `Math.sqrt(x)`.

The `main` method

The most common class method you will use is probably the `main` method.

Every Java program has a class method called `main`, which serves as the entry point to the program. To run a Java program, we need to tell the JVM the class whose `main` method should be invoked first. In the example that we have seen,

```
1  java Hello
```

will invoke the `main` method defined within the class `Hello` to kick start the execution of the program.

The `main` method must be defined in the following way:

```
1  public final static void main(String[] args) {
2  }
```

You have learned what `public` and `static` means. The return type `void` indicates that `main` must not return a value. We have discussed what `final` means on a field, but are not ready to explain what `final` means on a method yet.

The `main` method takes in an array (`[]`) of strings as parameters. These are the command-line arguments that we can pass in when invoking `java`. `String` (or `java.lang.String`) is another class provided by the Java library that encapsulates a sequence of characters.

Unit 9: Composition

After learning this unit, students should understand:

- how to compose a new class from existing classes using composition
- how composition models the HAS-A relationship
- how sharing reference values in composed objects could lead to surprising results

Adding more Abstractions

Our previous implementation of `Circle` stores the center using its Cartesian coordinate (x, y). We have a method `contains` that takes in the Cartesian coordinate of a point. As such, our implementation of `Circle` assumes that a 2D point is best represented using its Cartesian coordinate.

Recall that we wish to hide the implementation details as much as possible, protecting them with an abstraction barrier, so that the client does not have to bother about the details and it is easy for the implementer to change the details. In this example, what happens if the application finds that it is more convenient to use polar coordinates to represent a 2D point? We will have to change the code of the constructor to `Circle` and the method `contains`. If our code contains other shapes or other methods in `Circle` that similarly assume a point is represented with its Cartesian coordinate, we will have to change them as well. It is easy for bugs to creep in. For instance, we might pass in the polar coordinate (r, θ) to a method, but the method treats the two parameters as the Cartesian (x, y).

We can apply the principle of abstraction and encapsulation here, and create a new class `Point`. The details of which are omitted and left as an exercise.

With the `Point` class, our `Circle` class looks like the following:

```
1  // version 0.5
2  import java.lang.Math;
3
4  /**
5   * A Circle object encapsulates a circle on a 2D plane.
6   */
7  class Circle {
8    private Point c;   // the center
9    private double r;  // the length of the radius
```

```
10
11    /**
12     * Create a circle centered on Point c with given radius r
13    */
14    public Circle(Point c, double r) {
15      this.c = c;
16      this.r = r;
17    }
18
19    /**
20     * Return the area of the circle.
21     */
22    public double getArea() {
23      return Math.PI * this.r * this.r;
24    }
25
26    /**
27     * Return true if the given point p is within the circle.
28     */
29    public boolean contains(Point p) {
30      // TODO: Left as an exercise
31      return false;
32    }
33  }
```

This example also illustrates the concept of *composition*. Our class `Circle` has been upgraded from being a bundle of primitive types and its methods, to a bundle that includes a reference type `Point` as well. In OOP, composition is a basic technique to build up layers of abstractions and construct sophisticated classes.

We have mentioned that classes model real-world entities in OOP. The composition models that HAS-A relationship between two entities. For instance, a circle *has a* point as the center.

Example: `Cylinder`

Now let's build up another layer of abstraction and construct a 3D object -- a cylinder. A cylinder has a circle as its base and has a height value. Using composition, we can construct a `Cylinder` class:

```
1   class Cylinder {
2     private Circle base;
3     private double height;
4
5     public Cylinder(Circle base, double height) {
6       this.base = base;
7       this.height = height;
8     }
9       :
10  }
```

Sharing References (aka Aliasing)

Recall that unlike primitive types, reference types may share the same reference values. This is called *aliasing*. Let's look at the subtleties of how this could affect our code and catch us by surprise.

Consider the following, where we create two circles `c1` and `c2` centered at the origin (0, 0).

```
Point p = new Point(0, 0);
Circle c1 = new Circle(p, 1);
Circle c2 = new Circle(p, 4);
```

Let's say that we want to allow a Circle to move its center. For the sake of this example, let's allow mutators on the class `Point`. Suppose we want to move `c1` and only `c1` to be centered at (1,1).

```
p.moveTo(1, 1);
```

You will find that by moving `p`, we are actually moving the center of *both* `c1` and `c2`! This result is due to both circles `c1` and `c2` sharing the same point. When we pass the center into the constructor, we are passing the reference instead of passing a cloned copy of the center.

This is a common source of bugs and we will see how we can reduce the possibilities of such bugs later in this module, but let's first consider the following "fix" (that is still not ideal).

Let's suppose that instead of moving `p`, we add a `moveTo` method to the `Circle` instead:

```
class Circle {
  private Point c;   // the center
  private double r;  // the length of the radius
    :
  /**
   * move the center of this circle to the given point
   */
  void moveTo(Point c) {
      this.c = c;
  }

   :
}
```

Now, to move `c1`,

```
Point p = new Point(0, 0);
Circle c1 = new Circle(p, 1);
Circle c2 = new Circle(p, 4);
c1.moveTo(new Point(1, 1));
```

You will find that `c1` will now have a new center, but `c2`'s center remains at (0,0). Why doesn't this solve our problem then? Recall that we can further composed circles into other objects. Let's say that we have two cylinders:

```
Cylinder cylinder1 = new Cylinder(c1, 1);
Cylinder cylinder2 = new Cylinder(c1, 1);
```

that share the same base, then the same problem repeats itself!

One solution is to avoid sharing references as much as possible. For instance,

```
Point p1 = new Point(0, 0);
Circle c1 = new Circle(p1, 1);

Point p2 = new Point(0, 0);
Circle c2 = new Circle(p2, 4);

p1.moveTo(1, 1);
```

Without sharing references, moving `p1` only affects `c1`, so we are safe.

The drawback of not sharing objects with the same content is that we will have a proliferation of objects and the computational resource usage is not optimized. This is an example of the trade offs we mentioned in the introduction to this module: we are sacrificing the computational cost to save programmers from potential suffering!

Another approach to address this issue is *immutability*. We will cover this later in the module.

Unit 10: Heap and Stack

After taking this unit, students should:

- understand when memory are allocated/deallocated from the heap vs. from the stack
- understand the concept of call stack in JVM

Heap and Stack

The Java Virtual Machine (JVM) manages the memory of Java programs while its bytecode instructions are interpreted and executed. Different JVM implementations may implement these differently, but typically a JVM implementation partitions the memory into several regions, including:

- *method area* for storing the code for the methods;
- *metaspace* for storing meta information about classes;
- *heap* for storing dynamically allocated objects;
- *stack* for local variables and call frames.

Since the concepts of heap and stack are common to all execution environments (either based on bytecode or machine code), we will focus on them here.

The *heap* is the region in memory where all objects are allocated in and stored, while the *stack* is the region where all variables (including primitive types and object references) are allocated in and stored.

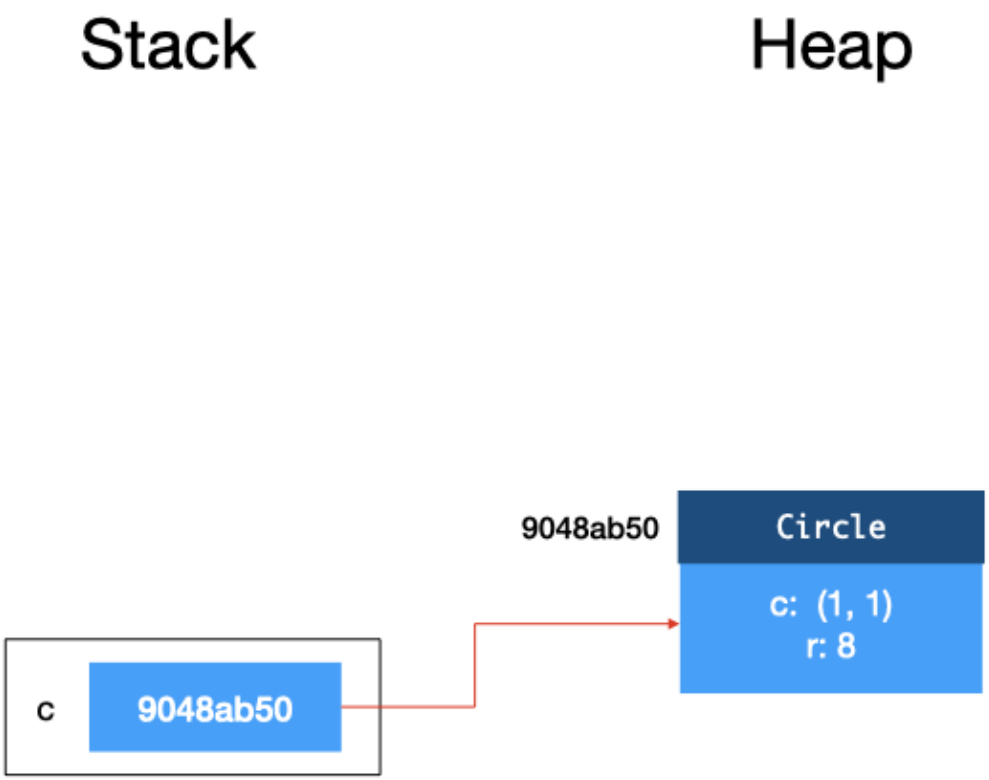
Considers the following two lines of code.

```
1  Circle c;
2  c = new Circle(new Point(1, 1), 8);
```

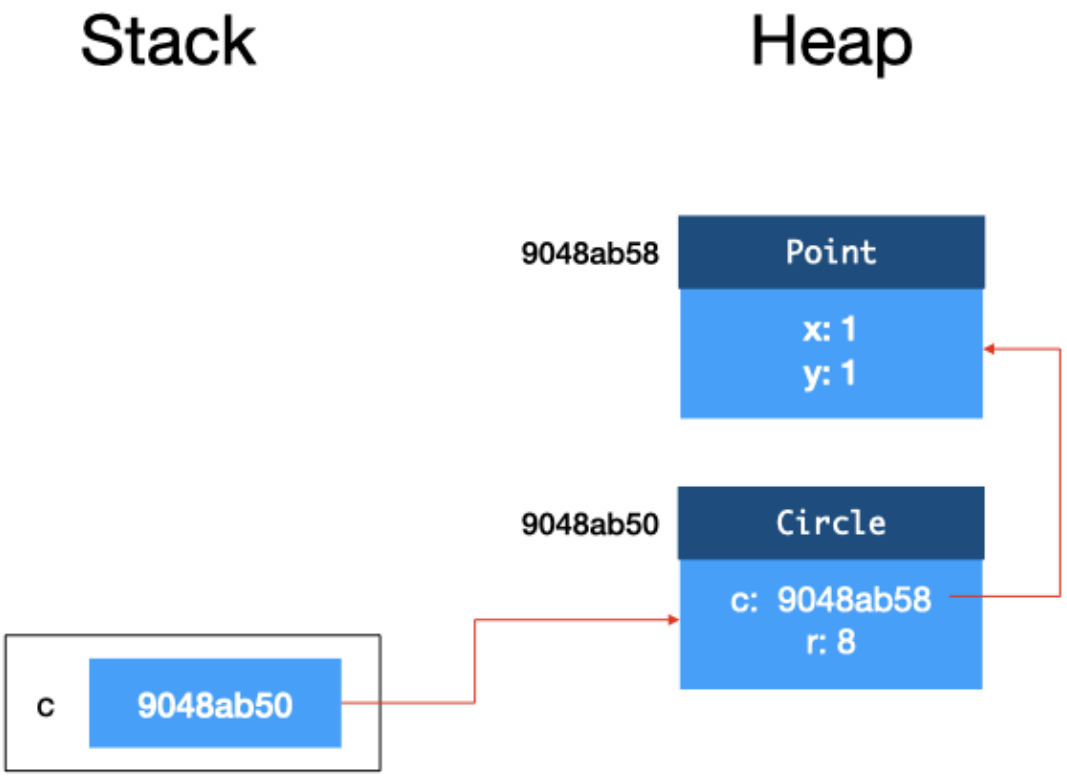
Line 1 declares a variable `c`. When the JVM executes this line of code, it allocates some memory space for an object reference for `c`, the content is initialized to `null`. Since `c` is a variable, it resides in the stack.

Line 2 creates a new Circle object. When the JVM executes this line of code, it allocates some memory space for a `Circle` object on the heap. The memory address of this memory space becomes the reference of the object and is assigned to the variable `c`.

This is shown in the figure below.



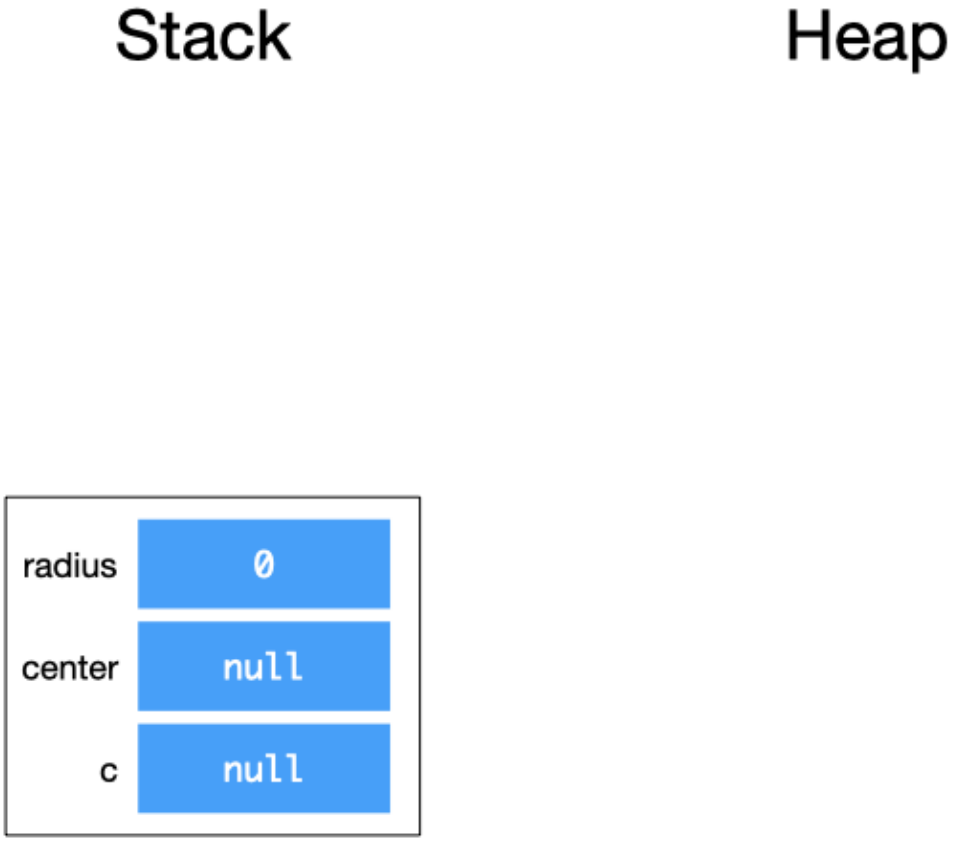
The first argument to the `Circle` constructor is also an object, so to be more precise, when Line 2 above is executed, a `Point` object is also created and allocated on the heap. So the field `c` inside `Circle` is actually a reference to this object.



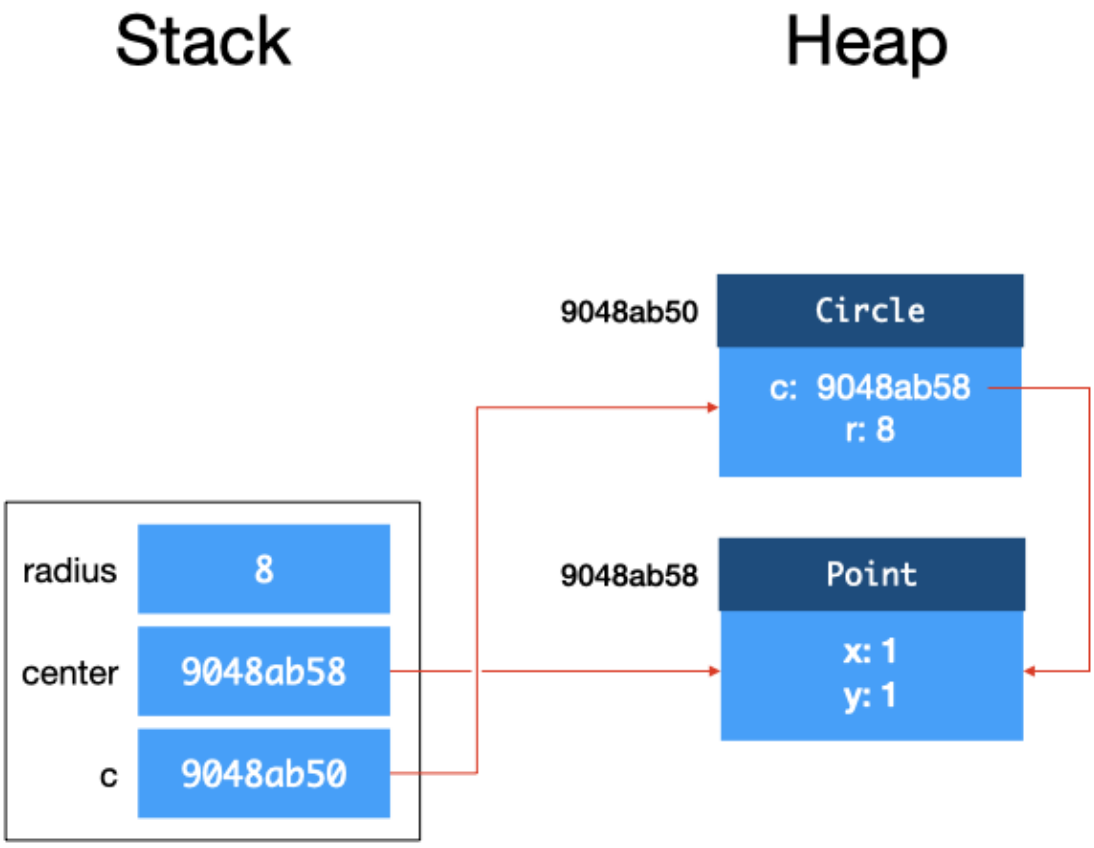
Now, let's look at a slightly different example.

```
1  Circle c;
2  Point center;
3  double radius;
4  radius = 8;
5  center = new Point(1, 1);
6  c = new Circle(center, radius);
```

In the second example, we have three variables, `c`, `center`, and `radius`. Lines 1-3 declare the variables, and as a result, we have three variables allocated on the stack. Recall that for object references, they are initialized to `null`. Primitive type variables (e.g., `radius`) are initialized to 0.



