Subprograms: Generic Programming

Programming Languages

William Killian

Millersville University

Outline

- Function Overloading
- Variable Number of Arguments
  - C
  - C++
  - Java
  - Javascript
  - Python/Ruby
- Generic Types
  - OCaml / F# wildcard type
  - Java Generics vs. Wildcards
  - C++ Templates with SFINAE/Concepts
  - Scripting Languages

Function Overloading

- A function is said to be overloaded when the **name** is the *same* but the **signature** is *different*.
- Usually resolved via parameter type analysis

```
int
median (int x, int y, int z);

double
median (double x, double y, double z);
```

Function Overloading Rules

1. The same function name is used for more than one function definition

2. The functions must differ either by the arity or types of their parameters

3. The usage within a program must be resolved at compile / analysis time rather than at runtime

Function Overloading

```
// circle
double area (double r) {
  return 3.1415926535 * r * r;
}
// rectangle
double area (double x, double y) {
  return x * y;
}
// triangle
double area (double a, double b, double c) {
  double s = (a + b + c) / 2.0;
  return sqrt (s * (s - a) * (s - b) * (s - c));
}
```

Function Overloading: C++/Java

- Constructors can also be overloaded!

```
class MyADT {
  MyADT (); // default
  MyADT (int size); // size constructor
  MyADT (MyADT other); // Java - copy constructor
  MyADT (const MyADT& other); // C++ - copy constructor
  MyADT (MyADT&& other); // C++ - move constructor
  . . .
}
```

Function Overloading Issues

Pros

Cons

Function Overloading Issues

Pros

- **Versatile** – use names that make sense and are common
- **Customizable** – introduce same behavior for new classes or datatypes

Cons

- Difficult to discern at function call
- Type coercion could result in unintended function being called
- Lots of overloads -> longer compilation / resolution

Variable Number of Arguments

Variable Number of Arguments

- Some functions should be able to accept any number of arguments, not just a limited number

What classes of functions should support this?

Variable Arguments: C

- The oldest language that supports variable arguments
- An afterthought – an extension to the language (`stdarg.h`)
- All variable arguments must come at the end

```
int sum (int n, ...) {
    va_list args;
    va_start (args, n);
    int s = 0;
    for (int i = 0; i < n; ++i) {
        s += va_arg (args, int);
    }
    va_end (args);
    return s;
}
```

Implemented via assistive macros:

- `va_list`
- `va_start`
- `va_arg`
- `va_end`

Variable Arguments: C++

- The template argument `typename ...Ts` is a *template parameter pack*
- The argument `Ts... ts` is a *parameter pack*
- The expression `(ts + ...)` is a fold expression
- You can do what C does, too!

```
template <typename ...Ts>
auto sum (Ts... ts) {
    return (ts + ...);
}
```

Variable Arguments: Java

- Java allows the user to specify any number of arguments with a triple-dot syntax
- The named formal parameter acts like an array
- No parameters may come after the variable argument

```
int sum (int... values) {
    int s = 0;
    for (int v : values) {
        s += v;
    }
    return s;
}
```

Variable Arguments: JavaScript

- JavaScript allows the user to specify any number of arguments with a triple-dot syntax
- The named formal parameter acts like an array
- No parameters may come after the variable argument

```
function sum (... values) {
  return values.reduce ((s, v) => s + v);
}
```

Variable Arguments: Python/Ruby

- Both Python and Ruby offer a special syntax for a parameter that allows for any number of arguments.
- An **asterisk** denotes that a formal parameter corresponds to zero or more *actual parameters*
- A **double asterisk** denotes that a formal parameter corresponds to zero or more *named actual parameters* (as a dictionary/map)
- Strict Ordering Requirement
  - Normal Parameters
  - *args (variable arguments)
  - **kwargs (keyword arguments)

Variable Arguments: Python/Ruby

- An **asterisk** denotes that a formal parameter corresponds to zero or more *actual parameters*

```
def mysum(*nums):
    s = 0
    for v in nums:
        s += v
    return s
```