After Lines 4-6, we have:



Call Stack

Now, let's look at what happens when we invoke a method. Take the `distanceTo` method in `Point` as an example:

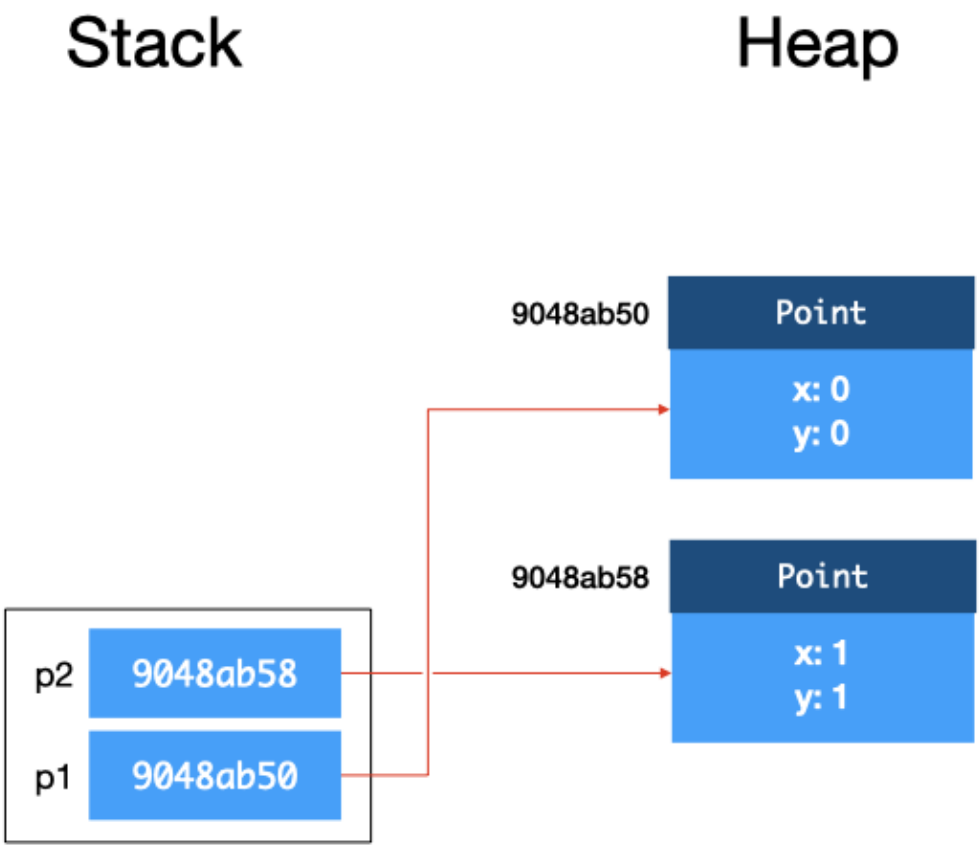
```
1  class Point {
2    private double x;
3    private double y;
4
5    public Point(double x, double y) {
6      this.x = x;
7      this.y = y;
8    }
9
10   public double distanceTo(Point q) {
11     return Math.sqrt((q.x - this.x)*(q.x - this.x)+(q.y - this.y)*(q.y -
12 this.y));
13
```

```
    }
}
```

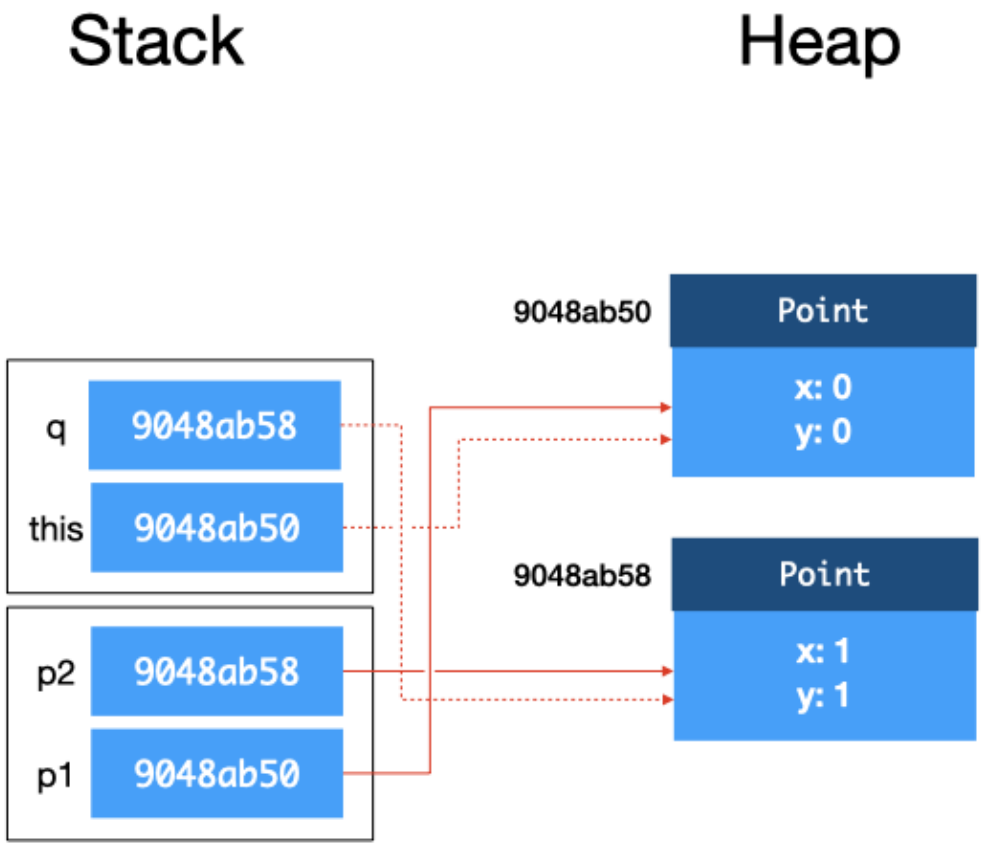
and the invocation:

```
Point p1 = new Point(0, 0);
Point p2 = new Point(1, 1);
p1.distanceTo(p2);
```

After declaring `p1` and `p2` and creating both objects, we have:



When `distanceTo` is called, the JVM creates a *stack frame* for this instance method call. This stack frame is a region of memory that tentatively contains (i) the `this` reference, (ii) the method arguments, and (iii) local variables within the method, among other things[12]. When a class method is called, the stack frame does not contain the `this` reference.



You can see that the *references* to the objects `p1` and `p2` are copied onto the stack frame. `p1` and `this` point to the same object, and `p2` and `q` point to the same object. Within the method, any modification done to `this` would change the object referenced to by `p1`, and any change made to `q` would change the object referenced to by `p2` as well. After the method returns, the stack frame for that method is destroyed.

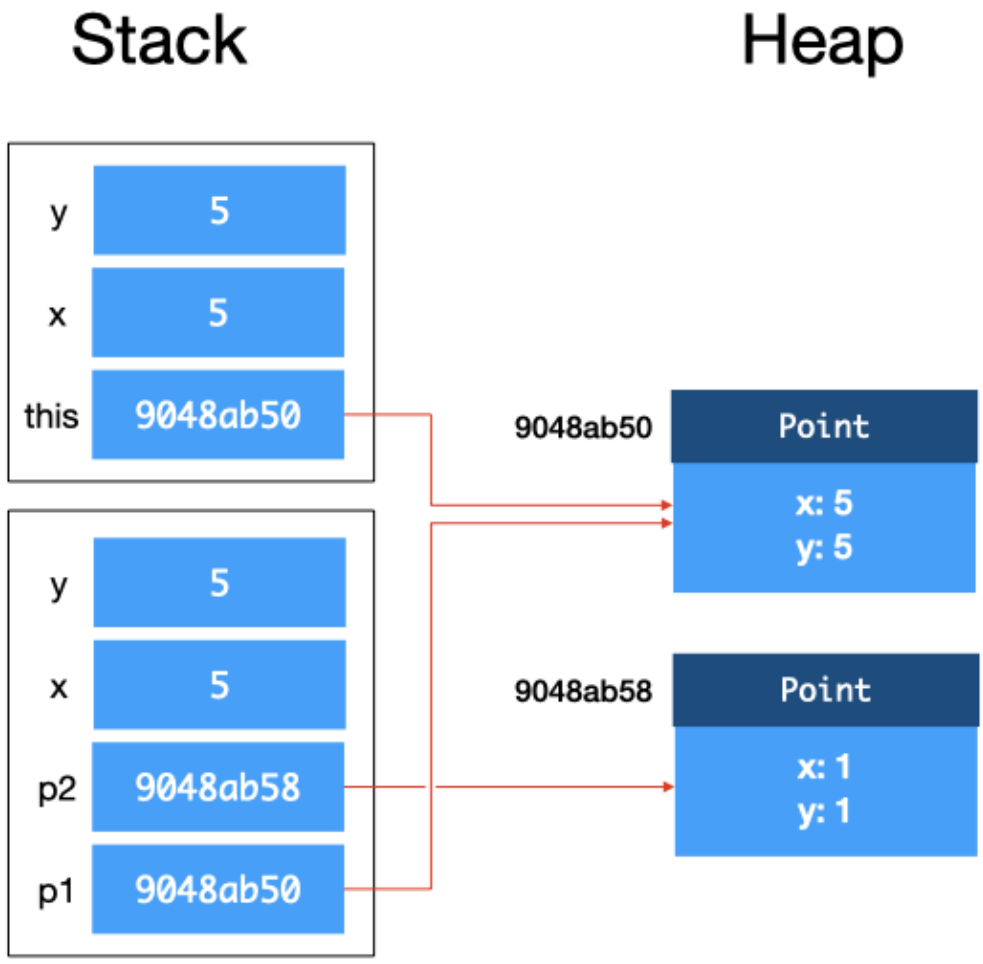
Let's consider a new `move` method for the class `Point` that has two parameters `(double x, double y)` and moves the `x` and `y` cordinates of the `Point`.

```
1  class Point {
2    private double x;
3    private double y;
4
5    public Point(double x, double y) {
6      this.x = x;
7      this.y = y;
8    }
```

```
9
10    public void move(double x, double y) {
11      this.x = x;
12      this.y = y;
13    }
14  }
```

```
1  Point p1 = new Point(0, 0);
2  Point p2 = new Point(1, 1);
3  double x = 5;
4  double y = 5;
5  p1.move(x, y);
```

Again, we create a stack frame, copy the reference to object `p1` into `this`, copy `x` from the calling method to `x` the argument within the method, copy `y` from the calling method to `y` the argument within the method.



What is important here is that, as `x` and `y` are primitive types instead of references, we copy the values onto the stack. If we change `x` or `y` within `move`, the `x` and `y` of the calling function will not change. This behavior is the same as you would expect in C. However, unlike in C where you can pass in a pointer to a variable, you cannot pass in a reference to a primitive type in any way in Java. If you want to pass in a variable of primitive type into a method and have its value changed, you will have to use a *wrapper class*. The details of how to do this are left as an exercise.

To summarize, Java uses *call by value* for primitive types, and *call by reference* for objects.

If we made multiple nested method calls, as we usually do, the stack frames get stacked on top of each other.

One final note: the memory allocated on the stack is deallocated when a method returns. The memory allocated on the heap, however, stays there as long as there is a reference to it (either from another object or from a variable in the stack). Unlike C or C++, in Java, you do not have to free the memory allocated to objects. The JVM runs a *garbage collector* that checks for unreferenced objects on the heap and cleans up the memory automatically.

---

1. This is not that different from how an OS handles function call in a machine code, as you will see in CS2100/CS2106. ↵
2. The other things are JVM implementation independent and not relevant to our discussion here. ↵

Unit 11: Inheritance

After taking this unit, students should:

- understand inheritance as a mechanism to extend existing code
- understand how inheritance models the IS-A relationship
- know how to use the `extends` keyword for inheritance
- understand inheritance as a subtype
- be able to determine the run-time type and compile-time type of a variable

Extension with Composition

We have seen how composition allows us to compose a new, more complex, class, out of existing classes, without breaking the abstraction barrier of existing classes. Sometimes, however, composition is not the right approach. Let's consider the following example. Let's suppose that we, as a client, want to add color as a property to our `Circle`.

Without penetrating the abstraction barrier of `Circle`, we can do the following:

```
1  // version 0.1 (using composition)
2  class ColoredCircle {
3    private Circle circle;
4    private Color color;
5
6    public ColoredCircle(Circle circle, Color color) {
7      this.circle = circle;
8      this.color = color;
9    }
10 }
```

where `Color` is another abstraction representing the color of shapes.

What should we do if we want to calculate the area of our colored circle? Suppose we already have a `ColoredCircle` instance called `coloredCircle`. We could make `circle` public and call `coloredCircle.circle.getArea()`, or we could add an accessor and call `coloredCircle.getCircle().getArea()`. Both of these are not ideal, since it breaks the abstraction barrier and reveals that the `ColoredCircle` class stores a `circle` (the latter being slightly better than the first).

A better alternative is to let `ColoredCircle` provide its own `getArea()` method, and *forward* its call to `Circle`.

```
// version 0.2 (using composition)
class ColoredCircle {
  private Circle circle;
  private Color color;

  public ColoredCircle(Circle circle, Color color) {
    this.circle = circle;
    this.color = color;
  }

  public double getArea() {
    return circle.getArea();
  }
}
```

Then, the client to `ColoredCircle` can just call `coloredCircle.getArea()` without knowing or needing to know how a colored circle is represented internally. The drawback of this approach is that we might end up with many such boilerplate forwarding methods.

Extension with Inheritance

Recall the concept of subtyping. We say that $S <: T$ if any piece of code written for type T also works for type S.

Now, think about `ColoredCircle` and `Circle`. If someone has written a piece of code that operates on `Circle` objects. Do we expect the same code to work on `ColoredCircle`? In this example, yes! A `ColoredCircle` object should behave just like a circle -- we can calculate its area, circumference, check if two circles intersect, check if a point falls within the circle, etc. The only difference, or more precisely, extension, is that it has a color, and perhaps has some methods related to this additional field. So, `ColoredCircle` *is a subtype of* `Circle`.

We now show you how we can introduce this subtype relationship in Java, using the `extends` keyword. We can reimplement our `ColoredCircle` class this way:

```
// version 0.3 (using inheritance)
class ColoredCircle extends Circle {
  private Color color;

  public ColoredCircle(Point center, double radius, Color color) {
    super(center, radius);  // call the parent's constructor
    this.color = color;
  }
}
```

We just created a new type `ColoredCircle` as a class that extends from `Circle`. We call `Circle` the *parent class* or *superclass* of `ColoredCircle`; and `ColoredCircle` a *subclass* of `Circle`.

We also say that `ColoredCircle` *inherits* from `Circle`, since all the public fields of `Circle` (center and radius) and public methods (like `getArea()`) are now accessible to `ColoredCircle`. Just like a parent-child relationship in real-life, however, anything private to the parent remains inaccessible to the child. This privacy veil maintains the abstraction barrier of the parent from the child, and creates a bit of a tricky situation -- technically a child `ColoredCircle` object has a center and a radius, but it has no access to it!

Line 6 of the code above introduces another keyword in Java: `super`. Here, we use `super` to call the constructor of the superclass, to initialize its center and radius (since the child has no direct access to these fields that it inherited).

The concept we have shown you is called *inheritance* and is one of the four pillars of OOP. We can think of inheritance as a model for the "*is a*" relationship between two entities.

With inheritance, we can call `coloredCircle.getArea()` without knowing or needing to know how a colored circle is represented internally and without forwarding methods.

When NOT to Use Inheritance

Inheritance tends to get overused. *In practice, we seldom use inheritance.* Let's look at some examples of how *not* to use inheritance, and why.

You may come across examples online or in books that look like the following:

```
1  class Point {
2    private double x;
3    private double y;
4      :
5  }
6
7  class Circle extends Point {
8    private double radius;
9      :
10 }
11
12 class Cylinder extends Circle {
13   private double height;
14     :
15 }
```

The difference between these implementations and the one you have seen in Unit 9 is that it uses inheritance rather than composition.

`Circle` implemented like the above would have the center coordinate inherited from the parent (so it has three fields, x, y, and radius); `Cylinder` would have the fields corresponding to a circle, which is its base and height. In terms of modeling the properties of circle and cylinder, we have all the right properties in the right class.

When we start to consider methods encapsulated with each object, things start to break down. Consider a piece of code written as follows:

```
void foo(Circle c, Point p) {
  if (c.contains(p)) {
    // do something
  }
}
```

Since `Cylinder` is a subtype of `Point` according to the implementation above, the code above should still work also if we replace `Point` with a `Cylinder` (according to the semantic of subtyping). But it gets weird -- what is the meaning of a `Circle` (in 2D) containing a Cylinder (in 3D)? We could come up with a convoluted meaning that explains this, but it is likely not what the original implementer of `foo` expects.

The message here is this: *Use composition to model a has-a relationship; inheritance for a is-a relationship. Make sure inheritance preserves the meaning of subtyping.*

Run-Time Type

Recall that Java allows a variable of type T to hold a value from a variable of type S only if $S <: T$. Since `ColoredCircle` <: `Circle`, the following is not allowed in Java:

```
ColoredCircle c = new Circle(p, 0); // error
```

but this is OK:

```
Circle c = new ColoredCircle(p, 0, blue); // OK
```

where `p` is a `Point` object and `blue` is a `Color` object.

Also, recall that `Circle` is called the compile-time type of `c`. Here, we see that `c` is now referencing an object of subtype `ColoredCircle`. Since this assignment happens during run-time, we say that the *run-time type* of `c` is `ColoredCircle`. The distinction between these two types will be important later.

Unit 12: Overriding

After reading this unit, students should

- be aware that every class inherits from `Object`
- be familiar with the `equals` and `toString` methods
- understand what constitutes a method signature
- understand method overriding
- appreciate the power of method overriding
- understand what Java annotations are for, and know when to use `@Override`
- be exposed to the `String` class and its associated methods, especially the `+` operator

`Object` and `String`

In Java, every class that does not extend another class inherits from the class `Object` implicitly. `Object` is, therefore, the "ancestor" of all classes in Java and is at the root of the class hierarchy.

The `Object` class does not encapsulate anything in particular. It is a very general class that provides useful methods common to all objects. The two useful ones that we are going to spend time with are:

- `equals(Object obj)`, which checks if two objects are equal to each other, and
- `toString()`, which returns a string representation of the object as a `String` object.

The `toString` Method

The `toString` method is very special, as this is invoked *implicitly* by Java, by default, to convert a reference object to a `String` object during string concatenation using the operator `+`.

We showed you that in Python, `4 + "Hello"` would result in a type mismatch error. In Java, however, `4 + "Hello"` will result in the string `"4Hello"`. In this example, the primitive value 4 is converted to a string before concatenation.

A more interesting scenario is what happens if we try to concatenate, say, a `Circle` object with a string. Let's say we have:

```
Circle c = new Circle(new Point(0, 0), 4.0);
String s = "Circle c is " + c;
```

You will see that `s` now contains the string "Circle c is Circle@1ce92674 " (the seemingly gibberish text after @` is the reference to the object and so your result will be different).

What happened here is that the `+` operator sees that one of the operands is a string but the other is not, so it converts the one that is not a string to a string by calling its `toString()` method automatically for us. This is equivalent to[1]

```
Circle c = new Circle(new Point(0, 0), 4.0);
String s = "Circle c is " + c.toString();
```

Recall that in our `Circle` class (up to version 0.5) we do not have any `toString` method. The `toString` method that we invoked here is the `toString` method inherited from its parent `Object`.

> **`jshell` and `toString`**
>
> Recall that `jshell` is a REPL tool. After evaluating an expression, `jshell` prints the resulting value out. If the resulting value is a reference type, `jshell` will invoke `toString` to convert the reference type to a string first, before printing the string.

Customizing `toString` for `Circle`

The `Object::toString` method (that is our notation for the method `toString` from the class `Object`) is not very user friendly. Ideally, when we print a `Circle` object, say, for debugging, we want to see its center and its radius. To do so, we can define our own `toString` method in `Circle`. Let's upgrade our `Circle` class to do this:

```
// version 0.6
import java.lang.Math;

/**
 * A Circle object encapsulates a circle on a 2D plane.
 */
class Circle {
  private Point c;   // the center
  private double r;  // the length of the radius

  /**
```

```
12      * Create a circle centered on Point c with given radius r
13     */
14     public Circle(Point c, double r) {
15       this.c = c;
16       this.r = r;
17     }
18
19     /**
20      * Return the area of the circle.
21      */
22     public double getArea() {
23       return Math.PI * this.r * this.r;
24     }
25
26     /**
27      * Return true if the given point p is within the circle.
28      */
29     public boolean contains(Point p) {
30       return false;
31       // TODO: Left as an exercise
32     }
33
34     /**
35      * Return the string representation of this circle.
36      */
37     @Override
38     public String toString() {
39         return "{ center: " + this.c + ", radius: " + this.r + " }";
40     }
41   }
```

The body of the method `toString` simply constructs a string representation for this circle object and returns it. With this `toString` implemented, the output will look something like this:

```
1  Circle c is { center: (0.0, 0.0), radius: 4.0 }
```

Note that when the center `this.c` is converted to a string, the `toString` method of `Point` is invoked. We leave the implementation of `Point::toString` as an exercise.

Method Overriding

What we just did is called *method overriding* in OOP. Inheritance is not only good for extending the behavior of an existing class but through method overriding, we can *alter* the behavior of an existing class as well.

Let's define the *method signature* of a method as the method name and the number, type, and order of its parameters, and the *method descriptor* as the method signature plus the return type.

When a subclass defines an instance method with the same *method descriptor* as an instance method in the parent class, we say that the instance method in the subclass *overrides* the instance method in the parent class[2]. In the example above, `Circle::toString` has overridden `Object::toString`.

The `@Override` Annotation

Line 37 in the example above contains the symbol `@Override`. This symbol is an example of *annotation* in Java. An annotation is not part of the program and does not affect the bytecode generated. Instead, it is a *hint* to the compiler. Remember that the compiler is our friend who will do its best to help detect errors early, during compilation. We must do our part to help the compiler help us. Here, `@Override` is a hint to the compiler that the following method, `toString`, is intended to override the method in the parent class. In case, there is a typo and overriding is not possible, the compiler will let us know.

It is therefore recommended and expected that all overriding methods in your code are annotated with `@Override`.

Using `super` To Access Overridden Methods

After a subclass overrides a method in the superclass, the methods that have been overridden can still be called, with the `super` keyword. For instance, the following `Circle::toString` calls `Object::toString` to prefix the string representation of the circle with `Circle@1ce92674`.

```
1  @Override
2  public String toString() {
3    return super.toString() + " { center: " + this.c + ", radius: " + this.r
4  + " }";
   }
```

---

1. Calling `toString` explicitly is not wrong, but we usually omit the call to keep the code readable and succinct. ↵
2. It is possible to override a method in some cases when the return type is different. We will discuss this during recitations. ↵

Unit 13: Overloading

After reading this unit, students should

- understand what is overloading
- understand how to create overloaded methods

Method overloading

In the previous unit, we introduced *method overriding*. That is, when a subclass defines an instance method with the same *method descriptor* as an instance method in the parent class.

In contrast, *method overloading* is when we have two or more methods in the same class with the same name but a differing *method signature*[1]. In other words, we create an overloaded method by changing the type, order, and number of parameters of the method but keeping the method name identical.

Lets consider an `add` method which allows us to add two numbers, and returns the result. What if we would like to create an `add` method to sum up three numbers?

```
1  public int add(int x, int y) {
2    return x + y;
3  }
4
5  public int add(int x, int y, int z) {
6    return x + y;
7  }
```

In the example above, the methods `add(int, int)` and `add(int, int, int)` are overloaded. They have the same name but a different number of parameters. We can see that this allows us to write methods to handle differing inputs.

Now lets consider our `Circle` class again. Our `Circle::contains(Point)` method allows us to check if a `Point` is within the radius of the current instance of the `Circle`. We would like to create a new method `Circle::contains(double, double)` which will allow us to check if an `x` and `y` co-ordinate (another valid representation of a point) is within our circle.

```
1  import java.lang.Math;
2
```

```
class Circle {
  private Point c;
  private double r;

  public Circle(Point c, double r) {
    this.c = c;
    this.r = r;
  }

  public double getArea() {
    return Math.PI * this.r * this.r;
  }

  public boolean contains(Point p) {
    return false;
    // TODO: Left as an exercise
  }

  public boolean contains(double x, double y) {
    return false;
    // TODO: Left as an exercise
  }

  @Override
  public String toString() {
      return "{ center: " + this.c + ", radius: " + this.r + " }";
  }
}
```

In the above example, `Circle::contains(Point)` and `Circle::contains(double, double)` are overloaded methods.

Recall that overloading requires changing the order, number, and/or type of parameters and says nothing about the names of the parameters. Consider the example below, where we have two `contains` methods in which we swap parameter names.

```
  public boolean contains(double x, double y) {
    return false;
    // TODO: Left as an exercise
  }

  public boolean contains(double y, double x) {
    return false;
    // TODO: Left as an exercise
  }
```

These two methods have the same method signature, and therefore `contains(double, double)` and `contains(double, double)` are not distinct methods. They are not overloaded, and therefore this above example will not compile.

As it is also a method, it is possible to overload the class *constructor* as well. As in the example below, we can see an overloaded constructor which gives us a handy way to instantiate a `Circle` object that is the unit circle.

```java
class Circle {
  private Point c;
  private double r;

  public Circle(Point c, double r) {
    this.c = c;
    this.r = r;
  }

  // Overloaded constructor
  public Circle() {
    this.c = new Point(0, 0);
    this.r = 1;
  }
  :
}
```

```java
// c1 points to a new Circle object with a centre (1, 1) and a radius of 2
Circle c1 = new Circle(new Point(1, 1), 2);
// c2 points to a new Circle object with a centre (0, 0) and a radius of 1
Circle c2 = new Circle();
```

It is also possible to overload `static` class methods in the same way as instance methods. In the next unit, we will see how Java chooses which method implementation to execute when a method is invoked.

---

1. Note that this is not the same as the *method descriptor*. You can not overload a method by changing the return type. ↩

Unit 14: Polymorphism

After reading this unit, students should

- understand dynamic binding and polymorphism
- be aware of the `equals` method and the need to override it to customize the equality test
- understand when narrowing type conversion and type casting are allowed

Taking on Many Forms

Method overriding enables *polymorphism*, the fourth and the last pillar of OOP, and arguably the most powerful one. It allows us to change how existing code behaves, without changing a single line of the existing code (or even having access to the code).

Consider the function `say` below:

```
1  void say(Object obj) {
2      System.out.println("Hi, I am " + obj.toString());
3  }
```

Note that this method receives an `Object` instance. Since both `Point` <: `Object` and `Circle` <: `Object`, we can do the following:

```
1  Point p = new Point(0, 0);
2  say(p);
3  Circle c = new Circle(p, 4);
4  say(c);
```

When executed, `say` will first print `Hi, I am (0.0, 0.0)`, followed by `Hi, I am { center: (0.0, 0.0), radius: 4.0 }`. *We are invoking the overriding* `Point::toString` *in the first call, and* `Circle::toString` *in the second call.* The same method invocation `obj.toString()` causes two different methods to be called in two separate invocations!

In biology, polymorphism means that an organism can have many different forms. Here, the variable `obj` can have many forms as well. Which method is invoked is decided *during run-time*, depending on the run-time type of the `obj`. This is called *dynamic binding* or *late binding* or *dynamic dispatch*.

Before we get into this in more detail, let consider overriding `Object::equals`.

The `equals` method

`Object::equals` compares if two object references refer to the same object. Suppose we have:

```
Circle c0 = new Circle(new Point(0, 0), 10);
Circle c1 = new Circle(new Point(0, 0), 10);
Circle c2 = c1;
```

`c2.equals(c1)` returns `true`, but `c0.equals(c1)` returns `false`. Even though `c0` and `c1` are *semantically* the same, they refer to the two different objects.

To compare if two circles are *semantically* the same, we need to override this method[1].

```
// version 0.7
import java.lang.Math;

/**
 * A Circle object encapsulates a circle on a 2D plane.
 */
class Circle {
  private Point c;   // the center
  private double r;  // the length of the radius

  /**
   * Create a circle centered on Point c with given radius r
  */
  public Circle(Point c, double r) {
    this.c = c;
    this.r = r;
  }

  /**
   * Return the area of the circle.
   */
  public double getArea() {
    return Math.PI * this.r * this.r;
  }

  /**
   * Return true if the given point p is within the circle.
   */
  public boolean contains(Point p) {
    return false;
    // TODO: Left as an exercise
  }

  /**
   * Return the string representation of this circle.
   */
  @Override
  public String toString() {
      return "{ center: " + this.c + ", radius: " + this.r + " }";
```

```
40    }
41
42    /**
43     * Return true the object is the same circle (i.e., same center, same
44  radius).
45     */
46    @Override
47    public boolean equals(Object obj) {
48      if (obj instanceof Circle) {
49        Circle circle = (Circle) obj;
50        return (circle.c.equals(this.c) && circle.r == this.r);
51      }
52      return false;
53    }
  }
```

This is more complicated than `toString`. There are a few new concepts involved here:

- `equals` takes in a parameter of compile-time type `Object`. It only makes sense if we compare (during run-time) a circle with another circle. So, we first check if the run-time type of `obj` is a subtype of `Circle`. This is done using the `instanceof` operator. The operator returns `true` if `obj` has a run-time type that is a subtype of `Circle`.
- To compare `this` circle with the given circle, we have to access the center `c` and radius `r`. But if we access `obj.c` or `obj.r`, the compiler will complain. As far as the compiler is concerned, `obj` has the compile-time type `Object`, and there is no such fields `c` and `r` in the class `Object`! This is why, after assuring that the run-time type of `obj` is a subtype of `Circle`, we assign `obj` to another variable `circle` that has the compile-time type `Circle`. We finally check if the two centers are equal (again, `Point::equals` is left as an exercise) and the two radii are equal[2].
- The statement that assigns `obj` to `circle` involves *type casting*. We mentioned before that Java is strongly typed and so it is very strict about type conversion. Here, Java allows type casting from type T to S if $S <: T$. [3]: This is called *narrowing type conversion*. Unlike widening type conversion, which is always allowed and always correct, a *narrowing type conversion* requires explicit typecasting and validation during run-time. If we do not ensure that `obj` has the correct run-time type, casting can lead to a run-time error (which if you recall, is bad).

All these complications would go away, however, if we define `Circle::equals` to take in a `Circle` as a parameter, like this:

```
1   class Circle {
2       :
3     /**
4      * Return true the object is the same circle (i.e., same center, same
5   radius).
6      */
7     @Override
```

```
8     public boolean equals(Circle circle) {
9         return (circle.c.equals(this.c) && circle.r == this.r);
10    }
    }
```

This version of `equals` however, does not override `Object::equals`. Since we hinted to the compiler that we meant this to be an overriding method, using `@Override`, the compiler will give us an error. This is not treated as method overriding, since the signature for `Circle::equals` is different from `Object::equals`.

Why then is overriding important? Why not just leave out the line `@Override` and live with the non-overriding, one-line, `equals` method above?

The Power of Polymorphism

Let's consider the following example. Suppose we have a general `contains` method that takes in an array of objects. The array can store any type of objects: `Circle`, `Square`, `Rectangle`, `Point`, `String`, etc. The method `contains` also takes in a target `obj` to search for, and returns true if there is an object in `array` that equals to `obj`.

```
1  // version 0.1 (with polymorphism)
2  boolean contains(Object[] array, Object obj) {
3    for (Object curr : array) {
4      if (curr.equals(obj)) {
5        return true;
6      }
7    }
8    return false;
9  }
```

With overriding and polymorphism, the magic happens in Line 4 -- depending on the run-time type of `curr`, the corresponding, customized version of `equals` is called to compare against `obj`.

However, if `Circle::equals` takes in a `Circle` as the parameter, the call to `equals` inside the method `contains` would not invoke `Circle::equals`. It would invoke `Object::equals` instead due to the matching method signature, and we can't search for `Circle` based on semantic equality.

To have a generic `contains` method without polymorphism and overriding, we will have to do something like this:

```
1  // version 0.2 (without polymorphism)
2  boolean contains(Object[] array, Object obj) {
3    for (Object curr : array) {
4      if (obj instanceof Circle) {
```

```
5         if (curr.equals((Circle)obj)) {
6           return true;
7         }
8       } else if (obj instanceof Square) {
9         if (curr.equals((Square)obj)) {
10          return true;
11        }
12      } else if (obj instanceof Point) {
13        if (curr.equals((Point)obj)) {
14          return true;
15        }
16      }
17       :
18    }
19    return false;
20  }
```

which is not scalable since every time we add a new class, we have to come back to this method and add a new branch to the `if-else` statement!

As this example has shown, polymorphism allows us *to write succinct code that is future proof*. By dynamically deciding which method implementation to execute during run-time, the implementer can write short yet very general code that works for existing classes as well as new classes that might be added in the future by the client, without even the need to re-compile!

---

1. If we override `equals()`, we should generally override `hashCode()` as well, but let's leave that for another lesson on another day. ↵
2. The right way to compare two floating-point numbers is to take their absolute difference and check if the difference is small enough. We are sloppy here to keep the already complicated code a bit simpler. You shouldn't do this in your code. ↵
3. This is not the only condition where type casting is allowed. We will look at other conditions in later units. ↵

Unit 15: Method Invocation

After this unit, the student should:

- understand the two step process that Java uses to determine which method implementation will be executed when a method is invoked
- understand that Class Methods do not support dynamic binding

How does Dynamic Binding work?

We have seen that, with the power of dynamic binding and polymorphism, we can write succinct, future-proof code. Recall that example below, where the magic happens in Line 4. The method invocation `curr.equals(obj)` will call the corresponding implementation of the `equals` method depending on the run-time type of `curr`.

```
1  // version 0.1 (with polymorphism)
2  boolean contains(Object[] array, Object obj) {
3    for (Object curr : array) {
4      if (curr.equals(obj)) {
5        return true;
6      }
7    }
8    return false;
9  }
```

How does dynamic binding work? To be more precise, when the method `equals` is invoked on the target `curr`, how does Java decide which method implementation is this invocation bound to? While we have alluded to the fact that the run-time type of the target `curr` plays a role, this is not the entire story. Recall that we may have multiple versions of `equals` due to overloading. So, Java also needs to decide, among the overloaded `equals`, which version of `equals` this particular invocation is bound to.

This unit elaborates on Java's decision process to resolve which method implemented in which class should be executed when a method is invoked. This process is a two-step process. The first occurs during compilation; the second during run time.

During Compile Time

During compilation, Java determines the method descriptor of the method invoked, using the compile-time type of the target.

For example, in the line

```
curr.equals(obj)
```

above, the target `curr` has the compile-time type `Object`.

Let's generalize the compile-time type of the target to C. To determine the method descriptor, the compiler searches for all methods that can be correctly invoked on the given argument.

In the example above, we look at the class `Object`, and there is only one method called `equals`. The method can be correctly invoked with one argument of type `Object`.

What if there are multiple methods that can correctly accept the argument? In this case, we choose the *most specific* one. Intuitively, a method M is more specific than method N if the arguments to M can be passed to N without compilation error. For example, let's say a class `Circle` implements:

```
      boolean equals(Circle c) { .. }

      @Override
      boolean equals(Object c) { .. }
```

Then, `equals(Circle)` is more specific than `equals(Object)`. Every `Circle` is an `Object`, but not every `Object` is a `Circle`.

Once the method is determined, the method's descriptor (return type and signature) is stored in the generated code.

In the example above, the method descriptor `boolean equals(Object)` will be stored in the generated binaries. Note that it does not include information about the class that implements this method. The class to take this method implementation from will be determined in Step 2 during run-time.

During Run Time

During execution, when a method is invoked, the method descriptor from Step 1 is first retrieved. Then, the run-time type of the target is determined. Let the run-time type of the target be R. Java then looks for an accessible method with the matching descriptor in R. If no such method is found, the search will continue up the class hierarchy, first to the parent class of R, then to the grand-parent class of R, and so on, until we reach the root

`Object`. The first method implementation with a matching method descriptor found will be the one executed.

For example, let's consider again the invocation in the highlighted line below again:

```
1  // version 0.1 (with polymorphism)
2  boolean contains(Object[] array, Object obj) {
3    for (Object curr : array) {
4      if (curr.equals(obj)) {
5        return true;
6      }
7    }
8    return false;
9  }
```

Let's say that `curr` points to a `Circle` object during run-time. Suppose that the `Circle` class does not override the method `equals` in `Object`. As a result, Java can't find a matching method descriptor `boolean equals(Object)` in the method `Circle`. It then looks for the method in the parent of `Circle`, which is the class `Object`. It finds the method `Object::equals(Object)` with a matching descriptor. Thus, the method `Object::equals(Object)` is executed.

Now, suppose that `Circle` overrides the method `Object::equals(Object)` with its own `Circle::equals(Object)` method. Since Java starts searching from the class `Circle`, it finds the method `Circle::equals(Object)` that matches the descriptor. In this case, `curr.target(obj)` will invoke the method `Circle::equals(Object)` instead.

Invocation of Class Methods

The description above applies to instance methods. Class methods, on the other hand, do not support dynamic binding. The method to invoke is resolved statically during compile time. The same process in Step 1 is taken, but the corresponding method implementation in class C will always be executed during run-time, without considering the run-time type of the target.

Unit 16: Liskov Substitution Principle

After this unit, the student should:

- understand the type of bugs that reckless developers can introduce when using inheritance and polymorphism
- understand the Liskov Substitution Principle and thus be aware that not all IS-A relationships should be modeled with inheritance
- know how to explicitly disallow inheritance when writing a class or disallow overriding with the `final` keyword

The Responsibility When Using Inheritance

As you have seen in Unit 14, polymorphism is a powerful tool that allows a client to change the behavior of existing code written by the implementer, behind the abstraction barrier.

As Ben Parker (aka Uncle Ben) said, "With great power, comes great responsibility." The client must use overriding and inheritance carefully. Since they can affect how existing code behaves, they can easily break existing code and introduce bugs. Since the client may not have access to the existing code behind the abstraction barrier, it is often tricky to trace and debug. Furthermore, the implementer would not appreciate it if their code was working perfectly until one day, someone overriding a method causes their code to fail, even without the implementer changing anything in their code.