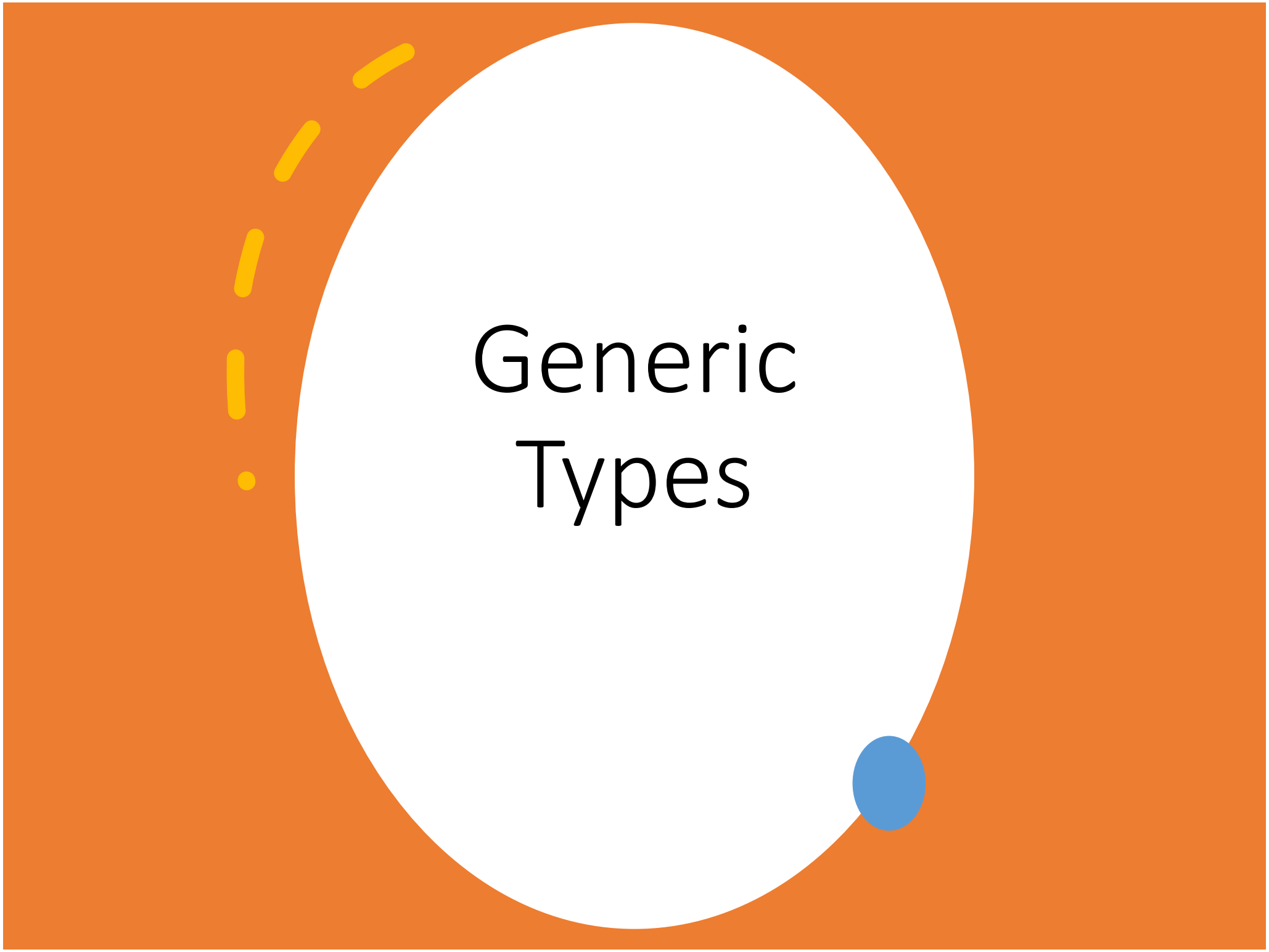
```
def mysum(*nums)
    s = 0
    nums.each do |v|
        s += v
    end
    return s
```

Variable Arguments: Python

- A **double asterisk** denotes that a formal parameter corresponds to zero or more *named actual parameters* (as a dictionary/map)

```
def my_arg_checker(**args):
    if "magic" in args:
        print (f'magic={args["magic"]}')
    else:
        return False
    return True

my_arg_checker (a=4)
my_arg_checker (b=3, magic=42)
```


Generic
Types

OCaml / F# Wildcard type

```
type int_option = Some of int | None
type 'a option = Some of 'a | None
```

- 'a is a type that means “any”
- All wildcard types will start with a single quote
- There’s a bunch more to OCaml type theory that we just AREN’T going to cover

Java Generics

- Allow a programmer to specify “any type” in Java
- Decays to Object type

```
public class Node<E> {
    private E data;
    private Node<E> next;
}
```