Ensuring this responsibility cannot be done by the compiler, unfortunately.
It thus becomes a developer's responsibility to ensure that any inheritance with method overriding does not introduce bugs to existing code. This brings us to the *Liskov Substitution Principle* (LSP), which says: "Let $\phi(x)$ be a property provable about objects x of type T. Then $\phi(y)$ should be true for objects y of type S where $S <: T$."

This is consistent with the definition of subtyping, $S <: T$, but spelled out more formally.

Let's consider the following example method, `Module::marksToGrade`, which takes in the marks of a student and returns the grade 'A', 'B', 'C', or 'F' as a `char`. How `Module::marksToGrade` is implemented is not important. Let's look at how it is used.

```
1  void displayGrade(Module m, double marks) {
2    char grade = m.marksToGrade(marks);
3    if (grade == 'A')) {
```

```
        System.out.println("well done");
    else if (grade == 'B') {
        System.out.println("good");
    else if (grade == 'C') {
        System.out.println("ok");
    } else {
        System.out.println("retake again");
    }
}
```

Now, suppose that one day, someone comes along and create a new class `CSCUModule` that inherits from `Module`, and overrides `marksToGrade` s that it now returns only 'S' and 'U'. Since `CSCUModule` is a subclass of `Module`, we can pass an instance to `displayGrade`:

```
displayGrade(new CSCUModule("GEQ1000", 100));
```

and suddenly `displayGrade` is displaying `retake again` even if the student is scoring 100 marks.

We are violating the LSP here. The object `m` has the following property: `m.marksToGrade` always returns something from the set { `'A'`, `'B'`, `'C'`, `'F'` }, that the method `displayGrade` depends on explicitly. The subclass `CSCUModule` violated that and makes `m.marksToGrade` returns `'S'` or `'U'`, sabotaging `displayGrade` and causing it to fail.

LSP cannot be enforced by the compiler[1]. The properties of an object have to be managed and agreed upon among programmers. A common way is to document these properties as part of the code documentation.

LSP Through the Lens of Testing

Another way to develop an intuition of the LSP is through the lens of testing. When we write a method, we may want to introduce test cases to check that our method is working correctly. These test cases are designed based on the *specification* of our method and not its implementation details[2]. That is, we test based on the expected inputs and resultant outputs.

Let's look at an example. We would like to model a restaurant booking system for a restaurant chain. Consider the following `Restaurant` class. Every restaurant in the chain opens at 12 pm and closes at 10 pm, and has a singular method `canMakeReservation` which allows us to check if the restaurant is available for reservations at a certain `time`. **The requirement given is that, the system must be able to process a reservation during its opening hours.**

```
public class Restaurant {
  public static final int OPENING_HOUR = 1200;
```

```
  public static final int CLOSING_HOUR = 2200;

  public boolean canMakeReservation(int time) {
    if (time <= CLOSING_HOUR && time >= OPENING_HOUR) {
      return true;
    }
    return false;
  }
}
```

The method `canMakeReservation` returns `true` when the argument passed in to `time` is between 12 pm and 10 pm. Let's think about how we would test this method. Two important edge cases to test is to check if the method returns true for the stated restaurant opening and closing hours.

```
Restaurant r = new Restaurant();
r.canMakeReservation(1200) == true; // Is true, therefore test passes
r.canMakeReservation(2200) == true; // Is true, therefore test passes
```

Note that these are simple `jshell` tests, in software engineering modules you will learn better ways to design and formalise these tests.

We can now rephrase our LSP in terms of testing. A *subclass* should not break the expectations set by the *superclass*. If a class `B` is substitutable for a parent class `A` then it should be able to pass all test cases of the parent class `A`. If it does not, then it is not substitutable and the LSP is violated.

Lets now consider two subclasses of `Restaurant`, `LunchRestaurant` and `DigitalReadyRestaurant`. Our `LunchRestaurant` does not take reservations during peak hours (12 to 2 pm).

```
public class LunchRestaurant extends Restaurant {
  private final int peakHourStart = 1200;
  private final int peakHourEnd = 1400;

  @Override
  public boolean canMakeReservation(int time) {
    if (time <= peakHourEnd && time >= peakHourStart) {
      return false;
    } else if (time <= CLOSING_HOUR && time >= OPENING_HOUR) {
      return true;
    }
    return false;
  }
}
```

As `LunchRestaurant` <: `Restaurant`, we can point our variable `r` to a new instance of `LunchRestaurant` and run the test cases of the parent class, as can be seen in the code below.

```
Restaurant r = new LunchRestaurant();
r.canMakeReservation(1200) == true; // Is false, therefore test fails
r.canMakeReservation(2200) == true; // Is true, therefore test passes
```

Whilst the second test passes, the first test does not since it falls within the peak lunch hour. Therefore `LunchRestaurant` is not substitutable for `Restaurant` and the LSP is violated. We have changed the expectation of the method in the child class.

Let's suppose the restaurant chain starts to roll out online reservation system for a subset of its restaurants. These restaurants can take reservations any time.
We create a subclass `DigitalReadyRestaurant`, as follows:

```
public class DigitalReadyRestaurant extends Restaurant {

  @Override
  public boolean canMakeReservation(int time) {
    return true;
  }
}
```

Similarly, as `DigitalReadyRestaurant` <: `Restaurant`, we can point our variable `r` to a new instance of `DigitalReadyRestaurant` and run the test cases of the parent class, as can be seen in the code below.

```
Restaurant r = new DigitalReadyRestaurant();
r.canMakeReservation(1200) == true; // Is true, therefore test passes
r.canMakeReservation(2200) == true; // Is true, therefore test passes
```

Both test cases pass. In fact, all test cases that pass for `Restaurant` would pass for `DigitalReadyRestaurant`. Therefore `DigitalReadyRestaurant` is substitutable for `Restaurant`. Anywhere we can use an object of type `Restaurant`, we can use `DigitalReadyRestaurant` without breaking any previously written code.

Preventing Inheritance and Method Overriding

Sometimes, it is useful for a developer to explicitly prevent a class to be inherited. Not allowing inheritance would make it much easier to argue for the correctness of programs, something that is important when it comes to writing secure programs. Both the two java classes you have seen, `java.lang.Math` and `java.lang.String`, cannot be inherited from. In Java, we use the keyword `final` when declaring a class to tell Java that we ban this class from being inherited.

```
final class Circle {
    :
```

```
3   }
```

Alternatively, we can allow inheritance but still prevent a specific method from being overridden, by declaring a method as `final`. Usually, we do this on methods that are critical for the correctness of the class.

For instance,

```
1   class Circle {
2       :
3     public final boolean contains(Point p) {
4       :
5     }
6   }
```

---

1. We can use `assert` to check some of the properties though. ↵
2. The test cases we are describing here are known as black-box tests and you will encounter these in later modules at NUS. We will not go into any further details in this module. ↵

Unit 17: Abstract Class

After this lecture, students should:

- be familiar with the concept of an abstract class
- know the use of the Java keyword `abstract` and the constraints that come with it
- understand the usefulness of defining and using an abstract class
- understand what makes a class concrete

High-Level Abstraction

Recall that the concept of abstraction involves hiding away unnecessary complexity and details so that programmers do not have to bogged down with the nitty-gritty.

When we code, we should, as much as possible, try to work with the higher-level abstraction, rather than the detailed version. Following this principle would allow us to write code that is general and extensible, by taking full advantage of inheritance and polymorphism.

Take the following example which you have seen,

```
1  // version 0.1 (with polymorphism)
2  boolean contains(Object[] array, Object obj) {
3    for (Object curr : array) {
4      if (curr.equals(obj)) {
5        return true;
6      }
7    }
8    return false;
9  }
```

The function above is very general. We do not assume and do not need to know, about the details of the items being stored or search. All we required is that the `equals` method compared if two objects are equal.

In contrast, someone whose mind focuses on finding a circle, might write something like this:

```
1  // version 0.3 (for Circle)
2  boolean contains(Circle[] array, Circle circle) {
3    for (Circle curr : array) {
```

```
      if (curr.equals(circle)) {
        return true;
      }
    }
    return false;
  }
```

which serves the purpose, but is not general enough. The only method used is `equals`, which `Circle` inherits/overrides from `Object` so that using `Circle` for this function is too constraining. We can reuse this for any other subclasses of Circle, but not other classes.

Abstracting Circles

Now, let's consider the following function, which finds the largest area among the circles in a given array:

```
// version 0.1
double findLargest(Circle[] array) {
  double maxArea = 0;
  for (Circle curr : array) {
    double area = curr.getArea();
    if (area > maxArea) {
      maxArea = area;
    }
  }
  return maxArea;
}
```

`findLargest` suffers from the same specificity as the version 0.3 of `contains`. It only works for `Circle` and its subclasses only. Can we make this more general? We cannot replace `Circle` with `Object`,

```
// version 0.2
double findLargest(Object[] array) {
  double maxArea = 0;
  for (Object curr : array) {
    double area = curr.getArea();
    if (area > maxArea) {
      maxArea = area;
    }
  }
  return maxArea;
}
```

since `getArea` is not defined for a generic object (e.g., what does `getArea` of a string mean?).

To allow us to apply `findLargest` to a more generic object, we have to create a new type -- something more specific than `Object` that supports `getArea()`, yet more general than `Circle`.

Shape

Let's create a new class called `Shape`, and redefine our `Circle` class as a subclass of `Shape`. We can now create other shapes, `Square`, `Rectangle`, `Triangle`, etc, and define the `getArea` method for each of them.

With the new `Shape` class, we can rewrite `findLargest` as:

```
// version 0.3
double findLargest(Shape[] array) {
  double maxArea = 0;
  for (Shape curr : array) {
    double area = curr.getArea();
    if (area > maxArea) {
      maxArea = area;
    }
  }
  return maxShape;
}
```

which now not only works for an array of `Square`, `Rectangle`, `Circle`, etc but also an array containing multiple shapes!

Let's actually write out our new `Shape` class:

```
class Shape {
    public double getArea() {
        // ?
    }
}
```

and rewrite our `Circle`:

```
// version 0.8
import java.lang.Math;

/**
 * A Circle object encapsulates a circle on a 2D plane.
 */
class Circle extends Shape {
  private Point c;   // the center
  private double r;  // the length of the radius

  /**
   * Create a circle centered on Point c with given radius r
  */
  public Circle(Point c, double r) {
    this.c = c;
    this.r = r;
  }

```

```
    /**
     * Return the area of the circle.
     */
    @Override
    public double getArea() {
      return Math.PI * this.r * this.r;
    }

    /**
     * Return true if the given point p is within the circle.
     */
    public boolean contains(Point p) {
      // TODO: Left as an exercise
      return false;
    }

    /**
     * Return the string representation of this circle.
     */
    @Override
    public String toString() {
        return "{ center: " + this.c + ", radius: " + this.r + " }";
    }

    /**
     * Return true the object is the same circle (i.e., same center, same
  radius).
     */
    @Override
    public boolean equals(Object obj) {
      if (obj instanceof Circle) {
        Circle circle = (Circle) obj;
        return (circle.c.equals(this.c) && circle.r == this.r);
      }
    }
  }
```

Notably, since our `Shape` is a highly abstract entity, it does not have any fields. One question that arises is, how are we going to write `Shape::getArea()`? We cannot compute the area of a shape unless we know what sort of shape it is.

One solution is make `Shape::getArea()` returns 0.

```
class Shape {
    public double getArea() {
        return 0;
    }
}
```

This is not ideal. It is easy for someone to inherit from `Shape`, but forget to override `getArea()`. If this happens, then the subclass will have an area of 0. Bugs ensue.

As we usually do in CS2030S, we want to exploit programming language constructs and the compiler to check and catch such errors for us.

Abstract Methods and Classes

This brings us to the concept of *abstract classes*. An abstract class in Java is a class that has been made into something so general that it cannot and should not be instantiated. Usually, this means that one or more of its instance methods cannot be implemented without further details.

The `Shape` class above makes a good abstract class since we do not have enough details to implement `Shape::getArea`.

To declare an abstract class in Java, we add the `abstract` keyword to the `class` declaration. To make a method abstract, we add the keyword `abstract` when we declare the method.

An `abstract` method cannot be implemented and therefore should not have any method body.

This is how we implement `Shape` as an abstract class.

```
abstract class Shape {
    abstract public double getArea();
}
```

An abstract class cannot be instantiated. Any attempt to do so, such as:

```
Shape s = new Shape();
```

would result in an error.

Note that our simple example of `Shape` only encapsulates one abstract method. An abstract class can contain multiple fields and multiple methods. Not all the methods have to be abstract. As long as one of them is abstract, the class becomes abstract.

To illustrate this, consider

```
abstract class Shape {
    private int numOfAxesOfSymmetry ;

    public boolean isSymmetric() {
        return numOfAxesOfSymmetry > 0;
    }

```

```
8       abstract public double getArea();
9   }
```

`Shape::isSymmetric` is a concrete method but the class is still abstract since `Shape::getArea()` is abstract.

Concrete Classes

We call a class that is not abstract as a *concrete class*. A concrete class cannot have any abstract method. Thus, any subclass of `Shape` must override `getArea()` to supply its own implementation.

Unit 18: Interface

After taking this unit, students should:

- understand interface as a type for modeling "can do" behavior
- understand the subtype-supertype relationship between a class and its interfaces

Modeling Behavior

We have seen how we can write our program using superclasses (including abstract ones) to make our code more general and flexible. In this unit, we will kick this up one more notch and try to write something even more general, through another abstraction.

Let's reexamine this method again:

```
1  // version 0.3
2  double findLargest(Shape[] array) {
3    double maxArea = 0;
4    for (Shape curr : array) {
5      double area = curr.getArea();
6      if (area > maxArea) {
7        maxArea = area;
8      }
9    }
10   return maxArea;
11 }
```

Note that all that is required for this method to work, is that the type of objects in `array` supports a `getArea` method. While `Shape` that we defined in the previous unit meets this requirement, it does not have to be. We could pass in an array of countries or an array of HDB flats. It is unnatural to model a `Country` or a `Flat` as a subclass of `Shape` (recall inheritance models the IS-A relationship).

To resolve this, we will look at an abstraction that models what can an entity do, possibly across different class hierarchies.

Interface

The abstraction to do this is called an *interface*. An interface is also a type and is declared with the keyword `interface`.

Since an interface models what an entity can do, the name usually ends with the -able suffix[1] .

Suppose we want to create a type that supports the `getArea()` method, be it a shape, a geographical region, or a real estate property. Let's call it `GetAreable`:

```
interface GetAreable {
  public abstract double getArea();
}
```

All methods declared in an interface are `public abstract` by default. We could also just write:

```
interface GetAreable {
  double getArea();
}
```

Now, for every class that we wish to be able to call `getArea()` on, we tell Java that the class `implements` that particular interface.

For instance,

```
abstract class Shape implements GetAreable {
  private int numOfAxesOfSymmetry ;

  public boolean isSymmetric() {
    return numOfAxesOfSymmetry > 0;
  }
}
```

The `Shape` class will now have a `public abstract double getArea()` thanks to it implementing the `GetAreable` interface.

We can have a concrete class implementing an interface too.

```
 class Flat extends RealEstate implements GetAreable {
     private int numOfRooms;
     private String block;
     private String street;
     private int floor;
     private int unit;

     @Override
     public double getArea() {
         :
     }
 }
```

For a class to implement an interface and be concrete, it has to override all abstract methods from the interface and provide an implementation to each, just like the example above. Otherwise, the class becomes abstract.

With the `GetAreable` interface, we can now make our function `findLargest` even more general.

```
// version 0.3
double findLargest(GetAreable[] array) {
  double maxArea = 0;
  for (GetAreable curr : array) {
    double area = curr.getArea();
    if (area > maxArea) {
      maxArea = area;
    }
  }
  return maxArea;
}
```

Note:

- A class can only extend from one superclass, but it can implement multiple interfaces.
- An interface can extend from one or more other interfaces, but an interface cannot extend from another class.

Interface as Supertype

If a class C implements an interface I, $C <: I$. This definition implies that a type can have multiple supertypes.

In the example above, `Flat` <: `GetAreable` and `Flat` <: `RealEstate`.

Casting using an Interface

Like any type in Java, it is also possible cast using an Interface. Lets consider an interface `I` and two classes `A` and `B`. Note that `A` does not implement `I`

```
interface I {
  :
}

class A {
  :
}

class B implements I {
```

```
    :
}
```

Now lets, consider the following code excerpt:

```
I i1 = new B(); // Compiles, widening type conversion
I i2 = (I) new A(); // Also compiles?
```

Note that even though `A` does not implement `I`, the Java compiler allows this code to compile. Constrast this with casting between classes which have no subtype relationship:

```
A a = (A) new B(); // Does not compile
B a = (B) new A(); // Does not compile
```

How do we explain this? Well, the Java compiler will not let us cast, when it is provable that it won't work, i.e. casting between two classes which have no subtype relationship. However, for interfaces, there is the *possibility* that a subclass *could* implement the interface and therefore Java allows it to compile. Consider one such potential subclass `AI`:

```
class AI extends A implements I{
  :
}
```

The lesson here is that when we are using typecasting, we are telling the compiler that *we know best*, and therefore it will not warn us or stop us from making bad decisions. It is important to always be sure when you use an explicit typecast.

Impure Interfaces

As we mentioned at the beginning of this module, it is common for software requirements, and their design, to continuously evolve. But once we define an interface, it is difficult to change.

Suppose that, after we define that `GetAreable` interface, other developers in the team starts to write classes that implement this interface. One fine day, we realize that we need to add more methods into the `getAreable`. Perhaps we need methods `getSqFt()` and `getMeter2()` in the interface. But, one cannot simply change the interface and add these abstract methods now. The other developers will have to change their classes to add the implementation of two methods, or else their code would not compile!

This is what happened to the Java language when they transitted from version 7 to version 8. The language needed to add a bunch of useful methods to standard interfaces provided

by the Java library, but doing so would break existing code in the 1990s that rely on these interfaces.

The solution that Java came up with is the allow an interface to provide a default implementation of methods that all implementation subclasses will inherit (unless they override). A method with default implementation is tagged with the `default` keyword. This leads to a less elegant situation where an `interface` has some abstract methods and some non-abstract default methods. In CS2030S, we refer to this as *impure interfaces* and it is a pain to explain since it breaks our clean distinction between a class and an interface. *We prefer not to talk about it* -- but it is there in Java 8 and up.

---

1. Although in recent Java releases, this is less common. ↵

Unit 19: Wrapper Class

After this unit, students should:

- be aware that Java provides wrapper classes around the primitive types
- be aware that Java will transparently and automatically box and unbox between primitive types and their corresponding wrapper classes

Writing General Code for Primitive Types

We have seen the following general code that takes in an array of `Object` objects, and searches if another object `obj` is in the given `array`.

```
1  // version 0.1 (with polymorphism)
2  boolean contains(Object[] array, Object obj) {
3    for (Object curr : array) {
4      if (curr.equals(obj)) {
5        return true;
6      }
7    }
8    return false;
9  }
```

Through polymorphism and overriding of the `equals` method, we can make sure that it is general enough to work on any reference type. But what about primitive types? Do we need to write a separate function for each primitive type, like this?

```
1  // version 0.4 (for int, a primitive type)
2  boolean contains(int[] array, int obj) {
3    for (int curr : array) {
4      if (curr == obj) {
5        return true;
6      }
7    }
8    return false;
9  }
```

Making Primitive Types Less Primitive

Java provides wrapper classes for each of its primitive types. A *wrapper class* is a class that encapsulates a *type*, rather than fields and methods. The wrapper class for `int` is called

`Integer`, for `double` is called `Double`, etc. A wrapper class can be used just like every other class in Java and behave just like every other class in Java. In particular, they are reference types and their instances can be created with `new`; instances are stored on the heap, etc.

For instance,

```
1  Integer i = new Integer(4);
2  int j = i.intValue();
```

The code snippet above shows how we can convert a primitive `int` value to a wrapper instance `i` of type `Integer`, and how the `intValue` method can retrieve the `int` value from an `Integer` instance.

With the wrapper type, we can reuse our `contains` method that takes in an `Object` array as a parameter to operate on an array of integers -- we just need to pass our integers into the method in an `Integer` array instead of an `int` array.

All primitive wrapper class objects are *immutable* -- once you create an object, it cannot be changed.

Auto-boxing and Unboxing

As conversion back-and-forth between a primitive type and its wrapper class is pretty common, Java provides a feature called auto-boxing/unboxing to perform type conversion between primitive type and its wrapper class.

For instance,

```
1  Integer i = 4;
2  int j = i;
```

The first statement is an example of auto-boxing, where the primitive value `int` of 4 is converted into an instance of `Integer`. The second statement converts an instance of `Integer` back to `int` (without affecting its value of 4).

Performance

Since the wrapper classes allow us to write flexible programs, why not use them all the time and forget about primitive types?

The answer: performance. Because using an object comes with the cost of allocating memory for the object and collecting garbage afterward, it is less efficient than primitive

types.

Consider the following two programs:

```
Double sum;
for (int i = 0; i < Integer.MAX_VALUE; i++)
{
    sum += i;
}
```

vs.

```
double sum;
for (int i = 0; i < Integer.MAX_VALUE; i++)
{
    sum += i;
}
```

The second one can be about 2 times faster. All primitive wrapper class objects are immutable -- once you create an object, it cannot be changed. Thus, every time the sum in the first example above is updated, a new `Double` object gets created. Due to autoboxing and unboxing, the cost of creating objects becomes hidden and is often forgotten.

The Java API in fact, provides multiple versions of the same method, one for all the reference types using `Object`, and one for each of the primitive types. This decision leads to multiple versions of the same code, but with the benefits of better performance. See the Arrays class for instance.

Unit 20: Run-Time Class Mismatch

After taking this unit, students should:

- Understand the need for narrowing type conversion and type casting when writing code that depends on higher-level abstraction
- Understand the possibility of encountering run-time errors if typecasting is not done properly.

We have seen in Unit 18 how we can write code that is reusable and general by making our code dependent on types at a higher-level of abstraction. Our main example is the following `findLargest` method, which takes in an array of objects that support the `getArea` method, and returns the largest area among these objects.

```
1  // version 0.3
2  double findLargest(GetAreable[] array) {
3    double maxArea = 0;
4    for (GetAreable curr : array) {
5      double area = curr.getArea();
6      if (area > maxArea) {
7        maxArea = area;
8      }
9    }
10   return maxArea;
11 }
```

The method served our purpose well, but it is NOT a very well-designed method. Just returning the value of the largest area is not as useful as returning the *object* with the largest area. Once the caller has a reference of the object, the caller can call `getArea` to find the value of the largest area.

Let's write our `findLargest` method to find which object has the largest area instead.

```
1  // version 0.4
2  GetAreable findLargest(GetAreable[] array) {
3    double maxArea = 0;
4    GetAreable maxObj = null;
5    for (GetAreable curr : array) {
6      double area = curr.getArea();
7      if (area > maxArea) {
8        maxArea = area;
9        maxObj = curr;
10     }
11   }
     return maxObj;
```

```
12   }
13
```

Let's see how `findLargest` can be used:

```
1  GetAreable[] circles = new GetAreable[] {
2    new Circle(new Point(1, 1), 2),
3    new Circle(new Point(0, 0), 5)
4  };
5
6  GetAreable ga = findLargest(circles);  // ok
7  Circle c1 = findLargest(circles); // error
8  Circle c2 = (Circle) findLargest(circles); // ok
```

The return type of `findLargest` (version 0.4) is now `GetAreable`. On Line 6 above, we assign the return object with a compile-time type of `GetAreable` to `ga`, which also has `GetAreable` as its compile-time type. Since the variable `ga` is of type `GetAreable`, however, it is not very useful. Recall that `GetAreable` is an interface with only one method `getArea`. We cannot use it as a circle.

On Line 7, we try to return the return object to a variable with compile-time type `Circle`. This line, however, causes a compile-time error. Since `Circle` <: `GetAreable`, this is a narrowing type conversion and thus is not allowed (See Unit 14). We will have to make an explicit cast of the result to `Circle` (on Line 8). Only with casting, our code can compile and we get a reference with a compile-time type of `Circle`.

Cast Carefully

Typecasting, as we did in Line 8 above, is basically is a way for programmers to ask the compiler to trust that the object returned by `findLargest` has a run-time type of `Circle` (or its subtype).

In the snippet above, we can be sure (even *prove*) that the returned object from `findLargest` must have a run-time type of `Circle` since the input variable `circles` contains only `Circle` objects.

The need to cast our returned object, however, leads to fragile code. Since the correctness of Line 8 depends on the run-time type, the compiler cannot help us. It is then up to the programmers to not make mistakes.

Consider the following two snippets, which will compile perfectly, but will lead to the program crashing at run-time.

```
1  GetAreable[] circles = new GetAreable[] {
2    new Circle(new Point(1, 1), 2),
```

```
    new Square(new Point(1, 1), 5)
};

Circle c2 = (Circle) findLargest(circles);
```

Or

```
GetAreable[] circles = new GetAreable[] {
    new Circle(new Point(1, 1), 2),
    new Circle(new Point(1, 1), 5)
};

Square sq = (Square) findLargest(circles);
```

We will see how to resolve this problem in later units.

Unit 21: Variance

After this unit, students should:

- understand the definition of the variance of types: covariant, contravariant, and invariant.
- be aware that the Java array is covariant and how it could lead to run-time errors that cannot be caught during compile time.

Both the methods `findLargest` and `contains` takes in an array of reference types as parameters:

```
1  // version 0.4
2  GetAreable findLargest(GetAreable[] array) {
3    double maxArea = 0;
4    GetAreable maxObj = null;
5    for (GetAreable curr : array) {
6      double area = curr.getArea();
7      if (area > maxArea) {
8        maxArea = area;
9        maxObj = curr;
10     }
11   }
12   return maxObj;
13 }
14
15 // version 0.1 (with polymorphism)
16 boolean contains(Object[] array, Object obj) {
17   for (Object curr : array) {
18     if (curr.equals(obj)) {
19       return true;
20     }
21   }
22   return false;
23 }
```

What are some possible arrays that we can pass into these methods? Let's try this:

```
1  Object[] objArray = new Object[] { new Integer(1), new Integer(2) };
2  Integer[] intArray = new Integer[] { new Integer(1), new Integer(2) };
3
4  contains(objArray, new Integer(1)); // ok
5  contains(intArray, new Integer(1)); // ok
```

Line 4 is not surprising since the type for `objArray` matches that of parameter `array`. Line 5, however, shows that it is possible to assign an instance with run-time type

`Integer[]` to a variable with compile-time type `Object[]`.

Variance of Types

So far, we have established the subtype relationship between classes and interfaces based on inheritance and implementation. The subtype relationship between *complex types* such as arrays, however, is not so trivial. Let's look at some definitions.

The *variance of types* refers to how the subtype relationship between complex types relates to the subtype relationship between components.

Let $C(S)$ corresponds to some complex type based on type S. An array of type S is an example of a complex type.

We say a complex type is:

- *covariant* if $S <: T$ implies $C(S) <: C(T)$
- *contravariant* if $S <: T$ implies $C(T) <: C(S)$
- *invariant* if it is neither covariant nor contravariant.

Java Array is Covariant

Arrays of reference types are covariant in Java[1]. This means that, if $S <: T$, then $S[] <: T[]$.

For example, because `Integer` <: `Object`, we have `Integer[]` <: `Object[]` and we can do the following:

```
Integer[] intArray;
Object[] objArray;
objArray = intArray; // ok
```

By making array covariant, however, Java opens up the possibility of run-time errors, even without typecasting!

Consider the following code:

```
Integer[] intArray = new Integer[2] {
  new Integer(10), new Integer(20)
};
Object[] objArray;
objArray = intArray;
objArray[0] = "Hello!"; // <- compiles!
```

On Line 5 above, we set `objArray` (with a compile-time type of `Object[]`) to refer to an object with a run-time type of `Integer[]`. This is allowed since the array is covariant.

On Line 6, we try to put a `String` object into the `Object` array. Since `String` <: `Object`, the compiler allows this. The compiler does not realize that at run-time, the `Object` array will refer to an array of `Integer`.

So we now have a perfectly compilable code, that will crash on us when it executes Line 6 -- only then would Java realize that we are trying to stuff a string into an array of integers!

This is an example of a type system rule that is unsafe. Since the array type is an essential part of the Java language, this rule cannot be changed without ruining existing code. We will see later how Java avoids this pitfall for other complex types (such as a list).

---

1. Arrays of primitive types are invariant. ↵

Unit 22: Exceptions

After this unit, students should:

- understand about handling java exceptions and how to use the `try` - `catch` - `finally` blocks
- understand the hierarchy of exception classes and the difference between checked and unchecked exceptions
- be able to create their own exceptions
- understand the control flow of exceptions
- be aware of good practices for exception handling

One of the nuances of programming is having to write code to deal with exceptions and errors. Consider writing a method that reads in a single integer value from a file. Here are some things that could go wrong:

- The file to read from may not exist
- The file to read from exists, but you may not have permission to read it
- You can open the file for reading, but it might contain non-numeric text where you expect numerical values
- The file might contain fewer values than expected
- The file might become unreadable as you are reading through it (e.g., someone unplugs the USB drive)

In C, we usually have to write code like this:

```
fd = fopen(filename,"r");
if (fd == NULL) {
  fprintf(stderr, "Unable to open file. ");
  if (errno == ENFILE) {
    fprintf(stderr, "Too many opened files.  Unable to open another\n");
  } else if (errno == ENOENT) {
    fprintf(stderr, "No such file %s\n", filename);
  } else if (errno == EACCES) {
    fprintf(stderr, "No read permission to %s\n", filename);
  }
  return -1;
}
scanned = fscanf(fd, "%d", &value);
if (scanned == 0) {
  fprintf(stderr, "Unable to scan for an integer\n");
```

```
16    fclose(fd);
17    return -2;
18  }
19  if (scanned == EOF) {
20    fprintf(stderr, "No input found.\n");
21    fclose(fd);
22    return -3;
23  }
```

Out of the lines above, only TWO lines correspond to the actual task of opening and reading in a file, the others are for exception checking/handling. The actual tasks are interspersed between exception checking code, which makes reading and understanding the logic of the code difficult.

The examples above also have to return different values to the calling method, because the calling method may have to do something to handle the errors. Note that the POSIX API has a global variable `errno` that signifies the detailed error. First, we have to check for different `errno` values and react accordingly (we can use `perror`, but that has its limits). Second, `errno` is global, and using a global variable is a bad practice. In fact, the code above might not work because `fprintf` in Line 3 might have changed `errno`.

Finally, there is the issue of having to repeatedly clean up after an error -- here we `fclose` the file if there is an error reading, twice. It is easy to forget to do so if we have to do this in multiple places. Furthermore, if we need to perform a more complex clean up, then we would end up with lots of repeated code.

Many modern programming languages support exceptions as a programming construct. In Java, this is done with `try`, `catch`, `finally` keywords, and a hierarchy of `Exception` classes. The `try` / `catch` / `finally` keywords group statements that check/handle errors together making code easier to read. The Java equivalent to the above is:

```
1   try {
2     reader = new FileReader(filename);
3     scanner = new Scanner(reader);
4     value = scanner.nextInt();
5   }
6   catch (FileNotFoundException e) {
7       System.err.println("Unable to open " + filename + " " + e);
8   }
9   catch (InputMismatchException e) {
10      System.err.println("Unable to scan for an integer");
11  }
12  catch (NoSuchElementException e) {
13      System.err.println("No input found");
14  }
15  finally {
16    if (scanner != null)
17      scanner.close();
18  }
```

Let's look at the example more carefully. The general syntax for `try` - `catch` - `finally` is the following:

```
try {
    // do something
} catch (an exception parameter) {
    // handle exception
} finally {
    // clean up code
    // regardless of there is an exception or not
}
```

In the example above, we have the `try` block:

```
try {
  reader = new FileReader(filename);
  scanner = new Scanner(reader);
  value = scanner.nextInt();
}
  :
```

which opens the file and reads an integer from it. Thus the main task for the code is put together in one place, making it easier to read and understand (and thus less bug-prone).

```
  :
catch (FileNotFoundException e) {
    System.err.println("Unable to open " + filename + " " + e);
} catch (InputMismatchException e) {
    System.err.println("Unable to scan for an integer");
} catch (NoSuchElementException e) {
    System.err.println("No input found");
}
```

The error handling comes under the `catch` clauses, each handling a different type of exception. In Java, exceptions are instances that are a subtype of the `Exception` class. Information about an exception is encapsulated in an exception instance and is "passed" into the `catch` block. In the example above, `e` is the variable containing an exception instance.

With the exception, we no longer rely on a special return value from a function nor a global variable to indicate exceptions.

```
  :
finally {
  if (scanner != null)
    scanner.close();
}
```

Finally, we have the optional `finally` clause for house-keeping tasks. Here, we close the `scanner` if it is opened.

In cases where the code to handle the exceptions is the same, you can avoid repetition by combining multiple exceptions into one catch statement:

```
catch (FileNotFoundException | InputMismatchException |
NoSuchElementException e) {
    System.err.println(e);
}
```

Throwing Exceptions

The `try` - `catch` - `finally` blocks above show you how to *handle* exceptions. Let's see how we can throw an exception. Let's revisit our `Circle` class. A circle cannot have a negative radius. Let's say that we wish our constructor to throw an `IllegalArgumentException` when a negative radius is passed in.

We need to do two things. First, we need to declare that the construct is throwing an exception, with the `throws` keyword. Second, we have to create a new `IllegalArgumentException` object and throw it to the caller with the `throw` keywords.

```
  public Circle(Point c, double r) throws IllegalArgumentException {
    if (r < 0) {
      throw new IllegalArgumentException("radius cannot be negative.");
    }
    this.c = c;
    this.r = r;
  }
}
```

Note that executing the `throw` statement causes the method to immediately return. In the example above, the initialization of the center `c` and radius `r` does not happen.

The caller then can catch and handle this exception:

```
try {
    c = new Circle(point, radius);
} catch (IllegalArgumentException e) {
    System.err.println("Illegal arguement:" + e.getMessage());
}
```

Checked vs Unchecked Exceptions

Java distinguishes between two types of exceptions: checked and unchecked.

An unchecked exception is an exception caused by a programmer's errors. They should not happen if perfect code is written. `IllegalArgumentException`, `NullPointerException`, `ClassCastException` are examples of unchecked exceptions. Generally, unchecked exceptions are not explicitly caught or thrown. They indicate that something is wrong with the program and cause run-time errors.

A checked exception is an exception that a programmer has no control over. Even if the code written is perfect, such an exception might still happen. The programmer should thus actively anticipate the exception and handle them. For instance, when we open a file, we should anticipate that in some cases, the file cannot be opened. `FileNotFoundException` and `InputMismatchException` are two examples of is an example of a checked exception. A checked exception must be either handled, or else the program will not compile.

In Java, unchecked exceptions are subclasses of the class `RuntimeException`.

Passing the Buck

The caller of the method that generates (i.e., `new` and `throws`) an exception need not catch the exception. The caller can pass the exception to its caller, and so on if the programmer deems that it is not the right place to handle it.

An unchecked exception, if not caught, will propagate automatically down the stack until either, it is caught or if it is not caught at all, resulting in an error message displayed to the user.

For instance, the following toy program would result in `IllegalArgumentException` being thrown out of `main` and displayed to the user.

```
1  class Toy {
2    static void createCircles() {
3      int radius = 10;
4      for (int i = 0; i <= 11; i++) {
5          new Circle(new Point(1, 1), radius--);
6      }
7    }
8    public static void main(String[] args) {
9      createCircles();
10   }
11 }
```

A checked exception, on the other hand, must be handled. Consider the following example:

```
// version 0.1 (won't compile)
class Toy {
  static FileReader openFile(String filename) {
    return new FileReader(filename);
  }
  public static void main(String[] args) {
    openFile();
  }
}
```

This program won't compile because the checked exception `FileNotFoundException` is not handled. As the example we have seen, we could handle it in `openFile`. In this case, `openFile` does not throw any exception.

```
// version 0.2 (handle where exception occur)
class Toy {
  static FileReader openFile(String filename) {
    try {
      return new FileReader(filename);
    } catch (FileNotFoundException e) {
      System.err.println("Unable to open " + filename + " " + e);
    }
  }
  public static void main(String[] args) {
    openFile();
  }
}
```

Alternatively, `openFile` can pass the buck to the caller instead of catching it.

```
// version 0.3 (passing exception to caller)
class Toy {
  static FileReader openFile(String filename) throws
FileNotFoundException {
    return new FileReader(filename);
  }
  public static void main(String[] args) {
    try {
      openFile();
    } catch (FileNotFoundException e) {
      // warn user and pop up dialog box to select another file.
    }
  }
}
```

Sometimes the caller is a better place to handle the exception. Where an exception should be handled is a design decision. We will see some considerations for this later in this unit.