Java Generics with Constraints

- Allow a programmer to specify “any type” in Java
- Called an *upper-bound*
- Decays to the constraint type (Comparable<E>)

```
public class Node<E extends Comparable<E>>
{
    private E data;
    private Node<E> next;
}
```

Java Wildcards

- Generics must be used at class-level declarations, but wildcards can be used for function arguments
- ? decays to Object type

```
public static int size(Collection<?> obj) {
    return obj.size();
}
```

Java Wildcards with Constraints

- We can specify a type *upper bound* with Wildcards
  - `extends`
  - decays to the constrained type
- Or we can specify a type *lower bound* with Wildcards
  - `super`
  - decays to Object

```
public void bar(List<? super Integer> list)

public void foo(List<? extends Number> nums)
```

Java Generics + Wildcards

- Generics can impose a **lower bound**
- Wildcards can impose an **upper bound**
- Full type safety!

```
public static <T extends Number>
void
adder(T elem, List<? super Number> list)
{
    list.add(elem);
}
```

C++ Templates

- Classes, types, functions can have template parameters
- Template Parameters can be any of:
  - Type
  - Enumeration
  - Integral Constant
- Template parameters are specified with the `template` keyword at definition, and contained within `<>` at the callsite/usage

C++ Templates: ADTs

```
template <typename T>
class
vector {
    T data;
    int size;
    int capacity;
public:
    . . .
};
```

Whatever type T has is the **exact type** that is used throughout

C++ Templates with Default: ADTs

```
template <
  typename T,
  typename Alloc = std::allocator<T>>
class
vector {
    T data;
    int size;
    int capacity;
public:
    . . .
};
```

If we omit the second type parameter, `std::allocator<T>` will be used for `Alloc`

C++ Templates: functions

```
template <typename T>
T
max (T a, T b) {
    if (a > b) {
        return a;
    } else {
        return b;
    }
}
```

Both parameters are copied by value!

BTW, `std::vector` is comparable

C++ Templates: functions

```
template <typename T>
T
max (T const &a, T const &b) {
    if (a > b) {
        return a;
    } else {
        return b;
    }
}
```

The result is still copied back!

C++ Templates: functions

```
template <typename T>
T const &
max (T const &a, T const &b) {
    if (a > b) {
        return a;
    } else {
        return b;
    }
}
```

No extra copies made now

C++ Templates: Type Constraints

```
template <typename T>
class
vector {
public:
  template <typename Iter>
  vector (Iter first, Iter last);

  vector (size_t size, T const& value);
  ...
};
```

What if T == size_t ?

C++ Templates: Type Constraints

```
template <typename T>
class
vector {
public:
  template <typename Iter>
  vector (Iter first, Iter last);

  vector (size_t size, size_t const& value);

  ...
};
```

What if T == size_t ?

Template overload resolution failure!
Multiple candidates found

C++ Templates: Type Constraints

```
template <typename T>  where T = size_t
class
vector {
public:
  template <typename Iter>
  vector (Iter first, Iter last);

  vector (size_t size, size_t const& value);
  ...
};
```

Solution:
Constrain the range-constructor to only work with iterators

C++ Templates: Type Constraints

- An Iterator is *Dereferencable* and *Incrementable*

Type Constraint:

```
std::enable_if_t<decltype(
  *std::declval<Iter>(),     // *iter
  ++std::declval<Iter>()    // ++iter
)>
```

- **`decltype`**`(e)`: the evaluated type of e
- `std::declval<T>`: give me some instance of T

If it fails, then it's not going to yield a compiler error

C++ Templates: Type Constraints

```
template <typename T>
class
vector {
public:
  template <
    typename Iter
    typename = std::enable_if_t<decltype(
       (++std::declval<Iter>(),
         *std::declval<Iter>()))>>
  vector (Iter first, Iter last);

  vector (size_t size, T const& value);
  ...
};
```

SFINAE:
Substitution
Failure
Is
Not
An
Error

C++ Templates: Concepts

- New for C++20

```
template <typename T>
class
vector {
public:
  template <std::input_iterator Iter>
  vector (Iter first, Iter last);

  vector (size_t size, T const& value);
  ...
};
```

Scripting Languages

- Scripting languages are arguably the most generic
- Why?
  - No type definitions are required
  - Often interpreted, so unreachable code is never evaluated.