What should not happen is the following:

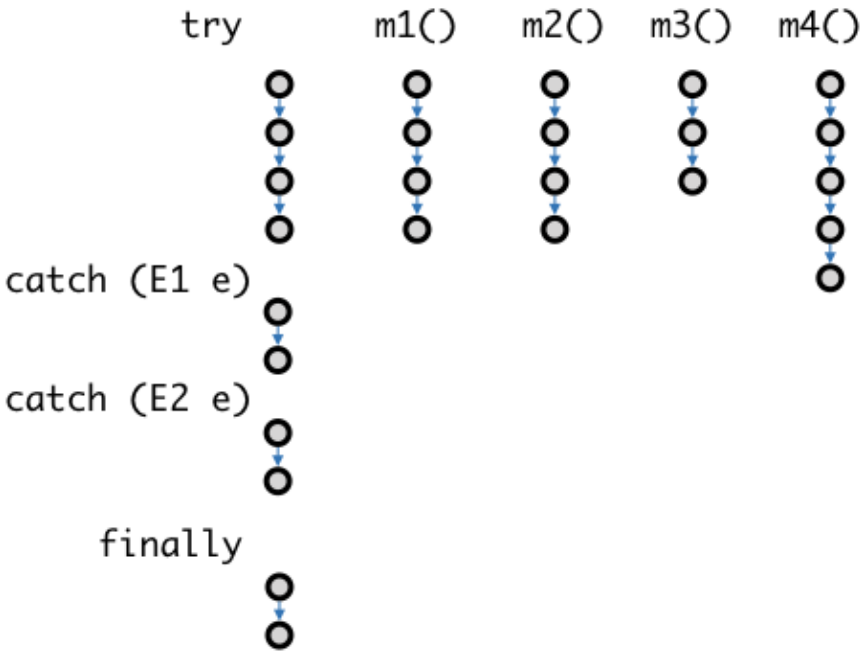
```
// version 0.4 (pass exception to user)
class Toy {
  static FileReader openFile(String filename) throws FileNotFoundException
{
    return new FileReader(filename);
  }
  public static void main(String[] args) throws FileNotFoundException {
    openFile();
  }
}
```

In the code above, every method passes the buck around. No one takes the responsibility to handle it and the user ends up with the exception. The ugly internals of the program (such as the call stack) is then revealed to the user.

A *good program always handle checked exception gracefully* and hide the details from the users.

Control Flow of Exceptions

Here is a more detailed description of the control flow of exceptions. Consider we have a `try` - `catch` - `finally` block that catches two exceptions `E1` and `E2`. Inside the try block, we call a method `m1()`; `m1()` calls `m2()`; `m2()` calls `m3()`, and `m3()` calls `m4()`.




```
try {
    m1();
} catch (E1 e) {
    :
} catch (E2 e) {
    :
} finally {
    :
}
```

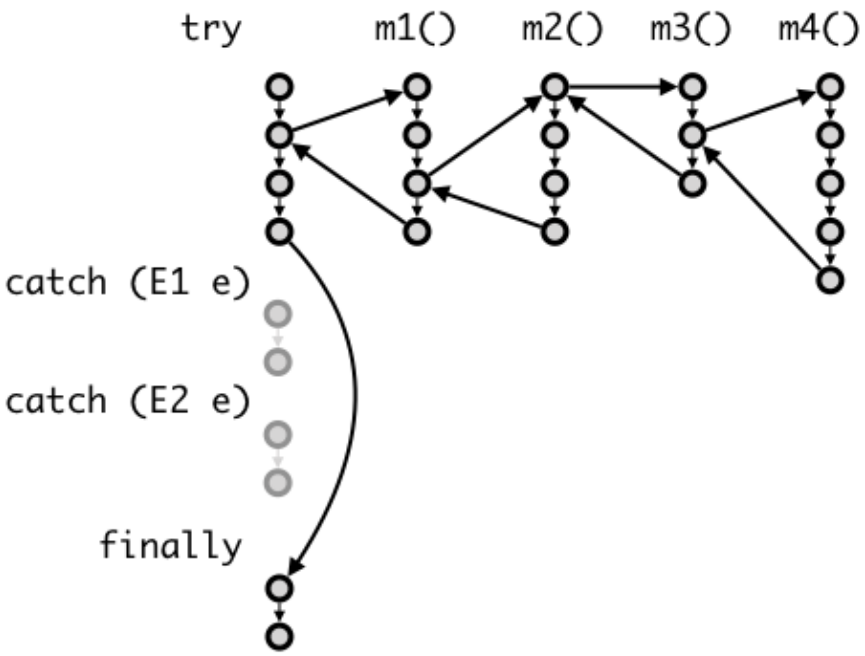
```
void m1() {
    :
    m2();
    :
}

void m2() {
    :
    m3();
    :
}

void m3() {
    :
    m4();
```

```
16      :
17  }
18
19  void m4() {
20      :
21      throw new E2();
22      :
23  }
```

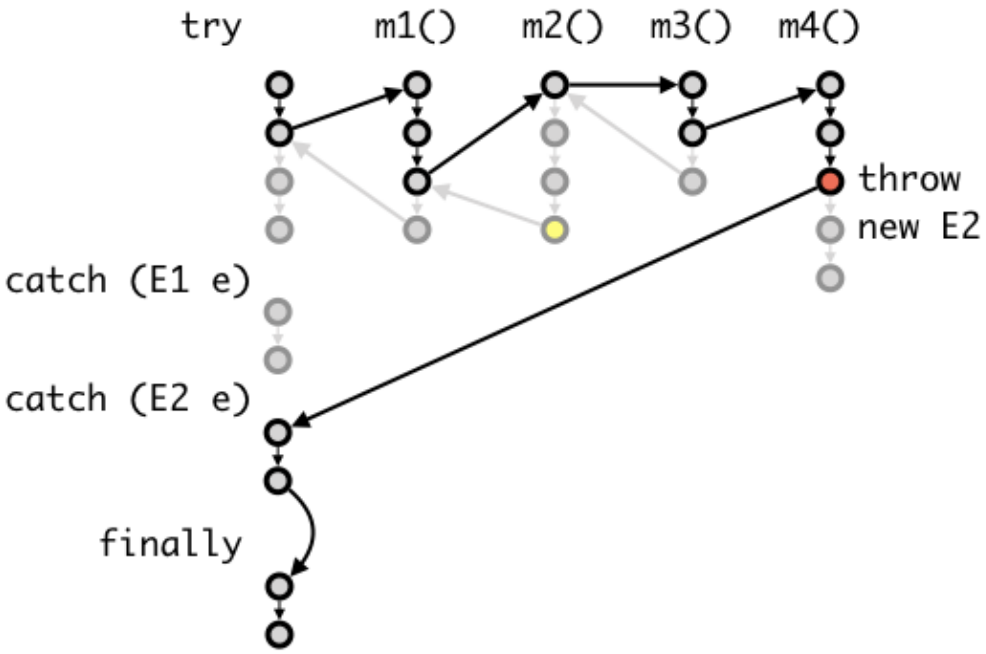
In a normal (no exception) situation, the control flow looks like this:



The statements in the try block are executed, followed by the statements in the `finally` block.

Now, let's suppose something went wrong deep inside the nested call, in `m4()`. One of the statement executes `throw new E2();`, which causes the execution in `m4()` to stop. JVM now looks for the block of code that catches `E2`, going down the call stack, until it can find a place where the exception is handled. In this example, we suppose that none of `m1()`-`m4()` handles (i.e., `catch`) the exception. Thus, JVM then jumps to the code that handles `E2`. Finally, JVM executes the `finally` block.

Note that the `finally` block is always executed even when return or throw is called in a catch block.



Creating Our Own Exceptions

If you find that none of the exceptions provided by Java meet your needs, you can create your own exceptions, by simply inheriting from one of the existing ones. But, you should only do so if there is a good reason, for instance, to provide additional useful information to the exception handler.

Here is an example:

```
class IllegalCircleException extends IllegalArgumentException {
  Point center;
  IllegalCircleException(String message) {
    super(message);
  }
  IllegalCircleException(Point c, String message) {
    super(message);
    this.center = c;
  }
```

```
  @Override
    public String toString() {
      return "The circle centered at " + this.center + " cannot be
created:" + getMessage();
    }
}
```

Overriding Method that Throws Exceptions

When you override a method that throws a checked exception, the overriding method must throw only the same, or a more specific checked exception, than the overridden method. This rule follows the Liskov Substitution Principle. The caller of the overridden method cannot expect any new checked exception beyond what has already been "promised" in the method specification.

Good Practices for Exception Handling

Catch Exceptions to Clean Up

While it is convenient to just pass the buck and let the calling method deals with exceptions ("Hey! Not my problem!"), it is not always responsible to do so. Consider the example earlier, where `m1()`, `m2()`, and `m3()` do not handle exception `E2`. Let's say that `E2` is a checked exception, and it is possible to react to this and let the program continues properly. Also, suppose that `m2()` allocated some system resources (e.g., temporary files, network connections) at the beginning of the method, and deallocated the resources at the end of the method. By not handling the exception, the code that deallocates these resources does not get called when an exception occurs. It is better for `m2()` to catch the exception, handle the resource deallocation in a `finally` block. If there is a need for the calling methods to be aware of the exception, `m2()` can always re-throw the exception:

```
public void m2() throws E2 {
  try {
    // setup resources
    m3();
  }
  catch (E2 e) {
    throw e;
  }
  finally {
    // clean up resources
  }
}
```

Do not catch-them-all!

Sometimes, you just want to focus on the main logic of the program and get it working instead of dealing with the exceptions. Since Java uses checked exceptions, it forces you to handle the exceptions, or else your code will not compile. One way to quickly get around this is to write:

```
try {
  // your code
}
catch (Exception e) {
  // do nothing
}
```

to stop the compiler from complaining. **DO NOT DO THIS.** Since `Exception` is the superclass of all exceptions, every exception that is thrown, checked or unchecked, is now silently ignored! You will not be able to figure out if something is wrong with your program. This practice is such a bad practice that there is a name for it -- this is called *Pokemon Exception Handling*.

Overreacting

Do not exit a program just because of an exception. This would prevent the calling function from cleaning up their resources. Worse, do not exit a program silently.

```
try {
  // your code
}
catch (Exception e) {
  System.exit(0);
}
```

Do Not Break Abstraction Barrier

Sometimes, letting the calling method handle the exception causes the implementation details to be leaked, and make it harder to change the implementation later.

For instance, suppose we design a class `ClassRoster` with a method `getStudents()`, which reads the list of students from a text file.

```
class ClassRoster {
   :
  public Students[] getStudents() throws FileNotFoundException {
    :
  }
}
```

Here, the fact that a `FileNotFoundException` is thrown leaks the information that the information is read from a file.

Suppose that, later, we change the implementation to reading the list from an SQL database. We may have to change the exception thrown to something else:

```
class ClassRoster {
    :
  public Students[] getStudents() throws SQLException {
    :
  }
}
```

The caller will have to change their exception handling code accordingly.

We should, as much as possible, handle the implementation-specific exceptions within the abstraction barrier.

Do NOT Use Exception As a Control Flow Mechanism

This is probably the most commonly seen mistakes among new programmers. Exceptions are meant to handle unexpected errors, not to handle the logic of your program. Consider the following snippet:

```
if (obj != null) {
  obj.doSomething();
} else {
  doTheOtherThing();
}
```

We use an `if` condition to handle the logic. Some programmers wrote this:

```
    try {
      obj.doSomething();
    } catch (NullPointerException e) {
      doTheOtherThing();
    }
```

Not only is this less efficient, but it also might not be correct, since a `NullPointerException` might be triggered by something else other than `obj` being null.

The `Error` class

Java has another class called `Error` for situations where the program should terminate as generally there is no way to recover from the error. For instance, when the heap is full

(`OutOfMemoryError`) or the stack is full (`StackOverflowError`). Typically we don't need to create or handle such errors.

Unit 23: Generics

After taking this unit, students should:

- know how to define and instantiate a generic type and a generic method
- be familiar with the term parameterized types, type arguments, type parameters
- appreciate how generics can reduce duplication of code and improve type safety

The `Pair` class

Sometimes it is useful to have a lightweight class to bundle a pair of variables together. One could, for instance, write a method that returns two values. The example defines a class `IntPair` that bundles two `int` variables together. This is a utility class with no semantics nor methods associated with it and so, we did not attempt to hide the implementation details.

```
1  class IntPair {
2    private int first;
3    private int second;
4
5    public IntPair(int first, int second) {
6        this.first = first;
7        this.second = second;
8    }
9
10   int getFirst() {
11       return this.first;
12   }
13
14   int getSecond() {
15       return this.second;
16   }
17 }
```

This class can be used, for instance, in a function that returns two `int` values.

```
1  IntPair findMinMax(int[] array) {
2      int min = Integer.MAX_VALUE;  // stores the min
3      int max = Integer.MIN.VALUE; // stores the max
4      for (int i : array) {
5          if (i < min)  {
6              min = i;
7          }
8          if (i > max) {
```

```
            max = i;
        }
    }
    return new IntPair(min, max);
}
```

We could similarly define a pair class for two doubles (`DoublePair`), two booleans (`BooleanPair`), etc. In other situations, it is useful to define a pair class that bundles two variables of two different types, say, a `Customer` and a `ServiceCounter`; a `String` and an `int`; etc.

We should not, however, create one class for each possible combination of types. A better idea is to define a class that stores two `Object` references:

```
class Pair {
  private Object first;
  private Object second;

  public Pair(Object first, Object second) {
      this.first = first;
      this.second = second;
  }

  Object getFirst() {
      return this.first;
  }

  Object getSecond() {
      return this.second;
  }
}
```

At the cost of using a wrapper class in place of primitive types, we get a single class that can be used to store any type of values.

You might recall that we used a similar approach for our `contains` method to implement a general *method* that works for any type of object. Here, we are using this approach for a general *class* that encapsulates any type of object.

Unfortunately, the issues we faced with narrowing type conversion and potential run-time errors apply to the `Pair` class as well. Suppose that a function returns a `Pair` containing a `String` and an `Integer`, and we accidentally treat this as an `Integer` and a `String` instead, the compiler will not be able to detect the type mismatch and stop the program from crashing during run-time.

```
Pair foo() {
  return new Pair("hello", 4);
}

```

```
Pair p = foo();
Integer i = (Integer) p.getFirst(); // run-time ClassCastException
```

To reduce the risk of human error, what we need is a way to specify the following: suppose the type of `first` is S and type of `second` is T, then we want the return type of `getFirst` to be S and of `getSecond` to be T.

Generic Types

In Java and many other programming languages, the mechanism to do this is called generics or templates. Java allows us to define a *generic type* that takes other types as *type parameters*, just like how we can write methods that take in variables as parameters.

Declaring a Generic Type

Let's see how we can do this for `Pair`:

```
class Pair<S,T> {
  private S first;
  private T second;

  public Pair(S first, T second) {
      this.first = first;
      this.second = second;
  }

  S getFirst() {
      return this.first;
  }

  T getSecond() {
      return this.second;
  }
}
```

We declare a generic type by specifying its type parameters between `<` and `>` when we declare the type. By convention, we use a single capital letter to name each type parameter. These type parameters are scoped within the definition of the type. In the example above, we have a generic class `Pair<S,T>` (read "pair of S and T") with `S` and `T` as type parameters. We use `S` and `T` as the type of the fields `first` and `second`. We ensure that `getFirst()` returns type `S` and `getSecond()` returns type `T`, so that the compiler will give an error if we mix up the types.

Note that the constructor is still declared as `Pair` (without the type parameters).

Using/Instanting a Generic Type

To use a generic type, we have to pass in *type arguments*, which itself can be a non-generic type, a generic type, or another type parameter that has been declared. Once a generic type is instantiated, it is called a *parameterized type*.

To avoid potential human errors leading to `ClassCastException` in the example above, we can use the generic version of `Pair` as follows, taking in two non-generic types:

```
Pair<String,Integer> foo() {
  return new Pair<String,Integer>("hello", 4);
}

Pair<String,Integer> p = foo();
Integer i = (Integer) p.getFirst(); // compile-time error
```

With the parameterized type `Pair<String,Integer>`, the return type of `getFirst` is bound to `String`, and the compiler now have enough type information to check and give us an error since we try to cast a `String` to an `Integer`.

Note that we use `Integer` instead of `int`, since *only reference types* can be used as type arguments.

Just like you can pass a parameter of a method to another method, we can pass the type parameter of a generic type to another:

```
class DictEntry<T> extends Pair<String,T> {
      :
}
```

We define a generic class called `DictEntry<T>` with a single type parameter `T` that extends from `Pair<String,T>`, where `String` is the first type argument (in place of `S`), while the type parameter `T` from `DictEntry<T>` is passed as the type argument for `T` of `Pair<String,T>`.

Generic Methods

Methods can be parameterized with a type parameter as well. Consider the `contains` method, which we now put within a class for clarity.

```
class A {
    // version 0.1 (with polymorphism)
    public static boolean contains(Object[] array, Object obj) {
      for (Object curr : array) {
        if (curr.equals(obj)) {
          return true;
        }
      }
      return false;
```

```
    }
}
```

While using this method does not involve narrowing type conversion and type casting, it is a little to general -- it allows us to call `contains` in a nonsensical way, like this:

```
String[] strArray = new String[] { "hello", "world" };
A.contains(strArray, 123);
```

Searching for an integer within an array of strings is a futile attempt! Let's constrain the type of the object to search for to be the same as the type of the array. We can make this type the parameter to this method:

```
class A {
    // version 0.4 (with generics)
    public static <T> boolean contains(T[] array, T obj) {
      for (T curr : array) {
        if (curr.equals(obj)) {
          return true;
        }
      }
      return false;
    }
}
```

The above shows an example of a *generic method*. The type parameter `T` is declared within `<` and `>` and is added before the return type of the method. This parameter `T` is then scoped within the whole method.

To call a generic method, we need to pass in the type argument placed before the name of the method[1]. For instance,

```
String[] strArray = new String[] { "hello", "world" };
A.<String>contains(strArray, 123); // type mismatch error
```

The code above won't compile since the compiler expects the second argument to also be a `String`.

Bounded Type Parameters

Let's now try to apply our newly acquired trick to fix the issue with `findLargest`. Recall that we have the following `findLargest` method (which we now put into an ad hoc class just for clarity), which requires us to perform a narrowing type conversion to cast from `GetAreable` and possibly leading to a run-time error.

```
class A {
    // version 0.4
    public static GetAreable findLargest(GetAreable[] array) {
      double maxArea = 0;
      GetAreable maxObj = null;
      for (GetAreable curr : array) {
        double area = curr.getArea();
        if (area > maxArea) {
          maxArea = area;
          maxObj = curr;
        }
      }
      return maxObj;
    }
}
```

Let's try to make this method generic, by forcing the return type to be the same as the type of the elements in the input array,

```
class A {
    // version 0.4
    public static <T> T findLargest(T[] array) {
        double maxArea = 0;
        T maxObj = null;
        for (T curr : array) {
            double area = curr.getArea();
            if (area > maxArea) {
                maxArea = area;
                maxObj = curr;
            }
        }
        return maxObj;
    }
}
```

The code above won't compile, since the compiler cannot be sure that it can find the method `getArea()` in type `T`. In contrast, when we run `contains`, we had no issue since we are invoking the method `equals`, which exists in any reference type in Java.

Since we intend to use `findLargest` only in classes that implement the `GetAreable` interface and supports the `getArea()` method, we can put a constraint on `T`. We can say that `T` must be a subtype of `GetAreable` when we specify the type parameter:

```
class A {
    // version 0.5
    public static <T extends GetAreable> T findLargest(T[] array) {
        double maxArea = 0;
        T maxObj = null;
        for (T curr : array) {
            double area = curr.getArea();
            if (area > maxArea) {
```

```
                maxArea = area;
                maxObj = curr;
            }
        }
        return maxObj;
    }
}
```

We use the keyword `extends` here to indicate that `T` must be a subtype of `GetAreable`. It is unfortunate that Java decides to use the term `extends` for any type of subtyping when declaring a bounded type parameter, even if the supertype (such as `GetAreable`) is an interface.

We can use bounded type parameters for declaring generic classes as well. For instance, Java has a generic interface `Comparable<T>`, which dictates the implementation of the following `int compareTo(T t)` for any concrete class that implements the interface. Any class that implements the `Comparable<T>` interface can be compared with an instance of type `T` to establish an ordering. Such ordering can be useful for sorting objects, for instance.

Suppose we want to compare two `Pair` instances, by comparing the first element in the pair, we could do the following:

```
class Pair<S extends Comparable<S>,T> implements Comparable<Pair<S,T>> {
  private S first;
  private T second;

  public Pair(S first, T second) {
      this.first = first;
      this.second = second;
  }

  S getFirst() {
      return this.first;
  }

  T getSecond() {
      return this.second;
  }

  @Override
  public int compareTo(Pair<S,T> s1) {
      return this.first.compareTo(s1.first);
  }

  @Override
  public String toString() {
    return this.first + " " + this.second;
  }
}
```

Let's look at what it means:

- We declared `Pair` to be a generic type of two type parameters: the first one `S` is bounded and must be a subtype of `Comparable<S>`. This bound is self-referential, but it is intuitive -- we say that `S` must be comparable to itself, which is common in many use cases.
- Since we want to compare two `Pair` instances, we make `Pair` implements the `Comparable` interface too, passing in `Pair<S,T>` as the type argument to `Comparable`.

Let's see this in action with `Arrays::sort` method, which sorts an array based on the ordering defined by `compareTo`.

```
1   Object[] array = new Object[] {
2     new Pair<String,Integer>("Alice", 1),
3     new Pair<String,Integer>("Carol", 2),
4     new Pair<String,Integer>("Bob", 3),
5     new Pair<String,Integer>("Dave", 4),
6   };
7
8   java.util.Arrays.sort(array);
9
10  for (Object o : array) {
11    System.out.println(o);
12  }
```

You will see the pairs are sorted by the first element.

---

1. Java actually can infer the type using the *type inference* mechanism and allows us to skip the type argument, but for clarity, we insist on specifying the type explicitly until students get used to the generic types and reasoning about types. ↵

Unit 24: Type Erasure

After taking this unit, students are expected to:

- understand that generics are implemented with type erasure in Java
- understand that type information is not fully available during run-time when generics are used, and problems that this could cause
- be aware that arrays and generics don't mix well in Java
- know the terms reifiable type and heap pollution.

Implementing Generics

There are several ways one could implement generics in a programming language.

For instance, in C#, every instantiation of a generic type causes new code to be generated for that instantiated type. For instance, instantiating `Pair<S,T>` into `Pair<String,Integer>` causes a new type to be generated during run-time. In C++ and in Rust, instantiating `Pair<String,Integer>` causes new code to be generated during compile-time. This approach is sometimes called *code specialization*.

Java takes a *code sharing* approach, instead of creating a new type for every instantiation, it chooses to *erase* the type parameters and type arguments during compilation (after type checking, of course). Thus, there is only one representation of the generic type in the generated code, representing all the instantiated generic types, regardless of the type arguments.

Part of the reason to do this is for compatibility with the older version of Java. Java introduces generics only from version 5 onwards. Prior to version 5, one has to use `Object` to implement classes that are general enough to works on multiple types, similar to what we did with `Pair` here:

```
1  class Pair {
2    private Object first;
3    private Object second;
4
5    public Pair(Object first, Object second) {
6        this.first = first;
7        this.second = second;
8    }
9
```

```
    Object getFirst() {
        return this.first;
    }

    Object getSecond() {
        return this.second;
    }
}
```

The Java type erasure process transforms:

```
class Pair<S,T> {
    private S first;
    private T second;

    public Pair(S first, T second) {
        this.first = first;
        this.second = second;
    }

    S getFirst() {
        return this.first;
    }

    T getSecond() {
        return this.second;
    }
}
```

to the version above. Note that each type parameter `S` and `T` are replaced with `Object`. If the type parameter is bounded, it is replaced by the bounds instead (e.g., If `T extends GetAreable`, then `T` is replaced with `GetAreable`).

Where a generic type is instantiated and used, the code

```
Integer i = new Pair<String,Integer>("hello", 4).getSecond();
```

is transformed into

```
Integer i = (Integer) new Pair("hello", 4).getSecond();
```

The generated code is similar to what we would write earlier, but this is generated by the compiler after type checking, it ensures that the casting will not lead to `ClassCastException` during run-time.

Type erasures have several important implications. We will explore some of them below, and a few others during recitation.

Generics and Arrays Can't Mix

Let's consider the hypothetical code below:

```
// create a new array of pairs
Pair<String,Integer>[] pairArray = new Pair<String,Integer>[2];

// pass around the array of pairs as an array of object
Object[] objArray = pairArray;

// put a pair into the array -- no ArrayStoreException!
objArray[0] = new Pair<Double,Boolean>(3.14, true);
```

This is similar to what we have in Unit 21, where we showed we could get an `ArrayStoreException` due to Java arrays being covariant. We would not, however, get an exception when we try to put a pair of double and boolean, into an array meant to store a pair of string and integer! This type checking is done during run-time, and due to type erasure, the run-time has no information about what is the type arguments to `Pair`. The run-time sees:

```
// create a new array of pairs
Pair[] pairArray = new Pair[2];

// pass around the array of pairs as an array of object
Object[] objArray = pairArray;

// put a pair into the array -- no ArrayStoreException!
objArray[0] = new Pair(3.14, true);
```

It checks that we have an array of pairs and we are putting another pair inside. Everything checks out. This would have caused a *heap pollution*, a term that refers to the situation where a variable of a parameterized type refers to an object that is not of that parameterized type.

Heap pollution is dangerous, as now, we will get a `ClassCastException` when we do:

```
// getting back a string?  -- now we get ClassCastException
String str = pairArray[0].getFirst();
```

The example above shows why generics and arrays don't mix well together. An array is what is called *reifiable* type -- a type where full type information is available during run-time. It is because Java array is reifiable that the Java run-time can check what we store into the array matches the type of the array and throw an `ArrayStoreException` at us if there is a mismatch. Java generics, however, is not reifiable due to type erasure. Java designers have decided not to mix the two.

The hypothetical code above actually is not a valid Java syntax. We can't compile this line:

```
Pair<String,Integer>[] pairArray = new Pair<String,Integer>[2];
```

The following is illegal as well:

```
new Pair<S,T>[2];
new T[2];
```

Unit 25: Unchecked Warnings

After this unit, students should:

- be aware of how to use generics with an array
- be aware of unchecked warnings that compilers can give when we are using generics
- be able to make arguments why a piece of code is type-safe for simple cases
- know how to suppress warnings from compilers
- be aware of the ethics when using the @SuppressWarnings("unchecked") annotation
- know what is a raw type
- be aware that raw types should never never be used in modern Java

Creating Arrays with Type Parameters

We have seen how arrays and generics do not mix well. One way to get around this is to use Java Collections, a library of data structures provided by Java, instead of arrays, to store our items. The `ArrayList` class provides similar functionality to an array, with some performance overhead.

```
1  ArrayList<Pair<String,Integer>> pairList;
2  pairList = new ArrayList<Pair<String,Integer>>(); // ok
3
4  pairList.add(0, new Pair<Double,Boolean>(3.14, true));  // error
5
6  ArrayList<Object> objList = pairList;  // error
```

`ArrayList` itself is a generic class, and when parameterized, it ensures type-safety by checking for appropriate types during compile time. We can't add a `Pair<Double,Boolean>` object to a list of `Pair<String,Integer>`. Furthermore, unlike Java array, which is covariant, generics are invariant. There is no subtyping relationship between `ArrayList<Object>` and `ArrayList<Pair<String,Integer>>` so we can't alias one with another, preventing the possibility of heap pollution.

Using `ArrayList` instead of arrays only *gets around* the problem of mixing arrays and generics, as a user. `ArrayList` is implemented with an array internally after all. As computing students, especially computer science students, it is important to know how to

implement your own data structures instead of using ones provided by Java or other libraries.

Let's try to build one:

```
1  // version 0.1
2  class Array<T> {
3    private T[] array;
4
5    Array(int size) {
6      this.array = (T[]) new Object[size];
7    }
8
9    public void set(int index, T item) {
10     this.array[index] = item;
11   }
12
13   public T get(int index) {
14     return this.array[index];
15   }
16
17   public T[] getArray() {
18     return this.array;
19   }
20 }
```

This generic class is a wrapper around an array of type `T`. Recall that we can't `new T[]` directly. On Line 6, to get around this restriction, we `new` an `Object` array instead, and cast it to an array of `T[]` instead.

The code now compiles, but we receive the following message:

```
1  $ javac Array.java
2  Note: Array.java uses unchecked or unsafe operations.
3  Note: Recompile with -Xlint:unchecked for details.
```

Let's do what the compiler tells us, and compile with the `-Xlint:unchecked" flags.

```
1  $ javac -Xlint:unchecked Array.java
2  Array.java:6: warning: [unchecked] unchecked cast
3      array = (T[]) new Object[size];
4                    ^
5    required: T[]
6    found:    Object[]
7    where T is a type-variable:
8      T extends Object declared in class Array
9  1 warning
```

We get a warning that our Line 6 is doing an unchecked cast.

Unchecked Warnings

An unchecked warning is basically a message from the compiler that it has done what it can, and because of type erasures, there could be a run-time error that it cannot prevent. Recall that type erasure generates the following code:

```
(String) array.get(0);
```

Since `array` is an array of `Object` instances and Java array is covariant, the compiler can't guarantee that the code it generated is safe anymore.

Consider the following:

```
Array<String> array = new Array<String>(4);
Object[] objArray = array.getArray();
objArray[0] = 4;
array.get(0);  // ClassCastException
```

The last line would generate a `ClassCastException`, exactly a scenario that the compiler has warned us.

It is now up to us humans to change our code so that the code is safe. Suppose we remove the `getArray` method from the `Array` class,

```
// version 0.2
class Array<T> {
  private T[] array;

  Array(int size) {
    this.array = (T[]) new Object[size];
  }

  public void set(int index, T item) {
    this.array[index] = item;
  }

  public T get(int index) {
    return this.array[index];
  }
}
```

Can we prove that our code is type-safe? In this case, yes. Since `array` is declared as `private`, the only way someone can put something into the `array` is through the `Array::set` method[1]. `Array::set` only put items of type `T` into `array`. So the only type of objects we can get out of `array` must be of type `T`. So we, as humans, can see that casting `Object[]` to `T[]` is type-safe.

If we are sure (and only if we are sure) that the line

```
    array = (T[]) new Object[size];
```

is safe, we can thank the compiler for its warning and assure the compiler that everything is going to be fine. We can do so with the `@SuppressWarning("unchecked")` annotation.

```
// version 0.3
class Array<T> {
  private T[] array;

  Array(int size) {
    // The only way we can put an object into array is through
    // the method set() and we only put object of type T inside.
    // So it is safe to cast `Object[]` to `T[]`.
    @SuppressWarnings("unchecked")
    T[] a = (T[]) new Object[size];
    this.array = a;
  }

  public void set(int index, T item) {
    this.array[index] = item;
  }

  public T get(int index) {
    return this.array[index];
  }
}
```

`@SuppressWarning` is a powerful annotation that suppresses warning messages from compilers. Like everything that is powerful, we have the responsibility to use it properly:

- `@SuppressWarning` can apply to declaration at a different scope: a local variable, a method, a type, etc. We must always use `@SuppressWarning` to the *most limited* scope to avoid unintentionally suppressing warnings that are valid concerns from the compiler.
- We must suppress a warning *only if* we are sure that it will not cause a type error later.
- We must always add a note (as a comment) to fellow programmers explaining why a warning can be safely suppressed.

Note that since `@SuppressWarnings` cannot apply to an assignment but only to declaration, we declare a local variable `a` in the example above before assigning `this.array` to `a`.

Raw Types

Another common scenario where we can get an unchecked warning is the use of *raw types*. A raw type is a generic type used without type arguments. Suppose we do:

```
Array a = new Array(4);
```

The code would compile perfectly. We are using the generic `Array<T>` as a raw type `Array`. Java allows this code to compile for backward compatibility. This is anyway what the code looks like after type erasure and how we would write the code in Java before version 5. Without a type argument, the compiler can't do any type checking at all. We are back to the uncertainty that our code could bomb with `ClassCastException` after it ships.

Mixing raw types with paramterized types can also lead to errors. Consider:

```
Array<String> a = new Array<String>(4);
populateArray(a);
String s = a.get(0);
```

where the method `populateArray` uses raw types:

```
void populateArray(Array a) {
    a.set(0, 1234);
}
```

Since we use raw types in this method, the compiler can't help us. It will warn us:

```
Array.java:24: warning: [unchecked] unchecked call to set(int,T) as a
member of the raw type Array
    a.set(0, 1234);
         ^
  where T is a type-variable:
    T extends Object declared in class Array
1 warning
```

If we ignore this warning or worse, suppress this warning, we will get a run-time error when we execute `a.get(0)`.

Raw types must not be used in your code, ever. The only exception to this rule is using it as an operand of the `instanceof` operator. Since `instanceof` checks for run-time type and type arguments have been erased, we can only use the `instanceof` operator on raw types.

---

1. Another win for information hiding! ↵

Unit 26: Wildcards

After going through this unit, students should:

- be aware of the meaning of wildcard `?` and bounded wildcards
- know how to use wildcards to write methods that are more flexible in accepting a range of types
- know that upper-bounded wildcard is covariant and lower-bounded wildcard is contravariant
- know the PECS principle and how to apply it
- be aware that the unbounded wildcard allows us to not use raw types in our programs

`contains` with `Array<T>`

Now that we have our `Array<T>` class, let's modify our generic `contains` method and replace the type of the argument `T[]` with `Array<T>`.

```
1  class A {
2    // version 0.5 (with generic array)
3    public static <T> boolean contains(Array<T> array, T obj) {
4      for (int i = 0; i < array.getLength(); i++) {
5        T curr = array.get(i);
6        if (curr.equals(obj)) {
7          return true;
8        }
9      }
10     return false;
11   }
12 }
```

Similar to the version that takes in `T[]`, using generics allows us to constrain the type of the elements of the array and the object to search for to be the same. This allows the following code to type-check correctly:

```
1  Array<String> stringArray;
2  Array<Circle> circleArray;
3  Circle circle;
4   :
5  A.<String>contains(stringArray, "hello"); // ok
6  A.<Circle>contains(circleArray, circle); // ok
```

But trying to search for a circle in an array of strings would lead to a type error:

```
A.<String>contains(stringArray, circle); // error
```

Consider now having an array of shapes.

```
Array<Shape> shapeArray;
Array<Circle> circleArray;
Shape shape;
Circle circle;
 :
A.<Shape>contains(shapeArray, shape); // ok
A.<Circle>contains(circleArray, circle); // ok
```

As expected, we can pass `Shape` as the argument for `T`, and search for a `Shape` in an instance of `Array<Shape>`. Similarly, we can pass `Circle` as the argument for `T` and search for a `Circle` in an instance of `Array<Circle>`.

We could also look for a `Circle` instance from `Array<Shape>` if we pass `Shape` as the argument for `T`.

```
A.<Shape>contains(shapeArray, circle); // ok
```

Note that we can pass in a `Circle` instance as a `Shape`, since `Circle` <: `Shape`.

Recall that generics are invariant in Java, i.e, there is no subtyping relationship between `Array<Shape>` and `Array<Circle>`. `Array<Circle>` is not a subtype of `Array<Shape>`. Otherwise, it would violate the Liskov Substitution Principle, we can put a square into an `Array<Shape>` instance, but we can't put a square into an `Array<Circle>` instance.

So, we can't call:

```
A.<Circle>contains(shapeArray, circle); // compilation error
```

The following would result in compilation errors as well:

```
A.<Shape>contains(circleArray, shape); // compilation error
A.<Circle>contains(circleArray, shape); // compilation error
```

Thus, with our current implementation, we can't look for a shape (which may be a circle) in an array of circles, even though this is something reasonable that a programmer might want to do. This constraint is due to the invariance of generics -- while we avoided the possibility of run-time errors by avoiding covariance arrays, our methods have become less general.

Let's see how we can fix this with bounded type parameters first. We can introduce another type parameter, say `S`, to remove the constraints that the type of the array must be the same as the type of the object to search for. I.e., we change from

```
1    public static <T> boolean contains(Array<T> array, T obj) { .. }
```

to:

```
1    public static <S,T> boolean contains(Array<T> array, S obj) { .. }
```

But we don't want to completely decouple `T` and `S`, as we want `T` to be a subtype of `S`. We can thus make `T` a bounded type parameter, and write:

```
1    public static <S, T extends S> boolean contains(Array<T> array, S obj) {
   .. }
```

Now, we can search for a shape in an array of circles.

```
1       A.<Shape,Circle>contains(circleArray, shape);
```

Copying to and from `Array<T>`

Let's consider another example. Let's add two methods `copyFrom` and `copyTo`, to `Array<T>` so that we can copy to and from one array to another.

```
1  // version 0.4 (with copy)
2  class Array<T> {
3    private T[] array;
4
5    Array(int size) {
6    // The only way we can put an object into the array is through
7    // the method set() and we only put an object of type T inside.
8    // So it is safe to cast `Object[]` to `T[]`.
9    @SuppressWarnings("unchecked")
10     T[] a = (T[]) new Object[size];
11   this.array = a;
12   }
13
14   public void set(int index, T item) {
15     this.array[index] = item;
16   }
17
18   public T get(int index) {
19     return this.array[index];
20   }
21
22   public void copyFrom(Array<T> src) {
23     int len = Math.min(this.array.length, src.array.length);
```

```
24      for (int i = 0; i < len; i++) {
25          this.set(i, src.get(i));
26      }
27    }
28
29    public void copyTo(Array<T> dest) {
30      int len = Math.min(this.array.length, dest.array.length);
31      for (int i = 0; i < len; i++) {
32          dest.set(i, this.get(i));
33      }
34    }
35  }
```

With this implementation, we can copy, say, an `Array<Circle>` to another `Array<Circle>`, an `Array<Shape>` to another `Array<Shape>`, but not an `Array<Circle>` into an `Array<Shape>`, even though each circle is a shape!

```
1  Array<Circle> circleArray;
2  Array<Shape> shapeArray;
3    :
4  shapeArray.copyFrom(circleArray); // error
5  circleArray.copyTo(shapeArray); // error
```

Upper-Bounded Wildcards

Let's consider the method `copyFrom`. We should be able to copy from an array of shapes, an array of circles, an array of squares, etc, into an array of shapes. In other words, we should be able to copy from *an array of any subtype of shapes* into an array of shapes. Is there such a type in Java?

The type that we are looking for is `Array<? extends Shape>`. This generic type uses the *wildcard* `?`. Just like a wild card in card games, it is a substitute for any type. A wildcard can be bounded. Here, this wildcard is upper-bounded by `Shape`, i.e., it can be substituted with either `Shape` or any subtype of `Shape`.

The upper-bounded wildcard is an example of covariance. The upper-bounded wildcard has the following subtyping relations:

- If `S` <: `T`, then `A<? extends S>` <: `A<? extends T>` (covariance)
- For any type `S`, `A<S>` <: `A<? extends S>`

For instance, we have:

- `Array<Circle>` <: `Array<? extends Circle>`
- Since `Circle` <: `Shape`, `Array<? extends Circle>` <: `Array<? extends Shape>`
- Since subtyping is transitive, we have `Array<Circle>` <: `Array<? extends Shape>`

Because `Array<Circle>` <: `Array<? extends Shape>`, if we change the type of the parameter to `copyFrom` to `Array<? extends T>`,

```
  public void copyFrom(Array<? extends T> src) {
    int len = Math.min(this.array.length, src.array.length);
    for (int i = 0; i < len; i++) {
        this.set(i, src.get(i));
    }
  }
```

We can now call:

```
shapeArray.copyFrom(circleArray); // ok
```

without error.

Lower-Bounded Wildcards

Let's now try to allow copying of an `Array<Circle>` to `Array<Shape>`.

```
circleArray.copyTo(shapeArray);
```

by doing the same thing:

```
  public void copyTo(Array<? extends T> dest) {
    int len = Math.min(this.array.length, dest.array.length);
    for (int i = 0; i < len; i++) {
        dest.set(i, this.get(i));
    }
  }
```

The code above would not compile. We will get the following somewhat cryptic message when we compile with the `-Xdiags:verbose` flag:

```
Array.java:32: error: method set in class Array<T> cannot be applied to
given types;
        dest.set(i, this.get(i));
          ^
  required: int,CAP#1
  found: int,T
  reason: argument mismatch; T cannot be converted to CAP#1
  where T is a type-variable:
    T extends Object declared in class Array
  where CAP#1 is a fresh type-variable:
    CAP#1 extends T from capture of ? extends T
1 error
```

Let's try not to understand what the error message means first, and think about what could go wrong if the compiler allows:

```
1           dest.set(i, this.get(i));
```

Here, we are trying to put an instance with compile-time type `T` into an array that contains elements with the compile-time type of `T` or subtype of `T`.

The `copyTo` method of `Array<Shape>` would allow an `Array<Circle>` as an argument, and we would end up putting instance with compile-time type `Shape` into `Array<Circle>`. If all the shapes are circles, we are fine, but there might be other shapes (rectangles, squares) in `this` instance of `Array<Shape>`, and we can't fit them into `Array<Circle>`! Thus, the line

```
1           dest.set(i, this.get(i));
```

is not type-safe and could lead to `ClassCastException` during run-time.

Where can we copy our shapes into? We can only copy them safely into an `Array<Shape>`, `Array<Object>`, `Array<GetAreable>`, for instance. In other words, into arrays containing `Shape` or supertype of `Shape`.

We need a wildcard lower-bounded by `Shape`, and Java's syntax for this is `? super Shape`. Using this new notation, we can replace the type for `dest` with:

```
1   public void copyTo(Array<? super T> dest) {
2     int len = Math.min(this.array.length, dest.array.length);
3     for (int i = 0; i < len; i++) {
4         dest.set(i, this.get(i));
5     }
6   }
```

The code would now type-check and compile.

The lower-bounded wildcard is an example of contravariance. We have the following subtyping relations:

- If `S` <: `T`, then `A<? super T>` <: `A<? super S>` (contravariance)
- For any type `S`, `A<S>` <: `A<? super S>`

For instance, we have:

- `Array<Shape>` <: `Array<? super Shape>`
- Since `Circle` <: `Shape`, `Array<? super Shape>` <: `Array<? super Circle>`

- Since subtyping is transitive, we have `Array<Shape>` <: `Array<? super Circle>`

The line of code below now compiles:

```
circleArray.copyTo(shapeArray);
```

Our new `Array<T>` is now

```
// version 0.5 (with flexible copy using wildcards)
class Array<T> {
  private T[] array;

  Array(int size) {
  // The only way we can put an object into the array is through
  // the method set() and we only put an object of type T inside.
  // So it is safe to cast `Object[]` to `T[]`.
  @SuppressWarnings("unchecked")
    T[] a = (T[]) new Object[size];
  this.array = a;
  }

  public void set(int index, T item) {
    this.array[index] = item;
  }

  public T get(int index) {
    return this.array[index];
  }

  public void copyFrom(Array<? extends T> src) {
    int len = Math.min(this.array.length, src.array.length);
    for (int i = 0; i < len; i++) {
        this.set(i, src.get(i));
    }
  }

  public void copyTo(Array<? super T> dest) {
    int len = Math.min(this.array.length, dest.array.length);
    for (int i = 0; i < len; i++) {
        dest.set(i, this.get(i));
    }
  }
}
```

PECS

Now we will introduce the rule that governs when we should use the upper-bounded wildcard `? extends T` and a lower-bounded wildcard `? super T`. It depends on the role of the variable. If the variable is a producer that returns a variable of type `T`, it should be

declared with the wildcard `? extends T`. Otherwise, if it is a consumer that accepts a variable of type `T`, it should be declared with the wildcard `? super T`.

As an example, the variable `src` in `copyFrom` above acts as a *producer*. It produces a variable of type `T`. The type parameter for `src` must be either `T` or a subtype of `T` to ensure type safety. So the type for `src` is `Array<? extends T>`.

On the other hand, the variable `dest` in `copyTo` above acts as a *consumer*. It consumes a variable of type `T`. The type parameter of `dest` must be either `T` or supertype of `T` for it to be type-safe. As such, the type for `dest` is `Array<? super T>`.

This rule can be remembered with the mnemonic PECS, or "Producer Extends; Consumer Super".

Unbounded Wildcards

It is also possible to have unbounded wildcards, such as `Array<?>`. `Array<?>` is the supertype of every parameterized type of `Array<T>`. Recall that `Object` is the supertype of all reference types. When we want to write a method that takes in a reference type, but we want the method to be flexible enough, we can make the method accept a parameter of type `Object`. Similarly, `Array<?>` is useful when you want to write a method that takes in an array of some specific type, and you want the method to be flexible enough to take in an array of any type. For instance, if we have:

```
1  void foo(Array<?> array) {
2  }
```

We could call it with:

```
1  Array<Circle> ac;
2  Array<String> as;
3  foo(ac); // ok
4  foo(as); // ok
```

A method that takes in generic type with unbounded wildcard would be pretty restrictive, however. Consider this:

```
1  void foo(Array<?> array) {
2     :
3    x = array.get(0);
4    array.set(0, y);
5
6  }
```

What should the type of the returned element `x` be? Since `Array<?>` is the supertype of all possible `Array<T>`, the method `foo` can receive an instance of `Array<Circle>`, `Array<String>`, etc. as an argument. The only safe choice for the type of `x` is `Object`.

The type for `y` is every more restrictive. Since there are many possibilities of what type of array it is receiving, we can only put `null` into `array`!

There is an important distinction to be made between `Array`, `Array<?>` and `Array<Object>`. Whilst `Object` is the supertype of all `T`, it does not follow that `Array<Object>` is the supertype of all `Array<T>` due to generics being invariant. Therefore, the following statements will fail to compile:

```
Array<Object> a1 = new Array<String>(0);
Array<Object> a2 = new Array<Integer>(0);
```

Whereas the following statements will compile:

```
Array<?> a1 = new Array<String>(0); // Does compile
Array<?> a2 = new Array<Integer>(0); // Does compile
```

If we have a function

```
void bar(Array<Object> array) {
}
```

Then, the method `bar` is restricted to *only* takes in an `Array<Object>` instance as argument.

```
Array<Circle> ac;
Array<String> as;
bar(ac); // compilation error
bar(as); // compilation error
```

What about raw types? Suppose we write the method below that accepts a raw type

```
void qux(Array array) {
}
```

Then, the method `qux` is also flexible enough to take in any `Array<T>` as argument.

```
Array<Circle> ac;
Array<String> as;
qux(ac);
qux(as);
```

Unlike `Array<?>`, however, the compiler does not have the information about the type of the component of the array, and cannot type check for us. It is up to the programmer to ensure type safety. For this reason, we must not use raw types.

Intuitively, we can think of `Array<?>`, `Array<Object>`, and `Array` as follows:

- `Array<?>` is an array of objects of some specific, but unknown type;
- `Array<Object>` is an array of `Object` instances, with type checking by the compiler;
- `Array` is an array of `Object` instances, without type checking.

Back to `contains`

Now, let's simplify our `contains` methods with the help of wildcards. Recall that to add flexibility into the method parameter and allow us to search for a shape in an array of circles, we have modified our method into the following:

```
1  class A {
2    // version 0.6 (with generic array)
3    public static <S,T extends S> boolean contains(Array<T> array, S obj) {
4      for (int i = 0; i < array.getLength(); i++) {
5        T curr = array.get(i);
6        if (curr.equals(obj)) {
7          return true;
8        }
9      }
10     return false;
11   }
12 }
```

Can we make this simpler using wildcards? Since we want to search for an object of type `S` in an array of its subtype, we can remove the second parameter type `T` and change the type of array to `Array<? extends S>`:

```
1  class A {
2    // version 0.7 (with wild cards array)
3    public static <S> boolean contains(Array<? extends S> array, S obj) {
4      for (int i = 0; i < array.getLength(); i++) {
5        S curr = array.get(i);
6        if (curr.equals(obj)) {
7          return true;
8        }
9      }
10     return false;
11   }
12 }
```

We can double-check that `array` is a producer (it produces `curr` on Line 5) and this follows the PECS rules. Now, we can search for a shape in an array of circles.

```
1       A.<Shape>contains(circleArray, shape);
```

Revisiting Raw Types

In previous units, we said that you may use raw types only in two scenarios. Namely, when using generics and `instanceof` together, and when creating arrays. However, with unbounded wildcards, we can now see it is possible to remove both of these exceptions. We can now use `instanceof` in the following way:

```
1  a instanceof A<?>
```

Recall that in the example above, `instanceof` checks of the run-time type of `a`. Previously, we said that we can't check for, say,

```
1  a instanceof A<String>
```

since the type argument `String` is not available during run-time due to erasure. Using `<?>` fits the purpose here because it explicitly communicates to the reader of the code that we are checking that `a` is an instance of `A` with some unknown (erased) type parameter.

Similarly, we can create arrays in the following way:

```
1  new Comparable<?>[10];
```

Previously, we said that we could not create an array using the expression `new Comparable<String>[10]` because generics and arrays do not mix well. Java insists that the array creation expression uses a *reifiable* type, i.e., a type where no type information is lost during compilation. Unlike `Comparable<String>`, however, `Comparible<?>` is reifiable. Since we don't know what is the type of `?`, no type information is lost during erasure!

Going forward now in the module, we will not permit the use of raw types in any scenario.

Unit 27: Type Inference

After this unit, students should:

- be familiar how Java infers missing type arguments

We have seen in the past units the importance of types in preventing run-time errors. Utilizing types properly can help programmers catch type mismatch errors that could have caused a program to fail during run-time, possibly after it is released and shipped.

By including type information everywhere in the code, we make the code explicit in communicating the intention of the programmers to the readers. Although it makes the code more verbose and cluttered -- it is a small price to pay for ensuring the type correctness of the code and reducing the likelihood of bugs as the code complexity increases.

Java, however, allows the programmer to skip some of the type annotations and try to infer the type argument of a generic method and a generic type, through the *type inference* process.

The basic idea of type inference is simple: Java will looking among the matching types that would lead to successful type checks, and pick the most specific ones.

Diamond Operator

One example of type inference is the diamond operator `<>` when we `new` an instance of a generic type:

```
1  Pair<String,Integer> p = new Pair<>();
```

Java can infer that `p` should be an instance of `Pair<String,Integer>` since the compile-time type of `p` is `Pair<String,Integer>`. The line above is equivalent to:

```
1  Pair<String,Integer> p = new Pair<String,Integer>();
```

Type Inferencing

We have been invoking

```
class A {
  // version 0.7 (with wild cards array)
  public static <S> boolean contains(Array<? extends S> array, S obj) {
    for (int i = 0; i < array.getLength(); i++) {
      S curr = array.get(i);
      if (curr.equals(obj)) {
        return true;
      }
    }
    return false;
  }
}
```

by explicitly passing in the type argument `Shape` (also called *type witness* in the context of type inference).

```
      A.<Shape>contains(circleArray, shape);
```

We could remove the type argument `<Shape>` so that we can call `contains` just like a non-generic method:

```
      A.contains(circleArray, shape);
```

and Java could still infer that `S` should be `Shape`. The type inference process looks for all possible types that match. In this example, the type of the two parameters must match. Let's consider each individually first:

- An object of type `Shape` is passed as an argument to the parameter `obj`. So `S` might be `Shape` or, if widening type conversion has occurred, one of the other supertypes of `Shape`. Therefore, we can say that `Shape <: S <: Object`.
- An `Array<Circle>` has been passed into `Array<? extends S>`. A widening type conversion occurred here, so we need to find all possible `S` such that `Array<Circle>` <: `Array<? extends S>`. This is true only if `S` is `Circle`, or another supertype of `Circle`. Therefore, we can say that `Circle <: S <: Object`.

Solving for these two constraints on `S`, we get the following:

```
Shape <: S <: Object
```

We therefore know that `S` could be `Shape` or one of its supertypes: `GetAreable` and `Object`. We choose the lower bound, so `S` is inferred to be `Shape`.

Type inferencing can have unexpected consequences. Let's consider an older version of `contains` that we wrote:

```
class A {
    // version 0.4 (with generics)
    public static <T> boolean contains(T[] array, T obj) {
      for (T curr : array) {
        if (curr.equals(obj)) {
          return true;
        }
      }
      return false;
    }
}
```

Recall that we want to prevent nonsensical calls where we are searching for an integer in an array of strings.

```
String[] strArray = new String[] { "hello", "world" };
A.<String>contains(strArray, 123); // type mismatch error
```

But, if we write:

```
A.contains(strArray, 123); // ok!  (huh?)
```

The code compiles! Let's go through the type inferencing steps to understand what happened. Again, we have two parameters:

- `strArray` has the type `String[]` and is passed to `T[]`. So `T` must be `String` or its superclass `Object` (i.e. `String <: T <: Object`). The latter is possible since Java array is covariant.
- `123` is passed as type `T`. The value is treated as `Integer` and, therefore, `T` must be either `Integer`, or its superclasses `Number`, and `Object` (i.e. `Integer <: T <: Object`).

Solving for these two constraints:

```
T <: Object
```

Therefore `T` can only have the type `Object`, so Java infers `T` to be `Object`. The code above is equivalent to:

```
A.<Object>contains(strArray, 123);
```

And our version 0.4 of `contains` actually is quite fragile and does not work as intended. We were bitten by the fact that the Java array is covariant, again.

Target Typing

The example above performs type inferencing on the parameters of the generic methods. Type inferencing can involve the type of the expression as well. This is known as *target typing*. Take the following upgraded version of `findLargest`:

```
// version 0.6 (with Array<T>)
public static <T extends GetAreable> T findLargest(Array<? extends T>
array) {
  double maxArea = 0;
  T maxObj = null;
  for (int i = 0; i < array.getLength(); i++) {
    T curr = array.get(i);
    double area = curr.getArea();
    if (area > maxArea) {
      maxArea = area;
      maxObj = curr;
    }
  }
  return maxObj;
}
```

and we call

```
Shape o = A.findLargest(new Array<Circle>(0));
```

We have a few more constraints to check:

- Due to target typing, the returning type of `T` must be a subtype of `Shape` (i.e. `T <: Shape`)
- Due to the bound of the type parameter, `T` must be a subtype of `GetAreable` (i.e. `T <: GetAreable`)
- `Array<Circle>` must be a subtype of `Array<? extends T>`, so `T` must be a supertype of `Circle` (i.e. `Circle <: T <: Object`)

Solving for all three of these constraints:

```
Circle <: T <: Shape
```

The lower bound is `Circle`, so the call above is equivalent to:

```
Shape o = A.<Circle>findLargest(new Array<Circle>(0));
```

Further Type Inference Examples

We now return to our `Circle` and `ColoredCircle` classes and the `GetAreable` interface. Recall that `Circle` implements `GetAreable` and `ColoredCircle` inherits from `Circle`.

Now lets consider the following method signature of a generic method `foo`:

```
1  public <T extends Circle> T foo(Array<? extends T> array)
```

Then we consider the following code excerpt:

```
1  ColoredCircle c = foo(new Array<GetAreable>());
```

What does the java compiler infer `T` to be? Lets look at all of the constraints on `T`.

- First we can say that the return type of `foo` must be a subtype of `ColoredCircle`, therefore we can say `T <: ColoredCircle`.
- `T` is also a bounded type parameter, and therefore we also know `T <: Circle`.
- Our method argument is of type `Array<GetAreable>` and must be a subtype of `Array<? extends T>`, so `T` must be a supertype of `GetAreable` (i.e. `GetAreable <: T <: Object`).

We can see that there no solution to our contraints, `T` can not be both a subtype of `ColoredCircle` and a supertype of `GetAreable` and therefore the Java compiler can not find a type `T`. The Java compiler will throw an error stating the inference variable `T` has incompatible bounds.

Lets consider, one final example using the following method signature of a generic method `bar`:

```
1  public <T extends Circle> T bar(Array<? super T> array)
```

Then we consider the following code excerpt:

```
1  GetAreable c = bar(new Array<Circle>());
```

What does the java compiler infer `T` to be? Again, lets look at all of the constraints on `T`.

- We can say that the return type of `bar` must be a subtype of `GetAreable`, therefore we can say `T <: GetAreable`.
- Our method argument is of type `Array<Circle>` and must be a subtype of `Array<? super T>`, so `T` must be a subtype of `Circle` (i.e. `T <: Circle`).

Solving for these two constraints:

```
T <: Circle
```

Whilst `ColoredCircle` is also a subtype of `Circle` it is not included in the above statement and therefore the compiler does not consider this class during type inference. Indeed, the compiler cannot be aware[1] of all subtypes of `Circle` and there could be more than one subtype. Therefore `T` can only have the type `Circle`, so Java infers `T` to be `Circle`.

Rules for Type Inference

We now summarize the steps for type inference. First, we figure out all of the type constraints on our type parameters, and then we solve these constraints. If no type can satisfy all the constraints, we know that Java will fail to compile. If in resolving the type constraints for a given type parameter `T` we are left with:

- `Type1 <: T <: Type2`, then `T` is inferred as `Type1`
- `Type1 <: T`[2], then `T` is inferred as `Type1`
- `T <: Type2`, then `T` is inferred as `Type2`

where `Type1` and `Type2` are arbitrary types.

---

1. Due to evolving specifications of software, at the time of compilation, a subtype may not have even been conceived of or written yet! ↩
2. Note that `T <: Object` is implicit here. We can see that this case could also be written as `Type1 <: T <: Object`, and would therefore also be explained by the previous case (`Type1 <: T <: Type2`). ↩